Майкл Дженесерет
Винай К. Чаудри

Введение в логическое программирование

ДМК
издательство

Майкл Дженесерет
Винай К. Чаудри

Введение в логическое программирование

Introduction to Logic Programming

Michael Genesereth
Vinay K. Chaudhri

MORGAN & CLAYPOOL
PUBLISHERS

Введение в логическое программирование

Майкл Дженесерет
Винай К. Чаудри

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2022

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.

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

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 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.

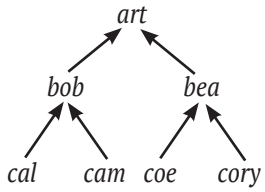

Рис. 2.1. Представление информации графом

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.

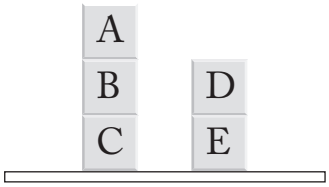

Рис. 2.3. Состояние мира блоков

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.

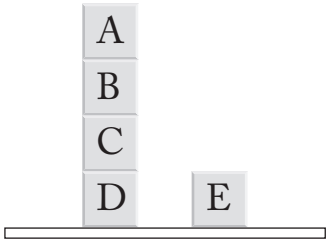

Рис. 2.4. Состояние мира блоков для упражнения

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?

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

Часть II
Запросы и обновления

Глава 3

Запросы

3.1. Введение

В главе 2 рассмотрено представление состояния области применения в виде набора данных. Если набор данных большой, может быть трудно ответить на вопросы, основанные на этом наборе данных. В этой главе мы рассмотрим различные способы *запроса* к набору данных, чтобы найти именно ту информацию, которая нам нужна.

Самая простая форма запроса – это вопрос «*Истина или ложь?*». Имея фактоид и набор данных, мы можем узнать, является ли фактоид истинным в этом наборе данных или нет. Например, мы хотим узнать, является ли Арт родителем Боба. Ответ на вопрос «Истина или ложь?» – это просто проверка того, входит ли данный фактоид в набор данных.

Более интересной формой запроса является запрос «*Заполнение пробелов*». Если имеется фактоид с пробелами, могут понадобиться значения, которые, будучи подставленными вместо пробелов, сделают запрос истинным. Например, мы можем захотеть найти детей Арта, родителей Билла или пары родителей и детей.

Еще более интересной формой запроса является *составной запрос*. Нам могут понадобиться значения, для которых булева комбинация условий является истинной. Например, мы захотим узнать, является ли Арт родителем Боба или родителем Бада. Или захотим найти всех людей, у которых есть сыновья и нет дочерей.

В начале этой главы мы рассмотрим расширение нашего языка наборов данных, которое позволяет выражать такие вопросы, в следующем разделе определим синтаксис языка, а в разделе далее – его семантику. Затем мы рассмотрим несколько примеров использования этого языка для запросов к наборам данных, после чего остановимся на важном синтаксическом ограничении, называемом безопасностью. И наконец, в заключение мы обсудим полезные предопределенные понятия (например, арифметические операторы), которые увеличивают возможности языка запросов.

3.2. Синтаксис запросов

Язык запросов включает в себя язык наборов данных, но предоставляет некоторые дополнительные возможности, которые делают его более выразительным, а именно переменные и правила запросов. Переменные позволяют нам писать запросы с заполнением пробелов. Правила запросов позволяют нам выражать составные запросы, в частности отрицания (чтобы сказать, что условие ложно), конъюнкции (чтобы сказать, что все несколько условий истинны) и дизъюнкции (чтобы сказать, что хотя бы одно из нескольких условий истинно).

В нашем языке запросов *переменная* – это либо единичное подчеркивание, либо строка букв, цифр и знаков подчеркивания, начинающаяся с заглавной буквы. Например, `_`, `X23`, `X_23` и `Somebody` являются переменными.

Атомарное предложение, или *атом*, аналогично фактоиду в наборе данных, за исключением того, что его аргументы могут включать как переменные, так и символы. Например, если `p` – бинарный предикат, `a` – символ, а `Y` – переменная, то `p(a,Y)` является атомарным предложением.

Литерал – это либо атом, либо отрицание атома. Атом называется положительным литералом. Отрицание атома называется отрицательным литералом. В дальнейшем мы будем писать отрицательные литералы с помощью знака отрицания `~`. Например, если `p(a,b)` – это атом, то `~p(a,b)` обозначает отрицание этого атома. Оба они являются литералами.

Правило запроса – это выражение, состоящее из выделенного атома, называемого *головой*, и набора из нуля или более литералов, называемого *телом*. Литералы в теле называются *подцелями*. Предикат в голове правила запроса должен быть новым предикатом (т. е. не входящим в словарь набора данных), а все предикаты в теле должны быть предикатами набора данных.

Далее мы будем писать правила так, как показано в примере ниже. Здесь `goal(a,b)` – это голова; `p(a,b) & ~q(b)` – тело; `p(a,b)` и `~q(b)` – подцели.

```
goal(a,b) :- p(a,b) & ~q(b)
```

Как мы увидим в следующем разделе, правило запроса – это нечто вроде обратного следствия. Это утверждение, что голова правила (т. е. общая цель) истинна всякий раз, когда истинны подцели. Например, правило выше утверждает, что `goal(a,b)` истинна, *если* `p(a,b)` истинно *и* `q(b)` *не* истинно.

Выразительная сила правил запроса значительно увеличивается благодаря использованию переменных. Рассмотрим, например, правило, показанное ниже. Это более общая версия правила, показанного выше. Вместо того чтобы применяться только к конкретным объектам `a` и `b`,

оно применяется ко *всем* объектам. В этом случае правило гласит, что цель истинна для *любого* объекта `X` и *любого* объекта `Y`, если `p` истинно для `X` и `Y`, а `q` не истинно для `Y`.

```
goal(X,Y) :- p(X,Y) & ~q(Y)
```

Запрос – это непустой конечный набор правил запроса. Как правило, запрос состоит всего из одного правила. На самом деле большинство систем логического программирования не поддерживает запросы с несколькими правилами (по крайней мере, напрямую). Однако запросы с несколькими правилами иногда полезны и не добавляют сложности, поэтому далее мы допускаем возможность запросов с несколькими правилами.

3.3. Семантика запроса

Экземпляр выражения (атома, литерала или правила) – это выражение, в котором все переменные заменены основными элементами (т. е. элементами без переменных). Например, если у нас есть язык с символами `a` и `b`, то ниже показаны экземпляры `goal(X,Y) :- p(X,Y) & ~q(Y)`.

```
goal(a,a) :- p(a,a) & ~q(a)
goal(a,b) :- p(a,b) & ~q(b)
goal(b,a) :- p(b,a) & ~q(a)
goal(b,b) :- p(b,b) & ~q(b)
```

Исходя из этого можно определить результат применения правила к набору данных. Если есть правило `r` и набор данных Δ, мы определяем $\upsilon(r,\Delta)$ как множество всех таких ψ, что:

- ψ является головой произвольного экземпляра r;
- каждая позитивная подцель в этом экземпляре является членом Δ;
- ни одна негативная подцель в экземпляре не является членом Δ.

Расширение запроса – это множество всех фактов, которые могут быть «выведены» на основе правил программы, т. е. это объединение $\upsilon(r_i,\Delta)$ для каждого r_i в нашем запросе.

Для иллюстрации этих определений рассмотрим набор данных, описывающий небольшой направленный граф. В предложениях ниже используем символы для обозначения узлов графа и отношение `p` для обозначения стрелок графа.

```
p(a,b)
p(b,c)
```

```
p(c,b)
```

Теперь предположим, что задан запрос, в котором предикат `goal` является истинным для каждого узла, имеющего исходящую стрелку к другому узлу, а также входящую стрелку от этого узла.

```
goal(X) :- p(X,Y) & p(Y,X)
```

Поскольку здесь имеются две переменные и три символа, существует девять экземпляров этого правила, которые показаны ниже.

```
goal(a) :- p(a,a) & p(a,a)
goal(a) :- p(a,b) & p(b,a)
goal(a) :- p(a,c) & p(c,a)
goal(b) :- p(b,a) & p(a,b)
goal(b) :- p(b,b) & p(b,b)
goal(b) :- p(b,c) & p(c,b)
goal(c) :- p(c,a) & p(a,c)
goal(c) :- p(c,b) & p(b,c)
goal(c) :- p(c,c) & p(c,c)
```

Тело в первом из них не удовлетворяется. Фактически тело истинно только в шестом и восьмом экземплярах. Следовательно, расширение этого запроса содержит только два атома, показанные ниже.

```
goal(b)
goal(c)
```

Определение семантики в терминах экземпляров правил просто и понятно. Однако системы логического программирования обычно не производят обработку запросов таким образом. Существуют более эффективные способы вычисления таких расширений. В последующих главах мы рассмотрим некоторые алгоритмы такого рода.

3.4. Безопасность

Правило запроса является *безопасным* тогда и только тогда, когда каждая переменная, которая появляется в голове или в любом отрицательном литерале тела, также встречается хотя бы в одном положительном литерале тела.

Правило, показанное ниже, является безопасным. Каждая переменная в голове и отрицательной подцели появляется и в положительной подцели тела. Обратите внимание, что тело может содержать переменные, которые не встречаются в голове.

```
goal(X) :- p(X,Y,Z) & ~q(X,Z)
```

В отличие от этого два правила, показанные ниже, не безопасны. Первое правило не безопасно, так как переменная Z появляется в голове, но не появляется ни в одной положительной подцели. Второе правило не безопасно, так как переменная Z появляется в отрицательной подцели, но не появляется ни в одной положительной подцели.

```
goal(X,Y,Z) :- p(X,Y)
goal(X,Y,X) :- p(X,Y) & ~q(Y,Z)
```

Чтобы понять, почему безопасность существенна в случае первого правила, предположим, что у нас есть база данных, в которой p(a,b) является истинным. Тогда тело первого правила будет выполнено, если X – это a, а Y – это b. В этом случае можно заключить, что каждый экземпляр головы истинен. Но чем мы должны заменить Z? Интуитивно мы можем подставить туда что угодно, но вариантов может быть много. Хотя концептуально это нормально, но практически проблематично.

Чтобы увидеть, почему безопасность имеет значение во втором правиле, предположим, что имеется база данных, содержащая всего два факта, а именно p(a,b) и q(b,c). В этом случае, если X – это a, Y – это b, а Z – что угодно, кроме c, обе подцели истинны, и мы можем заключить goal(a,b,a).

Основная проблема с этим заключается в том, что многие неправильно интерпретируют отрицание как утверждение того, что не существует Z, для которого q(Y,Z) истинно, тогда как правильное прочтение заключается в том, что q(Y,Z) должно быть ложным только для одного значения Z. Как мы увидим, существуют различные способы выразить это значение без написания не безопасных запросов.

3.5. Предопределенные понятия

В практических языках логического программирования принято предопределять полезные понятия. К ним обычно относятся арифметические функции (такие как сложение, умножение, максимум, минимум), строковые функции (такие как конкатенация), равенство и неравенство, агрегаты (например, countofall) и т. д.

В языке Epilog равенства и неравенства выражаются с помощью отношений same и distinct. Предложение same(σ,τ) истинно, когда σ и τ идентичны. Предложение distinct(σ,τ) истинно, когда σ и τ различны.

Отношение evaluate используется для представления уравнений, включающих предопределенные функции. Например, мы напишем evaluate(plus(times(3,3),times(2,3),1),16), чтобы представить уравнение $3 \times 3 + 2 \times 3 + 1 = 16$. Если height является бинарным предикатом, связывающим фигуру и ее высоту, и если width – бинарный предикат, относя-

щийся к фигуре и ее ширине, мы можем определить площадь объекта, как показано ниже. Площадь фигуры X равна A, если высота X равна H, ширина X равна W, а A – результат умножения H и W.

```
goal(X,A) :- height(X,H) & width(X,W) & evaluate(times(H,W),A)
```

В языках логического программирования, предоставляющих такие предопределенные понятия, обычно существуют синтаксические ограничения на их использование. Например, если запрос содержит подцель с отношением сравнения (например, same и distinct), то каждая переменная, которая встречается в этой подцели, должна встречаться по крайней мере в одном положительном литерале в теле запроса, и это появление должно предшествовать подцели с отношением сравнения. Если в запросе используется evaluate в подцели, то любая переменная, которая встречается в первом аргументе этой подцели, должна встречаться по крайней мере в одном положительном литерале в теле запроса, и это должно предшествовать подцели с арифметическим отношением. Подробности обычно можно найти в документации систем, которые предоставляют такие встроенные понятия.

В практических языках логического программирования также принято включать предопределенные агрегатные операторы, такие как setofall и countofall.

Агрегатные операторы обычно представляются в виде отношений со специальным синтаксисом. Например, следующее правило использует оператор countofall для запроса количества детей человека. N является числом детей X тогда и только тогда, когда N является числом всех Y таких, что X является родителем Y.

```
goal(X,N) :- person(X) & evaluate(countofall(Y,parent(X,Y)),N)
```

Как и в случае со специальными отношениями, существуют синтаксические ограничения на использование агрегатных операторов. В частности, их подцели должны быть безопасными в том смысле, что все переменные второго аргумента должны быть включены в первый аргумент или должны использоваться в положительных подцелях правила, содержащего агрегат.

3.6. Пример – родственные связи

Рассмотрим вариант примера родства, представленного в главе 2. В этом случае наш словарь состоит из символов (представляющих людей) и бинарного предиката parent (истинного тогда и только тогда, когда человек, указанный в качестве первого аргумента, является родителем человека, указанного в качестве второго аргумента).

Имея данные о родительских отношениях, выраженные с помощью этого словаря, можно написать запросы, чтобы извлечь информацию и о других отношениях. Например, можно найти бабушек, дедушек и внуков, написав запрос, показанный ниже. `X` является бабушкой или дедушкой `Z`, если `X` является родителем `Y`, а `Y` – родителем `Z`. Переменная `Y` здесь является потоковой переменной, которая связывает первую подцель со второй, но сама не появляется в голове правила.

```
goal(X,Z) :- parent(X,Y) & parent(Y,Z)
```

В общем случае можно писать запросы с несколькими правилами. Например, собрать всех людей, упомянутых в нашем наборе данных, написав следующий запрос с несколькими правилами. В этом случае условия дизъюнктивны (хотя бы одно из них должно быть истинным), тогда как условия для приведенного выше запроса являются конъюнктивными (оба должны быть истинными).

```
goal(X) :- parent(X,Y)
goal(Y) :- parent(X,Y)
```

В некоторых случаях в запросах полезно использовать встроенные отношения. Например, можно найти все пары людей, которые являются родными братьями и сестрами, написав правило запроса, показанное ниже. При этом мы используем условие `distinct`, чтобы не указывать человека как собственного родного брата или сестру.

```
goal(Y,Z) :- parent(X,Y) & parent(X,Z) & distinct(Y,Z)
```

Хотя возможно выразить многие распространенные родственные отношения с помощью нашего языка запросов, есть некоторые отношения, которые просто слишком сложны. Например, не существует способа найти всех предков человека (родителей, бабушек, дедушек, прабабушек, прадедушек и т. д.). Для этого нужна возможность писать *рекурсивные* запросы. Мы покажем, как писать такие запросы, в главе, посвященной определениям *представлений*.

3.7. Пример – раскрашивание карт

Рассмотрим проблему раскрашивания плоских карт, используя только четыре цвета, причем каждому региону надо присвоить цвет так, чтобы никакие два смежных региона не имели одинакового цвета.

Типичная карта показана ниже. На ней есть шесть регионов. Некоторые из них являются смежными друг с другом, это означает, что им нельзя присвоить один и тот же цвет. Другие не являются смежными, им может быть присвоен одинаковый цвет.

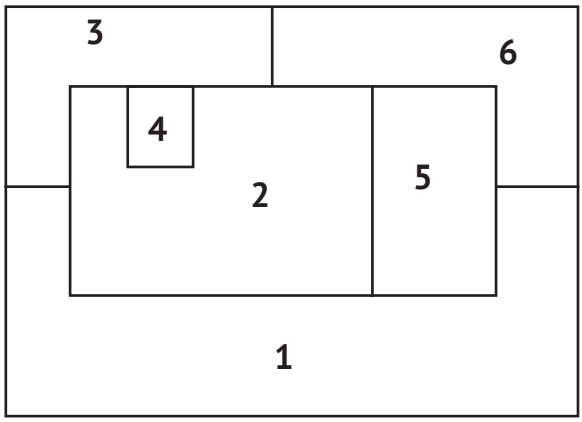

Рис. 3.1. Типичная карта

Можно перечислить оттенки, которые будут использоваться, как показано ниже. Константы `red`, `green`, `blue` и `purple` обозначают оттенки красного, зеленого, голубого и фиолетового соответственно.

```
hue(red)
hue(green)
hue(blue)
hue(purple)
```

В случае показанной выше карты наша цель – найти шесть оттенков (по одному для каждого региона карты) так, чтобы никакие два смежных региона не имели одинакового оттенка. Мы можем выразить эту цель, написав следующий запрос.

```
goal(C1,C2,C3,C4,C5,C6) :-
hue(C1) & hue(C2) & hue(C3) & hue(C4) & hue(C5) & hue(C6) &
distinct(C1,C2) & distinct(C1,C3) & distinct(C1,C5) & distinct(C1,C6) &
distinct(C2,C3) & distinct(C2,C4) & distinct(C2,C5) & distinct(C2,C6) &
distinct(C3,C4) & distinct(C3,C6) & distinct(C5,C6)
```

В результате выполнения этого запроса будут получены наборы по шесть оттенков, которые гарантируют, что никакие два смежных региона не имеют одинакового цвета. В задачах, подобных этой, мы обычно хотим получить только одно решение, а не все. Однако поиск даже одного решения в таких случаях может оказаться дорогостоящим. В главе 4 мы обсудим способы составления запросов, которые делают процесс поиска таких решений более эффективным.

3.8. Упражнения

3.1. Для каждой из следующих строк скажите, является ли она синтаксически допустимым запросом.

(a) `goal(X) :- p(a,f(f(X)))`
(b) `goal(X,Y) :- p(X,Y) & ~p(Y,X)`
(c) `~goal(X,Y) :- p(X,Y) & p(Y,X)`
(d) `goal(P,Y) :- P(a,Y)`
(e) `goal(X) :- p(X,b) & p(X,p(b,c))`

3.2. Скажите, является ли каждый из следующих запросов безопасным.

(a) `goal(X,Y) :- p(X,Y) & p(Y,X)`
(b) `goal(X,Y) :- p(X,Y) & p(Y,Z)`
(c) `goal(X,Y) :- p(X,X) & p(X,Z)`
(d) `goal(X,Y) :- p(X,Y) & ~p(Y,Z)`
(e) `goal(X,Y) :- p(X,Y) & ~p(Y,Y)`

3.3. Каков результат выполнения запроса `goal(X,Z) :- p(X,Y) & p(Y,Z)` для следующего набора данных?

```
p(a,b)
p(a,c)
p(b,d)
p(c,d)
```

3.4. Предположим, что существует набор данных с бинарным предикатом `parent` (который является истинным для двух людей, если человек, указанный в качестве первого аргумента, является родителем человека, указанного в качестве второго аргумента). Напишите запрос, определяющий свойство быть бездетным. *Подсказка*: используйте агрегатный оператор `countofall` (подсчет всех). И убедитесь, что ваш запрос безопасен. (Это упражнение несложное, но немного хитрое.)

3.5. Для каждой из следующих задач напишите запрос для ее решения. Значения должны включать только цифры 8, 1, 4, 7, 3, и каждая цифра должна быть использована не более одного раза в решении каждой задачи. Ваш запрос должен формулировать задачу в том виде, в котором она изложена, т. е. вы не должны сначала решить задачу самостоятельно, а затем просто вывести ответ.

(a) Произведение однозначного и двухзначного числа равно 284.
(b) Произведение двух двухзначных чисел плюс однозначное число равно 3 355.

(c) Произведение трехзначного числа и однозначного минус однозначное число равно 1 137.

(d) Произведение двухзначного числа и трехзначного находится между 13 000 и 14 000.

(e) При делении трехзначного числа на двухзначное результат будет между 4 и 6.

Глава 4

Обновления

4.1. Введение

В предыдущей главе было описано создание запросов для извлечения информации из набора данных. В данной главе мы рассмотрим, как обновить информацию в наборе данных, т. е. как преобразовать один набор данных в другой, в идеале не переписывая все фактоиды, а концентрируясь только на тех из них, которые изменили свои значения.

4.2. Синтаксис обновлений

Как и наш язык запросов, язык обновлений включает в себя язык наборов данных, но предоставляет некоторые дополнительные возможности. Опять же, в нем есть переменные, но в этом случае вместо правил запроса используются правила обновления.

Правило обновления – это выражение, состоящее из непустой коллекции литералов (называемых *условиями)* и второй непустой коллекции литералов (называемых *результатами*). Условия и результаты могут быть основными элементами или содержать переменные. Существует только одно ограничение: все переменные в результатах должны также присутствовать в положительных условиях.

Правила обновления записываются, как в примере, показанном ниже. Здесь `p(a,b)` и `~q(b)` – условия, а `~p(a,b)` и `p(b,a)` – результаты.

```
p(a,b) & ~q(b) ==> ~p(a,b) & p(b,a)
```

Как мы увидим в следующем разделе, правило обновления является чем-то вроде правила «условие–действие». Это утверждение о том, что всякий раз, когда условия истинны, отрицательные выводы должны быть удалены из набора данных, а положительные выводы должны быть добавлены в него. Например, приведенное выше правило гласит, что если `p(a,b)` истинно, а `q(b)` ложно, то `p(a,b)` должно быть удалено из набора данных, а `p(b,a)` – добавлено.

Как и в случае с правилами запроса, возможности правил обновления значительно расширяются благодаря использованию переменных. Рассмотрим, например, правило, показанное ниже. Это более общая версия показанного выше правила. Вместо того чтобы применяться только к конкретным объектам `a` и `b`, оно применимо ко всем объектам.

```
p(X,Y) & ~q(Y) ==> ~p(X,Y) & p(Y,X)
```

Обновление – это конечная коллекция правил обновления. Как правило, обновление состоит только из одного правила. Однако обновления с несколькими правилами иногда полезны и не добавляют значительной сложности, поэтому в дальнейшем мы допускаем возможность обновлений со многими правилами.

4.3. Семантика обновлений

Экземпляр правила обновления – это правило, в котором все переменные были заменены основными элементами (т. е. элементами без переменных). Например, если у нас есть язык с символами `a` и `b`, то ниже показаны случаи `p(X,Y) & ~q(Y) ==> ~p(X,Y) & p(Y,X)`.

```
p(a,a) & ~q(a) ==> ~p(a,a) & p(a,a)
p(a,b) & ~q(b) ==> ~p(a,b) & p(b,a)
p(b,a) & ~q(a) ==> ~p(b,a) & p(a,b)
p(b,b) & ~q(b) ==> ~p(b,b) & p(b,b)
```

Предположим, что дан набор данных Δ и правило обновления `r`. Мы говорим, что *экземпляр* `r` *активен* на Δ, если и только если все условия истинны для Δ. Мы определяем положительное множество обновлений $A(r, \Delta)$ как множество всех положительных результатов в некотором активном экземпляре `r`; и определяем отрицательное множество $D(r, \Delta)$ как множество всех отрицательных результатов в некотором активном экземпляре `r`.

Множество положительных обновлений $A(\Omega, \Delta)$ для набора правил Ω на наборе данных Δ является объединением положительных обновлений правил на Δ; и множество отрицательных обновлений $D(\Omega, \Delta)$ для Ω является объединением отрицательных обновлений правил на Δ.

Наконец, мы получаем результат применения набора правил обновления R к набору данных Δ путем удаления отрицательных обновлений и добавления положительных обновлений, т. е. результат есть $\Delta - D(\Omega, \Delta) \cap A(\Omega, \Delta)$.

Давайте рассмотрим несколько примеров, иллюстрирующих эту семантику. Рассмотрим набор данных, показанный ниже. В нем имеется четыре символа и один бинарный предикат `p`.

```
p(a,a)
p(a,b)
p(b,c)
p(c,c)
p(c,d)
```

Предположим, что мы хотим удалить все фактоиды `p` в наборе данных, в которых первый и второй аргументы одинаковы. Для этого зададим обновление, показанное ниже.

```
p(X,X) ==> ~p(X,X)
```

Ниже представлены два экземпляра, для которых условия истинны.

```
p(a,a) ==> ~p(a,a)
p(c,c) ==> ~p(c,c)
```

Следовательно, после выполнения этого обновления мы получим следующий набор данных.

```
p(a,b)
p(b,c)
p(c,d)
```

Теперь предположим, что мы хотим поменять местами аргументы всех фактоидов в полученном наборе данных. Для этого зададим обновление с `p(X,Y)` в качестве условия, `p(X,Y)` в качестве отрицательного результата и `p(Y,X)` в качестве положительного результата. В этом случае мы получим одно назначение переменной для каждого фактоида в нашем наборе данных; отрицательные результаты будут `{p(a,b), p(b,c), p(c,d)}`, т. е. каждый фактоид в нашем наборе данных; а положительные результаты будут `{p(b,a), p(c,b), p(d,c)}`. Обновление для этого случая показано ниже.

```
p(X,Y) ==> ~p(X,Y) & p(Y,X)
```

После его выполнения получим набор данных:

```
p(b,a)
p(c,b)
p(d,c)
```

4.4. Одновременные обновления

Обратите внимание, что иногда случается так, что фактоид является как положительным, так и отрицательным обновлением. В качестве примера рассмотрим обновление с `p(X,a)` в качестве условия, `p(X,a)` в

качестве отрицательного результата и с p(a,X) в качестве положительного результата.

```
p(X,a) ==> ~p(X,a) & p(a,X)
```

В случае с первым набором данных, показанным в предыдущем разделе, p(a,a) будет являться как положительным, так и отрицательным обновлением.

В таких случаях наша семантика диктует, что фактоид должен быть удален, а затем снова добавлен, в результате чего не произойдет никаких изменений. Это достаточно произвольное разрешение конфликта, но, по-видимому, именно ему чаще всего отдают предпочтение программисты.

4.5. Пример – родство

Предположим, как и раньше, что мы начинаем с единственного бинарного предиката parent (являющегося истинным для двух людей тогда и только тогда, когда человек, указанный в качестве первого аргумента, является родителем человека, указанного в качестве второго аргумента).

Фактоиды, показанные ниже, представляют собой набор данных, использующий этот словарь. Человек по имени art является родителем человека по имени bob и человека по имени bea, bob является родителем cal и cam, а bea – родителем coe и cory.

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

В главе 3 было рассмотрено составление запросов, характеризующих другие родственные отношения, с использованием предиката parent. В некоторых случаях мы можем захотеть хранить полученные фактоиды, чтобы к ним можно было получить доступ без повторных вычислений.

Предположим, например, что мы хотим хранить информацию о бабушках, дедушках и их внуках. Это можно сделать, написав обновление, подобное показанному ниже.

```
parent(X,Y) & parent(Y,Z) ==> grandparent(X,Z)
```

В этом случае к нашему набору данных будут добавлены следующие фактоиды:

```
grandparent(art,cal)
grandparent(art,cam)
grandparent(art,cat)
grandparent(art,coe)
```

Если впоследствии мы захотим удалить эти фактоиды, возможно выполнить обновление, показанное ниже, и мы вернемся к тому, с чего начали.

```
grandparent(X,Y) ==> ~grandparent(X,Z)
```

Теперь предположим, что мы хотим поменять местами аргументы предиката `parent`, связав детей и родителей, а не родителей и детей.

Чтобы сделать это, напишем следующее обновление:

```
parent(X,Y) ==> ~parent(X,Y) & parent(Y,X)
```

В результате его выполнения получим следующий набор данных:

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

Для понимания обновлений, подобных этому, важно помнить, что обновления происходят *атомарно* – сначала вычисляются все фактоиды, которые должны быть изменены, затем вносятся все эти изменения сразу. И только после этого можно рассматривать любые другие обновления.

4.6. Пример – цвета

Руби Красная, Уилла Белая и Бетти Синяя встречаются за обедом. На одной из них надета красная юбка, на другой – белая, на третьей – синяя. Никто не одет более чем в один цвет, и нет двух одинаковых цветов. Бетти Синяя говорит одной из своих спутниц: «Ты заметила, что мы все носим юбки разных цветов, отличающихся от наших имен?» – и другая женщина, которая в белой юбке, говорит: «Это верно!» Наша задача – выяснить, кто из женщин носит юбку какого цвета.

Один из способов решения подобных задач – перечисление возможностей и проверка каждой из них, удовлетворяет ли она ограничениям в постановке задачи. Этот подход работает, но он часто требует большого объема поиска. Чтобы сделать процесс поиска решений более эффективным, иногда возможно использовать уже известные нам

значения для вывода дополнительных значений и тем самым сократить количество рассматриваемых вариантов. В этом разделе мы рассмотрим, как можно использовать обновления для реализации этой техники. В этом особом случае, как мы увидим, эта техника исключает поиск вообще.

Для решения задачи мы используем словарь с шестью символами: `r`, `w`, `b`, `v`, `x` и `e`. Первые три обозначают людей или цвета, а последние три – «значения истинности»: истинное, ложное и неизвестное.

Чтобы выразить состояние нашей проблемы, мы используем константу троичного отношения `c`. Например, мы можем записать `c(r,w,v)`, чтобы обозначить, что Руби Красная носит белую юбку; `c(r,w,x)` означает, что Руби Красная не носит белую юбку; и `c(r,w,e)` означает, что мы не знаем, носит ли Руби Красная белую юбку или нет.

При решении этой задачи мы начинаем с набора данных, показанного ниже. Изначально мы ничего не знаем о том, кто во что одет.

```
c(r,r,e)
c(r,w,e)
c(r,b,e)
c(w,r,e)
c(w,w,e)
c(w,b,e)
c(b,r,e)
c(b,w,e)
c(b,b,e)
```

Мы можем отобразить эту ситуацию, как показано в табл. 4.1. При этом значение в каждой ячейке таблицы отражает наше мнение о том, носит ли человек, указанный в качестве первого аргумента в соответствующем `c` фактоиде, юбку того цвета, который указан в качестве второго аргумента. Для ясности – мы оставляем ячейки пустыми, если они имеют значение `e`.

Таблица 4.1. Начальное состояние задачи

	r	w	b
r			
w			
b			

Сначала применим ограничение, что ни одна из женщин не носит юбку того же цвета, что и ее имя.

```
c(C,C,e) ==> ~c(C,C,e) & c(C,C,x)
```

После этого обновления получим состояние, показанное в табл. 4.2. Теперь появляются значения × по диагонали, но остальные ячейки по-прежнему пустые.

Таблица 4.2. Состояние после обновления
`c(C,C,e) ==> ~c(C,C,e) & c(C,C,x)`

	r	w	b
r	×		
w		×	
b			×

Далее примем во внимание обращение Бетти Синей к кому-то, кто носит белую юбку, что означает, что Бетти Синяя не носит белую юбку.

```
c(b,w,e) ==> ~c(b,w,e) & c(b,w,x)
```

В результате получим ситуацию, показанную в табл. 4.3.

Таблица 4.3. Состояние после обновления
`c(b,w,e) ==> ~c(b,w,e) & c(b,w,x)`

	r	w	b
r	×		
w		×	
b		×	×

Теперь переходим к самому интересному. Во-первых, у нас есть обновления, которые говорят нам, что если есть два вхождения `x` в строке или столбце, а в третьей ячейке – `e`, то в этой ячейке должна быть `v`.

```
c(P,C1,x) & c(P,C2,x) & c(P,C3,e) ==> ~c(P,C3,e) & c(P,C3,v)
c(P1,C,x) & c(P2,C,x) & c(P3,C,e) ==> ~c(P3,C,e) & c(P3,C,v)
```

Однократное применение этих обновлений приводит к ситуации, изображенной в табл. 4.4. Так как ни Уилла Белая, ни Бетти Синяя не носят белую юбку, Руби Красная должна быть одета в белое.

Таблица 4.4. Определение цвета для Руби

	r	w	b
r	×	✓	
w		×	
b		×	×

Аналогично есть обновления, которые говорят нам, что если в строке или столбце встречается символ v и есть ячейка, содержащая e, то это e должно быть заменено на x.

```
c(P,C1,v) & c(P,C2,e) ==> ~c(P,C2,e) & c(P,C2,x)
c(P1,C,v) & c(P2,C,e) ==> ~c(P2,C,e) & c(P2,C,x)
```

Применение этих обновлений дает нам больше информации. Поскольку Руби Красная носит белую юбку, она не должна носить синюю юбку. Это указано в табл. 4.5.

Таблица 4.5. Состояние после обновлений

```
c(P,C1,v) & c(P,C2,e) ==> ~c(P,C2,e) & c(P,C2,x)
c(P1,C,v) & c(P2,C,e) ==> ~c(P2,C,e) & c(P2,C,x)
```

	r	w	b
r	✗	✓	✗
w		✗	
b		✗	✗

Применение этих обновлений еще три раза приводит к полному решению задачи, показанному в табл. 4.6. Поскольку ни Руби Красная, ни Бетти Синяя не носят синюю юбку, Уилла Белая должна носить синюю. Следовательно, Уилла Белая не может быть в красном. И, следовательно, Бетти Синяя должна быть одета в красное.

Таблица 4.5. Полное решение

	r	w	b
r	✗	✓	✗
w	✗	✗	✓
b	✓	✗	✗

Эта проблема специфична тем, что мы можем решить ее исключительно путем вывода значений из других значений. В проблемах удовлетворения ограничений, подобных этой, часто необходим некоторый поиск. Тем не менее подобные методы распространения ограничений часто могут сократить его объем, даже если они не могут быть использованы для решения проблемы полностью.

Этот случай также особенен тем, что в нем легко выразить все правила обновления, необходимые для решения проблемы. Для некоторых проблем, таких как решение головоломок судоку, нецелесообразно писать правила обновления, используя наш ограниченный язык обнов-

лений. К счастью, как мы увидим в последующих главах, есть возможность легко выражать более сложные правила, применяя определения представлений и определения действий.

4.7. Упражнения

4.1. Для каждой из следующих строк скажите, является ли она синтаксически корректным обновлением:

(a) `p(a,f(f(X))) ==> p(X,Y);`

(b) `P(a,Y) ==> P(Y,a);`

(c) `p(X,Y) & p(Y,Z) ==> ~(X,Y) & ~p(Y,Z) & p(X,Z);`

(d) `p(X,b) ==> f(X,f(b,c))`.

4.2. Каков результат применения правила обновления `p(X,Y) ==> ~p(X,Y) & p(Y,X)` к набору данных, показанному ниже?

```
p(a,a)
p(a,b)
p(b,a)
```

4.3. Предположим, у нас есть набор данных родства с бинарным предикатом `parent` (родитель) и унарным предикатом `male` (мужчина). Напишите правила обновления для замены всех фактоидов, использующих предикат `parent`, эквивалентными фактоидами, использующими бинарные предикаты `father` (отец) и `mother` (мать).

4.4. Эми, Боб, Ко и Дэн путешествуют в разные места. Один едет на поезде, другой – на машине, третий летит самолетом, еще один плывет на корабле. Эми ненавидит летать. Боб взял напрокат свой автомобиль. Ко страдает морской болезнью. А Дэн любит поезда. Напишите правила обновления для решения этой задачи с помощью распространения ограничений.

Глава 5

Оценка запросов

5.1. Введение

В главе 3 была описана семантика запросов в терминах экземпляров правил запросов. Хотя это определение легко понять и оно математически точно, перечисление экземпляров не является практическим методом для вычисления ответов на запросы. В этой главе будет представлен алгоритм, дающий те же результаты, но более эффективным способом.

Мы начинаем эту главу с обсуждения оценки запросов без переменных. В разделе 5.3 рассматривается способ сравнения выражений, содержащих переменные. В следующем разделе показано, как объединить эту технику с описанной здесь процедурой для оценки запросов с переменными. В заключение проведен анализ вычислительной сложности алгоритма оценки.

5.2. Оценка базовых запросов

Если запрос содержит несколько правил запроса, мы проверяем, истинно ли тело каждого правила. Если да, то голова правила добавляется к расширению запроса. Процедура определения истинности тела правила зависит от типа тела.

1. Если тело представляет собой один атом, мы проверяем, содержится ли этот атом в наборе данных. Если да, то тело истинно.
2. Если тело является отрицательным атомом, мы проверяем, содержится ли этот атом в наборе данных. Если да, то тело ложно. Если нет, то тело истинно.
3. Если тело является конъюнкцией литералов, то сначала мы выполняем эту процедуру для первого литерала. Если он истинен, мы переходим к следующему конъюнкту и т. д., пока не закончим. Если любой из конъюнктов ложен, то тело в целом ложно.

Рассмотрим набор данных, показанный ниже. В нем имеются четыре символа a, b, c и d; и один бинарный предикат p.

```
p(a,b)
p(b,c)
p(c,d)
```

Теперь представим, что нас просят оценить запрос, показанный ниже. В нем есть три правила. Чтобы получить результат, выполняем нашу процедуру на каждом из этих правил.

```
goal(a) :- p(a,c)
goal(b) :- p(a,b) & p(b,a)
goal(c) :- p(c,d) & ~p(d,c)
```

В первом правиле тело p(a,c) является атомом, поэтому проверяем, есть ли оно в наборе данных. Поскольку его там нет, в результат ничего не вносится.

Тело второго правила является конъюнкцией p(a,b) & p(b,a), поэтому оцениваем конъюнкты, чтобы проверить, все ли они истинны. В данном случае первый конъюнкт истинен, а второй – ложен, поэтому конъюнкция в целом ложна, и снова ничего не добавляется к нашему результату.

Тело третьего правила также представляет собой конъюнкцию p(c,d) & ~p(d,c). Опять же, проверяем конъюнкты по очереди. p(c,d) истинно, поэтому переходим к ~p(d,c). p(d,c) ложно, поэтому отрицание истинно. Поскольку оба конъюнкта конъюнкции истинны, конъюнкция в целом истинна. Следовательно, мы добавляем к результату голову правила goal(c).

5.3. Сопоставление

Сопоставление – это процесс определения соответствия *шаблона* (выражения с переменными или без них) образцу (выражению без переменных), т. е. можно ли сделать эти два выражения идентичными с помощью соответствующих замен переменных в шаблоне.

Подстановка – это конечное множество *привязок* переменных к элементам. В дальнейшем будем писать подстановки как наборы правил замены, как показано ниже. В каждом правиле переменная, на которую указывает стрелка, должна быть заменена элементом, от которого направлена стрелка. В нашем случае X связан с a, а Y – с b.

```
{X←a, Y←b}
```

Результатом *применения* подстановки σ к выражению φ является выражение φσ, полученное из выражения φ путем замены каждого вхождения переменной элементом, с которым она связана.

```
p(X,b) {X←a, Y←b} = p(a,b)
q(X,Y,X){X a, Y b} = q(a,b,a)
```

Подстановка является матчером для образца и экземпляра тогда и только тогда, когда применение подстановки к шаблону приводит к данному экземпляру. Одним из положительных моментов нашего языка является то, что существует простая и недорогая процедура для вычисления матчера шаблона и выражения, если он существует.

Эта процедура предполагает представление выражений в виде последовательностей подвыражений. Например, выражение `p(X,b)` можно представить как последовательность с тремя элементами – предикатом `p`, переменной `X` и символом `b`. Мы начинаем процедуру с двух выражений и подстановки (которая изначально пуста), затем рекурсивно обрабатываем оба выражения, сравнивая подвыражения на каждом шаге. Попутно мы расширяем подстановку с помощью описанного ниже присвоения значений переменным.

1. Если шаблон является символом, а экземпляр – тем же символом, то процедура завершается успешно, возвращая в качестве результата немодифицированный шаблон.
2. Если шаблон – символ, а экземпляр – другой символ или составное выражение, то процедура завершается неудачно.
3. Если шаблон является переменной с привязкой, сравниваем привязку для переменной с заданным экземпляром. Если они идентичны, процедура завершается успешно, возвращая в качестве результата немодифицированную подстановку; в противном случае процедура завершается неудачно.
4. Если шаблон является переменной без привязки, включаем привязку для переменной и возвращаем эту подстановку в качестве результата.
5. Если шаблон является составным выражением, а экземпляр – составным выражением такой же длины, выполняем итерацию по шаблону и экземпляру.
6. Если шаблон является составным выражением, а экземпляр – символом или составным выражением другой длины, процедура завершается неудачно.

Если не удается сопоставить подшаблон и подэкземпляр на любом шаге этого процесса, процедура в целом завершится неудачей. Если мы завершаем рекурсивное сравнение шаблона и экземпляра, процедура в целом завершается успешно, и накопленная в этот момент подстановка является результирующим матчером.

В качестве примера работы этой процедуры рассмотрим процесс сопоставления шаблона `p(X,Y)` и экземпляра `p(a,b)` с начальной подста-

новкой {}. Результаты выполнения процедуры для этого случая показаны ниже. Сравниваемые выражения и входная подстановка показаны в строке с меткой Compare, а результат каждого сравнения – в строке с меткой Result. Отступы показывают глубину рекурсии процедуры.

```
Compare: p(X,Y), p(a,b), {}
    Compare: p, p, {}
    Result: {}
    Compare: X, a, {}
    Result: {X←a}
    Compare: Y, b, {X←a}
    Result: {X←a, Y←b}
Result: {X←a, Y←b}
```

В качестве другого примера рассмотрим процесс сопоставления шаблона p(X,X) и экземпляра p(a,a). Ниже показана трассировка. К тому моменту, когда мы сравниваем последние аргументы двух выражений, X связан с a, поэтому сопоставление проходит успешно.

```
Compare: p(X,X), p(a,a), {}
    Compare: p, p, {}
    Result: {}
    Compare: X, a, {}
    Result: {X←a}
    Compare: X, a, {X←a}
    Result: {X←a}
Result: {X←a}
```

В качестве последнего примера рассмотрим процесс попытки сопоставления шаблона p(X,X) и выражения p(a,b). Основной интерес в этом примере представляет сравнение последнего аргумента в двух выражениях, а именно X и b. К тому времени, когда мы достигаем этой точки, X связан с a, а соответствующим экземпляром является b. Поскольку шаблон – это символ, а экземпляр – другой символ, попытка не удалась.

```
Compare: p(X,X), p(a,b), {}
    Compare: p, p, {}
    Result: {}
    Compare: X, a, {}
    Result: {X←a}
    Compare: X, b, {X←a}
    Result: false
Result: false
```

Эта процедура сопоставления довольно проста. Однако ее стоит понять досконально, так как она является основой для более сложных процедур сопоставления, определенных и используемых в последующих главах.

5.4. Оценка запросов с переменными

Оценка запросов с переменными осложняется тем, что может существовать несколько привязок переменных, которые делают тело правила истинным, и, следовательно, может быть несколько возможных ответов. В некоторых случаях нам нужен только один ответ, в некоторых – несколько ответов, а в других случаях нам нужны все ответы. Далее мы говорим о процедуре генерации всех ответов. Процедуры для других случаев аналогичны.

При поиске расширения для произвольного правила запроса (т. е. с переменными или без них) мы начинаем с правила запроса и пустой подстановки. Вместо того чтобы просто проверять, истинно или ложно тело, как в основном случае, мы вычисляем множество всех привязок переменных, для которых тело истинно, и для каждой из них мы включаем в наше расширение результат применения привязки переменной к голове правила. Процедура вычисления этих привязок зависит от типа тела правила.

1. Если тело правила является атомом, мы пытаемся сопоставить этот атом с фактоидами в нашем наборе данных. Для каждого соответствующего атому фактоида мы добавляем соответствующую подстановку к нашему набору ответов. И возвращаем набор всех подстановок, полученных таким образом.
2. Если запрос является отрицанием, мы выполняем нашу процедуру на аргументе отрицания и заданной подстановке. Если результатом является непустое множество (т. е. существуют подстановки, которые работают), то отрицание ложно, и мы возвращаем `false` в качестве ответа. Если результат рекурсивного выполнения является пустым множеством (т. е. нет ни одной подходящей замены), то отрицание в целом истинно, и в качестве результата мы возвращаем единственное множество, содержащее данную подстановку.
3. Если запрос является конъюнкцией, то мы выполняем нашу процедуру на первом конъюнкте и заданной подстановке. Затем перебираем список ответов, для каждой подстановки выполняя процедуру на оставшихся конъюнктах и заданной подстановке, и возвращаем результирующие подстановки.

Чтобы проиллюстрировать эту процедуру, рассмотрим набор данных, показанный ниже.

```
p(a,b)
p(a,c)
p(b,c)
p(c,d)
```

Теперь рассмотрим запрос `goal(Y) :- p(a,Y)`. Шаблон `p(a,Y)` соответствует двум фактоидам в нашем наборе данных, и поэтому есть два результата.

```
goal(b)
goal(c)
```

Предположим, что вместо этого у нас есть правило запроса `goal(Y) :- p(a,Y) & p(Y,d)`. И снова шаблон `p(a,Y)` соответствует только двум фактоидам в нашем наборе данных. Учитывая `{Y←b}`, шаблон `p(Y,d)` не соответствует ни одному фактоиду; для `{Y←c}` шаблон `p(Y,d)` соответствует только `p(c,d)`. Таким образом, существует только один ответ в этом случае.

```
goal(c)
```

Рассматривая конъюнктивный запрос `goal(Y) :- p(a,Y) & ~p(Y,d)`, мы снова найдем два ответа на первый конъюнкт, а именно `{Y←b}` и `{Y←c}`. Рассматривая первый из них, отрицание `~p(Y,d)` удовлетворяется, и поэтому конъюнкция истинна. Учитывая второй ответ на первый конъюнкт, отрицание не выполняется, поэтому в этом случае ответа нет. Как и в случае с предыдущим запросом, существует только один ответ.

```
goal(b)
```

Наконец, для запроса с более чем одним правилом мы получим набор ответов на отдельные правила.

5.5. Вычислительный анализ

Одной из приятных особенностей нашего алгоритма оценки запросов является простота вычислительного анализа. В этом разделе мы предполагаем стандартную реализацию оценки слева направо без *индексирования* наборов данных и без *кеширования* результатов после их вычисления. Рассмотрим запрос, показанный ниже.

```
goal(a,c) :- p(a,Y) & p(Y,c)
```

Каков объем вычислений по этому запросу? В худшем случае в базе данных имеется n^2 фактов, где n – количество основных элементов в на-

шем языке. Таким образом, нам нужно n^2 шагов, чтобы оценить первый конъюнкт. Существует не более n фактов, имеющих `a` в качестве первого аргумента. Для каждого из них мы рассматриваем n^2 возможностей для второго конъюнкта. Следовательно, объем вычислений экземпляра может быть выражен, как показано ниже.

$$n^2 + n \times n^2 = n^2 + n^3.$$

Теперь рассмотрим общую версию этого запроса, показанную ниже.

```
goal(X,Z) :- p(X,Y) & p(Y,Z)
```

Каков объем вычисления всех ответов на этот запрос? В худшем случае в базе данных имеется n^2 фактов, где n – количество объектов в домене. Таким образом, нам нужно n^2 шагов для оценки первой подцели. Существует не более n^2 возможных значений для `Y`. Для каждого из них мы рассматриваем n^2 возможностей для второй подцели. Результирующий объем вычислений показан ниже.

$$n^2 + n^2 \times n^2 = n^2 + n^4$$

Прежде чем завершить наш анализ сложности, поучительно сравнить объем вычислений с помощью этого алгоритма с объемом вычислений в соответствии с семантикой, описанной в предыдущей главе, т. е. путем перечисления всех экземпляров правил и проверки для каждого экземпляра, истинно ли его тело.

В предыдущем примере запрос содержит всего три переменные. Следовательно, для домена с n объектами существует n^3 экземпляров. Чтобы оценить каждый из них, мы должны сравнить каждую из двух подцелей с каждым фактоидом в нашем наборе данных, а их в худшем случае n^2.

Общий объем вычислений показан ниже.

$$n^3(n^2 + n^2) = n^5 + n^5 = 2n^5.$$

Наш алгоритм явно лучше в этом случае, и относительные преимущества больше, когда мы рассматриваем разреженные наборы данных, т. е. наборы данных, которые не включают все возможные фактоиды. В таких случаях «основанный на семантике» алгоритм должен по-прежнему рассматривать все экземпляры, а нашему алгоритму необходимо рассматривать только те экземпляры, которые были получены из фактоидов в наборе данных.

Обратите внимание, что при наличии индексирования набора данных или кеширования результатов детали этого анализа, вероятно, будут отличаться, но стиль анализа тот же самый, и относительные достоинства двух алгоритмов более или менее одинаковы.

5.6. Упражнения

5.1. Для каждого из следующих шаблонов и экземпляров скажите, соответствует ли экземпляр шаблону, и, если да, укажите соответствующий матчер:

(a) `p(X,Y)` и `p(a,a)`;

(b) `p(X,Y)` и `p(a,f(a))`;

(c) `p(X,f(Y))` и `p(a,f(a))`;

(d) `p(X,X)` и `p(2,min(2,4))`;

(e) `p(X,min(2,4))` и `p(2,2)`.

5.2. Предположим, что мы хотим найти все `goal(X,Y,Z)` такие, что `p(X,Y) & q(Y,Z)`. Выберите формулу, которая отражает наихудшую сложность нашего стандартного алгоритма оценки для этого запроса (при условии отсутствия индексации набора данных). Символ n здесь обозначает общее количество объектов в домене.

(a) $2n^2 + 2n^3$;

(b) $2n^2 + 2n^4$;

(c) $2n^2 + n^3 + n^4$.

5.3. Для каждого из приведенных ниже запросов выберите выражение, которое отражает наихудшую сложность нашего стандартного алгоритма оценки без индексации. Символ n обозначает общее количество объектов в домене.

```
goal(X,Y) :- p(X,Y) & q(Y) & q(Z)
```

(a) $n^4 + n^3 + n^2 + n$;

(b) $2n^4 + 2n^3 + n^2 + n$;

(c) $2n^4 + 2n^3$.

```
goal(X,Y) :- p(X,Y) & q(Y)
```

(a) $n^4 + n^3 + n^2 + n$;

(b) $2n^4 + 2n^3 + n^2 + n$;

(c) $2n^4 + 2n^3$.

Глава 6

Оптимизация просмотра

6.1. Введение

Два запроса *семантически эквивалентны* тогда и только тогда, когда они дают идентичные результаты для каждого набора данных. В таких случаях не имеет значения, какой запрос написан, если целью является получение правильных ответов. С другой стороны, возможно, что один запрос *вычислительно лучше* другого в том смысле, что наш алгоритм оценки быстрее получает такие ответы.

В этой главе рассматриваются различные методы оптимизации запросов, т. е. преобразования запросов в семантически эквивалентные запросы, которые являются менее сложными с вычислительной точки зрения. Мы начнем с обсуждения упорядочивания подцелей в правилах запроса, затем рассмотрим устранение необходимости в упорядочивании подцелей в правилах запроса и удаление бесполезных подцелей. И в заключение мы обсудим удаление правил из запросов с несколькими правилами.

6.2. Упорядочивание подцелей

Одним из очень распространенных источников неэффективности при оценке является неоптимальное упорядочивание подцелей в запросах. Положительно то, что часто можно найти лучшее упорядочивание, просто взглянув на форму запросов, даже не рассматривая данные, к которым эти запросы применяются.

В качестве примера неэффективности из-за плохого упорядочивания подцелей рассмотрим запрос, показанный ниже. Наша цель истинна для `X` и `Y`, если `p` истинно для `X`, `r` истинно для `X` и `Y` и `q` истинно для `X`.

```
goal(X,Y) :- p(X) & r(X,Y) & q(X)
```

Интуитивно кажется, что это плохой способ написать запрос для нашей обычной процедуры оценки. Кажется, что условие `q` должно стоять перед условием `r`, как в следующем запросе.

```
goal(X,Y) :- p(X) & q(X) & r(X,Y)
```

Действительно для этого есть веские основания. Для нашей стандартной процедуры оценки объем вычислений для первого запроса в худшем случае составляет n^4, где n – размер домена объектов. В отличие от этого наихудшая оценка для второго запроса составляет всего n^3.

Давайте рассмотрим эти два случая более подробно. В наихудшем случае в базе данных имеется $n^2 + 2n$ фактов, где n – количество объектов в домене.

При оценке первого запроса наш алгоритм сначала исследует все $n^2 + 2n$ фактов, чтобы найти те из них, которые соответствуют `p(X)`. Ответов будет не более n. Для каждого из них алгоритм снова просмотрит $n^2 + 2n$ фактов, чтобы найти соответствующие `r(X,Y)`. Их будет n для каждого значения `X`. Наконец, для каждой из этих пар алгоритм снова рассмотрит $n^2 + 2n$ фактов, чтобы найти соответствующие `q(X)`. Общий итог показан ниже.

$$(n^2 + 2n) + n * ((n^2 + 2n) + n * (n^2 + 2n)) = n^4 + 3n^3 + 3n^2 + 2n.$$

При оценке второго запроса алгоритм снова исследует все $n^2 + 2n$ фактов, чтобы найти соответствующие `p(X)`. Ответов будет не более n. Для каждого из них алгоритм снова просмотрит $n^2 + 2n$ фактов, чтобы найти соответствующие `q(X)`. Найдется по крайней мере один для каждого значения `X`. Наконец, для каждого `X` алгоритм рассмотрит $n^2 + 2n$ фактов, чтобы найти соответствующие `r(X,Y)`. Общий итог показан ниже.

$$(n^2 + 2n) + n * ((n^2 + 2n) + 1 * (n^2 + 2n)) = 2n^3 + 5n^2 + 2n.$$

Предположим, например, что в домене было 4 объекта. В этом случае будет максимум 4 `p` фактов, 4 `q` фактов и 16 `r` фактов. Оценка первого запроса потребует 504 совпадения в то время, как для оценки второго запроса их потребуется только 208.

При наличии индексации асимптотическая сложность одинакова для обоих упорядочений. Однако меньшее количество степенных членов для второго упорядочения говорит о том, что оно все же лучше. Более того, можно показать, что, усредняя по всем возможным базам данных, второй запрос лучше, чем первый.

К счастью, существует простой метод переупорядочивания подцелей в подобных ситуациях. Основная идея его заключается в том, чтобы собрать новое тело запроса постепенно, выбирая подцель на каждом шаге и удаляя ее из списка оставшихся подцелей, подлежащих рассмотрению. Делая выбор, метод рассматривает оставшиеся подцели слева направо. Если он встречает подцель, все переменные которой связаны с уже выбранными подцелями, то эта подцель добавляется в новый запрос и удаляется из списка оставшихся подцелей. Если нет, то метод удаляет

из списка первую оставшуюся подцель, добавляет ее в новый запрос, обновляет список связанных переменных и переходит к следующему шагу.

В качестве примера действия этого метода рассмотрим первый запрос, показанный выше. Сначала список оставшихся подцелей состоит из всех трех подцелей запроса. В этот момент ни одна из трех подцелей не является основой, поэтому метод выбирает первую подцель `p(X)`, добавляет ее к новому запросу и помещает `X` в список связанных переменных. На втором шаге метод рассматривает оставшиеся две подцели. Подцель `r(X,Y)` содержит несвязанную переменную `Y`, поэтому она не выбирается. Напротив, все переменные в подцели `q(X)` являются связанными, поэтому метод выводит эту подцель следующей. На третьем шаге добавляется последняя подцель, формируя второй запрос, показанный выше.

6.3. Удаление подцелей

Другой распространенный источник неэффективности при оценке связан с наличием избыточных подцелей в запросах. Во многих случаях можно обнаружить и устранить такую избыточность. В качестве простого примера рассмотрим запрос, показанный ниже. Его цель истинна для `X` и `Y`, если `p` истинно для `X` и `Y`, `q` истинно для `Y` и `q` также истинно для некоторого `Z`.

```
goal(X,Y) :- p(X,Y) & q(Y) & q(Z)
```

Очевидно, что подцель `q(Z)` здесь лишняя. Если существует значение `Y` такое, что `q(Y)` истинно, то это значение для `Y` также работает как значение для `Z`. Следовательно, можно отказаться от подцели `q(Z)` (при сохранении `q(Y)`), в результате чего получается запрос, показанный ниже.

```
goal(X,Y) :- p(X,Y) & q(Y)
```

Обратите внимание, что обратное неверно. Если отказаться от `q(Y)` и сохранить `q(Z)`, то мы потеряем ограничение на второй аргумент `p`, который также является аргументом `q` .

К счастью, легко определить, какие подцели следует удалить, а какие необходимо сохранить. Идея заключается в том, чтобы собирать новое тело запроса постепенно, выбирая подцель на каждом шаге, проверяя ее избыточность и добавляя подцель один раз, если она не избыточна.

В качестве иллюстрации действия этого метода рассмотрим пример, показанный выше. Метод сначала вычисляет переменные в голове запроса, т. е. `[X,Y]`, и инициализирует переменную `newquery` в новом запросе с той же головой, что и исходный запрос. Затем выполняется

итерация по телу запроса, добавляя подцели в новый запрос после их проверки на избыточность.

На первом шаге итерации метод фокусируется на `p(X,Y)`. Он создает набор данных, состоящий из экземпляров оставшихся подцелей, а именно `q(x1)` и `q(x2)`, затем пытается вывести `p(x0,x1)`, и в этом случае он терпит неудачу. Поэтому `p(X,Y)` добавляется к новому запросу.

На втором шаге метод фокусируется на `q(Y)`. Он создает набор данных, состоящий из экземпляров оставшихся подцелей, а именно `p(x0,x1)` и `q(x2)`; затем он пытается вывести `q(x1)`, и снова неудача. Поэтому `q(Y)` добавляется к новому запросу.

Наконец, метод фокусируется на `q(Z)`. Он создает набор данных, состоящий из экземпляров других подцелей, а именно `p(x0,x1)` и `q(x1)`, затем пытается вывести экземпляр `q(Z)`. Обратите внимание, что `Z` здесь не связан, поскольку он не встречается в качестве переменной в голове или в любой из других подцелей. В этом случае проверка проходит успешно, и поэтому `q(Z)` *не* добавляется в новый запрос.

К сожалению, данный метод не является абсолютным. Существуют избыточные подцели, которые он не обнаруживает. Проблема возникает, когда несколько избыточных подцелей имеют общие переменные, не позволяющие методу обнаружить избыточность.

В качестве примера рассмотрим запрос, показанный ниже. Его цель истинна для `X`, если `p` истинно для `X` и `Y`, `q` истинно для `X` и `Y`, `p` истинно для `X` и `Z`, `q` истинно для `X` и `Z`.

```
goal(X) :- p(X,Y) & q(X,Y) & p(X,Z) & q(X,Z)
```

Очевидно, что последние две подцели избыточны по отношению к первым двум подцелям. К сожалению, наш метод не обнаруживает, что одна из подцелей в любой паре является избыточной из-за наличия переменных, общих с другой подцелью этой пары. Проверьте.

Обнаружение такого рода избыточности можно сделать механически, рассматривая подмножества подцелей, а не только отдельные подцели. Однако это более дорогостоящий способ, чем простой метод, описанный выше.

6.4. Удаление правил

Аналогичная форма неэффективности при оценке возникает из-за наличия избыточных правил. Как и с избыточными подцелями внутри правил, часто легко обнаружить и устранить такие избыточности.

В качестве примера рассмотрим правила, показанные ниже. Для первого из них – цель истинна для `X`, если p истинно для `X` и `Y`, `q` истинно для `b` и `r` истинно для `Z`. Для второго – цель истинна для `X`, если p истинно для `X` и `Y` и `q` истинно для `Y`.

```
goal(X) :- p(X,b) & q(b) & r(Z)
goal(X) :- p(X,Y) & q(Y)
```

Здесь любой ответ, полученный по первому правилу, также получается по второму правилу, поэтому первое правило является избыточным и может быть исключено.

Тонкость обнаружения таких избытков заключается в том, чтобы понять, что второе правило *включает* в себя первое, т. е. все ответы, произведенные вторым правилом, производятся первым правилом. Если заменить некоторые или все переменные в теле второго правила, которые не встречаются в голове, то головы останутся прежними, а все подцели второго правила станут членами тела первого правила. В нашем случае достаточно заменить `Y` на `b`, и мы получим правило `goal(X) :- p(X,b) & q(b)`, которое аналогично первому правилу, только с меньшим количеством подцелей. Каждый вывод первого правила является, таким образом, выводом второго правила, и, следовательно, первое правило можно отбросить.

6.5. Пример – криптарифметика

Криптарифметическая проблема – это проблема удовлетворения ограничений, характеризующаяся конечным набором букв, конечным набором чисел и арифметическим ограничением, записанным в терминах букв. А решением проблемы является нахождение соответствия букв числам таким образом, что при замене букв соответствующими числами арифметическое ограничение выполняется.

Классическая криптарифметика показана ниже. Здесь буквы – это `{S, E, N, D, M, O, R, Y}`, а числа – это цифры от 0 до 9, и мы ищем соответствие букв цифрам такое, чтобы уравнение выполнялось.

```
 SEND
+MORE
MONEY
```

Мы можем сформулировать эту задачу в виде запроса примерно так же, как формализовали задачу раскраски карты, представленную в главе 3. Во-первых, у нас есть набор данных, в котором перечислены допустимые цифры.

```
digit(0)
digit(1)
digit(2)
digit(3)
digit(4)
digit(5)
```

```
digit(6)
digit(7)
digit(8)
digit(9)
```

Далее мы пишем запрос, перечисляющий восемь букв в качестве целевых значений, с подцелями, фиксирующими диапазоны этих переменных, ограничениями на нерасчлененность и дополнительными ограничениями для отражения арифметических условий. В интересах краткости здесь использованы обычные арифметические операторы вместо соответствующих встроенных, например мы использовали `E!=S` вместо использованного ранее `distinct(E,S)` и `1000*S` – вместо `times(1000,S)`.

```
goal(S,E,N,D,M,O,R,Y) :-
digit(S) & digit(E) & digit(N) & digit(D) &
digit(M) & digit(O) & digit(R) & digit(Y) &
S!=0 & E!=S & N!=S & N!=E & D!=S & D!=E & D!=N &
M!=0 & M!=S & M!=E & M!=N & M!=D &
O!=S & O!=E & O!=N & O!=D & O!=M &
R!=S & R!=E & R!=N & R!=D & R!=M & R!=O &
Y!=S & Y!=E & Y!=N & Y!=D & Y!=M & Y!=O & Y!=R &
evaluate(1000*S+100*E+10*N+D),Send) &
evaluate(1000*M+100*O+10*R+E),More) &
evaluate(10000*M+1000*O+100*N+10*E+Y),Money) &
evaluate(plus(Send,More,Money)/td>
```

Выразив таким образом цель, мы можем использовать процедуру оценки для генерирования ответа на эту проблему. Однако это не без труда. Учитывая то, как составлен этот запрос, процесс оценки займет много времени. Существует 10^8 = 100 000 000 возможных сочетаний переменных. В худшем случае (когда нет решения) придется проверить все из них. В среднем придется просмотреть значительную часть.

Хорошей новостью является то, что можно использовать упорядочивание подцелей для преобразования этого запроса в такой, который легче оценить. Мы просто перемещаемся до точки, где переменные связаны.

```
goal(S,E,N,D,M,O,R,Y) :-
digit(S) & S!=0 &
digit(E) & E!=S &
digit(N) & N!=S & N!=E &
digit(D) & D!=S & D!=E & D!=N &
digit(M) & M!=0 & M!=S & M!=E & M!=N & M!=D &
digit(O) & O!=S & O!=E & O!=N & O!=D & O!=M &
```

```
digit(R) & R!=S & R!=E & R!=N & R!=D & R!=M & R!=O &
digit(Y) & Y!=S & Y!=E & Y!=N & Y!=D & Y!=M & Y!=O & Y!=R &
evaluate(1000*S+100*E+10*N+D),Send) &
evaluate(1000*M+100*O+10*R+E),More) &
evaluate(10000*M+1000*O+100*N+10*E+Y),Money) &
evaluate(plus(Send,More,Money)/td>
```

Сделав это, мы устраним многие варианты еще до того, как они будут сгенерированы. В итоге для оценки такого запроса потребуется на порядки меньше времени, на большинстве компьютеров оно не превысит долей секунды.

6.6. Упражнения

6.1. Для каждой из следующих групп правил запросов скажите, какое правило является лучшим с точки зрения наихудшей сложности оценки с использованием нашего стандартного алгоритма без индексации.

(a)
```
goal(X,Y,Z) :- p(X,Y) & q(X,X) & r(X,Y,Z)
goal(X,Y,Z) :- p(X,Y) & r(X,Y,Z) & q(X,X)
goal(X,Y,Z) :- q(X,X) & p(X,Y) & r(X,Y,Z)
goal(X,Y,Z) :- q(X,X) & r(X,Y,Z) & p(X,Y)
goal(X,Y,Z) :- r(X,Y,Z) & p(X,Y) & q(X,X)
goal(X,Y,Z) :- r(X,Y,Z) & q(X,X) & p(X,Y)
```

(b)
```
goal(X,Y,Z) :- p(X,Y) & q(a,b) & r(X,Y,Z)
goal(X,Y,Z) :- p(X,Y) & r(X,Y,Z) & q(a,b)
goal(X,Y,Z) :- q(a,b) & p(X,Y) & r(X,Y,Z)
goal(X,Y,Z) :- q(a,b) & r(X,Y,Z) & p(X,Y)
goal(X,Y,Z) :- r(X,Y,Z) & p(X,Y) & q(a,b)
goal(X,Y,Z) :- r(X,Y,Z) & q(a,b) & p(X,Y)
```

(c)
```
goal(X,Y,Z) :- p(X,Y,Z) & q(X) & ~r(X,Y)
goal(X,Y,Z) :- p(X,Y,Z) & ~r(X,Y) & q(X)
goal(X,Y,Z) :- q(X) & p(X,Y,Z) & ~r(X,Y)
goal(X,Y,Z) :- q(X) & ~r(X,Y) & p(X,Y,Z)
goal(X,Y,Z) :- ~r(X,Y) & q(X) & p(X,Y,Z)
goal(X,Y,Z) :- ~r(X,Y) & p(X,Y,Z) & q(X)
```

6.2. Для каждой из следующих групп правил запроса выберите альтернативу, которая эквивалентна первому правилу в группе.

(a)
```
goal(X) :- p(X,Y) & q(Y) & q(Z)
goal(X) :- p(X,Y)
goal(X) :- p(X,Y) & q(Y)
goal(X) :- p(X,Y) & q(Z)
```

(b)
```
goal(X) :- p(X) & q(X) & q(W)
goal(X) :- p(X)
goal(X) :- p(X) & q(X)
goal(X) :- p(X) & q(W)
```

(c)
```
goal(X,Y,Z) :- p(X,Y) & q(Y) & q(Z) & q(W)
goal(X,Y,Z) :- p(X,Y) & q(Y) & q(Z)
goal(X,Y,Z) :- p(X,Y) & q(Y) & q(W)
goal(X,Y,Z) :- p(X,Y) & q(Z) & q(W)
```

6.3. Для каждой из следующих пар запросов скажите, включает ли первый запрос второй, т. е. содержит ли набор ответов на первый запрос ответы на второй.

(a)
```
goal(X) :- p(X,Y)
goal(X) :- p(X,a) & p(X,b)
```

(b)
```
goal(X) :- p(X,a)
goal(X) :- p(X,Y)
```

(c)
```
goal(X) :- p(X,Y) & p(X,Z)
goal(X) :- p(X,b) & q(b)
```

Часть III
Определения представлений

Глава 7

Определения представлений

7.1. Введение

Рассмотрим набор данных о родстве, показанный ниже. Ситуация здесь аналогична описанной в главе 2. Арт является родителем Боба и Би. Боб – родитель Кэла и Кэма. Би – родитель Коу и Кори.

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

Предположим теперь, что мы хотим выразить информацию об отношении бабушки и дедушки, а также и о родительском отношении. Как было показано ранее, это можно сделать, добавив факты к нашему набору данных. В данном случае добавим факты, показанные ниже. Арт является дедушкой Кэла, Кэма, Коу и Кори.

```
grandparent(art,cal)
grandparent(art,cam)
grandparent(art,coe)
grandparent(art,cory)
```

К сожалению, такой способ является расточительным. Отношение бабушки и дедушки может быть определено в терминах родительского отношения, и поэтому хранение данных о дедушке и бабушке, а также данных о родителе является излишним.

Лучшей альтернативой является написание правил для кодирования таких определений и использование этих правил для вычисления определенных ими отношений, когда это необходимо. Как показано в этой главе, можно написать такие определения, используя правила, аналогичные тем, которые использованы для определения отношений целей

в главе 3. Например, в приведенном выше случае вместо того, чтобы добавлять в набор данных факты о бабушках и дедушках, мы можем написать следующее правило, будучи уверенными в том, что можем использовать это правило для вычисления данных о бабушках и дедушках.

```
grandparent(X,Z) :- parent(X,Y) & parent(Y,Z)
```

В дальнейшем будем различать два разных типа отношений – *базовые отношения* и *отношения вида*. Мы определяем базовые отношения, записывая факты в наборе данных, и определяем отношения вида, записывая правила в *наборе правил*. В нашем примере parent – это базовое отношение, а grandparent – это отношение вида.

Имея набор данных, определяющий базовые отношения, и набор правил, определяющий отношения вида, можно использовать инструменты автоматического рассуждения для получения фактов об отношениях вида. Например, учитывая предыдущие факты об отношении parent и правило, определяющее отношение grandparent, можно вычислить факты об отношении grandparent.

Использование правил для определения отношений вида имеет множество преимуществ по сравнению с кодированием этих отношений в виде наборов данных. Прежде всего, как мы только что видели, это экономия: если отношения вида определены в терминах правил, не нужно хранить так много фактов в наборах данных. Во-вторых, уменьшается вероятность рассинхронизации, например, если мы изменим отношение parent и забудем изменить отношение grandparent. В-третьих, определения представлений работают для любого количества объектов, даже для приложений с бесконечно большим количеством объектов (например, целые числа), не требуя при этом бесконечного хранения.

В этой главе мы представляем синтаксис и семантику определений вида, а также описываем важное понятие стратификации. В последующих главах рассмотрено множество примеров использования правил для определения представлений, а в главах 11 и 12 – некоторые практические приемы использования правил для вычисления отношений вида.

7.2. Синтаксис

Синтаксис определений представлений практически идентичен синтаксису запросов, описанному в главе 3. Различные типы констант одинаковы, а понятия элемента, атома и литерала также одинаковы. Основное различие заключается в синтаксисе правил.

Как и раньше, *правило* – это выражение, состоящее из выделенного атома, называемого *головой*, и конъюнкции из нуля или более литера-

лов, называемой *телом*. Литералы в теле называются *подцелями*. Правила пишутся, как в примере, показанном ниже. Здесь `r(X,Y)` – это голова, `p(X,Y) & ~q(Y)` – тело; а `p(X,Y)` и `~q(Y)` – подцели.

```
r(X,Y) :- p(X,Y) & ~q(Y)
```

Несмотря на это сходство, есть два важных различия между правилами запросов и правилами, используемыми в определениях представлений.

Первое – при написании правил запросов используется один общий предикат (например, `goal`) в головах всех правил запросов. В отличие от этого в определениях представлений используются предикаты отношений, которые мы определяем (например, `r` в приведенном выше примере).

Второе – при написании правил запросов подцели правил могут содержать только предикаты отношений, описанных в наборе данных. В отличие от этого в определениях представлений подцели могут содержаться предикаты представлений, а также предикаты базовых отношений.

Одним из преимуществ более гибкого синтаксиса в определениях представлений является возможность определить несколько отношений в одном наборе правил. Например, следующие правила определяют и отношение `f`, и отношение `m` в терминах `p` и `q`.

```
f(X,Y) :- p(X,Y) & q(X)
m(X,Y) :- p(X,Y) & ~q(X)
```

Второе преимущество заключается в том, что можно использовать отношения вида при определении других отношений вида. Например, в следующем правиле используется отношение вида `f` в определении `g`.

```
g(X,Z) :- f(X,Y) & p(Y,Z)
```

Третье преимущество состоит в том, что представления могут быть использованы в своих собственных определениях, что позволяет определять отношения рекурсивно. Например, следующие правила определяют `a` как транзитивное замыкание `p`.

```
a(X,Z) :- p(X,Z)
a(X,Z) :- p(X,Y) & a(Y,Z)
```

К сожалению, наш язык позволяет создавать наборы правил с некоторыми неприятными свойствами. Чтобы избежать этих проблем, хорошей практикой является соблюдение некоторых синтаксических ограничений на наборы данных и наборы правил, а именно: совместимость, расслоение и безопасность.

Набор правил с*овместим* с набором данных тогда и только тогда, когда:

- все символы, общие для набора данных и набора правил, имеют один и тот же тип (символ, конструктор, предикат);
- все конструкторы и предикаты имеют одинаковую арность;
- ни один из предикатов набора данных не содержится в голове каких-либо правил в наборе правил.

7.3. Семантика

Семантика определений представлений сложнее, чем семантика запросов, из-за возможного появления предикатов представления в подцелях, поэтому мы используем несколько иной подход.

Чтобы определить результат применения набора определений представлений к набору данных, сначала объединяем факты в наборе данных с правилами, определяющими наши представления, в совместный набор фактов и правил, далее называемый *замкнутой логической программой*, а затем определяем расширение этой замкнутой логической программы следующим образом.

Вселенная Гербранда для замкнутой логической программы –это множество всех основных элементов, которые могут быть образованы из символов и конструкторов программы. Для программы без конструкторов вселенная Гербранда конечна (т. е. содержит только символы). Для программы с конструкторами вселенная Гербранда бесконечна (т. е. содержит символы и все составные элементы, которые могут быть образованы из этих символов).

Базой Гербранда для замкнутой логической программы является множество всех атомов, которые могут быть образованы из констант программы. Иначе говоря, это множество всех фактов вида $r(t_1,\ldots, t_n)$, где r – n-арный предикат, а $t_1,\ldots, t_n$ – основные элементы.

Интерпретация замкнутой логической программы – это произвольное подмножество базы Гербранда для этой программы. Как и в случае с наборами данных, идея здесь заключается в том, что фактоиды в интерпретации считаются истинными, а не включенные в нее – ложными.

Модель замкнутой логической программы – это интерпретация, *удовлетворяющая* программе. Мы определяем удовлетворение в два этапа – сначала рассмотрим случай основных правил, а затем – произвольных правил.

Интерпретация Г удовлетворяет основному атому φ тогда и только тогда, когда φ находится в Г. Г удовлетворяет основному отрицанию $\sim\varphi$ тогда и только тогда, когда φ *не находится* в Г. Г удовлетворяет основно-

му правилу φ :- φ_1 & … & φ_n тогда и только тогда, когда Г удовлетворяет φ всякий раз, когда оно удовлетворяет φ_1, …, φ_n.

Экземпляром правила в замкнутой логической программе является правило, в котором все переменные были последовательно заменены элементами из вселенной Гербранда, т. е. множеством базовых элементов, которые могут быть сформированы из словаря программы. Как и раньше, *последовательная замена* означает, что если вхождение переменной в предложении заменяется данным элементом, то все вхождения этой переменной в данном предложении заменяются тем же элементом.

Используя понятие экземпляра, можно определить понятие удовлетворения для произвольных замкнутых логических программ (с переменными или без них). Интерпретация Г *удовлетворяет* произвольной замкнутой логической программе Ω тогда и только тогда, когда Г удовлетворяет каждому основному экземпляру каждого предложения в Ω.

В качестве примера этих концепций рассмотрим набор данных, показанный ниже.

```
p(a,b)
p(b,c)
p(c,d)
p(d,c)
```

И предположим, что у нас есть следующее определение вида

```
r(X,Y) :- p(X,Y) & ~p(Y,X)
```

Следующая интерпретация удовлетворяет замкнутой логической программе, состоящей из данного набора данных и набора правил. Все факты из набора данных включены в интерпретацию, и каждое требуемое нашим правилом заключение также включено.

```
p(a,b)
p(b,c)
p(c,d)
p(d,c)
r(a,b)
r(b,c)
```

Напротив, следующие интерпретации **не** удовлетворяют программе. В той, что слева, отсутствуют выводы из правила, в средней отсутствуют факты из набора данных, а правая удовлетворяет правилам, но не содержит всех фактов из набора данных.

```
p(a,b)      r(b,c)      p(a,b)
p(b,c)      r(c,d)      p(b,c)
```

```
p(c,d)                  p(c,d)
p(d,c)                  r(a,b)
                        r(b,c)
                        r(c,d)
```

С другой стороны, модель, показанная выше (интерпретация, предшествующая этим трем не-моделям), не является единственной интерпретацией, которая работает. В целом закрытая логическая программа может иметь более одной модели, что означает, что может существовать более одного способа удовлетворить правилам программы. Следующие интерпретации также удовлетворяют нашей замкнутой логической программе.

```
p(a,b)      p(a,b)      p(a,b)
p(b,c)      p(b,c)      p(b,c)
p(c,d)      p(c,d)      p(c,d)
p(d,c)      p(d,c)      p(d,c)
r(a,b)      r(a,b)      r(a,b)
r(b,c)      r(b,c)      r(b,c)
r(c,d)      r(d,c)      r(c,d)
                        r(d,c)
```

Это кажется странным, поскольку нет причин включать `r(c,d)` или `r(d,c)` в нашу интерпретацию. С другой стороны, учитывая наше определение удовлетворенности, нет причин *не* включать их.

Причина, по которой это кажется неправильным, заключается в том, что мы обычно хотим, чтобы наши определения были «*если и только если*». Мы хотим включить в наши выводы только те факты, которые должны быть истинными. При этом:

- все фактоиды в нашем наборе данных должны быть истинными;
- все фактоиды, требуемые нашими правилами, должны быть истинными;
- все другие фактоиды должны быть исключены.

Это классическое определение того, что известно как *логическое следствие*. Фактоид *логически вытекает* из замкнутой логической программы тогда и только тогда, когда он истинен в каждой модели программы, т. е. множество выводов является пересечением всех моделей программы.

Один из способов обеспечить логическое следствие – взять пересечение всех интерпретаций, которые удовлетворяют программе. Это гарантирует, что мы получим только те заключения, которые истинны в каждой модели. Например, если взять пересечение трех моделей, показанных выше, получим нашу исходную модель.

Другой подход заключается в том, чтобы сосредоточиться на *минимальных моделях*. Модель Γ логической программы Ω *минимальна* тогда и только тогда, когда не существует подмножества Γ, которое является моделью для Ω. Если существует только одна минимальная модель замкнутой логической программы, то минимальность гарантирует логическое следствие. Например, первая модель, приведенная выше, минимальна, и каждый фактоид в этой модели должен присутствовать в каждой модели программы.

Многие замкнутые логические программы имеют единственную минимальную модель. Например, замкнутая логическая программа, не содержащая отрицаний, имеет одну и только одну минимальную модель. К сожалению, замкнутые логические программы с отрицанием могут иметь более одной минимальной модели.

Один из способов устранения подобных неясностей состоит в том, чтобы сосредоточиться на программах, которые являются полупозитивными или программами, которые стратифицированы по отношению к отрицанию. Мы определим эти типы программ и обсудим их семантику в следующих двух разделах.

7.4. Полупозитивные программы

Полупозитивная программа – это программа, в которой отрицания применяются только к базовым отношениям, т. е. нет подцелей с отрицаемыми представлениями.

Семантика полупозитивной программы может быть формализована путем определения результата применения определения вида к фактам в наборе данных программы. Мы используем слово *расширение* для обозначения множества всех фактов, которые могут быть «выведены» таким образом.

Экземпляр выражения (атома, литерала или правила) – это выражение, в котором все переменные были последовательно заменены основными элементами. Например, если у нас есть язык с константами `a` и `b`, то `r(a) :- p(a,a)`, `r(a) :- p(a,b)`, `r(b) :- p(b,a)` и `r(b) :- p(b,b)` – это все экземпляры `r(X) :- p(X,Y)`.

Используя это понятие, мы определяем результат применения одного правила к набору данных следующим образом. Пусть имеем правило r и набор данных Δ, тогда $\upsilon(r, \Delta)$ – это множество всех таких ψ, что:

- ψ является головой произвольного экземпляра r;
- каждая позитивная подцель в этом экземпляре является членом Δ;
- ни одна негативная подцель в экземпляре не является членом Δ.

Мы определяем результат многократного применения *единого слоя* правил Ω к набору данных Δ следующим образом. Рассмотрим последовательность наборов данных, определенную рекурсивно следующим образом. $\Gamma_0 = \Delta$ и $\Gamma_{n+1} = \cap\upsilon(r, \Gamma_0 \cap \ldots \cap \Gamma_n)$ для всех r в Ω. Замыкание Ω на Δ является объединением наборов данных в этой последовательности, т. е. $C(\Omega, \Delta) = \cap\Gamma_i$.

Чтобы проиллюстрировать это определение, начнем с набора данных, описывающего небольшой направленный граф. В приведенных ниже предложениях использован предикат `edge` для записи дуг конкретного графа.

```
edge(a,b)
edge(b,c)
edge(c,d)
edge(d,c)
```

Теперь напишем несколько правил, определяющих различные отношения между узлами графа. Здесь отношение p истинно для узлов с исходящей дугой. Отношение `q` истинно для двух узлов тогда и только тогда, когда существует ребро из первого во второй *или* ребро из второго в первый. Отношение `r` истинно для двух узлов тогда и только тогда, когда существует ребро из первого во второй *и* ребро из второго в первый. Отношение `s` является транзитивным замыканием отношения `edge`.

```
p(X) :- edge(X,Y)
q(X,Y) :- edge(X,Y)
q(X,Y) :- edge(Y,X)
r(X,Y,Z) :- edge(X,Y) & edge(Y,Z)
s(X,Y) :- edge(X,Y)
s(X,Z) :- edge(X,Y) & s(Y,Z)
```

Мы начинаем вычисления с инициализации нашего набора данных перечисленными выше фактами.

```
edge(a,b)
edge(b,c)
edge(c,d)
edge(d,c)
```

Рассматривая правило `p` и сопоставляя его подцели с данными в нашем наборе данных всеми возможными способами, мы видим, что можно добавить следующие факты. В нашем случае каждый узел графа имеет исходящее ребро, поэтому для каждого узла существует один факт `p`.

```
p(a)
p(b)
p(c)
p(d)
```

Рассматривая правила `q` и сопоставляя их подцели с данными в нашем наборе данных всеми возможными способами, можем добавить следующие факты. В этом случае получаем симметричное замыкание исходного графа.

```
q(a,b)
q(b,a)
q(b,c)
q(c,b)
q(c,d)
q(d,c)
```

Аналогично для правила `r` можно добавить следующие факты.

```
r(c,d)
r(d,c)
```

Наконец, рассмотрев первое правило для `s`, можем добавить следующие факты.

```
s(a,b)
s(b,c)
s(c,d)
s(d,c)
```

Однако мы еще не закончили. С только что добавленными фактами возможно использовать второе правило, чтобы получить следующие дополнительные данные.

```
s(a,c)
s(b,d)
s(c,c)
s(d,d)
```

Сделав это, можем снова использовать правило `s` и вывести следующий факт.

```
s(a,d)
```

Теперь ни одно из правил при применении к этому набору данных не дает никаких результатов, которые уже не находятся в этом наборе, поэтому процесс завершается. Полученная коллекция из 25 фактов является расширением этой программы.

7.5. Стратифицированные программы

Мы говорим, что набор определений вида *стратифицирован* тогда и только тогда, когда его правила могут быть разбиты на *слои* таким образом, что:

- каждый слой содержит хотя бы одно правило;
- правила, определяющие отношения, которые появляются в позитивных подцелях правила, находятся в том же слое, что и это правило, *или* в более низком слое;
- правила, определяющие отношения, которые появляются в негативных подцелях правила, появляются в *более низком* слое (не в том же самом).

В качестве примера предположим, что у нас есть унарное отношение `p`, которое истинно для всех объектов в некоторой области применения, и произвольное бинарное отношение `q`. Теперь рассмотрим набор правил, показанный ниже. Первые два правила определяют `r` как транзитивное замыкание `q`. Третье правило определяет `s` как дополнение транзитивного замыкания.

```
r(X,Y) :- q(X,Y)
r(X,Z) :- q(X,Y) & r(Y,Z)
s(X,Y) :- p(X) & p(Y) & ~r(X,Y)
```

Это сложный набор правил, но легко увидеть, что он стратифицирован. Первые два правила вообще не содержат отрицаний, и поэтому можно сгруппировать их вместе в самом нижнем слое. Третье правило имеет негативную подцель, содержащую отношение, определенное в самом нижнем слое, и поэтому мы помещаем его в более высокий слой, как показано ниже. Этот набор правил удовлетворяет нашему определению и, следовательно, он стратифицирован.

Таблица 7.1. Пример стратифицированного набора правил

Слой	Правила
2	`s(X,Y) :- p(X) & p(Y) & ~r(X,Y)`
1	`r(X,Y) :- q(X,Y)` `r(X,Z) :- q(X,Y) & r(Y,Z)`

Для сравнения рассмотрим следующий набор правил. Здесь отношение `r` определяется в терминах `p` и `q`, а отношение `s` определяется в терминах `r` и отрицания `s`.

```
r(X,Y) :- p(X) & p(Y) & q(X,Y)
s(X,Y) :- r(X,Y) & ~s(Y,X)
```

Не существует способа разделить правила этого набора на слои таким образом, чтобы удовлетворить приведенное выше определение. Следовательно, этот набор правил *не* является стратифицированным.

Проблема с нестратифицированными наборами правил заключается в том, что существует потенциальная неоднозначность. В качестве примера рассмотрим правила, приведенные выше, и предположим, что наш набор данных также включает факты `p(a)`, `p(b)`, `q(a,b)` и `q(b,a)`. Из этих фактов можно заключить, что `r(a,b)` и `r(b,a)` оба истинны. Пока что хорошо. Но что мы можем сказать об `s`? Если принять `s(a,b)` за истинное, а `s(b,a)` – за ложное, то второе правило выполнено. Если принять `s(a,b)` за ложное, а `s(b,a)` – за истинное, то второе правило снова будет выполнено. В итоге получается, что существует неоднозначность в отношении `s`. Сосредоточившись исключительно на логических программах, которые являются стратифицированными, мы избегаем таких двусмысленностей.

Хотя иногда можно стратифицировать правила более чем одним способом, это не вызывает никаких проблем. Пока программа стратифицирована относительно отрицания, определение дает одно и то же расширение независимо от того, какая стратификация используется.

Наконец, мы определяем *расширение* набора правил Ω на набор данных Δ следующим образом. Это определение опирается на декомпозицию Ω на слои $\Omega_1,\ldots, \Omega_k$. Поскольку в замкнутой логической программе существует только конечное число правил и каждый слой должен содержать хотя бы одно правило, то существует только конечное число наборов (хотя сами наборы могут быть бесконечными). Учитывая это, пусть $\Delta_0 = \Delta$ и $\Delta_{n+1} = \Delta_n \cap C(\Omega_{n+1}, \Delta_n)$. Расширение программы с k слоями равно просто Δ_k.

Расширение любой замкнутой логической программы без конструкторов должно быть конечным. Расширение любой нерекурсивной замкнутой логической программы также должно быть конечным. В обоих случаях можно вычислить расширение за конечное время. Можно показать, что объем вычислений полиномиально зависит от размера набора данных.

В случае рекурсивных программ без конструкторов результат все равно должен быть конечным. Однако объем вычислений расширения может иметь экспоненциальную зависимость от объема данных, но результат может быть вычислен за конечное время.

Для рекурсивных программ с конструкторами возможно, что расширение бесконечно. В таких случаях расширение все еще хорошо определено; и, хотя мы, очевидно, не можем сгенерировать все расширение за конечное время, но если фактоид находится в расширении, то это возможно.

Предыдущий раздел иллюстрирует наш метод вычисления расширений для полупозитивных программ. Теперь расширим наш пример, чтобы показать, как вычислить расширение стратифицированной программы.

Предположим, что мы добавили правило, показанное ниже, к программе из предыдущего раздела. Отношение `t` здесь является дополнением транзитивного замыкания краевого отношения.

```
t(X,Y) :- p(X) & p(Y) & ~s(X,Y)
```

Поскольку это правило содержит негативное отношение, оно обязательно появится в более высоком слое, чем отношение `s`, и поэтому мы не будем вычислять выводы до тех пор, пока не закончим с отношением `s`.

В этом случае существует шестнадцать способов удовлетворить первые две подцели нашего правила, и, как мы видели в предыдущем разделе, девять из них удовлетворяют отношению `s`. Оставшиеся семь фактов удовлетворяют отношению `t`. Добавим их в наше расширение.

```
s(a,a)
s(b,a)
s(b,b)
s(c,a)
s(c,b)
s(d,a)
s(d,b)
```

Обратите внимание, что при наличии правил с негативными подцелями иногда возможно стратифицировать правила более чем одним способом. Но это не вызывает никаких проблем. Пока программа стратифицирована по отрицанию, только что приведенное определение дает один и тот же набор данных независимо от того, какая стратификация используется. Следовательно, существует только одно расширение для любой безопасной, стратифицированной логической программы.

7.6. Упражнения

7.1. Скажите, является ли каждое из следующих выражений синтаксически законным определением вида.

(a) `r(X,Y) :- p(X,Y) & q()`
(b) `r(X,Y) :- p(X,Y) & ~q(Y,X)`
(c) `~r(X,Y) :- p(X,Y) & q(Y,X)`
(d) `p(X,Y) & q(Y,X) :- r(X,Y)`
(e) `p(X,Y) & ~q(Y,X) :- r(X,Y)`

7.2. Пусть у нас есть набор данных с двумя символами `a` и `b` и двумя унарными отношениями `p` и `q`, где все возможные факты истинны, т. е. набор данных имеет вид `{p(a), p(b), q(a), q(b)}`. Предположим, что у нас есть замкнутая логическая программа, состоящая из этого набора данных и правила `r(X) :- p(X) & ~q(X)`.

(a) Сколько интерпретаций имеет эта программа?

(b) Сколько моделей она имеет?

(c) Сколько минимальных моделей она имеет?

7.3. Скажите, является ли каждый из следующих наборов правил стратифицированным.

(a)
```
r(X,Y) :- p(X,Y) & ~q(Y,X)
r(X,Y) :- p(X,Y) & ~q(X,Y)
```

(b)
```
r(X,Z) :- p(X,Z) & q(X,Z)
r(X,Z) :- r(X,Y) & ~r(Y,Z)
```

(c)
```
r(X,Z) :- p(X,Z) & ~q(X,Z)
r(X,Z) :- r(X,Y) & r(Y,Z)
```

7.4. Что такое $v(r, \Delta)$, где r – это `r(X,Y) :- p(X,Y) & p(Y,X)` и Δ – набор данных, показанный ниже?

```
p(a,a)
p(a,b)
p(b,a)
p(b,c)
```

7.5. Что такое $C(\Omega, \Delta)$, где Ω – это `{r(X,Z) :- p(X,Z), r(X,Z) :- r(X,Y) & r(Y,Z)}`, и является ли Δ набор данных, показанный ниже?

```
p(a,a)
p(a,b)
p(b,a)
p(b,c)
```

7.6. Каково расширение слоев Ω_1 и Ω_2 на Δ, где Ω_1 – это `{q(X) :- p(X,Y)}`, Ω_2 – `{r(X,Y) :- p(X,Y) & ~q(Y)}` и Δ – набор данных, показанный ниже?

```
p(a,b)
p(a,c)
p(b,d)
p(c,d)
```

Глава 8

Оценка вида

8.1. Введение

В предыдущей главе мы определили результат конструктивного применения стратифицированной логической программы – начиная с набора данных и последовательно применяя слои программы для получения расширения программы в целом. Это определение легко приводит к практическому методу вычисления таких расширений, известному как оценка *снизу вверх*.

Хотя оценка снизу вверх используется в некоторых системах логического программирования, многие механизмы оценки используют *нисходящий* подход к ответам на вопросы. Вместо того чтобы начинать с данных и работать вверх, такие механизмы начинают с запроса, на который нужно ответить, и работают вниз, используя правила для сокращения целей до подцелей, пока они не достигнут подцелей, полностью записанных в терминах базовых отношений. Преимущество такого подхода заключается в том, что эти механизмы оценки позволяют избежать генерации большого количества выводов, которые не имеют никакого отношения к рассматриваемому вопросу. Более того, в случаях, когда существует бесконечно много возможных выводов, они часто помогают найти ответы на конкретные вопросы, не проделывая бесконечно много работы.

Одним из недостатков нисходящей оценки является то, что для некоторых людей она более сложна для понимания, чем оценка снизу вверх. Существует также опасность ненужных бесконечных циклов, если правила написаны плохо. Однако эту опасность можно минимизировать или устранить, если понять, как работает процедура. Небольшое знакомство с нисходящей обработкой может помочь понять принцип ее работы и избежать написания плохих правил.

В этой главе будет представлена конкретная процедура нисходящей оценки. Мы начнем с определения нисходящего, обратного подхода к обработке целей и правил без переменных, затем опишем ключевой процесс унификации, затем объединяем эти два процесса в нисходящую процедуру для произвольных целей и правил.

8.2. Нисходящая обработка основных целей и правил

В этом разделе мы начнем обсуждение нисходящей оценки, сосредоточившись на целях и правилах без переменных. В следующем разделе рассмотрим способ сравнения выражений, содержащих переменные. Затем мы покажем, как объединить эту технику с процедурой, описанной здесь, чтобы получить процедуру оценки для произвольных целей и правил.

Нисходящая оценка – это рекурсивная процедура. Мы начинаем с цели, которую нужно «доказать». Мы либо доказываем цель напрямую, либо сводим ее к одной или нескольким подцелям и пытаемся доказать эти подцели. Способ обработки цели зависит от ее типа.

1. Если цель является атомом, и предикат в цели является базовым отношением, мы просто проверяем, содержится ли цель в нашем наборе данных. Если она там есть, мы добиваемся успеха. Если нет, то мы терпим неудачу.
2. Если цель является негативным литералом, выполняем процедуру на аргументе отрицания. Если удается доказать аргумент, то отрицание в целом ложно, и процедура завершается неудачно. Если доказать аргумент не удается, то отрицание в целом истинно, и значит, мы добились успеха.
3. Если наша цель – конъюнкция литералов, то сначала выполняем процедуру на первом конъюнкте. Если удается доказать эту цель, переходим к следующему конъюнкту и т. д. до тех пор, пока не закончим. Если мы не смогли доказать любую из целей, то мы не смогли доказать конъюнкцию в целом.
4. Если цель является атомом, а предикат в цели является отношением вида, то рассматриваем все правила с нашей целью в качестве головы. Для каждого такого правила мы выполняем нашу процедуру в теле правила. Мы добиваемся успеха в нашей цели тогда и только тогда, когда можем добиться успеха в теле какого-либо правила, в противном случае терпим неудачу.

В качестве примера рассмотрим набор данных, показанный слева внизу, и набор правил, показанный справа. Здесь есть три базовых отношения – `p`, `q`, `r` – и два отношения вида – `s` и `t`.

```
p(a)    s(b) :- p(a) & q(b) & r(c)
q(a)    s(b) :- p(a) & ~q(b) & ~t(c)
r(a)    t(c) :- r(c)
        t(c) :- r(d)
```

Теперь представим, что нас просят оценить цель `s(b)`. Поскольку `s` является отношением вида, мы рассматриваем правила, содержащие `s(b)` в голове, и выполняем процедуру на телах этих правил, одно за другим, пока не найдем одно успешное.

Используя первое правило для `s(b)`, мы сводим нашу цель к конъюнкции (`p(a) & q(b) & r(c)`) и оцениваем эту подцель. Так как p является базовым отношением, мы просто проверяем набор данных на наличие литерала `p(a)`. Поскольку `p(a)` есть в наборе данных, эта подцель считается истинной, и мы переходим ко второму конъюнкту `q(b)`. Так как `q` является базовым отношением, мы снова проверяем наш набор данных на наличие литерала `q(b)`. К сожалению, в этом случае мы терпим неудачу, поскольку `q(b)` не является членом набора данных. На этом этапе мы прекращаем обработку конъюнкции. (Поскольку конъюнкция в целом ложна, нет смысла в проверке `r(c)`.)

Не сумев доказать тело первого правила, мы переходим ко второму правилу и попытаемся сделать это снова, на этот раз с `p(a) & ~q(b) & ~t(c)` в качестве цели. Как и раньше, мы обнаруживаем, что `p(a)` истинно, и переходим ко второму конъюнкту. В этом случае у нас есть отрицание, поэтому выполняем процедуру рекурсивно на `q(b)`. Как и раньше, мы терпим неудачу. Следовательно, подцель `~q(b)` истинна. В итоге на этот раз мы продолжаем и выполняем процедуру на `~t(c)`. Поскольку `t` является отношением вида, мы выполняем процедуру на телах правил, содержащих `t(c)` в голове. Сначала пробуем `r(c)` и терпим неудачу, затем пробуем `r(d)` и снова терпим неудачу. Исчерпав все правила, определяющие `t(c)`, мы не можем доказать `t(c)`. Это означает, что отрицание `~t(c)` истинно. Из этого следует, что конъюнкция (`p(a) & ~q(b) & ~t(c)`) истинна, и, следовательно, наша общая цель `s(b)` истинна.

8.3. Унификация

Унификация – это процесс определения того, можно ли *унифицировать* два выражения, т. е. сделать их идентичными путем соответствующих замен переменных. Как мы увидим, такое определение является важной частью нисходящей оценки.

Подстановка – это конечная замена переменных элементами. В дальнейшем мы будем записывать подстановки в виде наборов правил замены, как показано ниже. В каждом правиле переменная, на которую указывает стрелка, должна быть заменена элементом, из которого направлена стрелка. В данном случае `X` заменяется на a, `Y` заменяется на `f(b)`, а `Z` – на `V`.

```
{X←a, Y←f(b), Z←V}
```

Заменяемые переменные вместе составляют домен подстановки, а заменяющие их элементы составляют *диапазон*. Например, в предыдущей подстановке домен – {X, Y, Z}, а диапазон – {a, f(b), V}.

Результатом применения подстановки σ к выражению φ является выражение $\varphi\sigma$, полученное из исходного выражения путем замены каждого вхождения каждой переменной в домене подстановки на элемент, с которым она связана.

```
q(X,Y){X←a, Y←f(b), Z←V} = q(a,f(b))
q(X,X){X←a, Y←f(b), Z←V} = q(a,a)
q(X,W){X←a, Y←f(b), Z←V} = q(a, W)
q(Z,V){X←a, Y←f(b), Z←V} = q(V,V)
```

Для двух или более подстановок можно определить одну подстановку, которая будет иметь тот же эффект, что и последовательное применение этих подстановок. Например, подстановки {X←a, Y←U, Z←V} и {U←d, V←e} могут быть объединены в одну подстановку {X←a, Y←d, Z←e, U←d, V←e}, которая имеет тот же эффект, что и первые две, при применении к любому выражению.

Вычислить *объединение* подстановки σ и τ очень просто. Есть два шага:

- сначала применяем τ к диапазону σ;
- затем присоединяем к σ все пары из τ с не содержащимися в σ переменными домена.

В качестве примера рассмотрим объединение, показанное ниже. В правой части первого равенства мы применили вторую подстановку для замен в первой подстановке. Во втором равенстве объединили правила из этой новой подстановки с не противоречащими правилами из второй подстановки.

```
{X←a, Y←U, Z←V}{U←d, V←e, Z←g}
  = {X←a, Y←d, Z←e}{U←d, V←e, Z←g}
  = {X←a, Y←d, Z←e, U←d, V←e}
```

Подстановка σ является *унификатором* для выражений φ и ψ тогда и только тогда, когда $\varphi\sigma = \psi\sigma$, т. е. результат применения φ к σ совпадает с результатом применения σ к ψ. Если два выражения имеют унификатор, то говорят, что они *унифицируемы*.

Выражения p(X,Y) и p(a,V) имеют унификатор, например {X←a, Y←b, V←b}, и следовательно, они унифицируемы. Результаты применения этой подстановки к двум выражениям показаны ниже.

```
p(X,Y) {X←a, Y←b, V←b} = p(a,b)
p(a,V) {X←a, Y←b, V←b} = p(a,b)
```

Обратите внимание, что, хотя эта подстановка является унификатором для двух выражений, она не единственный унификатор. Нам не обязательно заменять `b` на `Y` и `V`, чтобы объединить эти два выражения. Можно с равным успехом заменить `c` или `f(c)`, или `f(W)`. На самом деле возможно объединить выражения без изменения `V`, просто заменив `Y` на `V`.

При рассмотрении этих альтернатив должно быть ясно, что некоторые подстановки являются более общими, чем другие. Мы говорим, что подстановка σ является *такой же или более общей, чем* подстановка τ, тогда и только тогда, когда существует другая подстановка δ такая, что $\sigma\delta = \tau$. Например, подстановка `{X←a, Y←V}` является более общей, чем `{X←a, Y←c, V←b}`, так как существует подстановка `{V←c}`, которая, будучи применена к первой, дает вторую.

```
{X←a, Y←V}{V←c} = {X←a, Y←c, V←c}
```

При нисходящей оценке нас интересуют только унификаторы с максимальной общностью. А *наиболее общий унификатор* двух выражений σ обладает тем свойством, что он является таким же или более общим, чем любой другой унификатор.

Хотя возможно, что у двух выражений может быть более одного наиболее общего унификатора, все эти унификаторы структурно одинаковы, т. е. уникальны до переименования переменных. Например, `p(X)` и `p(Y)` могут быть объединены либо подстановкой `{X←Y}`, либо подстановкой `{Y←X}`, и любая из этих подстановок может быть получена из другой путем применения третьей. Это не относится к подстановкам, упомянутым ранее.

Одно из достоинств нашего языка заключается в том, что существует простая и недорогая процедура для вычисления наиболее общего унификатора любых двух выражений, если он существует.

Эта процедура предполагает представление выражений в виде последовательностей подвыражений. Например, выражение `p(a,b,Z)` можно представить как последовательность с четырьмя элементами – предикатом `p`, символом `a`, символом `b` и переменной `Z`.

Процедура также предполагает, что два выражения не имеют общих переменных. Как мы увидим в следующем разделе, это можно гарантировать, переименовав переменные в одном из выражений.

Мы начинаем процедуру с двух выражений и подстановки, которая изначально является пустой. Затем рекурсивно обрабатываем два выражения, сравнивая подвыражения на каждом шаге. Попутно мы расширяем подстановку с помощью присвоения значений переменным, как описано ниже. Если нам не удается объединить какую-либо пару подвыражений в любой точке этого процесса, процедура в целом терпит неудачу. Если мы завершаем это рекурсивное сравнение выраже-

ний, то процедура в целом успешна, и накопленная подстановка является наиболее общим унификатором.

При сравнении двух подвыражений мы сначала применяем подстановку к каждому из двух выражений, а затем выполняем следующую процедуру над двумя измененными выражениями.

1. Если одно из выражений является символом, а другое выражение – тем же символом, то процедура завершается успешно, возвращая в качестве результата немодифицированную подстановку.
2. Если одно из выражений является символом, а другое выражение – другим символом или составным выражением, то процедура завершается неудачно.
3. Если одно из выражений является переменной, а другое выражение является той же переменной, то процедура завершается успешно, возвращая в качестве результата немодифицированную подстановку.
4. Если хотя бы одно из выражений является переменной, а другое – любым другим выражением, мы действуем следующим образом. Сначала проверяем, содержит ли другое выражение переменную. Если переменная встречается в другом выражении, мы терпим неудачу (по причинам, описанным ниже). В противном случае мы обновляем нашу подстановку до объединения старой и новой подстановки, в которой мы связываем переменную со вторым измененным выражением.
5. Если два выражения являются последовательностями одинаковой длины, мы выполняем итерацию по всем выражениям, сравнивая, как описано выше.
6. Если выражения являются составными выражениями разной длины, процедура завершается неудачно.

В качестве примера работы этой процедуры рассмотрим вычисление наиболее общего унификатора для выражений `p(X,b)` и `p(a,Y)` с начальной подстановкой `{}`. Трассировка выполнения процедуры для этого случая показана ниже. Начало сравнения показано строкой с надписью «Compare» вместе со сравниваемыми выражениями и входной подстановкой. Результат каждого сравнения отображается в строке с меткой «Result» (либо подстановкой в случае успеха, либо «false» в случае неудачи). Отступы показывают глубину рекурсии процедуры.

```
Compare: p(X,b), p(a,Y), {}
    Compare: p, p, {}
    Result: {}
    Compare: X, a, {}
```

```
    Result: {X←a}
    Compare: Y, b, {X←a}
    Result: {X←a, Y←b}
Result: {X←a, Y←b}
```

В качестве другого примера рассмотрим процесс объединения выражения `p(X,X)` и выражения `p(a,Y)`. Ниже показана трассировка. Основной интерес в этом примере представляет сравнение последних аргументов в двух выражениях, т. е. `X` и `Y`. К моменту достижения этой точки `X` привязан к `a`, поэтому мы вызываем процедуру рекурсивно для `a` и `Y`, что приводит к привязке `Y` к `a`.

```
Compare: p(X,X), p(a,Y), {}
    Compare: p, p, {}
    Result: {}
    Compare: X, a, {}
    Result: {X←a}
    Compare: X, Y, {X←a}
        Compare: a, Y, {X←a}
        Result: {X←a, Y←a}
    Result: {X←a, Y←a}
Result: {X←a, Y←a}
```

Одной из примечательных частей процедуры объединения является проверка того, встречается ли переменная в выражении до того, как переменная будет привязана к этому выражению. Эта проверка называется проверкой на возникновение, так как она применяется для проверки того, встречается ли переменная в элементе, с которым она объединяется. Например, пытаясь объединить `p(X,X)` и `p(Y,f(Y))`, мы не хотим связывать `Y` с `f(Y)`, так как эти выражения никогда нельзя сделать похожими друг на друга, подставляя значение для `Y` последовательно во всем выражении.

8.4. Нисходящая обработка неосновных запросов и правил

Используя унификацию, возможно преобразовать процедуру нисходящей оценки основных запросов и правил в процедуру для оценки произвольных запросов и правил. Для этого необходимо три существенных изменения:

- процедура начинается с цели и подстановки;
- вместо того чтобы проверять, идентичны ли цель и фактоид или голова правила, процедура проверяет, являются ли они унифицируемыми;

- вместо того чтобы возвращать булево значение при каждом рекурсивном вызове, процедура возвращает подстановку, которая делает выбранную цель истинной, и использует эту подстановку при обработке всех оставшихся подцелей.

Шаги процедуры описаны ниже.

1. Если предикат в цели является базовым отношением, мы итерационно проходим по нашему набору данных, сравнивая цель с каждым фактоидом по очереди. Если существует расширение данной подстановки, которое объединяет цель и фактоид, добавляем эту расширенную подстановку в наш список ответов. Закончив изучение всех соответствующих фактоидов, мы возвращаем список подстановок, накопленных за время работы. (Если не находим ни одного фактоида, объединяемого с целью, возвращаем пустой список.)
2. Если наша цель – негативный литерал, мы выполняем процедуру на аргументе отрицания и заданной подстановке. Если результат пуст, возвращаем список, содержащий только заданную подстановку. В противном случае возвращаем пустой список, что свидетельствует о невозможности доказать отрицание.
3. Если наша цель – конъюнкция литералов, мы сначала выполняем процедуру на первом конъюнкте и заданной подстановке, чтобы получить список подстановок, удовлетворяющих этому конъюнкту. Затем рассматриваем список подстановок, вызывая процедуру рекурсивно для оставшихся конъюнктов с каждой подстановкой по очереди. Мы собираем ответы от этих рекурсивных вызовов и возвращаем список ответов в качестве значения процедуры.
4. Если наша цель – атом, а предикат – отношение вида, мы перебираем правила нашей программы. Сначала копируем каждое правило, заменяя переменные новыми переменными (чтобы избежать возможных конфликтов с переменными в цели). Затем мы пытаемся найти наиболее общий унификатор для заданной цели и головы правила, начинающегося с данной подстановки. Если это удается, вызываем процедуру рекурсивно для тела правила и полученного унификатора. Мы добавляем все подстановки, полученные в результате этого рекурсивного вызова, в наш выходной список. Закончив рассмотрение всех правил, возвращаем все найденные подстановки.

Снова рассмотрим набор данных, который мы видели ранее (показан слева внизу), и версию логической программы, в которой некоторые константы заменены переменными (показана справа внизу).

```
p(a)    s(X) :- t(X) & ~r(X)
p(b)    s(X) :- p(X) & ~q(X) & ~t(c)
p(c)    t(X) :- p(X) & q(X)
q(b)    t(X) :- r(X)
r(c)
```

Чтобы увидеть нашу процедуру в действии, давайте начнем с простого случая. Пусть мы хотим найти все объекты, которые встречаются как в отношении `p`, так и в отношении `q`. Вызываем нашу процедуру с `p(X) & q(X)` в качестве цели и пустой подстановкой `{}` в качестве начальной подстановки. Поскольку цель – это конъюнкция, сначала вызываем процедуру рекурсивно на `p(X)` и `{}`. Наша цель `p(X)` с начальной подстановкой `{}` объединяет все три фактоида `p` в нашем наборе данных, и поэтому результатом рекурсивного вызова является список полученных подстановок, т. е. `{X←a}`, `{X←b}` и `{X←c}`. Для каждой из этих подстановок мы затем рекурсивно вызываем процедуру для второго конъюнкта `q(X)`. Не существует фактоида, который объединяется с `q(X)` при подстановке `{X←a}`, поэтому в этом случае мы возвращаем пустой список. Во втором случае нам повезло больше. `q(X)` и `q(b)` объединяются с подстановкой `{X←b}`, поэтому возвращаем список, содержащий эту подстановку. Третий случай похож на первый тем, что в нем не существует унифицируемого фактоида, поэтому мы снова получаем пустой список. Проверив второй конъюнкт для каждого из ответов первого конъюнкта, мы возвращаем список найденных подстановок. В данном случае список состоит из единственной подстановки `{X←b}`.

В качестве более интересного примера представим, что мы хотим оценить цель `s(X)`, т. е. получить все объекты, которые удовлетворяют отношению `s`. Мы вызываем нашу процедуру для `s(X)` и пустой подстановки `{}`. Поскольку `s` является отношением вида, рассматриваем правила, в которых `s` появляется в голове. Мы копируем первое правило, в результате чего получаем новое правило `s(X1) :- p(X1) & q(X1) & r(c)`, и пытаемся объединить нашу цель с головой этого правила. В этом случае мы добиваемся успеха с помощью подстановки `{X←X1}`. Затем вызываем процедуру рекурсивно для тела правила и этой подстановки и действуем, как и раньше, в результате чего получаем окончательный ответ, содержащий единственную подстановку `{X←X1, X1←b}`.

Только что описанная процедура вычисляет все ответы на заданный запрос. Если нам нужно только несколько ответов, можно использовать «конвейерную» версию алгоритма, которая возвращает по одному ответу за раз. При обработке правила вместо того, чтобы вычислять все ответы на подцель перед тем, как продолжить, при получении одного ответа мы проверяем, приводит ли это решение к ответу на оставшиеся подцели. Если да, то мы возвращаем этот ответ. Если нет, генерируем другой ответ на нашу подцель и пробуем снова.

8.5. Упражнения

8.1. Предположим, что нам нужно запустить наш метод нисходящей оценки на наборе данных, показанном ниже слева, и наборе правил, показанном справа. Сколько обращений к набору данных потребуется для того, чтобы оценить `s(b)` (каждый поиск фактоида считается одним обращением.)?

```
p(a)    s(b) :- p(a) & q(b) & r(c)
q(a)    s(b) :- p(a) & ~q(b) & ~t(c)
r(a)    t(c) :- r(c)
        t(c) :- r(d)
```

8.2. Для каждой из следующих пар предложений скажите, являются ли эти предложения унифицируемыми, и дайте наиболее общий унификатор для тех из них, которые являются унифицируемыми.

(a) `rp(X,X)` и `p(a,Y)`
(b) `p(X,X)` и `p(f(Y),Z)`
(c) `p(X,X)` и `p(f(Y),Y)`
(d) `p(f(X,Y),g(Z,Z))` и `p(f(W,Z),V),W)`

8.3. Предположим, что мы запустили наш метод нисходящей оценки на наборе данных, показанном ниже слева, и наборе правил, показанном справа, с целью `r(a,d)`. Покажите трассировку подцелей в том порядке, в котором они обрабатываются, и результаты.

```
p(a,b)    r(X,Z) :- p(X,Z)
p(a,c)    r(X,Z) :- p(X,Y) & p(Y,Z)
p(c,d)
```

Глава 9

Примеры

9.1. Введение

В этой главе мы рассмотрим несколько простых примеров определений представлений. Приведенные здесь примеры являются простыми в том смысле, что в них не используются конструкторы или составные элементы. В следующих главах мы рассмотрим более сложные примеры, где конструкторы и составные элементы играют важную роль.

9.2. Пример – родство

Чтобы проиллюстрировать использование правил при определении представлений, рассмотрим еще раз мир родственных отношений. Начиная с некоторых базовых отношений, мы можем определить различные интересные отношения вида.

Например, первое предложение ниже определяет отношение `father` (отец) в терминах `parent` (родитель) и `male` (мужчина). Второе предложение определяет `mother` (мать) в терминах `parent` (родитель) и `female` (женщина).

```
father(X,Y) :- parent(X,Y) & male(X)
mother(X,Y) :- parent(X,Y) & female(X)
```

Приведенное ниже правило определяет отношение `grandparent` (бабушка или дедушка) в терминах отношения `parent`. Лицо `X` является дедушкой или бабушкой лица `Z`, если `X` является родителем лица `Y`, а `Y` является родителем `Z`. Переменная `Y` здесь является *потоковой переменной*, которая связывает первую подцель со второй, но сама не появляется в голове правила.

```
grandparent(X,Z) :- parent(X,Y) & parent(Y,Z)
```

Обратите внимание, что одно и то же отношение может появляться в голове более чем одного правила. Например, отношение персоны истинно для лица `Y`, если существует `X` такой, что `X` является родителем `Y`,

или если Y является родителем некоторого лица Z. В этом случае условия дизъюнктивны (хотя бы одно из них должно быть истинным), тогда как условия в случае с дедом являются конъюнктивными (оба должны быть истинными).

```
person(X) :- parent(X,Y)
person(Y) :- parent(Y,Z)
```

Лицо X является предком лица Z, если X является родителем Z или если существует лицо Y такое, что X является родителем Y, а Y является предком Z. Этот пример показывает, что отношение может появляться в собственном определении (см. ниже обсуждение стратификации для ограничения этой возможности).

```
ancestor(X,Y) :- parent(X,Y)
ancestor(X,Z) :- parent(X,Y) & ancestor(Y,Z)
```

Человек является родителем тогда и только тогда, когда у него есть дети, а бездетный человек – это тот, у которого нет детей. Мы можем определить эти свойства с помощью правил, показанных ниже. Первое правило гласит, что isparent истинно для X, если X является родителем некоторого человека Y. Второе правило гласит, что X бездетен, если X является человеком и ложно то, что X является родителем.

```
isparent(X) :- parent(X,Y)
childless(X) :- person(X) & ~isparent(X)
```

Обратите внимание на использование отношения isparent в определении бездетности. Возникает соблазн написать правило бездетности как childless(X) :- person(X) & ~parent(X,Y). Однако это было бы неправильно. Это определило бы X как бездетного, если X является человеком и существует *некоторый* Y такой, что ~parent(X,Y) истинно. Но мы действительно хотим сказать, что ~parent(X,Y) истинно для *всех* Y. Определение isparent и использование его отрицания в определении бездетности позволяет нам выразить эту *универсальную количественную оценку*.

9.3. Пример – мир блоков

Еще раз рассмотрим мир блоков, представленный в главе 2. Сцена мира блоков, описанная ранее, повторена ниже.

Как и раньше, мы используем словарь с пятью символами для обозначения пяти блоков в сцене – a, b, c, d и e. Для обозначения того, что объект является блоком, используем унарный предикат block. Бинарный предикат on используем для выражения того факта, что один блок находится непосредственно на другом. Мы используем above, чтобы сказать, что

блок находится где-то над другим блоком. Унарный предикат `cluttered` служит для выражения того, что блок имеет другие блоки поверх него, и мы используем унарный предикат `clear`, чтобы сказать, что блок не имеет ничего поверх него. Используем унарный предикат `supported`, чтобы сказать, что блок опирается на другой блок, и мы используем унарный предикат `table`, чтобы сказать, что блок лежит на столе.

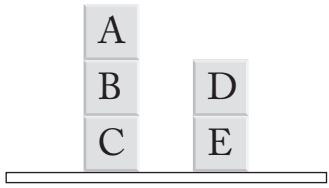

Рис. 9.1. Одно состояние мира блоков

Имея этот словарь, мы можем описать сцену на рис. 9.1, написав атомарные предложения, в которых указывается, какие отношения существуют между какими объектами или группами объектов. Начнем с блоков. В данном случае имеется пять блоков, названных `a`, `b`, `c`, `d` и `e`.

```
block(a)
block(b)
block(c)
block(d)
block(e)
```

Некоторые из блоков находятся друг над другом, а некоторые – нет. Следующие предложения отражают отношения на рис. 9.1.

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

Возможно сделать то же самое для других отношений. Однако есть более простой способ. Каждое из оставшихся отношений может быть определено в терминах `block` и `on`. Эти определения вместе с фактами об отношениях `block` и `on` логически влекут за собой все остальные предложения базовых отношений или их отрицание. Следовательно, учитывая эти определения, нам не нужно записывать никаких дополнительных данных.

Блок удовлетворяет отношению `cluttered` тогда и только тогда, когда существует блок, лежащий на нем. Блок удовлетворяет отношению `clear` тогда и только тогда, когда на нем ничего нет.

```
cluttered(Y) :- on(X,Y)
clear(X) :- block(X) & ~cluttered(X)
```

Блок удовлетворяет отношению supported тогда и только тогда, когда он опирается на некоторый блок. Блок удовлетворяет отношению table тогда и только тогда, когда он не опирается на какой-либо блок.

```
supported(X) :- on(X,Y)
table(X) :- block(X) & ~supported(X)
```

Три блока удовлетворяют отношению stack тогда и только тогда, когда первый лежит на втором и второй – на третьем.

```
stack(X,Y,Z) :- on(X,Y) & on(Y,Z)
```

Отношение above немного сложно определить правильно. Один блок находится над другим блоком тогда и только тогда, когда первый блок находится на втором или на другом блоке, который находится над вторым блоком. Учитывая полное определение для отношения on, следующие два правила определяют уникальное отношение above.

```
above(X,Y) :- on(X,Y)
above(X,Z) :- on(X,Y) & above(Y,Z)
```

Одним из преимуществ определения отношений в терминах других отношений является экономия. Если мы записываем информацию о каждом объекте и кодируем связь между отношением on и этими другими отношениями, нет необходимости записывать какие-либо базовые реляционные предложения для этих отношений.

Еще одним преимуществом является то, что эти общие предложения применимы к сценам мира блоков, отличным от изображенной здесь. Можно создать сцену мира блоков, в которой ни одно из перечисленных нами предложений не будет истинным, но эти общие определения все равно верны.

9.4. Пример – модульная арифметика

В этом примере мы покажем, как определить модульную арифметику. В модульной арифметике существует только конечное количество чисел. Например, в модульной арифметике с модулем 4 есть только четыре целых числа – 0, 1, 2, 3, и все. Наша цель – дать определение сложению. Это скромная цель, но, как только мы увидим, как это сделать, станет возможно использовать тот же подход для определения других арифметических понятий.

Начнем с отношения number, которое справедливо для каждого числа. Мы можем полностью охарактеризовать отношение number, написав основные предложения по одному для каждого числа.

```
number(0)
number(1)
```

```
number(2)
number(3)
```

Теперь определим следующее отношение, которое для каждого числа дает следующее большее число, возвращаясь к 0 после того, как мы достигнем 3.

```
next(0,1)
next(1,2)
next(2,3)
next(3,0)
```

Таблица сложения для модульной арифметики – это обычная таблица сложения для произвольных чисел, за исключением того, что мы возвращаемся к ее началу, когда переходим через 3. Для такой маленькой арифметики легко записать основные факты для сложения, как показано ниже.

```
add(0,0,0)  add(1,0,1)  add(2,0,2)  add(3,0,3)
add(0,1,1)  add(1,1,2)  add(2,1,3)  add(3,1,0)
add(0,2,2)  add(1,2,3)  add(2,2,0)  add(3,2,1)
add(0,3,3)  add(1,3,0)  add(2,3,1)  add(3,3,2)
```

Это один из вариантов, но можно сделать лучше. Вместо того чтобы записывать все эти факты, мы можем создать правила для определения сложения в терминах `number` и `next`, затем использовать эти правила для фактов о сложении. Соответствующие правила показаны ниже.

```
add(0,Y,Y) :- number(Y)
add(X2,Y,Z2) :- next(X,X2) & distinct(X2,0) & add(X,Y,Z) & next(Z,Z2)
```

Во-первых, у нас есть правило тождества. Прибавление 0 к любому числу дает то же самое число. Во-вторых, есть правило преемника. Если `X2` является преемником `X`, `Z` является суммой `X` и `Y`, а `Z2` является преемником `Z`, то `Z2` – это сумма `X2` и `Y`.

Как упоминалось ранее, одним из преимуществ такого способа является экономия. С помощью этих предложений нам не нужно выписывать факты о сложении, приведенные выше. Все они могут быть вычислены из фактов о `next` и правил, определяющих сложение. Второе преимущество – универсальность. Наши предложения определяют сложение в терминах `next` для арифметики с любым модулем, а не только с модулем 4.

9.5. Пример – направленные графы

Рассмотрим проблему описания конечных графов и определения свойств этих графов. Начнем с описания небольшого направленного

графа. Для обозначения узлов графа мы используем символы, а для обозначения дуг графа – отношение `edge`. Например, приведенный ниже набор данных описывает граф с четырьмя узлами и четырьмя дугами – от `a` к `b`, от `b` к `c`, от `c` к `d` и дугой от `d` к `c`.

```
node(a)
node(b)
node(c)
node(d)

edge(a,b)
edge(b,c)
edge(c,d)
edge(d,c)
```

Теперь дополним эту программу некоторыми правилами, определяющими различные отношения между узлами в графе.

```
p(X) :- edge(X,X)
q(X,Y) :- edge(X,Y)
q(X,Y) :- edge(Y,X)
r(X,Y,Z) :- edge(X,Y) & edge(Y,Z)
s(X,Y) :- edge(X,Y)
s(X,Z) :- edge(X,Y) & s(Y,Z)
```

Здесь отношение `p` истинно для каждого узла, который имеет дугу к самому себе. Отношение `q` истинно для двух узлов тогда и только тогда, когда существует дуга из первого узла во второй или из второго в первый. Отношение `r` истинно для трех узлов тогда и только тогда, когда существует дуга из первого узла во второй и дуга из второго в третий. Отношение `s` является транзитивным замыканием отношения `edge`.

Определение свойств графа в целом часто оказывается сложнее, чем определение свойств отдельных узлов, поскольку обычно мы должны убедиться, что свойства относятся ко всем узлам графа. Сложность в таких ситуациях заключается в том, чтобы охарактеризовать те случаи, когда граф *не* обладает желаемым свойством, и затем определить желаемое свойство как отрицание этих случаев.

Предположим, например, что мы хотим определить понятие рефлексивности. Граф является *рефлексивным* тогда и только тогда, когда каждый его узел имеет дугу к самому себе. Чтобы определить это понятие, мы сначала определим, что означает для графа быть нерефлексивным. Граф *нерефлексивен* тогда и только тогда, когда существует узел, который не имеет дугу к самому себе. Учитывая это определение, мы можем определить рефлексивность как отрицание этого свойства.

```
nonreflexive :- node(X) & ~edge(X,X)
reflexive :- ~nonreflexive
```

Мы также можем определить это понятие с помощью агрегата `countofall`, как показано ниже. Граф *нерефлексивен* тогда и только тогда, когда число всех узлов с дугами к себе равно нулю.

```
nonreflexive :- evaluate(countofall(X,edge(X,X)),0)
```

При таком подходе нам не нужно определять вспомогательное отношение, как в варианте выше. Однако здесь использован агрегатный оператор, который некоторые считают более сложным.

9.6. Упражнения

9.1. Два человека являются родными братьями или сестрами (`sibling`) тогда и только тогда, когда у них есть общий родитель. Напишите правила, определяющие бинарное отношение `sibling` в терминах отношения `parent` (*подсказка*: вам понадобится встроенное отношение `distinct`, чтобы получить правильное определение `sibling`).

9.2. Напишите правила, определяющие бинарное отношение `uncle` (дядя) и бинарное отношение `aunt` (тетя) в терминах `parent` (родитель), `male` (мужчина) и `female` (женщина).

9.3. Два блока находятся на одинаковой высоте тогда и только тогда, когда они опираются на одинаковое количество блоков. Определите отношение одинаковой высоты `sameheight` в терминах блока `block` и таким образом, чтобы оно работало независимо от числа блоков в мире блоков.

9.4. Определите умножение `mul` для модульной арифметики в терминах `number` и `next`. Чтобы упростить задачу, вы можете определить дополнительные предикаты в терминах `number` и `next`.

9.5. Рассмотрим направленный граф, заданный узлом с унарным отношением `node` и бинарным базовым отношением `edge`. Напишите правила для определения того, является ли граф *асимметричным*, т. е. если существует дуга от одного узла ко второму, то граф не содержит дуги от второго узла к первому.

9.6. Рассмотрим направленный граф, заданный узлом с унарным отношением `node` и бинарным базовым отношением `edge`. Напишите правила, позволяющие определить, является ли граф *симметричным*, т. е. если существует дуга от одного узла ко второму, то существует и дуга от второго узла к первому.

9.7. Рассмотрим направленный граф, заданный узлом с унарным отношением `node` и бинарным базовым отношением `edge`. Напишите

правила, позволяющие определить, является ли граф *транзитивным*, т. е. если существует дуга из x в y и дуга из y в z, то существует дуга из x в z.

9.8. Рассмотрим направленный граф, заданный узлом с унарным отношением node и бинарным базовым отношением edge. Напишите правила для определения того, является ли граф *ациклическим*, т. е. не существует последовательности дуг, соединяющих узел с самим собой.

Глава 10

Списки, множества, деревья

10.1. Введение

В этой главе мы начнем рассмотрение определений представлений, включающих конструкторы и составные элементы. Приведенные здесь примеры касаются представления информации о списках, деревьях и наборах. В последующих главах мы рассмотрим представление информации о динамических системах и представление метазнаний (т. е. информации об информации).

10.2. Пример – арифметика Пеано

Арифметика Пеано – это раздел математики, связанный с целыми неотрицательными числами, функцией сложения и отношением «*меньше чем*».

Арифметика Пеано сложнее модульной арифметики тем, что в ней мы имеем бесконечное множество объектов для рассмотрения, т. е. целые числа 0, 1, 2, 3… Поскольку таких чисел бесконечно много, нам нужно бесконечно много элементов для их описания в нашем языке.

Один из способов получить бесконечно много элементов – расширить словарь, включив в него бесконечное множество символов. К сожалению, это сделает невозможной работу по определению арифметики, так как нам пришлось бы написать бесконечно много предложений.

К счастью, существует лучший способ. В частности, мы можем представлять числа с помощью одного символа (например, `0`) и одного унарного конструктора (например, `s`). При таком подходе мы представляем число 0 символом `0`, а каждое другое натуральное число n представляем, применяя конструктор `s` ровно *n* раз. Например, в этой кодировке `s(0)` представляет 1; `s(s(0))` представляет 2; `s(s(s(0))` представляет 3 и т. д. При таком кодировании мы автоматически получаем бесконечное множество основных элементов.

К сожалению, даже при таком представлении определение арифметики Пеано является более сложной задачей, чем определение модульной арифметики. Мы не можем записать все факты, характеризующие

сложение, умножение и т. д., потому что существует бесконечно много случаев для рассмотрения. Для арифметики Пеано мы должны полагаться на определения вида не только потому, что они более экономны, но и потому, что это единственный способ, которым возможно охарактеризовать эти понятия в конечном пространстве.

Давайте сначала рассмотрим предикат `number`. Приведенные здесь правила определяют отношение `number` в терминах `0` и `s`.

```
number(0)
number(s(X)) :- number(X)
```

Предикат `next` относится к любому натуральному числу и числу, которое следует за ним. Например, у нас есть `next(0,s(0))` и `next(s(0),s(s(0)))` и т. д. Мы можем определить `next` в общем случае, как показано ниже.

```
next(X,s(X)) :- number(X)
```

Имея `number` и `next`, можно определить обычные арифметические отношения. Например, следующие предложения определяют отношение сложения. При добавлении 0 к любому числу получается это число. Если при добавлении числа `X` к числу `Y` получается число `Z`, то добавление преемника `X` к `Y` дает преемника `Z`.

```
add(0,Y,Y) :- number(Y)
add(s(X),Y,s(Z)) :- add(X,Y,Z)
```

Используя `next`, мы также можем определить отношение «*меньше чем*» аналогичным образом. Число `X` меньше числа `Z`, если `next` выполняется для `X` и `Z` *или* если существует число `Y` такое, что `Y` – это число после `X` и `Y` меньше, чем `Z`.

```
less(X,Z) :- next(X,Z)
less(X,Z) :- next(X,Y) & less(Y,Z)
```

Прежде чем мы оставим обсуждение арифметики, полезно рассмотреть понятие диофантовых уравнений. Полиномиальное уравнение – это предложение, составленное с использованием только сложения, умножения и возведения в степень с фиксированными показателями (т. е. числами, а не переменными). Например, приведенное ниже в традиционной математической нотации выражение является полиномиальным уравнением:

$$x^2 + 2y = 4z$$

Натуральное диофантово уравнение – это полиномиальное уравнение, в котором переменные ограничены целыми неотрицательными числами. Например, приведенное полиномиальное уравнение также

является уравнением Диофанта и имеет решение в неотрицательных числах, т. е. $x = 4$, $y = 8$ и $z = 8$.

Диофантовы уравнения могут быть легко выражены в виде предложений арифметики Пеано. Например, мы можем представить приведенное выше уравнение с помощью правила, показанного ниже.

```
solution(X,Y,Z) :-
mul(X,X,X2) &
mul(s(s(0)),Y,2Y) &
mul(s(s(s(s(0)))),Z,4Z) &
add(X2,2Y,4Z)
```

Это немного беспорядочно, но вполне выполнимо. И мы всегда можем навести порядок, добавив немного синтаксического «сахара» в нашу нотацию, чтобы она выглядела как традиционная математическая нотация.

10.3. Списки

Список – это конечная последовательность объектов. Списки могут быть простыми, например `[a, b, f(c), d]`. Списки также могут быть вложенными внутри других списков, например `[a, [b, f(c)], d]`.

Говоря о списках произвольной длины, мы используем бинарный конструктор `cons`, а для обозначения пустого списка – символ `nil`. В частности, элемент вида `cons(`τ`1, `τ`2)` обозначает список, в котором τ1 обозначает первый элемент, а τ2 – остальную часть списка.

Например, используя этот подход, мы можем представить список `[a, b, c]` составным элементом, показанным ниже.

```
cons(a,cons(b,cons(c,nil)))
```

Правила, определяющие примитивы и списки.

```
primitive(a)
primitive(b)
primitive(c)
list(nil)
list(cons(X,Y)) :- object(X) & list(Y)
object(X) :- primitive(X) object(X) :- list(X)
```

Преимущество этого представления заключается в том, что оно позволяет нам описывать отношения в списках, не обращая внимания на длину или глубину.

В качестве примера рассмотрим определение бинарного отношения `mem`, которое содержит объект и список, если объект является `mem` верхнего уровня списка. Используя конструктор `cons`, мы можем охарактеризо-

вать отношение mem, как показано ниже. Очевидно, что объект является mem, если он является первым элементом. Однако он также является mem, если он является mem остальной части списка.

```
mem(X,cons(X,Y)) :- объект(X) & список(Y)
mem(X,cons(Y,Z)) :- объект(Y) & mem(X,Z)
```

Аналогичным образом можно определить и другие отношения в списках. Например, следующие правила определяют отношение app. Значением app (его последним аргументом) является список, состоящий из элементов в списке, представленном в качестве первого аргумента, за которым следуют элементы в списке, представленном в качестве второго аргумента. Например, мы имеем app(cons(a,nil), cons(b, cons(c, nil)), cons(a, cons(b, cons(c, nil)))).

```
app(nil,Y,Y) :- list(Y)
app(cons(X,Y),Z,cons(X,W)) :- объект(X) & app(Y,Z,W)
```

И наконец, замечание по синтаксису. Существует три способа записи списков – с помощью квадратных скобок, с помощью cons и с помощью оператора !.

Если известны все элементы списка, мы можем записать список, заключив элементы в квадратные скобки и разделяя их запятыми, как в примере, показанном ниже.

```
[a,b,c]
```

Этот список также можно представить с помощью конструктора cons, как показано ниже.

```
cons(a, cons(b, cons(c, nil)))
```

Чтобы сократить представление списков, можно использовать оператор ! вместо cons. Например, мы можем записать только что рассмотренный список так:

```
a!b!c!nil
```

Все три этих представления эквивалентны. Фактически они обычно преобразуются в одно и то же внутреннее представление. Однако каждое из них имеет значение в различных обстоятельствах, и поэтому все три представления разрешены.

Списки являются чрезвычайно универсальным средством представления информации, и читателю рекомендуется как можно лучше ознакомиться с техникой написания определений для отношений в списках. Как и в случае со многими другими задачами, лучшим подходом к приобретению навыков является практика.

10.4. Пример – сортированные списки

Сортированный список – это список, в котором последовательные элементы удовлетворяют заданному отношению упорядочивания. Например, список [1,2,3] является сортированным с отношением упорядочивания «меньше чем», а [1,3,2] – нет.

Обратите внимание, что часто объекты появляются в сортированном списке более одного раза. Например, [1,2,2,3] – это сортированный список с отношением упорядочивания «меньше или равно». Однако это не так, если отношение упорядочивания не является рефлексивным. Например, список [1,2,2,3] не является отсортированным для отношения упорядочивания «меньше чем».

Учитывая отношение упорядочивания (например, отношение «меньше или равно» – `leq`), мы можем легко определить предикат `sorted`, который будет истинным для сортированных списков и ложным для всего остального (см. ниже). Пустой список отсортирован. Любой список из одного элемента отсортирован. Список из двух или более элементов отсортирован, если первый элемент меньше или равен второму, а конец списка отсортирован.

```
sorted(nil)
sorted([X]) :- object(X)
sorted(cons(X,cons(Y,L))) :- leq(X,Y) & sorted(cons(Y,L))
```

Как и в случае с несортированными списками, мы можем определять отношения и на сортированных списках. Однако при этом необходимо убедиться, что результирующие списки отсортированы. Хотя отношение `app`, определенное ранее, может быть применено к сортированным спискам, результат может быть не отсортирован.

Одним из способов решения этой проблемы является применение сортировщика к списку, чтобы обеспечить его сортировку. Например, следующее определение представления определяет отношение сортировки `sortappend` в терминах `app` и `sort`.

```
sortappend(X,Y,Z) :- app(X,Y,W) & sort(W,Z)
```

Этот подход работает и дает правильный ответ (даже если входные списки не отсортированы). Однако, если мы знаем, что входные списки отсортированы, можно определить отношение сортировки другим способом.

```
merge(nil,Y,Y) :- list(Y)
merge(X!L,Y,Z) :- merge(L,Y,W) & insert(X,W,Z)
```

Для многих эта версия является более подходящей, чем приведенная выше. Более того, как мы увидим, когда перейдем к выполнению

программы, она может выполняться более эффективно, чем предыдущая.

10.5. Пример – множества

Множество – это коллекция объектов. Множество отличается от списка двумя признаками:

- объекты могут появляться в списках более одного раза, в то время как в множестве это невозможно;
- порядок следования элементов в списке имеет важное значение, в то время как для множеств порядок не имеет значения.

В дальнейшем мы будем представлять множества как упорядоченные списки. Поскольку порядок в множествах не имеет значения, неважно, какой порядок мы выберем, а сохранение порядка облегчает определение отношений в множествах; и, как мы увидим в дальнейшем, это делает ответы на запросы более эффективными.

Если мы представляем множества в виде списков, можно узнать, является ли объект членом множества, используя отношение `mem`, определенное ранее. Однако, если множества представлены в виде упорядоченных списков, есть лучший способ.

```
mem(X,X!L) :- list(L)
mem(X,Y!L) :- less(Y,X) & mem(X,L)
```

Как и в определении членства в списке, эта версия циклически перебирает элементы множества. Также если мы попадаем в подмножество, в котором объект является первым элементом, то мы успешно завершаем работу. Основное отличие заключается в том, что иногда мы можем остановиться раньше, если объект не является элементом множества. В частности, если, перебирая подсписки подсписка, мы доберемся до подсписка, в котором указанный объект меньше первого элемента, то мы можем прекратить поиск, так как знаем, что все последующие объекты в списке больше этого элемента.

Одно множество является подмножеством другого тогда и только тогда, когда каждый элемент первого множества является элементом второго. Мы можем определить отношение подмножества `subset`, как показано ниже.

```
subset(nil,Y) :- list(Y)
subset(X!L,Y) :- mem(X,Y) & subset(L,Y)
```

Пересечение двух множеств – это множество, состоящее из всех объектов, которые встречаются в обоих множествах.

```
intersection(nil,Y,nil) :- list(Y)
intersection(X!L,Y,X!Z) :- mem(X,Y) & intersection(L,Y,Z)
intersection(X!L,Y,Z) :- ~mem(X,Y) & intersection(L,Y,Z)
```

Объединение двух множеств – это множество, состоящее из всех элементов любого из множеств. Если мы представляем множества как сортированные списки, то объединение идентично отношению слияния `merge`, определенному ранее.

10.6. Пример – деревья

Отношение `cons` также может быть использовано для представления произвольных деревьев. Например, `cons(cons(a,b),cons(c,d))` представляет бинарное дерево с `a`, `b`, `c` и `d` в качестве листьев.

Отношение `among` истинно для объекта и дерева, если объект появляется где-то в дереве.

```
among(X,X) :- object(X)
among(X,cons(Y,Z)) :- among(X,Y)
among(X,cons(Y,Z)) :- among(X,Z)
```

Обратите внимание, что если указанный объект сам является деревом, то это отношение аналогично отношению `subtree` для деревьев.

10.7. Упражнения

10.1. Скажите, является ли каждое из следующих предложений расширением отношения `app`, определенного в разделе 10.3.

(a) `app(nil,nil,nil)`
(b) `app(cons(a,nil),nil,cons(a,nil))`
(c) `app(cons(a,nil),cons(b,nil),cons(a,b))`
(d) `app(cons(cons(a,nil),nil),cons(b,nil),cons(a,cons(b,nil)))`

10.2. `last` – это бинарное отношение, которое имеет место для объекта и списка тогда и только тогда, когда объект является *последним* элементом списка. Например, `last(c,[a,b,c])` истинно. Напишите логическую программу, определяющую отношение `last`.

10.3. `rev` – бинарное отношение для списков. Это отношение выполняется для двух списков тогда и только тогда, когда второй содержит те же элементы, что и первый, только в обратном порядке. Например, `rev([a,b,c],[c,b,a])` истинно. Напишите логическую программу, определяющую отношение `rev`. *Подсказка*: полезно определить вариацию `app`, а затем использовать эту вариацию при определении `rev`.

10.4. `delete` – это тернарное отношение, которое содержит объект и два списка и выполняется тогда и только тогда, когда второй список является копией первого с удалением всех вхождений данного объекта. Например, `delete(b,[a,b,c,b,d],[a,c,d])` истинно. Напишите логическую программу, которая определяет отношение `delete`.

10.5. `substitute` – это 4-арное отношение, которое выполняется для двух объектов и двух списков тогда и только тогда, когда второй список является копией первого со всеми вхождениями второго объекта, замененными первым объектом. Например, `substitute(b,d,[a,d,d,c],[a,b,b,c])` истинно. Напишите логическую программу, определяющую отношение `substitute`.

10.6. `adjacent` – это тернарное отношение, которое имеет место для двух объектов и списка тогда и только тогда, когда первый объект и второй являются соседними в списке. Например, `adjacent(b,c,[a,b,c,d])` истинно. Напишите логическую программу, определяющую отношение `adjacent`.

10.7. `sublist` – это бинарное отношение, которое имеет место для двух списков тогда и только тогда, когда первый список является смежным подсписком второго. Например, `sublist([b,c],[a,b,c,d])` истинно, в то время как `sublist([b,d],[a,b,c,d])` – нет. Напишите логическую программу, определяющую отношение `sublist`.

10.8. `sort` – это бинарное отношение, которое имеет место для двух списков тогда и только тогда, когда второй список является версией первого в отсортированном порядке. Например, `sort([2,1,3,2],[1,2,2,3])` истинно. Напишите логическую программу, определяющую отношение `sort` в терминах `min`.

10.9. `powerset` – это бинарное отношение, которое имеет место для двух множеств тогда и только тогда, когда второе множество является `powerset` первого, т. е. второе множество является множеством всех подмножеств первого множества. Например, `powerset([a,b],[[],[a],[b],[a,b]])` истинно. Напишите логическую программу, которая определяет отношение `powerset`.

Глава 11

Динамические системы

11.1. Введение

Динамическая система – это система, состояние которой изменяется с течением времени. В некоторых случаях изменения происходят в ответ на чисто внутренние события (например, тиканье часов). В некоторых случаях эти изменения вызываются внешними воздействиями. В этой главе мы рассмотрим использование логического программирования для моделирования чисто реактивных систем, т. е. таких, которые изменяются в ответ на внешние воздействия.

Снова рассмотрим мир блоков, представленный в главе 2. Одно из состояний мира блоков показано ниже. Здесь блок C находится на блоке A, блок A находится на блоке B, а блок E находится на блоке D.

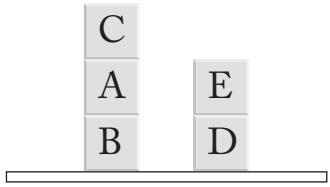

Рис. 11.1. Исходное состояние мира блоков

Теперь рассмотрим динамическую вариацию этого мира, в которой у нас есть операции, способные изменять состояние мира. Например, отсоединение C от A приводит к состоянию, показанному на рис. 11.2 слева, а отсоединение E от D – к состоянию, показанному там же справа.

Рис. 11.2. Полученные состояния мира блоков

В этой главе мы рассмотрим распространенный способ моделирования поведения подобных систем. В следующем разделе мы представим флюенты (факты, которые меняют истинностное значение от одного состояния к другому), затем рассмотрим действия (входы, которые изменяют истинностное значение). Затем мы рассмотрим, как писать правила представления, определяющие влияние действий на флюенты. Учитывая эту аксиоматизацию, мы покажем, как моделировать эффекты действий и планировать последовательности действий, которые приводят к желаемым состояниям.

11.2. Представление

В главе 9 мы рассмотрели описание состояние мира блоков в виде набора данных, используя унарное отношение `block` и бинарное отношение `on`, а также как определить другие отношения, такие как `clear` и `table`, в терминах этих базовых отношений.

К сожалению, такого представления недостаточно для описания динамического поведения. В динамической версии мира блоков свойства блоков и их отношения меняются со временем, и мы должны это учитывать. К счастью, мы можем сделать это, слегка изменив элементы нашего предыдущего словаря и добавив несколько дополнительных элементов.

Прежде всего мы добавим в наш язык символ `s1` для обозначения начального состояния мира. Можно было бы также добавить символы для других состояний мира; но, как мы увидим в следующем разделе, возможно ссылаться на эти другие состояния более удобным способом.

Во-вторых, мы вводим понятие флюента и соответствующее представление. *Флюент* – это условие, которое может менять истинностное значение от одного состояния к другому. В нашей формализации мы принимаем основные атомы нашего статического представления за флюенты. Однако теперь мы рассматриваем их как *элементы*, а не как фактоиды. Например, мы рассматриваем `on(a,b)` и `clear(a)` как элементы, а не фактоиды. Они больше не являются условиями, которые всегда истинны или ложны; теперь они являются элементами, которые истинны в одних состояниях и ложны в других.

Чтобы говорить об истинности флюентов в определенных состояниях, мы вводим бинарный предикат `tr` и используем его для обозначения того факта, что флюент истинен в определенном состоянии.

Например, мы можем представить исходное состояние мира, показанное на рис. 11.1, с помощью предложений, показанных ниже. Блок `c` находится на блоке `a` в состоянии `s1`; блок `a` находится на блоке `b`; и т. д.

```
tr(clear(c),s1)
tr(on(c,a),s1)
```

```
tr(on(a,b),s1)
tr(table(b),s1)
tr(clear(e),s1)
tr(on(e,d),s1)
tr(table(d),s1)
```

Чтобы говорить о действиях, которые могут изменить мир, мы вводим конструкторы, по одному для каждого действия. Например, в нашем мире блоков мы добавим два новых бинарных конструктора `u` и `s`. `u(X,Y)` представляет действие по снятию блока `X` с блока `Y`, а `s(X,Y)` – действие по установке блока `X` на блок `Y`.

Чтобы определить эффекты наших действий, мы добавляем бинарный конструктор `do`, чтобы говорить о результате выполнения данного действия в данном состоянии. Например, мы напишем `do(u(c,a),s1)` для обозначения состояния, которое возникает в результате выполнения действия `u(c,a)` в состоянии `s1`.

Чтобы передать физику мира, мы пишем правила, которые говорят, как мир изменяется в результате выполнения каждого из наших действий (см. ниже).

```
tr(table(X),do(u(X,Y),S)) :- tr(clear(X),S) & tr(on(X,Y),S)
tr(clear(Y),do(u(X,Y),S)) :- tr(clear(X),S) & tr(on(X,Y),S)

tr(on(X,Y),do(s(X,Y),S)) :- tr(clear(X),S) & tr(table(X),S) &
tr(clear(Y),S)
```

Обратите внимание, что, помимо описания *изменений*, нам также необходимо написать предложения, фиксирующие *инерцию* (то, что остается неизменным). Например, если мы снимаем один блок с другого, то все блоки, которые были `clear`, остаются `clear`; все блоки, лежавшие на столе, остаются на столе; и все блоки, стоявшие друг на друге, остаются стоять друг на друге, *за исключением* блоков, участвующих в операции снятия. Следующие предложения отражают инерционное поведение мира блоков.

```
tr(clear(U),do(u(X,Y),S)) :- tr(clear(U),S)
tr(table(U),do(u(X,Y),S)) :- tr(table(U),S)
tr(on(U,V),do(u(X,Y),S)) :- tr(on(U,V),S) & distinct(U,X)
tr(on(U,V),do(u(X,Y),S)) :- tr(on(U,V),S) & distinct(V,Y)

tr(clear(U),do(s(X,Y),S)) :- tr(clear(U),S) & distinct(U,Y)
tr(table(U),do(s(X,Y),S)) :- tr(table(U),S) & distinct(U,X)
tr(on(U,V),do(s(X,Y),S)) :- tr(on(U,V),S)
```

Подобные предложения, выражающие то, что *остается неизменным*, часто называют *аксиомами каркаса*. Многих раздражает необходимость

формализации того, что остается неизменным при представлении динамики. К счастью, существуют альтернативные способы формализации динамики, которые избавляют от необходимости использовать аксиомы каркаса. Например, вместо формализации того, что является *истинным* в состоянии, которое возникает в результате выполнения действия, мы можем формализовать то, что *изменяется* в текущем состоянии (в виде *добавления* и *удаления* списков). Подробнее об этой технике см. главу 14.

11.3. Моделирование

Моделирование – это процесс определения состояния, которое возникает в результате выполнения действия или последовательности действий в данном состоянии. Если у нас есть представление физики нашего мира, моделирование становится простым.

В качестве примера рассмотрим начальное состояние, описанное в предыдущем разделе, и следующую последовательность из двух действий. Сначала мы снимаем `c` с `a`, а затем устанавливаем `c` на `e`. Если бы нас интересовало состояние дел после выполнения этих действий, мы могли бы написать запрос, показанный ниже.

```
goal(P) :- tr(P,do(s(c,d),do(u(c,a),s1)))
```

Начальное состояние снова показано ниже.

```
tr(clear(c),s1)
tr(on(c,a),s1)
tr(on(a,b),s1)
tr(table(b),s1)
tr(clear(e),s1)
tr(on(e,d),s1)
tr(table(d),s1)
```

Используя эти данные, правила изменения и аксиомы каркаса, мы видим, что выполнение `u(c,a)` в этом состоянии приводит к данным, показанным ниже.

```
tr(clear(c),do(u(c,a),s1))
tr(table(c),do(u(c,a),s1))
tr(clear(a),do(u(c,a),s1))
tr(on(a,b),do(u(c,a),s1))
tr(table(b),do(u(c,a),s1))
tr(clear(e),do(u(c,a),s1))
tr(on(e,d),do(u(c,a),s1))
tr(table(d),do(u(c,a),s1))
```

Применяя правила изменения и правила каркаса еще раз, мы получаем следующие выводы.

```
tr(clear(c),do(s(c,e),do(u(c,a),s1)))
tr(on(c,e),do(s(c,e),do(u(c,a),s1)))
tr(clear(a),do(s(c,e),do(u(c,a),s1)))
tr(on(a,b),do(s(c,e),do(u(c,a),s1)))
tr(table(b),do(s(c,e),do(u(c,a),s1)))
tr(on(e,d),do(s(c,e),do(u(c,a),s1)))
tr(table(d),do(s(c,e),do(u(c,a),s1)))
```

Наконец, используя правило, определяющее предикат `goal`, мы получаем данные, показанные ниже.

```
goal(clear(c))
goal(on(c,e))
goal(clear(a))
goal(on(a,b))
goal(table(b))
goal(on(e,d))
goal(table(d))
```

Частичные результаты, показанные выше, делают этот процесс сложно выглядящим, но в действительности он довольно прост и не требует больших затрат. Поиск плана для достижения цели не так прост.

11.4. Планирование

Планирование в некотором смысле противоположно моделированию. При моделировании мы начинаем с *начального состояния* и *плана*, т. е. последовательности действий, и используем моделирование для определения результата выполнения плана в начальном состоянии. При планировании мы начинаем с начального состояния и *цели*, т. е. набора желательных состояний, и используем планирование для вычисления плана, который достигает одного из состояний цели.

В дальнейшем мы снова используем унарный предикат `goal`, только в этом случае определяем его как истинный для желательных состояний, а не для флюентов, как в предыдущем разделе. Например, следующее правило определяет, что цель истинна для состояния тогда и только тогда, когда флюенты `on(a,b)` и `on(b,c)` истинны в этом состоянии.

```
goal(S) :- tr(on(a,b),S) & tr(on(b,c),S)
```

Используя это определение цели и правила из раздела 11.4, легко убедиться, что следующее заключение истинно, т. е. `on(a,b)` и `on(b,c)` оба

истинны в состоянии `do(s(a,b),do(s(b,c),do(u(a,b),do(u(c,a),s1))))`, т. е. в состоянии, которое является результатом снятия `c`, снятия `a`, установки `b` на `c` и установки `a` на `b`.

```
goal(do(s(a,b),do(s(b,c),do(u(a,b),do(u(c,a),s1)))))
```

В принципе, мы должны быть в состоянии сделать этот вывод посредством восходящей или нисходящей оценки. К сожалению, при оценке снизу вверх исследуется множество планов, которые никак не связаны с целями. Нисходящая оценка остается сфокусированной на цели. Но с правилами, показанными выше, она, скорее всего, попадет в бесконечный цикл, исследуя все более и более длинные планы, хотя простой четырехшаговый план, показанный выше, работает.

Решением этой дилеммы является использование гибрида двух подходов. Мы определяем бинарный предикат `plan`, как показано ниже. `plan` истинен для пустого плана и состояния тогда и только тогда, когда это состояние удовлетворяет цели. Он истинен для непустой последовательности действий тогда и только тогда, когда «хвост» этой последовательности достигает цели при выполнении первого действия в данном состоянии.

```
plan(nil,S) :- goal(S)
plan(A!L,S) :- plan(L,do(A,S))
```

Учитывая это определение, мы можем сделать запрос `plan(L,s1)`, и нисходящая оценка будет пробовать короткие планы, чтобы увидеть, достигают ли они цели, и переходить к более длинным планам, только если ни один из них не достигнет цели.

Этот подход несколько быстрее, чем выполнение снизу вверх, и гарантирует, что будет получен самый короткий из возможных планов. Тем не менее этот метод все же является дорогостоящим. Для поиска способов более эффективного составления планов было проведено значительное количество исследований. Тем не менее можно показать, что иногда требуется полный поиск в пространстве планов.

11.5. Упражнения

11.1. Предположим, что мы хотим дополнить мир блоков действием `move(X,Y,Z)`, которое перемещает блок `X` с блока `Y` на блок `Z`. Напишите аксиомы изменения и аксиомы каркаса для этого нового действия в терминах словаря, введенного в этой главе.

11.2. Учитывая аксиомы изменения и каркаса, приведенные в этой главе, и данные, показанные ниже, оцените запрос `goal(P) :- tr(P,do(s(b,e),do(u(a,b),do(u(c,a),s1))))`.

```
tr(clear(c),s1)
tr(on(c,a),s1)
tr(on(a,b),s1)
tr(table(b),s1)
tr(clear(e),s1)
tr(on(e,d),s1)
tr(table(d),s1)
```

11.3. Приняв аксиомы изменения, аксиомы каркаса и определение цели из этой главы и данные, показанные ниже, дайте один ответ на запрос `result([X,Y]) :- plan([X,Y],s2)`.

```
tr(clear(a),s2)
tr(table(a),s2)
tr(clear(b),s2)
tr(on(b,c),s2)
tr(table(c),s2)
tr(clear(e),s2)
tr(on(e,d),s2)
tr(table(d),s2
```

Глава 12

Метазнания

12.1. Введение

Одной из интересных особенностей нашего языка является то, что он позволяет кодировать информацию об информации. Сложность заключается в том, чтобы представить *предложения* в виде *элементов* в нашем языке, а затем записать предложения об этих *элементах*, тем самым эффективно составляя предложения о предложениях. Существует множество примеров использования этой техники. В этой главе мы рассмотрим два из них – описание синтаксиса и семантики других языков в нашем языке и представление булевой логики в логическом программировании.

12.2. Обработка естественного языка

Псевдоанглийский – это формальный язык, который предназначен для приближения к синтаксису английского языка. Один из способов определить синтаксис псевдоанглийского языка – записать грамматические правила в форме Бэкуса–Наура (БНФ). Приведенные ниже правила иллюстрируют этот подход для небольшого подмножества псевдоанглийского языка. Предложение (`sentence`) – это `noun phrase` (`np`, фраза существительного), за которой следует `verb phrase` (`vp`, фраза глагола). Фраза существительного – это либо существительное, либо два существительных, разделенных словом `and`. Фраза глагола – это глагол, за которым следует фраза существительного. Существительное – это либо слово `mary`, либо слово `pat`, либо слово `quincy`. Глагол – это либо `like`, либо `likes`.

```
<sentence> ::= <np> <vp>
<np> ::= <noun>
<np> ::= <noun> "and" <noun>
<vp> ::= <verb> <np>
<noun> ::= "mary" | "pat" | "quincy"
<verb> ::= "like" | "likes"
```

В качестве альтернативы возможно использовать правила для формализации синтаксиса псевдоанглийского языка. Предложения, показанные ниже, выражают грамматику, описанную выше в правилах БНФ. Здесь мы используем отношение app, чтобы говорить о результате добавления слов.

```
sentence(Z) :- np(X) & vp(Y) & app(X,Y,Z)
np(X) :- noun(X)
np(W) :- noun(X) & noun(Y) & app(X,and,Z) & app(Z,Y,W)
vp(Z) :- verb(X) & np(Y) & app(X,Y,Z)
noun(mary)
noun(pat)
noun(quincy)
verb(like)
verb(likes)
```

Используя эти правила, мы можем проверить, является ли данная последовательность слов синтаксически законным предложением в псевдоанглийском языке, и перечислить синтаксически законные предложения, как показано ниже.

```
mary likes pat
pat and quincy like mary
mary likes pat and quincy
```

Слабость наших БНФ и соответствующей аксиоматизации заключается в том, что в них не учитывается согласование по числу между подлежащими и глаголами. Следовательно, используя эти правила, мы можем получить следующие выражения, которые являются безграмотными в естественном английском языке.

```
mary like pat
pat and quincy likes mary
```

К счастью, можно решить эту проблему, немного усовершенствовав наши правила. В частности, мы можем добавить аргумент к некоторым отношениям, чтобы указать, имеет ли выражение единственное или множественное число, и затем использовать его для блокировки последовательностей слов, в которых числа не совпадают.

```
sentence(Z) :- np(X,W) & vp(Y,W) & app(X,Y,Z)
np(X,singular) :- noun(X)
np(W,plural) :- noun(X) & noun(Y) & app(X,and,Z) & app(Z,Y,W)
vp(Z,W) :- verb(X,W) & np(Y,V) & app(X,Y,Z)
noun(mary)
noun(pat)
noun(quincy)
```

```
verb(like,plural)
verb(likes,singular)
```

С помощью этих правил синтаксически корректные предложения, показанные выше, по-прежнему гарантированно являются предложениями, но неграмотные последовательности блокируются. Другие грамматические характеристики могут быть формализованы аналогичным образом, например согласование родов в местоимениях (он или она), притяжательных прилагательных (его или ее), рефлексивных прилагательных (его или ее) и т. д.

12.3. Булева логика

Везде в этой книге мы использовали английский язык, чтобы говорить о логическом программировании. Естественно задать вопрос: можно ли формализовать логическое программирование в рамках логического программирования? Ответ – да, но есть некоторые ограничения на то, что можно сделать.

В этом разделе мы рассмотрим простую версию этой проблемы, а именно использование логического программирования для формализации синтаксиса и семантики булевой логики. Предложения в булевой логике проще, чем в логическом программировании. Словарь состоит из атомарных *предложений*, а предложения являются либо предложениями, либо сложными выражениями, образованными из *логических операторов*. Показанное ниже предложение является примером. Это утверждение о том, что либо `p` истинно и `q` ложно, либо `p` ложно и `q` истинно.

```
(p ∧ ¬q) ∨ (¬p ∨ q)
```

В дальнейшем мы будем ассоциировать символ с каждым предложением в нашем языке булевой логики. Например, мы используем `p`, `q` и `r` для обозначения предложений *p*, *q* и *r*.

Далее мы вводим конструкторы для формирования сложных предложений. Существует конструктор для каждого логического оператора – `not` для ¬, `and` для ∧, и `or` для ∨. Используя эти конструкторы, мы можем представлять предложения булевой логики как элементы нашего языка. Например, мы можем представить приведенное выше предложение следующим образом.

```
or(and(and(p,not(q)),and(not(p),q)))
```

Наконец, мы вводим несколько предикатов для выражения типов различных выражений в нашем языке булевой логики. Мы используем унарный предикат `proposition`, чтобы указать, что выражение является утверждением; `negation` – чтобы указать, что выражение является отри-

цанием; `conjunction` – выражение является конъюнкцией; `disjunction` – выражение является дизъюнкцией; и `sentence`, чтобы указать, что перед нами предложение.

С помощью этого словаря мы можем охарактеризовать синтаксис нашего языка следующим образом. Мы начинаем с объявления констант предложений.

```
proposition(p)
proposition(q)
proposition(r)
```

Далее определим типы выражений, в которых используются логические операторы.

```
negation(not(X)) :- sentence(X)
conjunction(and(X,Y)) :- sentence(X) & sentence(Y)
disjunction(or(X,Y)) :- sentence(X) & sentence(Y)
```

Наконец, мы определяем предложения как выражения этих типов.

```
sentence(X) :- proposition(X)
sentence(X) :- negation(X)
sentence(X) :- conjunction(X)
sentence(X) :- disjunction(X)
```

Задание истинности – это отображение констант предложения на булевы значения (`true` или `false`). Мы можем закодировать задание истинности с помощью значения бинарного отношения `value`, которое связывает константу предложения с соответствующим значением. Например, следующие факты представляют собой задание истинности для приведенных выше констант предложения. В этом случае `p` – истинно, `q` – ложно, а `r` – истинно.

```
value(p,true)
value(q,false)
value(r,true))
```

Имея задание истинности, мы можем определить истинностное значение для каждого предложения в нашем языке. Предложение истинно тогда и только тогда, когда ему присвоено значение `true`. Отрицание истинно тогда и только тогда, когда его аргумент ложен. Конъюнкция истинна тогда и только тогда, когда оба конъюнкта истинны. Дизъюнкция истинна тогда и только тогда, когда хотя бы один из ее дизъюнктов истинен.

```
truth(P) :- value(P,true)
truth(not(P)) :- ~truth(P)
```

```
truth(and(P,Q)) :- truth(P) & truth(Q)
truth(or(P,Q)) :- truth(P)
truth(or(P,Q)) :- truth(Q)
```

Мы можем сделать нашу формализацию более интересной, переформулировав задания истинности в объекты. Тогда мы сможем говорить о таких свойствах предложений, как действительность и удовлетворительность. Предложение действительно тогда и только тогда, когда оно истинно в каждом задании истинности. Предложение является удовлетворительным тогда и только тогда, когда некоторое задание истинности удовлетворяет ему. Предложение фальсифицируемо тогда и только тогда, когда некоторое задание истинности делает его ложным. Предложение является неудовлетворительным тогда и только тогда, когда ни одно задание истинности не делает его истинным.

12.4. Упражнения

12.1. Предположим, что нам нужно добавить слова `himself` и `herself` к грамматике псевдоанглийского языка. Измените правила, определяющие нашу псевдоанглийскую грамматику так, чтобы эти слова появлялись только в качестве объектов предложений и чтобы, когда одно из этих слов использовалось в предложении, его число и род соответствовали числу и роду подлежащего этого предложения.

12.2. Скажите, является ли каждое из следующих предложений следствием предложений из раздела «Булева логика».

(a) `conjunction(and(not(p),not(q)))`
(b) `conjunction(not(or(not(p),not(q))))`
(c) `sentence(not(not(p)))`
(d) `sentence(or(not(p),not(q),not(r)))`
(e) `sentence(and(p,not(p))`

12.3. Какие из следующих предложений являются следствиями задания истинности и правил раздела «Булева логика»?

(a) `truth(or(not(p),not(q)))`
(b) `truth(not(and(not(p),not(q))))`
(c) `truth(and(p,not(p))`

12.4. Предположим, что мы хотим добавить оператор `xor` в наш язык булевой логики. `xor(p,q)` истинен тогда и только тогда, когда оценка `p` отличается от оценки `q`. Напишите правило для расширения нашего определения истины с учетом оператора `xor`.

Часть IV
Определения операций

Глава 13

Операции

13.1. Введение

В предыдущей части (главы 7–12) мы рассмотрели, как писать правила для определения *отношений вида* в терминах базовых отношений. После этого определения мы можем использовать такие представления в запросах и определениях других представлений.

В этой части (главы 13–16) мы рассмотрим, как писать правила, определяющие *операции* в терминах изменений базовых отношений. После этого определения мы можем использовать эти операции в обновлениях и определениях других операций.

Как мы видели, правила, используемые при написании определений вида, обобщают правила, используемые при написании запросов; и, как мы увидим, правила, используемые при написании определений операций, обобщают правила, используемые при написании обновлений. При этом важно помнить о различиях между представлениями и операциями – представления используются для обсуждения *фактов*, которые верны в состояниях, в то время как операции используются для обсуждения *изменений* в состояниях.

В этой главе мы определяем синтаксис и семантику определений операций. В следующей главе мы увидим, как определения операций могут использоваться для определения обработки событий в динамических системах (когда поведение системы меняется в ответ на внешние воздействия). В главе 15 мы рассмотрим, как использовать определения операций в управлении базами данных, а в главе 16 – как использовать определения операций при создании интерактивных рабочих листов.

13.2. Синтаксис

Синтаксис определений операций основывается на синтаксисе обновлений, описанном в главе 4. Различные типы констант, понятия элемента, атома и литерала там и здесь одинаковы. Однако к ним мы добавляем несколько новых элементов.

Для обозначения операций мы обозначаем некоторые константы как *операционные*. Как и в случае с конструкторами и константами отношений, каждая операционная константа имеет фиксированную арность – унарную, бинарную и т. д.

Действие – это применение операции к определенным объектам. В дальнейшем мы обозначаем действия, используя синтаксис, аналогичный синтаксису атомарных предложений, а именно n-арную операционную константу, за которой следуют n элементов, заключенных в круглые скобки и разделенных запятыми. Например, если `f` – это бинарная операционная константа, а `a` и `b` – символы, то `f(a,b)` обозначает действие применения операции `f` к `a` и `b`.

Правило определения операции (или, проще говоря, *правило операции*) – это выражение вида, показанного ниже. Каждое правило состоит из (1) выражения действия, (2) двойного двоеточия, (3) литерала или конъюнкции литералов, (4) двойной стрелки вправо, и (5) литерала или выражения действия или конъюнкции литералов и выражений действия. Выражение действия, расположенное слева от двойного двоеточия, называется *головой*, литералы слева от стрелки называются *условиями*, а литералы справа от нее – *эффектами*.

$$\gamma \text{ :: } [\sim]\varphi_1 \text{ \&...\& } [\sim]\varphi_m \text{ ==> } [\sim]\psi_1 \text{ \&...\& } [\sim]\psi_n \text{ \& } \gamma_1 \text{ \&...\& } \gamma_k$$

Смысл правила операции прост. Если условия правила истинны в любом состоянии, то выполнение действия в голове требует выполнения эффектов правила.

Например, следующее правило гласит, что в любом состоянии, в котором `p(a,b)` истинно, а `q(a)` ложно, выполнение `click(a)` требует, чтобы мы удалили `p(a,b)` из нашего набора данных, добавили `q(a)` и выполнили действие `click(b)`.

```
click(a) :: p(a,b) & ~q(a) ==> ~p(a,b) & q(a) & click(b)
```

Как и правила, определяющие представления, правила операций могут содержать переменные, чтобы выразить информацию в компактной форме. Например, мы можем написать следующее правило, чтобы обобщить предыдущее правило на все объекты.

```
click(X) :: p(X,Y) & ~q(X) ==> ~p(X,Y) & q(X) & click(Y)
```

Как и в случае с правилами вида, необходимо учитывать *безопасность*. Безопасность в данном случае означает, что каждая переменная среди эффектов правила или в негативных условиях также появляется в голове правила или в позитивных условиях.

Оба правила операции, показанные выше, являются безопасными. Однако правила, показанные ниже, не являются таковыми. Второй эф-

фект первого правила содержит переменную, которая не появляется в голове или в каком-либо позитивном условии. Во втором правиле есть переменная, которая появляется в негативном условии и не появляется ни в голове, ни в каком-либо позитивном условии.

```
click(X) :: p(X,Y) & ~q(X) ==> ~p(X,Y) & q(Z) & click(Y)
click(X) :: p(X,Y) & ~q(Z) ==> ~p(X,Y) & q(X) & click(Y)
```

В некоторых правилах операций нет условий, т. е. эффекты правила перехода имеют место во всех наборах данных. Конечно, мы можем написать такие правила, используя условие `true`, как в следующем примере.

```
click(X) :: true ==> ~p(X) & q(X)
```

Для простоты изложения наших примеров мы иногда сокращаем такие правила, опустив условия и оператор перехода, и вместо этого пишем только эффекты перехода как тело правила операции. Например, можем сократить приведенное выше правило следующим образом.

```
click(X) :: ~p(X) & q(X)
```

Определение операции – это набор правил операций, в которых одна и та же операция появляется в заголовке каждого правила. Как и в случае с определениями вида, нас интересуют в первую очередь наборы правил, которые являются конечными. Однако, анализируя определения операций, мы иногда говорим о множестве всех основных экземпляров правил, и в некоторых случаях эти множества бесконечны.

13.3. Семантика

Семантика определений операций сложнее, чем семантика обновлений, из-за возможного появления представлений в условиях правила и возможного появления операций в его эффектах. Далее мы сначала определим расширение действия в контексте заданного набора данных, а затем определим результат выполнения этого действия над этим набором данных.

Предположим, что нам дан набор правил Ω, набор действий Γ (фактоиды, негативные фактоиды и действия) и набор данных Δ. Мы говорим, что *экземпляр* правила в Ω является *активным* по отношению к Γ и Δ тогда и только тогда, когда голова правила находится в Γ и все условия правила истинны в Δ.

Учитывая это понятие, мы определяем *расширение* действия γ по отношению к набору правил Ω и набору данных Δ следующим образом. Пусть Γ_0 будет $\{\gamma\}$, и пусть Γ_{i+1} будет множеством всех эффектов в любом

экземпляре любого правила в Ω относительно Γ_i и Δ. Мы определяем наше расширение U(γ, Ω, Δ) как точку фиксации этой серии. Это эквивалентно объединению множеств Γ_i для всех неотрицательных целых чисел *i*.

Далее, определяем позитивные обновления A(γ, Ω, Δ) как позитивные базовые фактоиды в U(γ, Ω, Δ). Мы определяем негативные обновления D(γ, Ω, Δ) как множество всех негативных фактоидов в U(γ, Ω, Δ).

Наконец, мы определяем результат применения действия γ к набору данных Δ, как результат удаления из Δ негативных обновлений и добавления позитивных обновлений, т. е. результатом является $(\Delta - D(\gamma, \Omega, \Delta)) \cap A(\gamma, \Omega, \Delta)$.

Чтобы проиллюстрировать эти определения, рассмотрим приложение с набором данных, представляющим собой направленный ациклический граф. В приведенных ниже предложениях мы используем символы для обозначения узлов графа, а для обозначения дуг графа используем отношение `edge`.

```
edge(a,b)
edge(b,d)
edge(b,e)
```

Следующее определение операции создает тернарную операцию `copy`, которая копирует исходящие дуги первого аргумента на второй.

```
copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)
```

Учитывая это определение операции и набор данных, показанный выше, расширение `copy(b,c)` состоит из изменений, показанных ниже. В данном случае фактоиды, представляющие две дуги, исходящие из `b`, копируются в `c`.

```
edge(c,d)
edge(c,e)
```

После выполнения этого события мы получим следующий набор данных.

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

Следующее правило определяет унарную операцию `invert`, которая инвертирует входящие дуги узла, указанного в качестве ее аргумента.

```
invert(Y) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)
```

Расширение функции `invert(c)` показано ниже. В этом случае аргументы в фактоиде с `c` в качестве второго аргумента были изменены на противоположные.

```
~edge(c,d)
~edge(c,e)
edge(d,c)
edge(e,c)
```

После выполнения этого события мы получим следующий набор данных.

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

Наконец, следующие правила операций определяют бинарную операцию, которая вставляет новый узел в граф (первый аргумент) с дугой ко второму аргументу и дугами ко всем узлам, достижимым из второго аргумента.

```
insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
```

Расширение `insert(w,b)` показано ниже. Первое правило добавляет `edge(w,b)` к расширению. Второе правило добавляет `insert(w,d)` и `insert(w,e)`. Учитывая эти события, на следующем раунде расширения первое правило добавляет `edge(w,d)` и `edge(w,e)`, а второе правило добавляет `insert(w,c)`. На третьем раунде расширения мы получаем `edge(w,c)`. С этого момента ни одно из правил не добавляет никаких дополнительных элементов к нашему расширению, и процесс завершается.

```
insert(w,b)
edge(w,b)
insert(w,d)
insert(w,e)
edge(w,d)
edge(w,e)
insert(w,c)
edge(w,c)
```

Применение этого события к предыдущему набору данных дает результат, показанный ниже.

```
edge(a,b)
edge(b,d)
edge(b,e)
edge(d,c)
edge(e,c)
edge(w,b)
edge(w,d)
edge(w,e)
edge(w,c)
```

Обратите внимание, что вставку можно определить и другим способом. Например, мы можем определить представление `edge`, которое связывает каждый узел с каждым узлом, который может быть достигнут из этого узла, затем использовать это представление в нерекурсивном определении `insert`. Однако это потребовало бы введения нового представления в наш словарь, что менее понятно для многих, чем определение, показанное выше.

13.4. Упражнения

13.1. Для каждой из следующих строк скажите, является ли она синтаксически законным определением операции.

(a) `a(X) :: p(X,Y) ==> q(Y,X)`
(b) `a(X) :: p(X,Y) & a(Y) ==> q(Y,X)`
(c) `a(X) :: p(X,Y) ==> q(Y,X) & a(Y)`
(d) `a(X) :: p(X,Y) ==> q(Y,X) & ~a(Y)`
(e) `a(X) :: p(Y,Y) ==> q(X,Y)`

13.2. Скажите, является ли безопасным каждый из следующих запросов.

(a) `a(X) :: p(X,Y) & p(Y,Z) ==> p(X,Z)`
(b) `a(X) :: p(X,Y) & ~p(Y,Z) ==> p(X,Z)`
(c) `a(X) :: p(Y,Z) ==> p(X,Z)`
(d) `a(X) :: p(Y,Z) ==> ~p(X,Z)`
(e) `a(X) :: p(Y,Z) ==> p(Z,Y)`

13.3. Учитывая определение `fix(X) :- p(X,Y) & p(Y,Z) ==> p(X,Z)`, какой результат будет получен после выполнения действия `fix(a)` на наборе данных, показанном ниже?

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

13.4. Учитывая определение `fix(X) :- p(X,Y) & p(Y,Z) ==> p(X,Z) & fix(Y)`, каков будет результат выполнения действия `fix(a)` на наборе данных, показанном ниже?

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

13.5. Рассмотрим иерархию типов, подобную той, что показана ниже.

```
subtype(giraffe,mammal)
subtype(rabbit,mammal)
subtype(mammal,vertebrate)
subtype(earthworm,vertebrate)
subtype(vertebrate,animal)
subtype(invertebrate,animal)
```

Определите операцию `classify`, которая принимает объект и тип в качестве аргументов и добавляет фактоиды, утверждающие, что объект имеет данный тип и все супертипы этого типа. Например, выполнение действия `classify(george,giraffe)` должно привести к тому, что в набор данных будут добавлены следующие фактоиды.

```
type(george,giraffe)
type(george,mammal)
type(george,vertebrate)
type(george,animal))
```

Глава 14

Динамические логические программы

14.1. Введение

В главе 11 мы разобрали, как использовать определения вида для описания поведения динамических систем. В этой главе мы рассмотрим альтернативный подход с использованием определений операций. Когда системы определены таким образом, они называются *динамическими логическими программами*.

В следующем разделе мы рассмотрим пример реактивной системы, т. е. системы, которая реагирует на внешние воздействия. Затем изучим пример закрытой динамической системы, т. е. системы, которая работает без внешнего воздействия. После этого – пример смешанной инициативной системы, т. е. системы, которая управляется комбинацией внутренних и внешних воздействий. И наконец, разберем вопросы, связанные с обработкой одновременных входных сигналов.

14.2. Реактивные системы

Простейшая форма реактивной системы – это система, поведение которой полностью определяется внешними входами. До поступления входного сигнала система находится в состоянии покоя, т. е. в ней ничего не меняется; после поступления входного сигнала система изменяет состояние в соответствии с входным сигналом; и после этого система снова становится спокойной (до появления следующего входного сигнала).

В качестве примера рассмотрим систему с тремя кнопками и тремя лампочками. В каждый момент времени некоторые из лампочек включены, а некоторые выключены. Если пользователь нажимает на первую кнопку, система переключает состояние первой лампочки. Если на вторую кнопку – система циклически меняет состояния первой и второй

лампочек (т. е. первая переключается в состояние второй, а вторая – в состояние первой). Если пользователь нажимает третью кнопку, система циклически меняет состояния второй и третьей лампочек.

Чтобы формализовать поведение этой системы, мы даем имена компонентам состояния. Мы используем булев предикат `p` для обозначения того, что первая лампочка включена, `q` означает, что вторая лампочка включена, и `r` означает, что третья лампочка включена. Далее даем имена трем возможным событиям. Мы используем булев предикат `a` для обозначения нажатия первой кнопки, `b` обозначает нажатие второй кнопки и `c` нажатие третьей.

Исходя из этого словаря, мы представляем состояние нашей системы как набор данных, состоящий из некоторого подмножества фактоидов `p`, `q` и `r`, а возникновение события – как одно из трех действий, т. е. `a`, `b` или `c`.

Используя эту новую терминологию, мы можем описать поведение системы с помощью показанных ниже определений операций. Если пользователь нажимает кнопку `a` и значение `p` истинно, система делает `p` ложным. Если пользователь нажимает кнопку `a`, а `p` ложно, система делает `p` истинным. Если пользователь нажимает кнопку `b`, система циклически меняет местами `p` и `q`. А если пользователь нажимает `c`, система циклически меняет местами `q` и `r`. (Обратите внимание, что, если выполняется действие `b`, а `p` и `q` одинаковы, ничего не меняется. Аналогично, если выполняется действие `c`, а `q` и `r` одинаковы, ничего не меняется. Следовательно, нам не нужны правила для этих случаев.)

```
a :: p ==> ~p
a :: ~p ==> p
b :: p & ~q ==> ~p & q
b :: ~p & q ==> p & ~q
c :: q & ~r ==> ~q & r
c :: ~q & r ==> q & ~r
```

Заметим, что, если система находится в состоянии, в котором все три условия ложны, она может достичь состояния, в котором они истинны, выполнив последовательность действий `a`, `b`, `c`, `a`, `b` и `a`. Можете ли вы придумать другую последовательность действий, которая бы справилась с этой задачей? Сколько существует последовательностей, которые приводят к желаемому состоянию?

14.3. Замкнутые системы

Замкнутая динамическая система – это система, которая работает без внешнего воздействия. Ее поведение является результатом только внутренних воздействий, таких как тиканье часов.

В качестве примера представьте себе вариацию мира кнопок и лампочек, описанного в предыдущем разделе. В этом случае нет кнопок и, следовательно, нет внешнего сигнала. Вместо этого система циклически проходит состояния, начиная с состояния, когда все лампочки выключены, проходя состояния в определенном порядке до тех пор, пока все лампочки не будут включены, сбрасывается, а затем процесс повторяется до бесконечности.

При описании поведения этой системы мы используем тот же словарь, что и в предыдущем разделе, за исключением того, что вместо действий `a`, `b` и `c` у нас есть одно внутреннее действие `tick`, представляющее тиканье часов.

Используя эту терминологию, мы можем описать поведение системы с помощью определения операции, показанного ниже. Когда часы тикают, система изменяет состояние в соответствии с текущим состоянием на данный момент. Когда все лампочки выключены, включается первая лампочка. Когда включена только первая лампочка, она гаснет и включается вторая лампочка и т. д.

```
tick :: ~p & ~q & ~r ==> p & ~q & ~r
tick :: p & ~q & ~r ==> ~p & q & ~r
tick :: ~p & q & ~r ==> ~p & ~q & r
tick :: ~p & ~q & r ==> p & ~q & r
tick :: p & ~q & r ==> ~p & q & r
tick :: ~p & q & r ==> p & q & r
tick :: p & q & r ==> ~p & ~q & ~r
```

Обратите внимание, что эта последовательность состояний совпадает с последовательностью, которая возникнет в результате выполнения последовательности действий, упомянутых в конце предыдущего раздела, а именно: `a`, `b`, `c`, `a`, `b` и `a`.

При желании мы могли бы также формализовать это поведение, используя действия, определенные в предыдущих разделах (за исключением того, что действия были бы внутренними действиями, а не внешними воздействиями).

Как и раньше, у нас будут определения действий, но к ним мы добавим операцию `reset`, чтобы выключить все лампочки (см. ниже).

```
a :: p ==> ~p
a :: ~p ==> p
b :: p & ~q ==> ~p & q
b :: ~p & q ==> p & ~q
c :: q & ~r ==> ~q & r
c :: ~q & r ==> q & ~r
reset :: ~p & ~q & ~r
```

Учитывая эти определения, мы можем переписать приведенное выше описание так, как показано ниже. Правила имеют те же условия, но, вместо перечисления изменений базовых отношений, правила в данном случае определяют, какое внутреннее действие выполняется в том или ином состоянии.

```
tick :: ~p & ~q & ~r ==> a
tick :: p & ~q & ~r ==> b
tick :: ~p & q & ~r ==> c
tick :: ~p & ~q & r ==> a
tick :: p & ~q & r ==> b
tick :: ~p & q & r ==> a
tick :: p & q & r ==> reset
```

Этот стиль описания иногда называют *универсальным планом*. Для каждого состояния он определяет действие, которое должно быть выполнено в этом состоянии.

14.4. Система со смешанной инициативой

Система со смешанной инициативой – это система, поведение которой определяется либо внешними, либо внутренними воздействиями. Интересно, что в системе со смешанной инициативой один внешний сигнал может привести к единственному изменению состояния или к последовательности изменений.

В качестве примера рассмотрим вариант замкнутой системы кнопок и лампочек, описанной в предыдущем разделе. В этом варианте вместо кнопок, описанных в разделе 14.2, есть две разные кнопки. Если пользователь нажимает на первую кнопку, система начинает вести себя так, как как описано в предыдущем разделе. При нажатии на вторую кнопку система приостанавливает свою работу. Если пользователь снова нажимает первую кнопку, система продолжает работу с того места, на котором остановилась.

Как и раньше, мы используем `p`, `q` и `r` для описания состояния трех лампочек. К ним мы добавляем один 0-арный предикат `running` для описания состояния процесса. Мы используем символы `play` и `stop` для обозначения двух внешних сигналов. Наконец, как и раньше, мы используем символ `tick` для обозначения тиканья внутренних часов.

Приведенные ниже правила операций определяют желаемое поведение нашей системы. Это определения `play` и `stop`, а также обычные правила, определяющие `tick`, единственным отличием является зависимость от `running`.

```
play :: running
stop :: ~running
```

```
tick :: running & ~p & ~q & ~r ==> p
tick :: running & p & ~q & ~r ==> ~p & q
tick :: running & ~p & q & ~r ==> c
tick :: running & ~p & ~q & r ==> a
tick :: running & p & ~q & r ==> b
tick :: running & ~p & q & r ==> a
tick :: running & p & q & r ==> reset
tick :: running & p & q & r ==> reset
```

Учитывая эти правила, система будет демонстрировать желаемое поведение. Когда пользователь нажимает кнопку `play`, она включает запущенный фактоид. В итоге, по мере того как внутренние часы системы тикают, она проходит через цикл состояний. Когда пользователь нажимает кнопку `stop`, система ничего не будет делать в последующие такты часов, пока кнопка `play` не будет нажата еще раз.

14.5. Одновременные действия

В предыдущих разделах мы рассмотрели проблему обработки одного входного сигнала за один раз. В некоторых приложениях мы должны иметь дело с возможностью одновременного поступления нескольких входных сигналов. Например, роботу может быть дана команда одновременно двигаться и вытягивать руку, или пользователь компьютера может нажать две клавиши одновременно.

В некоторых случаях эффекты таких событий не зависят друг от друга. Тогда можно обрабатывать эти события независимо друг от друга, рассматривая их так, как будто они произошли независимо друг от друга.

Операции `a` и `c` в оригинальной версии мира кнопок и лампочек (см. раздел 14.2) иллюстрируют это. Поведение, связанное с этими входными сигналами, является независимым. Нажатие кнопки `a` переключает `p` и не имеет никакого отношения к `q` и `r`. Нажатие кнопки `c` меняет местами `q` и `r` и не имеет никакого отношения к `p`. Поэтому эти входы могут обрабатываться независимо и одновременно в соответствии с правилами, определяющими их индивидуальное поведение.

К сожалению, рассмотрение одновременных входов независимо друг от друга возможно не всегда. В некоторых случаях одновременные действия могут взаимодействовать таким образом, что приводят к результатам, которые отличаются от результатов их независимого выполнения.

В качестве примера рассмотрим состояние, в котором первая и вторая лампочки в мире кнопок и лампочек (см. раздел 14.2) горят, и представьте, что пользователь одновременно нажимает и первую, и вторую кнопки, т. е. он одновременно выполняет действия `a` и `b`. В этой ситуации определение действия `a` предписывает, что `p` должно стать ложным,

а определение действия b предписывает, что p должно стать истинным. Что же происходит?

Проблема в этом случае заключается в том, что наши определения операций, как они написаны, предполагают, что только одно действие происходит одновременно. Чтобы справиться с возможными взаимодействиями, нам нужно описать эффекты одновременных входов.

Один из способов сделать это – создать терминологию для обсуждения *составных действий*, а затем написать определения операций для таких комбинаций.

Если количество возможных действий невелико, то для определения различных комбинаций входов принято использовать *аккорды*. Например, в мире кнопок и лампочек мы могли бы использовать тернарный конструктор press и указать булевы значения в качестве аргументов. Если первый аргумент истинен, это означает, что кнопка a нажата. Если второй аргумент истинен – кнопка b нажата. Если третий аргумент истинен – нажата кнопка c. Задавая различные комбинации булевых значений, мы можем характеризовать различные комбинации действий.

Если количество возможных действий велико, то представление составных действий в виде аккордов является непрактичным. В таких случаях принято представлять составные действия в виде *списков действий*. Например, в мире кнопок и лампочек мы бы придумали представлять сочетание действий a и c списком [a,c] и могли бы указать выполнение этого составного действия, написав выражение типа execute([a,c]).

Имея описание составных действий, мы можем написать определения операций, используя это описание. Например, правила, показанные ниже, определяют один из возможных вариантов поведения для системы «мир кнопок и лампочек», которая допускает до двух одновременных действий. Если пользователь нажимает одновременно a и b, то a обычным образом воздействует на p, а b обычным образом воздействует на q. Если пользователь нажимает одновременно a и c, то эти действия имеют свои обычные эффекты (как определено в разделе 14.2). Если пользователь нажимает одновременно b и c, то b оказывает свое обычное влияние на q, а c – на r.

```
execute([a,b]) :: ~p ==> p & ~q
execute([a,b]) :: p ==> ~p & q
execute([a,c]) :: a
execute([a,c]) :: c
execute([b,c]) :: q & ~r ==> ~q & r
execute([b,c]) :: ~q & r ==> q & ~r
```

Обратите внимание, что если все действия в приложении независимы друг от друга, то определить поведение составных действий очень просто. Предположим, например, что у нас есть система с выбором от-

дельных действий `execute(a)`, `execute(b)` и т. д., и все эти действия независимы. Тогда мы могли бы определить поведение системы в ответ на произвольное подмножество этих действий с помощью правила, показанного ниже (вместе с правилами, определяющими отдельные действия).

```
execute(L) :: member(A,L) ==> execute(A)
```

Формализация поведения одновременных входов может быть утомительной. Однако, используя определения операций для составных действий, по крайней мере, можно правильно формализовать такое поведение в ситуациях, где независимая обработка была бы невозможна.

14.6. Упражнения

14.1. Рассмотрим мир кнопок и лампочек, описанный в этой главе, но предположим, что есть четвертая кнопка `d`, которая переключает все лампочки одновременно. Напишите определение операции для `d`.

14.2. Перепишите определения операций для мира кнопок и лампочек, позволяющих одновременное выполнение трех действий `a`, `b` и `c`. Предположим, что эти действия имеют свои обычные эффекты, когда выполняются независимо друг от друга, а когда все три кнопки нажимаются одновременно, все три лампочки выключаются.

Глава 15

Управление базами данных

15.1. Введение

В предыдущей главе мы рассмотрели процесс обновления наборов данных в ответ на входные сигналы. В этой главе мы рассмотрим особый случай этого более общего процесса, в котором входами являются добавление и удаление конкретных фактов.

Чтобы определить такие входы, мы используем две новые операции, `add` и `delete`. `add` принимает в качестве аргумента основной атом и формирует запрос на добавление указанного атома в текущий набор данных. `delete` принимает основной атом в качестве аргумента и формирует запрос на удаление указанного атома из текущего набора данных.

Обратите внимание, что система может выполнять или не выполнять полученные запросы. Она также может добавлять или удалять факты, не указанные во входных данных. Правила операций позволяют программисту точно определить, как именно должна изменяться база данных в ответ на входные данные.

В этой главе мы сначала рассмотрим использование определений операций для обеспечения того, чтобы ограничения набора данных продолжали действовать после обновления. В следующем разделе поговорим об использовании определений операций для обновления материализованных представлений. Затем мы изучим использование определений операций для определения обновлений базовых отношений при получении запросов на обновление отношений вида.

15.2. Обновление с ограничениями

Рассмотрим систему базы данных с ограничениями и предположим, что база данных получает запрос на обновление, который, если его выполнить буквально, приведет к набору данных, нарушающему ограничения.

Одним из вариантов действий в этой ситуации является простое отклонение запроса, возможно, с указанием обнаруженных проблем

(как предлагается в разделе об оповещении в блоке устранения несоответствий).

В качестве альтернативы система может *исправить* набор обновлений, пополнив его дополнениями и удалениями таким образом, чтобы результирующий набор обновлений привел к набору данных, который отражает запрошенные обновления *и* удовлетворяет всем ограничениям.

Плохая новость заключается в том, что не существует метода исправления, который был бы удовлетворительным для всех приложений. Хорошая новость в том, что, используя определения операций, администратор может указать, как производить исправления в таких ситуациях.

В качестве примера действия этого механизма рассмотрим правила, показанные ниже. Первое предписывает системе удалять предложение вида `male(X)` всякий раз, когда пользователь добавляет предложение вида `female(X)` (в дополнение к добавлению `male(X)`). Второе правило аналогично первому, но в нем `male` и `female` поменялись местами. Вместе эти два правила обеспечивают взаимное исключение для `male` и `female`.

```
add(female(X)) :: female(X) & ~male(X)
add(male(X)) :: male(X) & ~female(X)
```

Аналогично мы можем ввести «зависимость включения» для родителя и взрослого, написав следующее правило. Если пользователь просит систему добавить предложение вида `parent(X,Y)`, то система также добавит предложение вида `adult(X)`.

```
add(parent(X,Y)) :: adult(X)
```

Обратите внимание, что не все ограничения могут быть реализованы с помощью правил обновления. Например, если пользователь предлагает добавить предложение `parent(art,art)` в базу данных в примере с родством, то система ничего не сможет сделать, чтобы исправить эту ошибку. И если она хочет сохранить непротиворечивость, она должна отклонить обновление.

Отдельная проблема возникает, когда существует несколько возможных вариантов исправления и ни один из них не кажется лучше других. Например, у нас может быть ограничение, говорящее, что каждый человек – либо мужчина, либо женщина. Если пользователь указывает факт, включающий нового человека, но не указывает его пол, у системы не будет возможности самостоятельно определить пол. В таких случаях система может отклонить запрос, собрать дополнительную информацию или сделать произвольный выбор и позволить пользователю изменить набор данных с помощью дополнительных обновлений.

15.3. Ведение материализованных представлений

Материализованное представление – это определенное отношение, которое хранится в базе данных в явном виде. Преимущество материализованных представлений в том, что система может просто искать ответы на вопросы о материализованном представлении, а не вычислять ответы из других хранимых отношений. Недостаток заключается в том, что необходимо поддерживать такие отношения в актуальном состоянии по мере внесения изменений в базовые отношения. Определения операций могут помочь в этом, обеспечивая автоматическое обновление материализованных представлений.

Предположим, например, что есть база данных с базовыми отношениями `parent` и `male`, и необходимо определить `father` как представление `parent` и `male`. И предположим, что мы должны материализовать отношение `father`. Тогда мы могли бы написать следующие правила обновления для поддержания этого материализованного представления. Согласно первому правилу система должна добавлять предложение вида `father(X,Y)` всякий раз, когда пользователь добавляет `parent(X,Y)` и известно, что `male(X)` истинно. Остальные правила охватывают все остальные случаи.

```
add(parent(X,Y)) :: male(X) ==> father(X,Y)
add(male(X)) :: parent(X,Y) ==> father(X,Y)
delete(parent(X,Y)) :: ~father(X,Y)
delete(male(X)) :: ~father(X,Y)
```

Этот подход работает, но написание таких правил обновления может быть немного утомительным. К счастью, есть несколько альтернатив. Во-первых, можно построить программу, которая различает определения вида и производит такие определения операций автоматически. Во-вторых, можно встроить обработку дифференциальных обновлений такого рода непосредственно в механизм обновления системы. В этом случае явные определения операций не нужны. Однако они остаются полезными для объяснения того, как система обрабатывает запросы на обновление.

15.4. Обновление через представления

В предыдущем разделе мы обсудили процесс обновления представлений в результате изменений базовых отношений в наборе данных. В этом разделе мы обсудим обратную сторону этого процесса, а именно – обновление базовых отношений при изменении представлений этих отношений. Этот процесс часто называют *обновлением через представления*.

Обновление представлений при изменении базовых отношений является простым, поскольку существует функциональная связь между представлениями и базовыми отношениями, в терминах которых они определены. Сложность обновления через представления заключается в том, что может существовать множество расширений базовых отношений, которые приводят к одному и тому же расширению представления. В результате изменение отношения представления может иногда быть выполнено различными способами, т. е. обновление может быть неоднозначным.

В качестве примера рассмотрим дедуктивную базу данных, в которой хранятся отношения `p` и `q`, и предположим, что нам дано следующее определение `r` в терминах `p` и `q`.

```
r(X) :- p(X)
r(X) :- q(X)
```

Предположим теперь, что пользователь просит нас добавить факт `r(a)`. Как мы должны изменить набор данных, чтобы этот вывод был верен? Должны ли мы добавить `p(a)` или `q(a)` или оба факта?

Конечно, система может просто отклонить обновление из-за неоднозначности. Но в некоторых случаях программист может предпочесть, чтобы система сделала выбор и предоставила пользователю возможность исправить этот выбор, если он ошибочен.

Прелесть определений операций заключается в том, что они позволяют программисту указать, какое из различных возможных обновлений базового отношения является подходящим, учитывая изменение обновления отношения вида в логической программе.

В качестве примера в случае, упомянутом выше, программист может выбрать запись `p(a)` (например, если объектов `p` намного больше, чем объектов `q`). В этом случае он мог бы указать эту политику, написав следующее определение операции.

```
add(r(X)) :: ~q(X) ==> p(X)
```

15.5. Упражнения

15.1. Предположим, что отношение `likes` является симметричным, т. е. если одному человеку нравится другой, то второму нравится первый. Определите операции `add` и `delete` для обновления отношения `likes` таким образом, чтобы соблюдалась симметрия.

15.2. Родительские отношения – это асимметричные отношения. Невозможно, чтобы человек был сам себе родителем. Определите операцию `add` таким образом, чтобы она обеспечивала асиммет-

рию родительского отношения, когда пользователь запрашивает добавление факта `parent`.

15.3. Предположим, что отношение `r` определено, как показано ниже. Определите `add` и `delete` для обновления отношения `r`, когда пользователь просит систему добавить или удалить фактоид `p` или `q`.

```
r(X,Y) :- p(X,Y) & ~q(Y,X)
```

15.4. Предположим, что отношение `t` определено, как показано ниже. Определите `add` и `delete` для обновления отношения `t`, когда пользователь просит систему добавить или удалить фактоид `p` или `q`.

```
t(X,Y) :- p(X,Y)
t(X,Y) :- q(X,Y)
```

15.5. Учитывая определение представления, показанное ниже, существует три общих способа обновления отношений `p` и `q` при получении запроса на добавление или удаление атома с участием отношения `r`. Определите `add` и `delete` для каждой из этих возможностей.

```
r(X,Y) :- p(X,Y) & q(X,Y)
```

Глава 16

Интерактивные рабочие листы

16.1. Интерактивные рабочие листы

Интерактивные рабочие листы – это простой, но мощный способ управления данными и решения связанных с данными проблем. Примеры интерактивных рабочих листов варьируются от простых, однопользовательских электронных таблиц (таких как интерактивные сетки ячеек в системах типа Numbers и Excel) до совместных , инструментов планирования и проектирования с участием нескольких организаций.

Сила и популярность интерактивных рабочих листов обусловлена сочетанием следующих особенностей.

1. Смысловое отображение данных. Данные обычно представляются в формах, соответствующих типу данных, – таблицах, диаграммах, графиках и т. д.
2. Возможность модификации. Данные могут быть непосредственно изменены пользователем по принципу «что видишь, то и получаешь». Важно, что данные могут быть изменены в любом удобном для пользователя порядке.
3. Проверка ограничений. Данные автоматически проверяются на полноту и согласованность со статическими и динамическими ограничениями. Пользователи предупреждаются о проблемах, и, где это возможно, им дается руководство по устранению этих проблем.
4. Автоматический расчет результатов. Результаты приемлемых изменений автоматически вычисляются, и представление обновляется, чтобы отразить эти изменения.

Хотя эти свойства могут быть использованы во многих сферах управления информацией, они имеют особую ценность в определенных типах приложений, например задачах конфигурации (таких как рабочие листы конфигурации продукта и листы академической программы),

обучения (интерактивные упражнения и симулированные среды), онлайн-играх (таких как шахматы, шашки, пентаго) и т. д.

Процесс реализации интерактивных рабочих листов с использованием традиционных технологий программирования занимает много времени и стоит дорого. Логическое программирование может значительно упростить этот процесс. Разработчикам становится проще создавать и сопровождать рабочие листы. А во многих случаях это становится возможным и для тех, кто не является разработчиком. Создание и обслуживание рабочих листов может и должно быть деятельностью типа «сделай сам». Точно так же, как пользователи без опыта программирования могут создавать и управлять традиционными электронными таблицами, так и пользователи без такого опыта должны иметь возможность создавать и управлять рабочими листами самостоятельно.

В этой главе мы рассмотрим некоторые способы использования методов логического программирования при создании интерактивных рабочих листов, которые работают в браузерах сети интернет. Хотя наше обсуждение сосредоточено на этом классе интерактивных рабочих листов, эти методы могут быть легко применены для создания интерактивных рабочих листов в других технологиях.

16.2. Пример

В качестве примера интерактивного рабочего листа, реализованного в виде веб-страницы, рассмотрим лист академической программы, показанный на странице **http://logicprogramming.stanford.edu/chapters/demo.html**. Этот рабочий лист предоставляет студенту возможность разработать программу обучения, которая удовлетворяет его академическим целям и в то же время соответствует академическим требованиям университета.

Просканируйте код камерой смартфона и пройдите по ссылке

Лист включает в себя список предметов, доступных студенту. Внизу слева расположена круговая диаграмма, показывающая долю выбранных студентом курсов в различных подобластях компьютерных наук. В середине указано количество зачетных единиц, которые студент запрашивает за каждый выбранный курс. А справа находится список профессоров, отвечающих за эти курсы.

Студент может изменить свою программу, выбирая курсы в любом удобном для него порядке. Нажатие на пустой флажок добавляет соответствующий курс в его программу обучения. Нажатие на уже установленный флажок удаляет соответствующий курс из его программы. Как только курс выбран, студент может изменить количество зачетных единиц для каждого курса с помощью ползунка, связанного с этим курсом.

Важной частью процесса обновления является проверка ограничений. По мере внесения каждого изменения рабочий лист проверяет выполнение всех академических требований. Если есть нарушение, соответствующее требование становится красным, указывая на несоответствие. Как только требование снова будет выполнено, оно становится черным.

По мере модификации программы, внесения изменений, рабочий лист обновляется соответствующим образом. Например, при установке галочки в каждом квадрате он добавляется в список курсов, и появляется фотография связанного с ним профессора. Перемещение ползунка на курсе изменяет запрашиваемый кредит, и по мере внесения изменений круговая диаграмма автоматически корректируется, чтобы показать часть времени, которое студент уделяет различным курсам.

Это простой пример, но он иллюстрирует ключевые особенности интерактивных рабочих листов – видимость всех соответствующих данных, возможность изменять эти данные, автоматическую проверку ограничений, а также автоматический расчет и отображение результатов изменений.

16.3. Данные страницы

Данные, лежащие в основе веб-страницы, отображаемой в браузере, обычно имеют форму иерархической структуры данных, называемой *DOM* (сокращение от *Document Object Model*). Верхний узел в этой структуре данных представляет документ, а ее дочерние узлы – его компоненты. Узлы в DOM обычно имеют атрибуты различного рода (например, ширину и высоту таблицы), а в некоторых случаях эти атрибуты являются объектами со своими собственными атрибутами (например, атрибут `style` узла имеет собственные атрибуты (семейство шрифтов, размер шрифта и т. д.)).

Чтобы использовать динамическое логическое программирование для задания внешнего вида и поведения веб-страницы, нам нужен словарь для представления состояния DOM в виде фактоидов, которые выражают это состояние.

Прежде всего мы присваиваем идентификаторы узлам DOM, которые нас интересуют. Чтобы придать им смысл, мы назначаем каждый из наших идентификаторов как значение атрибута `id` соответствующего узла. Например, если мы хотим использовать идентификатор `mynode` для ссылки на элемент `input` в HTML-фрагменте, показанном ниже, укажем этот идентификатор как атрибут `id` виджета ввода, как показано в этом примере.

```
<center>
    <input id='mynode' type='text' value='hello'
```

```
    size='30' style='color:black'/>
</center>
```

Далее придумываем предикаты для описания различных свойств этих узлов. Наиболее часто используемые предикаты приведены ниже. Например, мы используем бинарный предикат `value`, чтобы связать входной сигнал узла (селектор, поле ввода или текстовую область) с его значением.

`value(widget,value)`

Этот предикат истинен всегда, когда значение, ассоциированное с `widget`, равно `value`. Виджет здесь может быть текстовым полем, селектором, полем радиокнопки, ползунком и т. д.

`holds(widget,value)`

Этот фактоид истинен всякий раз, когда одно из значений связано с многозначным узлом `widget`. `widget` в данном случае – многозначный селектор или поле с флажком.

`attribute(widget,property,value)`

Этот фактоид истинен всякий раз, когда атрибутом `property` виджета `widget` является `value`.

`style(widget,property,value)`

Этот фактоид истинен, когда свойством `style` виджета `widget` является `value`.

`innerhtml(widget,content)`

Этот фактоид истинен, если `innerHTML` виджета `widget` является `content`. Обратите внимание, что `content` обычно является строкой символов.

Имея этот словарь, мы можем закодировать соответствующую информацию в виде набора данных.

Например, соответствующее состояние фрагмента DOM, показанного выше, может быть представлено набором данных, показанным ниже.

```
value(mynode,hello)
attribute(mynode,size,30)
style(mynode,color,black)
```

16.4. Жесты

Взаимодействие пользователя с веб-страницей происходит в виде жестов (например, нажатий клавиш и щелчков мышью). Чтобы говорить об этих жестах, нам нужен соответствующий словарь. Например, мы ис-

пользуем константу `click` для обозначения операции нажатия на кнопку. Мы используем константу `select` для обозначения операции выбора определенного параметра из селектора.

`select(widget,value)`

Это действие происходит, когда пользователь вводит или выбирает значение `value` в качестве значения виджета `widget`. Виджетом здесь может быть объект или текстовое поле, селектор, многозначное меню, флажок, поле радиокнопки, ползунок и т. д.

`deselect(widget,value)`

Это действие происходит, когда пользователь стирает или отменяет выбор значения `value` виджета `widget`.

`click(widget)`

Это действие происходит, когда пользователь кликает на виджет.

`tick`

Это действие происходит периодически. По умолчанию оно происходит один раз в секунду или когда пользователь нажимает кнопку **Run** или кнопку **Step** в окне управления моделированием.

`load`

Это действие происходит при первой загрузке страницы.

`unload`

Это действие происходит, когда пользователь покидает страницу.

Мы применяем этот словарь для представления жестов пользователя. Например, если пользователь нажимает на кнопку с идентификатором `a`, мы представляем это как действие `click(a)`. Если пользователь выбирает `3` из селектора с идентификатором `b`, мы представляем это как `select(b,3)`.

16.5. Определения операций

Имея словарь для кодирования данных и жестов, мы можем описать поведение рабочего листа, написав соответствующие определения операций. Следующие примеры иллюстрируют, как это можно сделать.

Рассмотрим кнопки с идентификаторами `orange`, `blue`, `purple` и `black`, показанные на рис. 16.1.

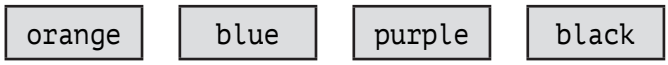

Рис. 16.1. Кнопки управления цветом документа

Предположим, что мы хотим, чтобы рабочий лист изменял цвет документа (обозначенного как page) всякий раз, когда пользователь нажимает на одну из этих кнопок. Такое поведение можно описать с помощью следующих правил операций.

```
click(orange) :: style(page,color,orange)
click(blue) :: style(page,color,blue)
click(purple) :: style(page,color,purple)
click(black) :: style(page,color,black)
```

В качестве альтернативы мы можем записать эти правила более компактно, используя переменную, как показано ниже.

```
click(X) :: style(page,color,X)
```

Это правило гласит следующее. Если пользователь нажимает на кнопку с X в качестве id, то в следующем состоянии рабочего листа цветовой стиль узла, идентифицированного как page, должен быть X.

Хотя эти правила операций работают хорошо, они не совсем полные. Это происходит потому, что после нажатия вышеуказанных кнопок состояние рабочего листа может включать более одного факта вида style(page,color,X). Чтобы полностью определить желаемое поведение, при нажатии на кнопку нужно удалить существующие фактоиды стиля для page. Это можно сделать с помощью следующего правила операции.

```
click(X) :: style(page,color,Y) & distinct(X,Y) ==> ~style(page,color,Y)
```

Это правило гласит следующее. Если пользователь нажимает на кнопку X, а цвет page – Y и Y отличается от X, то в следующем состоянии рабочего листа цвет страницы не должен быть Y.

Теперь рассмотрим другой пример. Здесь мы заменим четыре кнопки переключателем с идентификатором pagecolor и четырьмя вариантами – orange, blue, purple и black.

Рис. 16.2. Результат выбора цвета переключателем pagecolor

Предположим, что мы хотим изменить цвет текста этого документа в зависимости от выбранного значения. Это поведение можно описать с помощью следующих правил.

```
select(pagecolor,X) :: style(page,color,X)
select(pagecolor,X) :: style(page,color,Y) ==> ~style(page,color,Y)
```

Первое правило гласит, что если пользователь выбирает значение X

для pagecolor, то style(page,color,X) должен быть истинным в следующем состоянии, т. е. цвет текста страницы должен быть X. Второе правило гласит, что если пользователь выбирает значение X для pagecolor, а стиль страницы – Y и Y отличается от X, то style(page,color,X) *не должен быть* истинным в следующем состоянии.

К сожалению, этого недостаточно. Наш переход изменяет цвет страницы, но значение атрибута pagecolor при этом не меняется. В результате после обработки жеста цвет будет сброшен на черный. Следующие правила перехода обновляют переключатель.

```
select(pagecolor,X) :: value(pagecolor,X)
select(pagecolor,X) :: value(pagecolor,Y) ==> ~value(pagecolor,Y)
```

В качестве последнего примера рассмотрим пример взаимодействия между виджетами ввода. Правила операций для четырех кнопок в первом примере правильно изменяют цвет страницы. Однако они не обновляют цвет, указанный в переключателе.

Правило перехода, показанное ниже, предписывает желаемое поведение. Когда пользователь нажимает на кнопку с идентификатором X, то мы хотим, чтобы значение переключателя было обновлено и любое предыдущее значение было удалено.

```
click(X) :: value(pagecolor,X)
click(X) :: value(pagecolor,Y) ==> ~value(pagecolor,Y)
```

Комбинация этих правил с правилами, показанными выше, позволяет пользователю либо нажимать на кнопки, либо делать выбор и получать одинаковые эффекты в обоих случаях.

16.6. Определения вида

В предыдущем разделе мы увидели, что один жест может иметь несколько эффектов. Например, изменение значения переключателя с именем pagecolor устанавливает значение переключателя *и* изменяет цвет страницы. Чтобы реализовать такое поведение, нам нужно управлять обоими условиями в правилах перехода и хранить оба условия в наборе данных. Более того, если мы не будем осторожными, наши определения могут рассинхронизироваться друг с другом, и мы не получим нужного нам поведения.

Хорошая новость заключается в том, что иногда можно написать *определения вида*, которые описывают такое поведение более экономично и менее подвержены ошибкам. Определив некоторые из предикатов как *представления* других, можно не хранить так много информации и обойтись меньшим количеством правил перехода.

В случае из предыдущего раздела предположим, что мы должны были

определить цвет узла page в терминах значения узла pagecolor. Определение показано ниже.

```
style(page,color,X) :- value(pagecolor,X)
```

Имея это определение, мы можем заменить правила операций, показанные в предыдущем разделе, четырьмя, показанными ниже.

```
click(X) :: value(pagecolor,X)
click(X) :: value(pagecolor,Y) ==> ~value(pagecolor,Y)
select(pagecolor,X) :: value(pagecolor,X)
select(pagecolor,X) :: value(pagecolor,Y) ==> ~value(pagecolor,Y)
```

Здесь меньше правил, и в них упоминается меньше предикатов. В частности, здесь нет упоминания о стиле страницы. Это свойство полностью определяется значением pagecolor, и поэтому не нужно хранить или обновлять эту информацию в наших правилах. Вместо этого рабочий лист вычисляет стиль, используя определение вида, приведенное выше.

16.7. Семантическое моделирование

До сих пор мы говорили о чисто *реактивных* рабочих листах, в которых поведение определяется непосредственно в терминах видимых функций и жестов пользователя. В этом разделе мы рассмотрим *семантическое* моделирование – где поведение определяется в терминах отношений между объектами в прикладной области рабочего листа (люди, места, фильмы и т. д.). Мы рассмотрим, как можно использовать правила операций для обновления этих семантических данных, а также то, как можно использовать определения вида для определения синтаксических атрибутов в терминах семантической информации.

Рассмотрим рабочий лист планирования курсов, который выглядит следующим образом (см. рис. 16.3). В этой схеме переключатели имеют идентификаторы course1, course2, course3, course4 и варианты autumn (осень), winter (зима), spring (весна) и summer (лето).

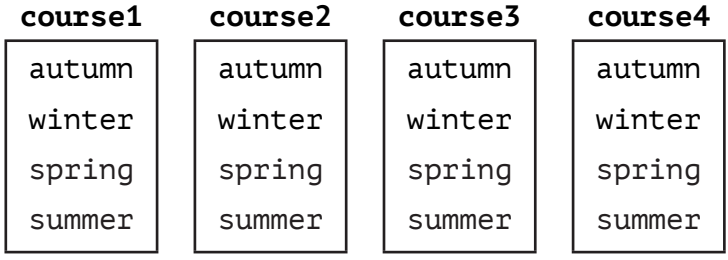

Рис. 16.3. Рабочий лист планирования курсов

Если пользователь этого рабочего листа выбирает варианты `autumn` и `spring` в `course1`, то в набор данных добавляются фактоиды `holds(course1,autumn)` и `holds(course1,spring)`. Семантически это означает, что пользователь выбрал курс 1 для осенней и весенней четверти.

Теперь рассмотрим альтернативную схему рабочего листа планирования курсов, показанную на рис. 16.4, где переключатели имеют идентификаторы `autumn`, `winter`, `spring`, `summer` и варианты `course1`, `course2`, `course3` и `course4` (обозначающие курсы, которые можно изучать в четверти).

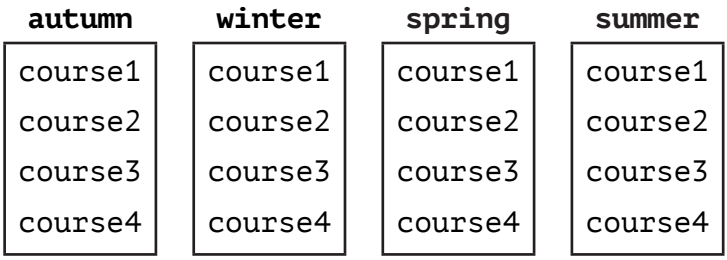

Рис. 16.4. Альтернативный вариант рабочего листа планирования курсов

В этой альтернативной схеме выбор курса пользователем соответствует фактам `holds(autumn,course1)` и `holds(spring,course1)`. Обратите внимание, что эти факты отличаются от фактов, сохраненных в предыдущем рабочем листе, т. е. `hold(course1,autumn)` и `hold(course1,spring)`. Однако на концептуальном уровне ничего не изменилось! В обоих случаях пользователь выбрал курс 1 как для осенней, так и для весенней четверти. Разница между состояниями этих двух концептуально идентичных рабочих листов обусловлена разницей в их оформлении.

Один из способов создания семантической модели рабочего листа заключается в том, чтобы отделить то, *что хранится* в состоянии рабочего листа, от того, *что отображается*, например стиль, значение, факты. Первый шаг – написать правила операций для жестов на виджетах рабочего листа, чтобы эффекты соответствовали семантически значимым отношениям.

Например, в первом рабочем листе планирования курсов мы будем использовать правила операций, показанные ниже.

```
select(Course,Quarter) :: taken(Course,Quarter)
deselect(Course,Quarter) :: ~taken(Course,Quarter)
```

Во втором рабочем листе планирования курсов, который концептуально идентичен первому, мы напишем следующие правила.

```
select(Quarter,Course) :: taken(Course,Quarter)
deselect(Quarter,Course) :: ~taken(Course, Quarter)
```

Второй шаг в создании семантического рабочего листа – это определение макета рабочего листа как представления этих семантически значимых отношений.

Например, в рабочем листе расписания первого курса мы определим `holds` как представление `taken`.

```
holds(Course,Quarter) :- taken(Course,Quarter)
```

В рабочем листе планирования второго курса мы определим `holds`, как показано ниже.

```
holds(Quarter,Course) :- taken(Course,Quarter)
```

Теперь предположим, что пользователь выбирает курс 1 в осенней и весенней четвертях. Факты, хранящиеся в обоих рабочих листах, будут идентичны.

```
taken(course1,autumn)
taken(course1,summer)
```

В итоге правила в каждом листе, по сути, представляют собой таблицу стилей для отображения и обновления этих данных. Важным преимуществом этого способа является то, что данными приложения, скрытыми в рабочем листе, можно обмениваться с другими рабочими листами, которые управляют данными иначе.

Часть V
Заключение

Глава 17

Вариации

17.1. Введение

В этой главе мы даем краткие описания нескольких дополнительных типов логического программирования: логических производственных систем, логического программирования с ограничениями, дизъюнктивного логического программирования, экзистенциального логического программирования, программирования наборов ответов и индуктивного логического программирования.

17.2. Логические производственные системы

Производственные системы – это парадигма языка программирования, которая широко используется как для компьютерных приложений, таких как экспертные системы, так и для представления процессов в человеческом мышлении. Правила в производственной системе имеют вид: *условия → действия*. Правила выполняются в *цикле*: факты в *рабочей памяти* сопоставляются с условиями, чтобы вывести действия, которые должны быть выполнены. Рабочая память аналогична набору данных, рассматриваемому в данной книге. Если условиям соответствует более одного правила, выполняется выбор правила, которое должно быть выполнено. Выполнение правила включает в себя добавление и удаление фактов из рабочей памяти. Примером стратегии выбора правил является ассоциирование приоритетов с правилами и выбор правила с более высоким приоритетом вместо правила с более низким. Повторное выполнение правил генерирует последовательные состояния рабочей памяти, и это поведение обеспечивает *операционную семантику* для правил.

Производственные правила использовались для моделирования трех видов ситуаций – ассоциации «воздействие–реакция», прямой цепочки и сокращения целей. Мы видели пример ассоциации «воздействие–реакция» в разделе 14.2 в системе с тремя лампочками: вся-

кий раз, когда пользователь нажимает на кнопку, что является воздействием, откликом системы является переключение соответствующей лампочки.

```
push_button :: p ==> ~p
push_button :: ~p ==> p
```

Правило, которое мы рассматривали в разделе 4.5, является примером прямой цепочки, так как каждый раз, когда мы добавляем факт *parent*, возникают новые факты *grandparent*, которые добавляются в набор данных. Такие правила могут быть также обобщены как операции обновления материализованных представлений, как показано в разделе 15.3.

```
parent(X,Y) & parent(Y,Z) ==> grandparent(X,Z)
```

В качестве примера сокращения целей и в контексте проблемы планирования, рассмотренной в разделе 11.4, рассмотрим следующее правило, в котором мы утверждаем, что если цель может быть достигнута путем выполнения действия `a` в состоянии `s` и возможно выполнить действие `a` в состоянии `s`, то эта цель может быть сведена к цели достижения состояния `s`.

```
goal(do(a,s)) & possible(a,s) ==> goal(s)
```

Из приведенных выше примеров видно, что существует естественный аналог между производственными правилами, динамическими правилами и операциями, рассматриваемыми в данной книге. В логических производственных системах (LPS) производственные правила первого вида представлены как динамические правила, а производственные правила второго и третьего видов – как определения вида. В дополнение к тому, что такая система дает динамическим правилам операционную семантику, аналогичную семантике производственных систем, LPS также наделяет динамические правила логической семантикой. Она интерпретирует динамические правила как декларативные предложения, которые необходимо сделать истинными в модели, содержащей последовательность значений отношений с временными метками, внешних событий и действий.

17.3. Логическое программирование с ограничениями

Рассмотрим арифметику Пеано из раздела 10.2 и следующий запрос.

```
number(L) & number(M) & add(L,M,N) & add(L,M,s(N))
```

Поскольку данный запрос эквивалентен доказательству N = N + 1, он тривиально ложен, но в системе логического программирования он будет выполняться бесконечно. Это происходит потому, что алгоритм оценки, рассмотренный в главе 8, не проверяет удовлетворение ограничений по подцелям. В логической программе с ограничениями (CLP) набор всех ограничений проверяется на удовлетворительность на каждом шаге оценки, и поэтому программа может ответить, что вышеприведенный запрос ложен.

В дополнение к проверке глобального удовлетворения ограничений система CLP позволяет сформулировать ограничения непосредственно в виде уравнений. Она позволяет ограничениям появляться в запросах, а во время оценки запросов она может генерировать новые ограничения. Чтобы проиллюстрировать эти возможности, рассмотрим следующую программу, которая вычисляет сумму S целых чисел от 0 до N.

```
sumto(0,0)
sumto(N,S) :- N≥1 & N≤S & sumto(N-1,S-1)
```

Во втором правиле, приведенном выше, ограничения задаются непосредственно с помощью терминов уравнений, а арифметические выражения выступают в качестве аргументов подцели. Более того, второе правило является небезопасным, поскольку `N` и `S` используются в его теле до того, как они ограничены. Система CLP способна обрабатывать такие небезопасные правила. Она также может генерировать новые ограничения во время оценки. Например, во время оценки запроса `S≤1` второе правило, приведенное выше, приведет к следующей расширенной версии его тела.

```
N=N1 & S=S1 & N1≥1 & N1≤S1 & sumto(N1-1,S1-1)
```

Приведенные выше примеры иллюстрировали простейшую форму ограничений, обрабатываемых системами CLP, – арифметические равенства и неравенства. Такие арифметические ограничения могут встречаться в определениях вида, рассматриваемых в этой книге. Мы видели пример использования арифметических ограничений неравенства в задаче криптарифметики раздела 6.5, хотя решение этой задачи не требует проверки ограничений по подцелям. В рамках логического программирования с ограничениями процедура унификации, описанная в главе 8, обобщается для получения решения всякий раз, когда выражения, которые должны быть сопоставлены, содержат ограничения. На каждом шаге оценки мы должны найти унификатор между выбранной подцелью и доказываемой целью и генерировать любые новые ограничения. Кроме того, необходимо проверить согласованность теку-

щего набора ограничений с ограничениями в теле определения вида. Таким образом, задействованы унификация и учет используемых ограничений.

Многочисленные расширения базовой структуры CLP существуют для задач, выходящих за рамки простых арифметических ограничений равенства и неравенства. В некоторых системах CLP возможно разрешить ограничения, в которых значения являются числами с плавающей запятой или определяются с помощью полиномиальных уравнений. CLP также используется для решения задач комбинаторного поиска, например задачи раскрашивания карты из главы 3. В некоторых комбинаторных задачах целью является не просто найти одно решение, а найти оптимальное решение в соответствии с одним или несколькими критериями оптимизации, или все решения, или замена (некоторых или всех) ограничений предпочтениями, или рассмотрение распределенной настройки, когда ограничения распределены между несколькими агентами.

17.4. Дизъюнктивное логическое программирование

В главе 2 мы определили набор данных как коллекцию простых фактов, которые характеризуют состояние области применения. Предполагается, что факты в наборе данных являются истинными, а факты, не включенные в набор данных, являются ложными. Бывают ситуации, когда наши знания о прикладной области неполны, т. е. при наличии набора фактов мы знаем, что один или несколько из них должны быть истинными, но не знаем, какие. Например, при наличии объекта `person(joe)` мы знаем, что `male(joe)` или `emale(joe)` должно быть истинным, но можем не знать, какое из них является истинным.

Чтобы понять сложность, возникающую из-за неполной информации, рассмотрим мир, в котором есть два объекта `a` и `b`, унарное отношение `p` и дизъюнктивное предложение `(p(a) | p(b))` (где `|` – оператор *или*). Вспомним из главы 7, что фактоид *логически вытекает* из замкнутой логической программы тогда и только тогда, когда он истинен в каждой модели этой программы. В данном примере множество, содержащее `p(a)`, множество, содержащее `p(b)`, и множество, содержащее как `p(a)`, так и `p(b)`, являются моделями программы, но их пересечение пусто, и поэтому единственной моделью является пустое множество. Это позволяет нам принять и `~p(a)`, и `~p(b)`, что противоречит нашему дизъюнктивному предложению. Было проведено обширное исследование дизъюнктивного логического программирования с целью изучения различных методов, позволяющих эффективно рассуждать при наличии такой неполноты. Далее мы рассмотрим один из таких методов.

Мы вычисляем набор определенных фактов как те основные атомарные факты, которые встречаются во всех минимальных моделях или ни в одной из минимальных моделей. Если нам нужно определить, является ли `(p(a) | p(b))` истинным, достаточно проверить, истинно ли `p(a)` или истинно ли `p(b)` в каждой минимальной модели. Если это верно, мы можем заключить, что `(p(a) | p(b))` истинно. Чтобы лучше оценить эту технику, рассмотрим более сложную дизъюнктивную логическую программу, показанную ниже.

```
q(a)
p(a) | p(b)
```

Эта программа имеет две минимальные модели: одна содержит `q(a)` и `p(a)`, а другая – `q(a)` и `p(b)`. Здесь набор определенных фактов включает `q(a)`, так как оно встречается в обеих минимальных моделях, и `q(b)`, так как оно не появляется ни в одной из минимальных моделей. Мы можем установить истинность `p(a) | p(b)`, проверив, содержит ли каждая минимальная модель либо `p(a)`, либо `p(b)`.

17.5. Экзистенциальное логическое программирование

Экзистенциальное правило – это правило, в голове которого находится атом с функциональным элементом. Такие правила также известны как зависимости, создающие кортежи в системах баз данных. Для наших целей логическая программа, содержащая экзистенциальные правила, называется экзистенциальной логической программой. Рассмотрим следующие экзистенциальные правила.

```
owns(X,house(X)) :- instance_of(X,person)
has_parent(X,f(X)) :- instance_of(X,person)
has_parent(X,g(X)) :- instance_of(X,male)
instance_of(f(X),person) :- instance_of(X,person)
instance_of(g(X),person) :- instance_of(X,person)
```

Первое правило гласит, что если `X` – человек, то `X` владеет домом `house(X)`. Второе правило утверждает, что если `X` – человек, то `f(X)` является родителем этого человека. Третье правило гласит, что если `X` – мужчина, то `g(X)` является родителем `X`. Четвертое и пятое правила утверждают, что для каждого человека `f(X)` и `g(X)` также являются экземплярами человека. Каждое из этих пяти правил имеет функциональный элемент в голове и, следовательно, является экзистенциальным правилом.

Экзистенциальные правила могут быть записаны в базовом логическом программировании, но их эффективное использование связано с

двумя новыми проблемами – окончанием рассуждений и рассуждениями с неполным знанием.

Первое экзистенциальное правило, показанное выше, является простейшей формой экзистенциального правила; и само по себе оно не представляет никакой проблемы с окончанием рассуждений. Но четвертое экзистенциальное правило приводит к непрекращающемуся поведению, потому что оно может быть рекурсивно применено к самому себе, что приводит к бесконечному множеству выводов.

Далее рассмотрим пример неполного знания. Из второго правила мы заключаем `has_parent(X,f(john))`, а из третьего – `has_parent(X,g(john))`, но отношения между `f(john)` и `g(john)` не определены. Логически `f(john)` и `g(john)` – это два отдельных объекта, но бывают ситуации, когда желательно заключить, что они относятся к одному и тому же человеку.

17.6. Программирование наборов ответов

При определении семантики логических программ в разделе 7.3 мы сказали, что интерпретация `D` удовлетворяет основному атому `p` тогда и только тогда, когда `p` находится в `D`. Далее мы сказали, что `D` удовлетворяет основному отрицанию `~p` тогда и только тогда, когда `p` *не входит* в `D`. Этот подход к определению семантики также известен как *отрицание как неудача*, потому что мы предполагаем, что отрицаемый атом удовлетворяется из-за его отсутствия в `D`. Для безопасной и стратифицированной логической программы семантика отрицания как неудачи гарантирует, что существует уникальная модель.

Программирование наборов ответов (ASP) – это подход к определению семантики логических программ, которые могут быть не стратифицированы. Например, рассмотрим следующие правила.

```
p(1)             p(2)                    p(3)
q(3) :- ~r(3)    r(X) :- p(X) & ~q(X)
```

Приведенные выше правила не являются стратифицированными. В ASP вышеприведенные правила приводят к двум *наборам ответов*, показанным ниже.

Набор ответов 1:

```
p(1)    p(2)    p(3)    q(3)    r(1)    r(2)
```

Набор ответов 2:

```
p(1)    p(2)    p(3)    r(3)    r(1)    r(2)
```

Решатель набора ответов – это программа, которая принимает на вход программу с набором ответов и выводит все наборы ответов этой программы. Типичный решатель набора ответов не требует входного

запроса. В этом разделе мы рассмотрим семантику логических программ с набором ответов и некоторые из их важных расширений.

Первым шагом в определении семантики набора ответов является вычисление множества всех экземпляров правил. Например, для программы, рассмотренной выше, основная программа показана ниже.

```
p(1)                    p(2)                    p(3)
q(3) :- ~r(3)           r(1) :- p(1) & ~q(1)
r(2) :- p(2) & ~q(2)    r(3) :- p(3) & ~q(3)
```

Программа, которая не содержит отрицаемых атомов или содержит отрицаемые атомы, но является безопасной и стратифицированной, будет иметь ровно один набор ответов. Набор ответов такой программы идентичен ее расширению, что определено в разделе 7.3.

Далее мы рассмотрим программы, которые не стратифицированы и содержат отрицаемые атомы. Чтобы решить, является ли множество `S` основных атомов множеством ответов, мы формируем *сокращение* основной программы относительно `S` следующим образом. Для каждого правила основной программы, такого что `S` не содержит ни одного отрицаемого элемента в теле правила, мы удаляем отрицаемые атомы из этого правила и сохраняем только его положительные атомы. Все остальные правила исключаются из основной программы. Сокращение не содержит отрицаемых атомов, и мы вычисляем его расширение, как определено в разделе 7.3. Если расширение совпадает с `S`, то `S` является набором ответов данной программы.

В качестве примера предположим, что мы хотим проверить, является ли `S = {p(1), p(2), p(3)}` набором ответов основной программы, показанной выше. Сокращение программы относительно `S` показано ниже.

```
p(1)            p(2)            p(3)
q(3)            r(1) :- p(1)
r(2) :- p(2)    r(3) :- p(3)
```

Расширением сокращения является `{p(1), p(2), p(3), r(1), r(2), r(3), q(3)}`, что отличается от `S`. Следовательно, `S = {p(1), p(2), p(3)}` не является набором ответов этой программы.

Теперь предположим, что `S = {p(1), p(2), p(3), q(3), r(1), r(2)}`. Сокращение программы относительно этого нового набора ответов показано ниже.

```
p(1)    p(2)            p(3)
q(3)    r(1) :- p(1)    r(2) :- p(2)
```

Расширением этой программы является `{p(1), p(2), p(3), q(3), r(1),`

r(2)}, что идентично S. Следовательно, S = {p(1), p(2), p(3), q(3), r(1), r(2)} является набором ответов этой программы.

Семантика наборов ответов обеспечивает элегантный способ определения значения нестратифицированных логических программ. Было обнаружено, что ASP является полезным подходом для описания широкого круга комбинаторных задач, особенно тех, которые включают в себя определение сложных ограничений. Кроме того, структура ASP легко поддается обобщению для работы с арифметикой и дизъюнкциями. В настоящее время существуют общедоступные и коммерческие решатели ASP, которые имеют впечатляющую производительность при обработке небольших задач.

17.7. Индуктивное логическое программирование

Индукция – это рассуждение от частного к общему. Например, рассмотрим следующий набор данных о родстве, который аналогичен рассмотренному в предыдущих главах.

```
parent(a,b)    parent(a,c)    parent(d,b)
father(a,b)    father(a,c)    mother(d,b)
male(a)        female(c)      female(d)
```

Имея этот набор данных, мы можем использовать индуктивное рассуждение, чтобы вывести следующие правила (или определения вида).

```
father(X,Y) :- parent(X,Y) & male(X)
mother(X,Y) :- parent(X,Y) & female(X)
```

В индуктивном логическом программировании, имея набор данных, набор начальных определений вида и целевой предикат, возможно вывести определение вида для целевого предиката. В приведенном выше примере дан набор данных, нет начальных определений вида, и мы можем вывести определения вида father и mother.

В контексте индуктивного логического программирования набор данных также называется набором позитивных примеров. Некоторые алгоритмы индуктивного рассуждения также принимают на вход набор негативных примеров. Если негативные примеры не предоставлены, они могут быть вычислены как набор основных атомов в базе Гербранда, которые отсутствуют в наборе данных. Объединенный набор позитивных и негативных примеров, взятых вместе, также известен как обучающие данные.

Существует два широких класса алгоритмов индуктивного логического программирования: сверху вниз и обратная дедукция. При подходе к

обучению сверху вниз мы начинаем с общего определения вида и ограничиваем его до тех пор, пока оно не удовлетворит всем позитивным и негативным примерам. В подходе обратной дедукции мы начинаем с известных фактов и ищем определения вида, которые необходимы для получения этих фактов.

Приложение A

Предопределенные понятия в EpilogJS

A.1. Введение

EpilogJS – это библиотека подпрограмм на языке Javascript, предназначенная для обработки логических программ, написанных на языке Epilog. Это приложение является руководством пользователя по предопределенным функциям, предопределенным отношениям и различным операторам, поддерживаемым EpilogJS.

A.2. Отношения

same(*выражение*,*выражение*)

Предложение `same(x,y)` истинно тогда и только тогда, когда `x` и `y` идентичны. Например, `same(f(b),f(b))` истинно.

distinct(*выражение*,*выражение*)

Предложение `distinct(x,y)` истинно тогда и только тогда, когда `x` и `y` не идентичны. Например, `distinct(f(a),f(b))` истинно.

evaluate(*выражение*,*выражение*)

Предложение `evaluate(x,y)` истинно тогда и только тогда, когда значение `x` равно `y`. Например, `evaluate(plus(2,3),5)` истинно.

member(*выражение*,*список*)

Предложение `member(x,l)` истинно тогда и только тогда, когда `x` является членом списка `l`. Например, `member(b,[a,b,c])` истинно.

true(*предложение*,*выражение*)

Предложение `true(p,d)` истинно тогда и только тогда, когда предложение `p` истинно в наборе данных с именем `d`. Например, если набор данных с именем `mydataset` содержит предложение `p(a,b)`, то `true(p(a,b),mydataset)` истинно.

А.3. Математические функции

abs(*число*) → *число*

Значение `abs(x)` – это абсолютная величина `x`. Например, значение `abs(-8)` равно 8.

acos(*число*) → *число*

Значение `acos(x)` – это обратный косинус `x`. Например, значение `acos(1)` равно `0`.

acosh(*число*) → *число*

Значение `acosh(x)` – это обратный гиперболический косинус `x`. Например, значение `acosh(1)` равно `0`.

asin(*число*) → *число*

Значение `asin(x)` – это обратный синус `x`. Например, значение `asin(0)` равно `0`.

asinh(*число*) → *число*

Значение `asinh(x)` – это обратный гиперболический синус `x`. Например, значение `asinh(0)` равно `0`.

atan(*число*) → *число*

Значение `atan(x)` – это обратный тангенс `x`. Например, значение `atan(0)` равно `0`.

atan2(*число, число*) → *число*

Значение `atan2(x,y)` – это обратный тангенс `x/y`. Например, значение `atan2(0,1)` равно `0`.

atanh(*число*) → *число*

Значение `atanh(x)` – это обратный гиперболический тангенс `x`. Например, значение `atanh(0)` равно 0.

cbrt(*число*) → *число*

Значение `cbrt(x)` – это кубический корень из `x`. Например, значение `cbrt(8)` равно `2`.

ceil(*число*) → *число*

Значение `ceil(x)` – это наименьшее целое число, которое больше `x`. Например, значение `cbrt(2.2)` равно `3`.

clz32(*число*) → *число*

Значение `clz32(x)` – это количество ведущих нулей в 32-битном представлении `x`. Например, значение `clz32(2147483647)` равно `1`.

cos(*число*) → *число*

Значение `cos(x)` – это косинус числа `x`. Например, значение `cos(0)` равно `1`.

cosh(*число*) → *число*

Значение `cosh(x)` – это гиперболический косинус `x`. Например, значение `cosh(0)` равно `1`.

exp(*число*) → *число*

Значение `exp(x)` – это `e` в степени `x`. Например, значение `exp(1)` равно `~2.718281828459045`.

expm1(*число*) → *число*

Значение `expm1(x)` равно `e` в степени `x` минус `1`. Например, значение `expm1(0)` равно `1`.

floor(*число*) → *число*

Значение `floor(x)` – это наибольшее целое число, меньшее `x`. Например, значение `floor(1.6)` равно `1`.

fround(*число*) → *число*

Значение `fround(x)` – это ближайшее к `x` число с плавающей точкой одинарной точности.

hypot(*число*,…,*число*) → *число*

Значение `hypot(x1,...,xk)` – квадратный корень из суммы квадратов `x1,...,xk`. Например, значение `hypot(3,4)` равно `5`.

imul(*число*,*число*) → *число*

Значение `imul(x,y)` – это произведение `x` и `y` (где сомножители – 32-битные целые числа со знаком). Например, значение `imul(4294967295,-5)` равно `5`.

log(*число*) → *число*

Значение `log(x)` – натуральный логарифм от `x`. Например, значение `log(1)` равно `0`.

log1p(*число*) → *число*

Значение `log1p(x)` – натуральный логарифм `x+1`. Например, значение `log1p(0)` равно `0`.

log2(*число*) → *число*

Значение `log2(x)` – это логарифм `x` по основанию `2`. Например, значение `log2(8)` равно `3`.

log10(*число*) → *число*

Значение `log10(x)` – это логарифм `x` по основанию `10`. Например, значение `log10(100)` равно `2`.

max(*число*,…,*число*) → *число*

Значение `max(x1,...,xk)` – это максимум из `x1,...,xk`. Например, значение `max(3,4,1,2)` равно `4`.

min(*число*,…,*число*) → *число*

Значение `min(x1,...,xk)` – это минимум из `x1,...,xk`. Например, значение `min(3,4,1,2)` равно `1`.

minus(*число*,…,*число*) → *число*

Значение `minus(x1,...,xk)` – это разность `x1,...,xk`. Например, значение `minus(9,4,3)` равно `2`.

plus(*число*,…,*число*) → *число*

Значение `plus(x1,...,xk)` – это сумма `x1,...,xk`. Например, значение `plus(2,3,4)` равно `9`.

pow(*число*,*число*) → *число*

Значение `pow(x,y)` равно `x`, возведенному в степень `y`. Например, значение `pow(2,3)` равно `8`.

quotient(*число*,…,*число*) → *число*

Значение `quotient(x1,...,xk)` – это частное от деления `x1,...,xk`. Например, значение `quotient(12,3,2)` равно `2`.

random() → *число*

Значение `random()` – случайное число в диапазоне от `0` (включительно) до `1` (исключительно). Например, одно из возможных значений `random` равно `0,23`.

round(*число*) → *число*

Значение `round(x)` – это `x`, округленное до ближайшего целого числа. Например, значение `round(1.6)` равно `2`.

sin(*число*) → *число*

Значение `sin(x)` – это синус `x`. Например, значение `sin(0)` равно `0`.

sinh(*число*) → *число*

Значение `sinh(x)` – это гиперболический синус `x`. Например, значение `sinh(0)` равно `0`.

sqrt(*число*) → *число*

Значение `sqrt(x)` – это положительный квадратный корень из `x` (где `x` – любое неотрицательное число). Например, значение `sqrt(4)` равно `2`.

tan(*число*) → *число*

Значение `tan(x)` – это тангенс `x`. Например, значение `tan(0)` равно `0`.

tanh(*число*) → *число*

Значение `tanh(x)` – это гиперболический тангенс `x`. Например, значение `tanh(0)` равно `0`.

times(*число*,...,*число*) → *число*

Значение `times(x1,...,xk)` – это произведение `x1,...,xk`. Например, значение `times(2,3,4)` равно `24`.

trunc(*число*) → *число*

Значение `trunc(x)` – это целая часть `x`. Например, значение `trunc(2.3)` равно `2`, а значение `trunc(-2.3)` равно `-2`.

A.4. Строковые функции

stringappend(*строка*,...,*строка*) → *строка*

Значением `stringappend(s1,...,sk)` является конкатенация `s1,...,sk`. Например, значение `stringappend("Hello",",","World","!")` равно `"Hello, World!"`.

stringmin(*строка*,...,*строка*) → *строка*

Значение `stringmin(s1,...,sk)` – это лексикографически наименьшее значение среди указанных строк. Например, значение `stringmin("def","abc","efg")` равно `"abc"`.

matches(*строка*,...,*строка*) → *строка*

Если строка `str` соответствует регулярному выражению `pat`, то значением `matches(str,pat)` является список, состоящий из подстроки `str`, которая соответствует `pat`, и подстрок, которые соответствуют компонентам `pat`, заключенным в круглые скобки. Например, значение `matches("321-1245","(.)-(.)")` будет `["1-1","1", "1"]`.

submatches(*строка*,...,*строка*) → *строка*

Значение `submatches(str,pat)` – это список всех подстрок строки `str`, которые соответствуют регулярному выражению `pat`. Например, значение `matches("321-1245",".2.")` равно `["321","124"]`.

A.5. Функции списков

append(*список*,...,*список*) → *список*

Значением `append(l1,...,lk)` является конкатенация `l1,...,lk`. Например, значением `append([a,b,c],[d,e,f])` будет `[a,b,c,d,e,f]`.

revappend(*строка*,*строка*) → *строка*

Значение `revappend(l1,l2)` – это результат конкатенации обратного `x` и `y`. Например, значение `revappend([a,b,c],[d,e,f])` равно `[c,b,a,d,e,f]`.

reverse(*список*) → *список*

Значение функции `reverse([x1,...,xk])` равно `[xk,...,x1]`. Например, значением `reverse([a,b,c])` будет `[c,b,a]`.

length(*список*) → *число*

Значение `length(l)` – это длина `l`. Например, значение `length([a,b,c])` равно `3`.

А.6. Арифметические функции списков

maximum([*число, ..., число*]) → *число*

Значением `maximum([x1,...,xk])` является максимальный элемент в указанном списке. Например, значение `maximum([3,4,1,2])` равно `4`.

minimum(*список*) → *список*

Значение `minimum([x1,...,xk])` – минимальный элемент в указанном списке. Например, значение `minimum([3,4,1,2])` равно `1`.

sum(*список*) → *число*

Значение `sum([x1,...,xk])` – это сумма элементов указанного списка. Например, значение `sum([3,4,1,2])` равно `10`.

range(*список*) → *число*

Значение `range([x1,...,xk])` – это диапазон элементов указанного списка, т. е разность между максимальным и минимальным элементом. Например, значение `range([3,4,2,1])` равно `3`.

midrange(*список*) → *число*

Значение `midrange([x1,...,xk])` – это среднее значение элементов указанного списка, т. е. половина суммы максимального и минимального элементов. Например, значение `midrange([3,4,2,1])` равно `2,5`.

mean(*список*) → *число*

Значение `mean([x1,...,xk])` – это среднее арифметическое значение элементов указанного списка. Например, значение `mean([3,4,2])` равно `3`.

median(*список*) → *число*

Значение `median([x1,...,xk])` – это медиана элементов указанного списка. Например, значение `median([3,4,2])` равно `3`.

variance(*список*) → *число*

Значение `variance([x1,...,xk])` – это дисперсия элементов в указанном списке. Например, значение variance([3,4,2,1]) равно `1.25`.

stddev(*список*) → *число*

Значение `stddev([x1,...,xk])` – это стандартное отклонение элементов указанного списка. Например, значение `stddev([3,4,2,1])` равно `~1.118033988749895`.

A.7. Функции преобразования

symbolize(*строка*) → *символ*

Значением функции `symbolize(str)` является символ, состоящий только из букв, знаков подчеркивания и цифр в строке `str`, в которой все заглавные буквы преобразованы в строчные. Например, значением функции `symbolize("Your name.")` будет `yourname`.

newsymbolize(*строка*) → *символ*

Значением `newsymbolize(str)` является символ, состоящий только из букв, знаков подчеркивания и цифр в строке `str`, в которой все прописные буквы были преобразованы в строчные, а все пробелы заменены на знаки подчеркивания. Например, значением `newsymbolize("Your name.")` является `your_name`.

readstring(*строка*) → *выражение*

Значением `readstring(str)` является первое выражение, которое можно выделить из символов в строке `str`. Например, значением `readstring("p(a) p(b)")` является `p(a)`.

readstringall(*строка*) → *выражение*

Значение `readstring(str)` – это список всех выражений, которые могут быть извлечены из символов в строке `str`. Например, значением `readstring("p(a) p(b)")` будет `[p(a), p(b)]`.

stringify(*выражение*) → *строка*

Значение `stringify(expression)` – это строковое представление выражения `expression`. Например, значение `stringify(p(a) & p(b))` равно `"p(a) & p(b)"`.

stringifyall(*список*) → *строка*

Значение `stringifyall([x1,...,xk])` – строковое представление `[x1,...,xk]`. Например, значением `stringifyall([p(a), p(b)])` является `"p(a) p(b)"`.

listify(*выражение*) → *список*

Значение `listify(expression)` – это представление выражения `expression` в виде списка. Например, значением `listify(p(a,b))` является `[p,a,b]`.

delistify(*список*) → *выражение*

Значение `delistify(l)` – это представление списка `l` в виде выражения. Например, значением `delistify([p,a,b])` является `p(a,b)`.

A.8. Агрегаты

setofall(*выражение,предложение*) → *список*

Значением `setofall([x,p])` является список, состоящий из всех

экземпляров `x`, для которых соответствующий экземпляр `p` является истинным. Например, для набора данных, содержащего `p(a,b)`, `p(a,c)` и `p(a,d)`, значение `setofall(X,p(a,X))` будет `[b,c,d]`.

countofall(*выражение*,*предложение*) → *число*

Значение `countofall([x,p])` – это количество экземпляров `x`, для которых соответствующий экземпляр `p` является истинным. Например, для набора данных, содержащего `p(a,b)`, `p(a,c)` и `p(a,d)`, значение `countofall(X,p(a,X))` равно `3`.

A.9. Операторы

nil

Символ `nil` – это другое представление пустого списка, т. е. `nil` и `[]` являются синонимами.

cons(*выражение*,*список*)

Символ `cons` – это основной оператор, используемый в списках языка Epilog. Например, список `[a,b,c]` эквивалентен `cons(a, cons(b,cons(c,nil)))`. Обратите внимание, что `a!b!c!nil` – это другой способ записи этого выражения.

not(*предложение*)

Символ `not` является основным оператором отрицания. Например, выражение `~p(a)` эквивалентно `not(p(a))`.

and(*выражение*,…,*выражение*)

Символ `and` является основным оператором в конъюнкции. Например, `(p(X) & q(X))` эквивалентно `and(p(X),q(X))`.

or(*выражение*,…,*выражение*)

Символ `or` является основным оператором в дизъюнкции. Например, `(p(X) | q(X))` эквивалентно `or(p(X),q(X))`.

rule(*выражение*,…,*выражение*)

Символ `rule` является основным оператором правил в определениях вида. Например, правило `r(X) :- p(X) & q(X)` эквивалентно `rule(r(X),p(X),q(X))`.

definition(*выражение*,*выражение*)

Символ `definition` является основным оператором определения функций. Например, `definition f(X) := g(h(X))` эквивалентно `definition(f(X),g(h(X)))`.

transition(*выражение*,*выражение*)

Символ `transition` является основным оператором правил перехода. Например, правило перехода `p(X) ==> q(X)` эквивалентно `transition(p(X),q(X))`.

if(*условие1*,*выражение1*,...,*условиеN*,*выражениеN*)

Символ `if` является основным условным оператором в определениях функций. Значением `if(condition1,expression1,condition2, expression2,...,conditionN,expressionN)` является `expression1`, если `condition1` истинно, иначе `expression2`, если `condition2` истинно, ..., иначе `expressionN`, если `conditionN` истинно. Например, значением `if(p(a),"yes",true,"no")` является `"yes"`, если `p(a)` истинно, иначе – `"no"`.

Эта встроенная функция является *вариативной*, т. е. количество ее аргументов может изменяться.

choose(*выражение*,*предложение*)

Значением `choose(expression,sentence)` является случайный член множества {выражение или предложение, являющееся истинным}. Например, для набора данных `{r(a), r(b)}` значение `choose(f(X), r(X))` может быть либо `f(a)`, либо `f(b)`.

Приложение Б

Sierra

Б.1. Введение

Sierra – это браузерная интерактивная среда разработки (IDE) для Epilog. Она позволяет пользователям просматривать и редактировать наборы данных и наборы правил, предоставляет разнообразные инструменты для запросов и изменения наборов данных и наборов правил. При внесении изменений происходит автоматическое обновление видимых наборов данных в виде электронной таблицы в соответствии с правилами пользователя. В ней также предусмотрены инструменты для анализа наборов данных и правил, отслеживания выполнения программы, а также инструменты для сохранения и загрузки файлов.

В данном документе представлен ознакомительный тур по основным возможностям Sierra. Мы рассмотрим, как загрузить Sierra, как создавать, просматривать и редактировать наборы данных и наборы правил, а также сохранять свою работу для последующего использования. Рекомендуем по мере прохождения тура повторять в своем браузере показанные здесь шаги.

Б.2. Начало работы

Поскольку Sierra основана на браузере, мы начнем с загрузки подходящего браузера. (Sierra работает в Safari, Chrome, Firefox и других браузерах. В наших примерах мы используем Safari, хотя внешний вид и взаимодействие практически одинаковы во всех основных браузерах.) После этого перейдем по ссылке **http://epilog.stanford.edu/homepage/sierra.php**. Откроется страница, которая выглядит следующим образом.

Просканируйте код камерой смартфона и пройдите по ссылке

Расположенная сверху командная строка обеспечивает доступ к меню, касающемуся файлов, наборов данных, каналов, наборов правил, различных инструментов и системных настроек. Мы будем знакомиться с этими меню по очереди по мере того, как будем продолжать наше путешествие.

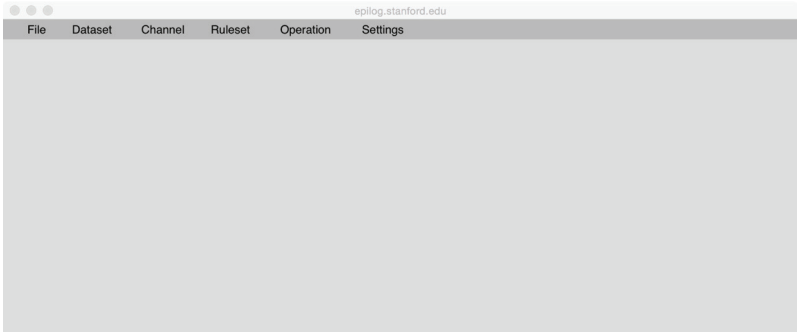

Рис. Б.1. Стартовая страница Sierra

Б.3. Данные

Щелкнув на меню **Datasets** (Наборы данных), мы увидим два варианта – **Lambda** (Лямбда – набор данных по умолчанию) и **New Dataset** (Новый набор данных), который используется для создания нового набора данных. Давайте начнем с нажатия на **Lambda**. Откроется окно, показывающее содержимое набора данных с именем **Lambda**. Изначально он пуст.

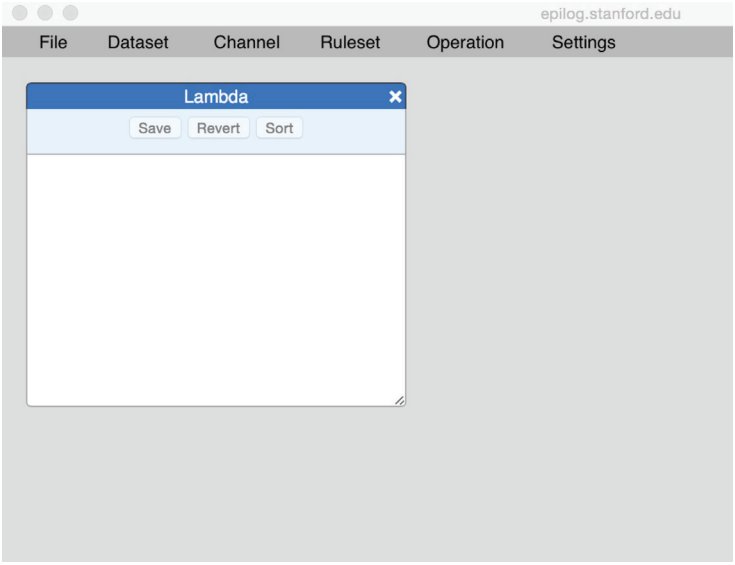

Рис. Б.2. Страница набора данных **Lambda**

Мы можем добавлять данные, вводя их в окно. Здесь мы ввели факты `p(c,d)`, `p(a,b)` и `p(b,c)`. Окно выделено красным цветом, что указывает на то, что мы внесли изменения, но пока не сохранили эти изменения в базе данных.

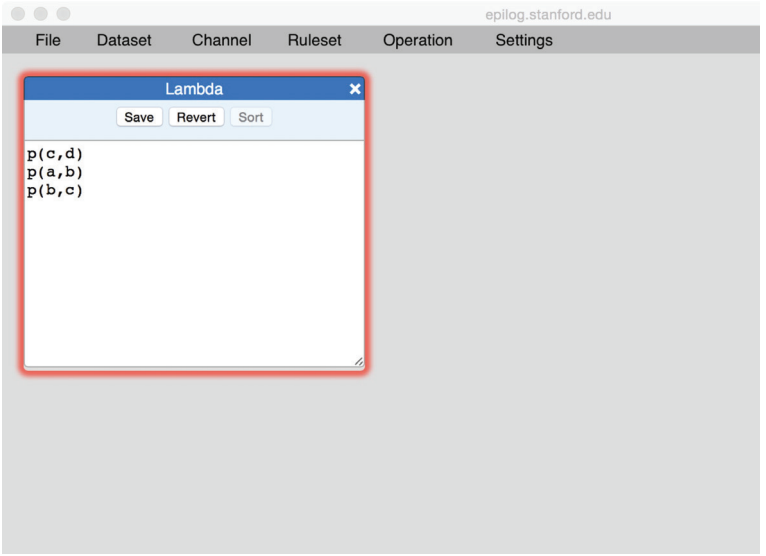

Рис. Б.3. Изменения набора данных не сохранены

Нажатие кнопки **Save** (Сохранить) сохраняет данные и убирает подсветку, указывая на то, что в окне показаны текущие данные.

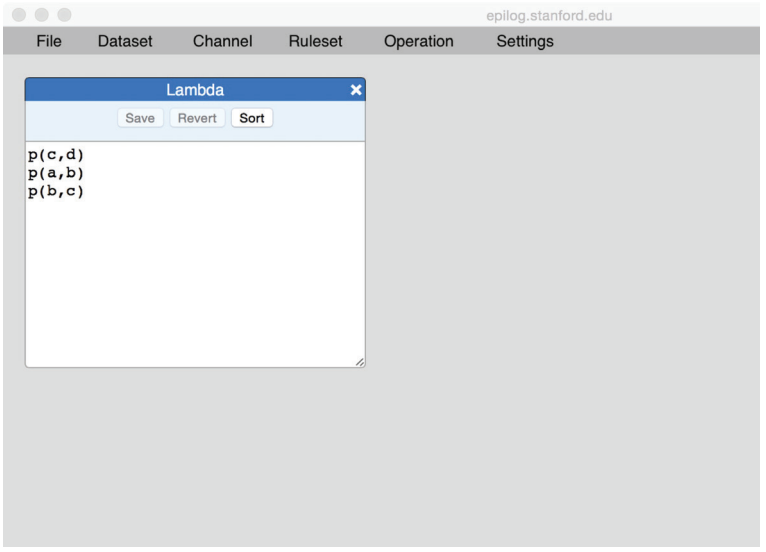

Рис. Б.4. Результат изменения набора данных

На этом этапе мы можем добавить или удалить данные, или изменить их другими способами. Одной из полезных функций является сортировка данных. Это можно сделать, нажав кнопку **Sort** (Сортировка). Обратите внимание, что окно снова выделено, что указывает на то, что результат операции сортировки не был сохранен в базе данных.

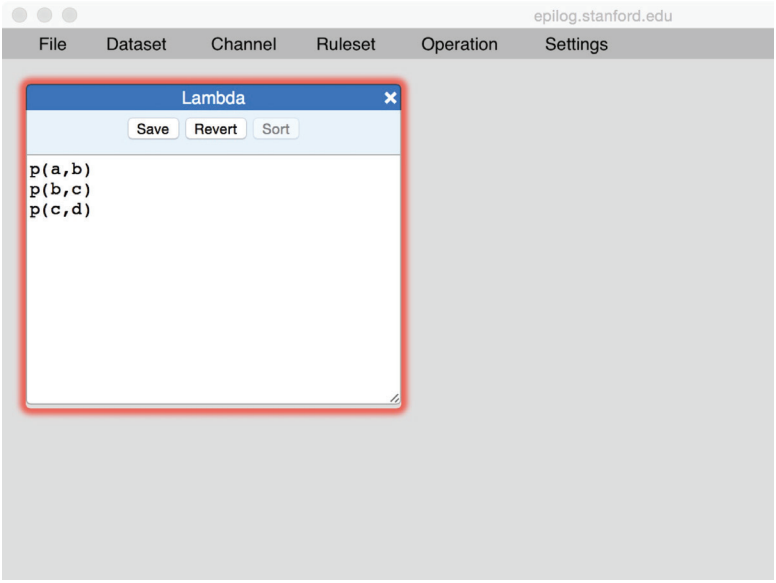

Рис. Б.5. База данных отсортирована

Повторное нажатие кнопки **Save** фиксирует эти изменения в базе данных.

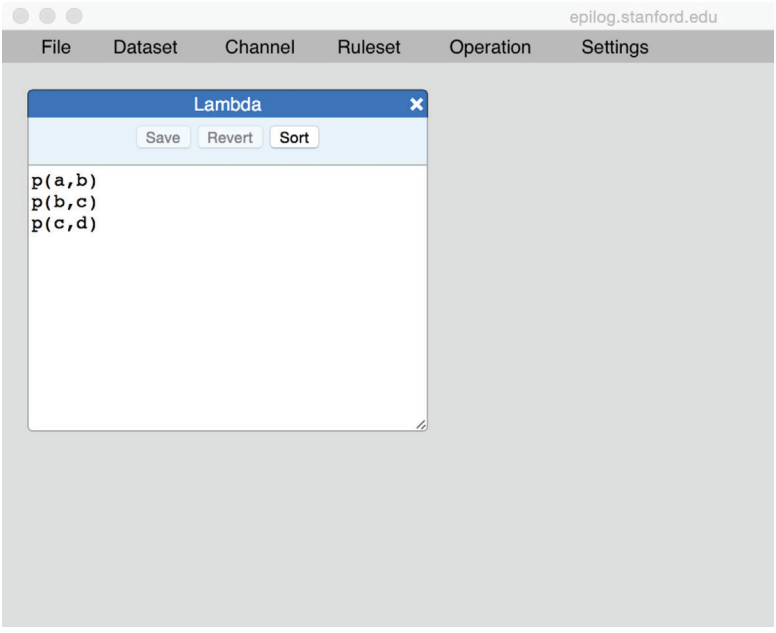

Рис. Б.6. Сохраненная отсортированная база данных

Предположим, мы отредактируем данные, и результат будет синтаксически незаконным, как показано ниже.

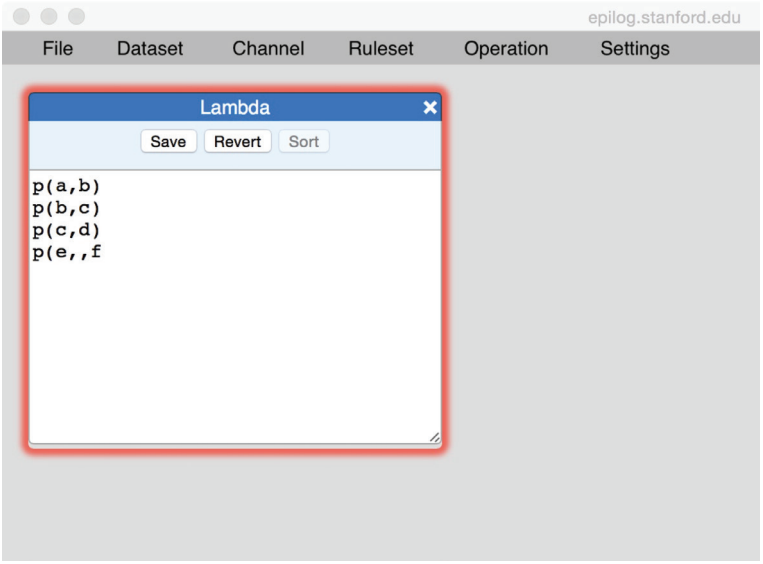

Рис. Б.7. Синтаксически незаконная операция

Если мы попытаемся сохранить эти данные, то получим сообщение об ошибке, как показано ниже, и база данных не будет изменена.

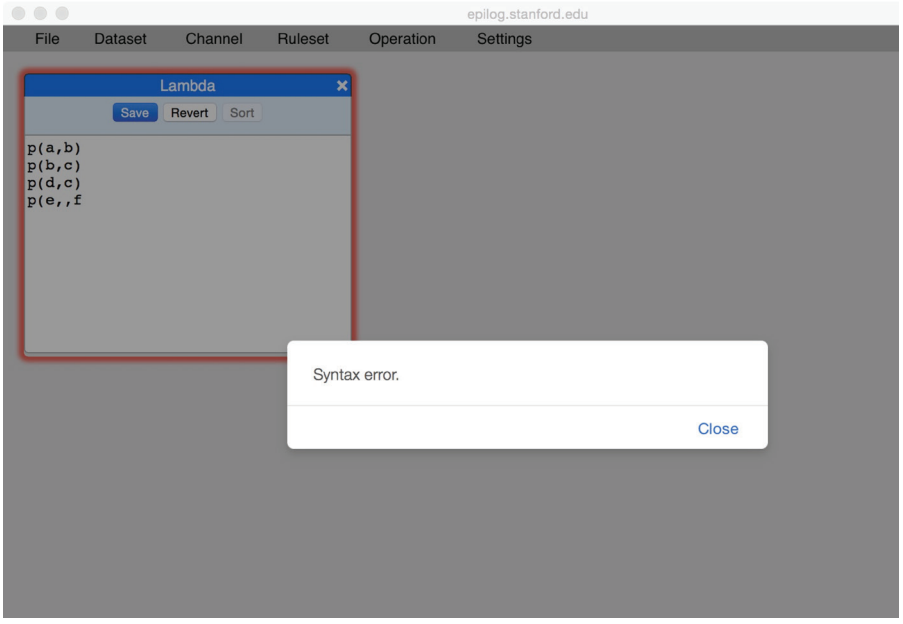

Рис. Б.8. Окно с сообщением об ошибке

В этот момент мы можем либо устранить проблему и попробовать снова, либо нажать кнопку **Revert** (Отмена), чтобы вернуться к текущему состоянию базы данных, как мы это сделали здесь.

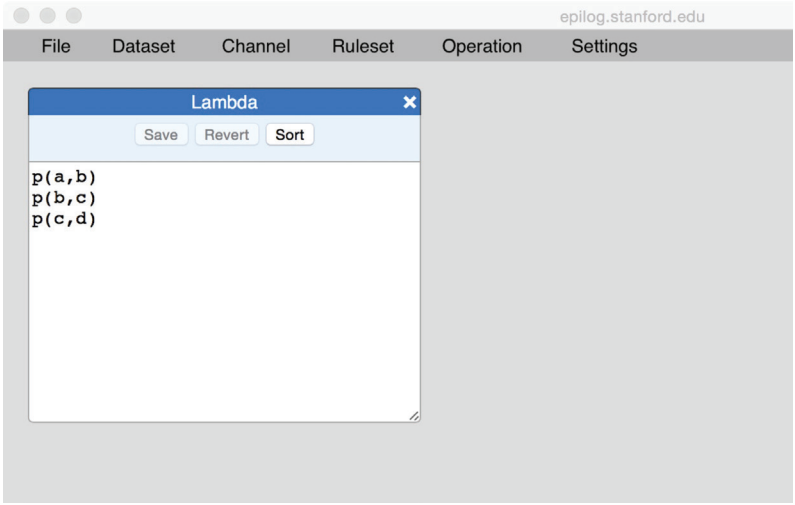

Рис. Б.9. Результат нажатия кнопки **Revert** (Отмена)

Б.4. Запросы

Меню **Operation** (Действия) содержит инструменты для программного запроса и обновления данных. Если мы выберем инструмент **Query** (Запрос), то получим окно, подобное показанному на рис Б.10. Обратите внимание, что окно может появиться поверх существующего окна. Чтобы переместить окно, щелкните на строке заголовка окна и перетащите окно в нужное место. Если вы хотите изменить размер окна, щелкните в его правом нижнем углу и перетащите его до нужного размера. Здесь мы перетащили окно вправо от окна **Lambda**.

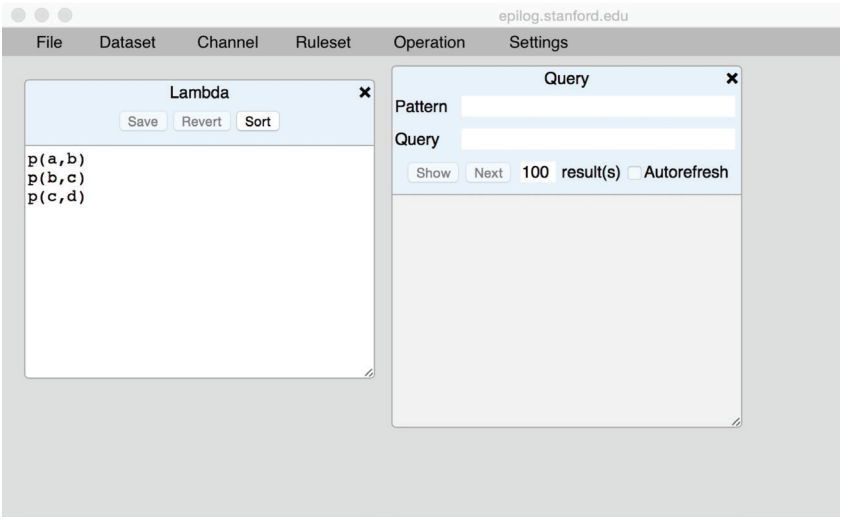

Рис. Б.10. Открыто окно **Query** (Запрос)

Для формирования запроса мы вводим выражение для желаемого ответа в поле **Pattern** (Шаблон) и вводим запрос в поле **Query** (Запрос). Здесь мы запрашиваем все выражения вида `goal(X,Z)`, для которых `p(X,Y) & p(Y,Z)` истинно.

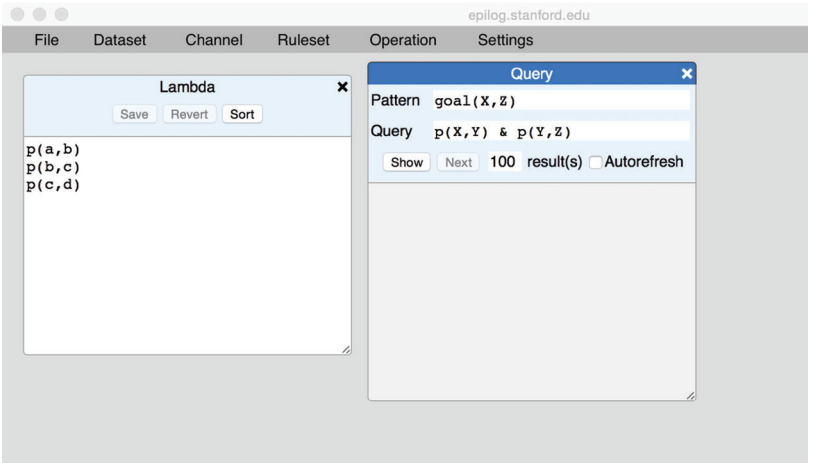

Рис. Б.11. Запрос сформирован

Нажатие кнопки **Show** (Показать) выполняет запрос и помещает результаты в окно запроса. В данном случае есть только два ответа, а именно `goal(a,c)` и `goal(b,d)`.

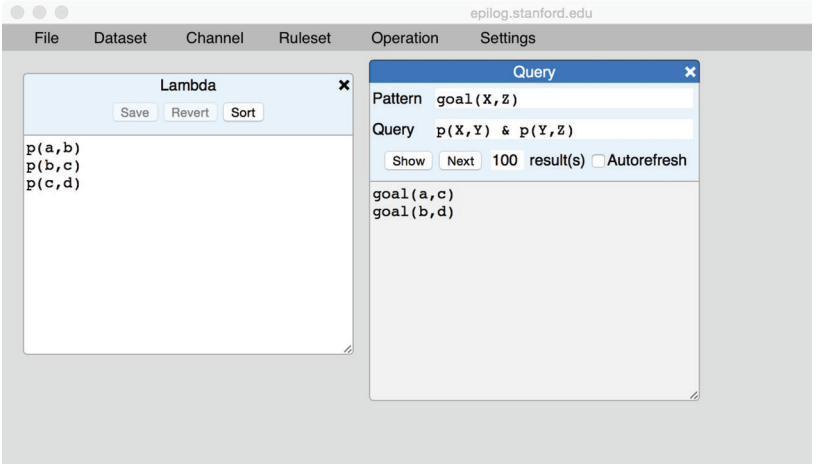

Рис. Б.12. Результат выполнения запроса

В общем виде запросы могут иметь много ответов. По умолчанию инструмент **Query** показывает только 100 ответов, как показано на рис. Б.13. Мы можем изменить это значение, отредактировав соответствующее поле. В случае дорогих запросов часто желательно получить только один результат. После того как результат или результаты будут

показаны, мы можем получить следующую порцию ответов, нажав кнопку **Next**.

Флажок **Autorefresh** (Автообновление) указывает системе, хотим ли мы, чтобы запрос пересчитывался автоматически в ответ на изменения в базе данных и базе правил. Здесь мы установили флажок (см. рис. Б.13) и тем самым даем команду на автоматическое обновление.

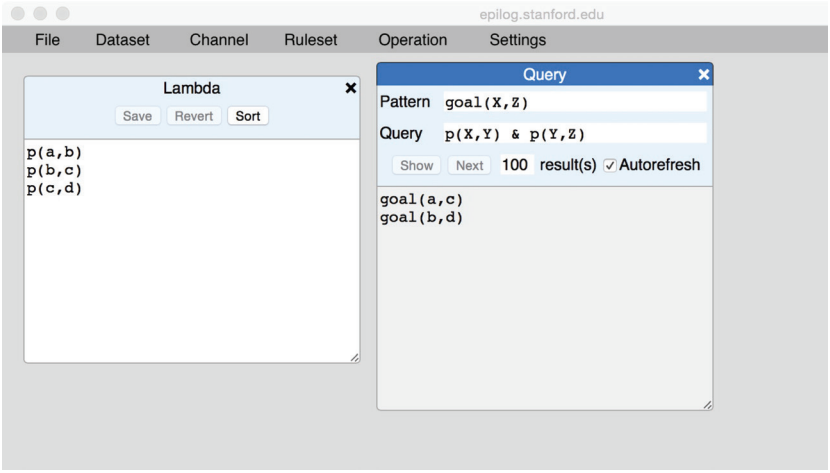

Рис. Б.13. Дана команда **Autorefresh**

Теперь давайте вернемся к набору данных **Lambda** и добавим еще один факт.

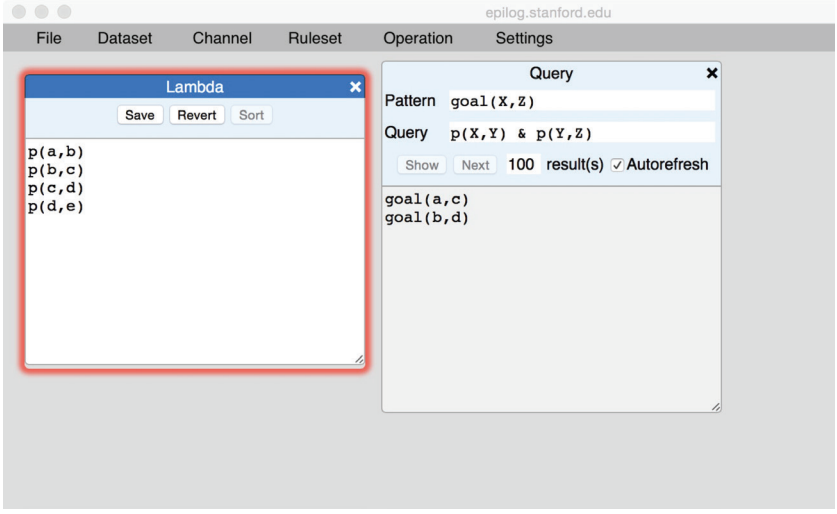

Рис. Б.14. В набор данных добавлен один факт

Когда мы нажимаем кнопку **Save**, данные сохраняются, и окно **Query** автоматически обновляется, как показано на рис. Б.15.

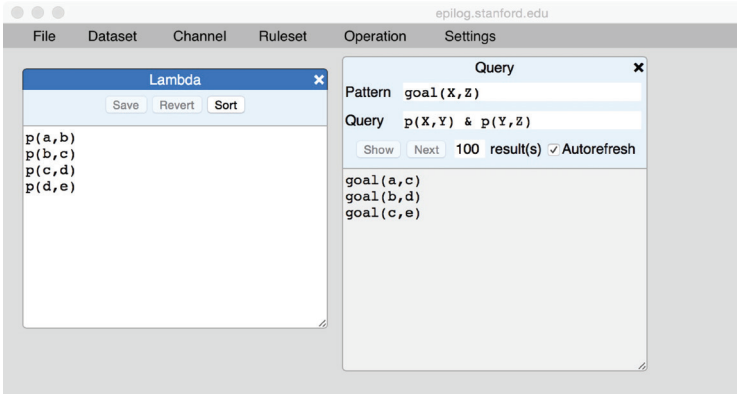

Рис. Б.15. Изменение результата запроса при обновлении данных

Обратите внимание, что обычно у пользователей одновременно открыто несколько окон запросов, и в них установлены флажки автообновления. Когда в базу данных вносятся изменения, все эти окна автоматически обновляются в стиле электронных таблиц.

Б.5. Обновления

Меню **Operation** (Действия) также содержит инструменты для программной модификации базы данных. При нажатии на кнопку **Transform** (Преобразовать) появляется окно, подобное показанному на рис. Б.16.

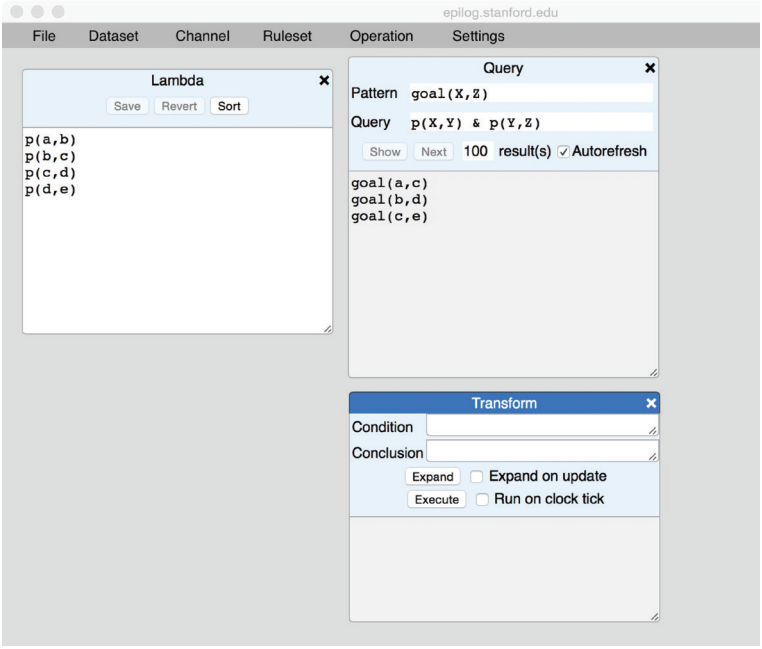

Рис. Б.16. Нажата кнопка **Transform** в меню **Operation**

Этот инструмент позволяет выполнять преобразования базы данных. Чтобы задать преобразование, мы вводим шаблон в поле **Condition** (Условие) и шаблон в поле **Conclusion** (Заключение), как показано ниже (рис. Б.17).

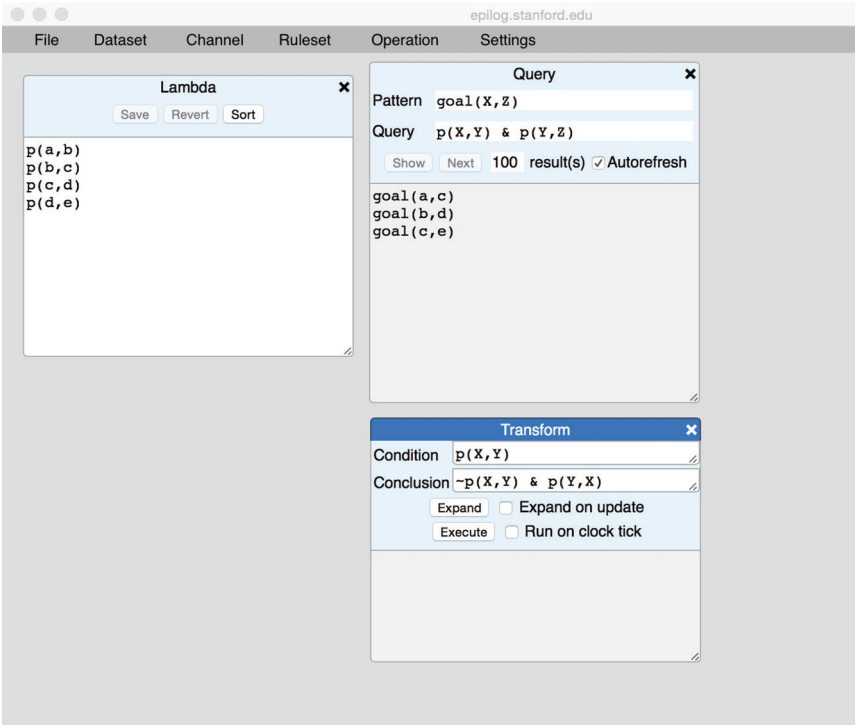

Рис. Б.17. Задание преобразования базы данных

При выполнении преобразования Sierra находит все привязки переменных, которые удовлетворяют заданному условию, и для каждой из них изменяет базу данных в соответствии с экземплярами заключения, соответствующего этой привязке переменной. В данном случае мы просим Sierra найти все факты вида `p(X,Y)`, и для каждого из них мы хотим, чтобы она удалила этот факт и заменила его фактом вида `p(Y,X)`, т. е. тем же самым фактом с обратными аргументами.

Нажатие кнопки **Execute** (Выполнить) в этом случае приводит к ситуации, показанной ниже (см. рис. Б.18). Обратите внимание, что факты в **Lambda** были изменены в соответствии с указаниями. Обратите также внимание, что окно запроса было обновлено, чтобы отразить новые данные.

Кнопка **Expand** (Расширить) запрашивает у Sierra отображение добавлений и удалений, которые были бы выполнены, если бы преобразование было выполнено (см. рис. Б.19). Это чрезвычайно полезно при отладке, чтобы увидеть изменения до того, как они будут сделаны.

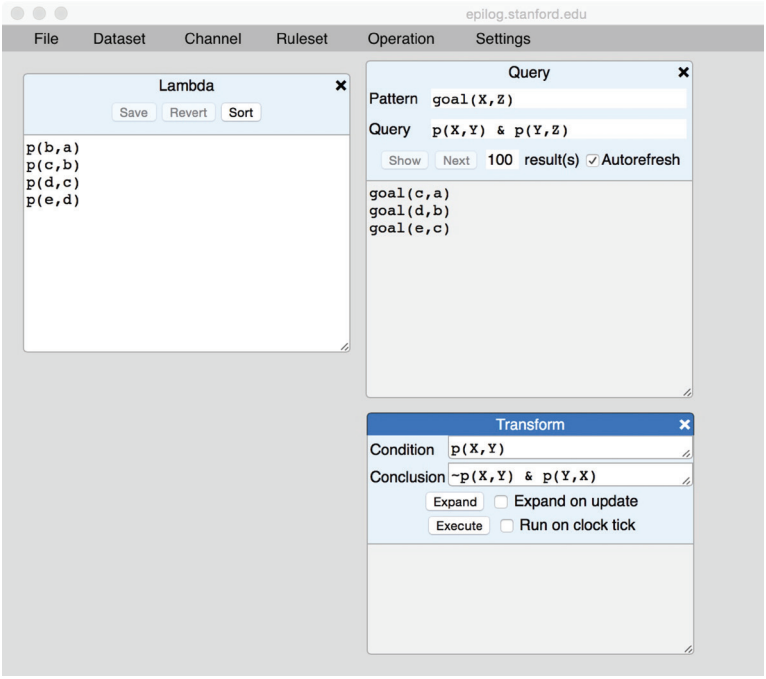

Рис. Б.18. Действие кнопки **Execute**

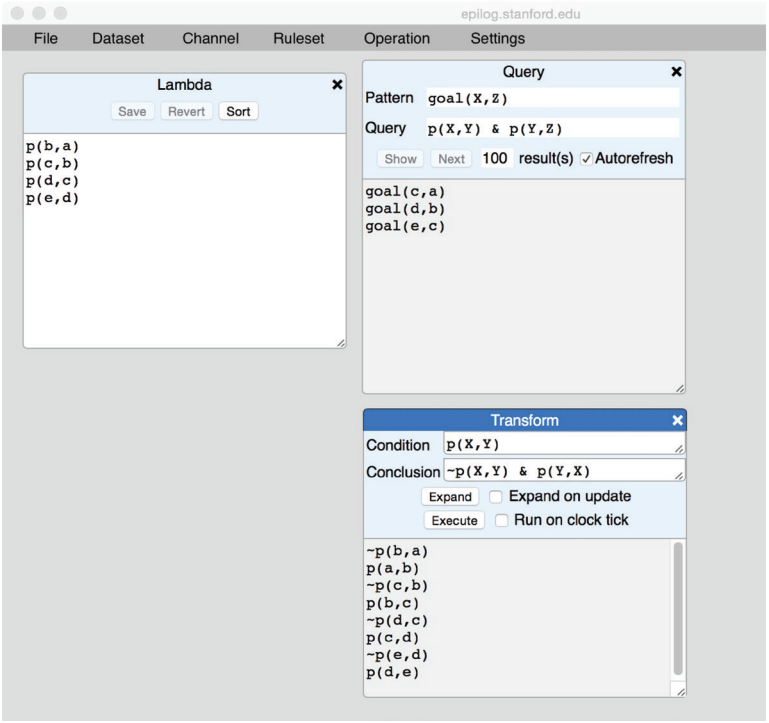

Рис. Б.19. Действие кнопки **Expand**

Флажок **Expand on update** (Расширить при обновлении) направляет Sierra на обновление изменений, которые потребуются при изменении базы данных. Например, если установить этот флажок и нажать кнопку **Execute**, база данных вернется в исходное состояние и отобразит другой набор изменений (показан на рис. Б.20).

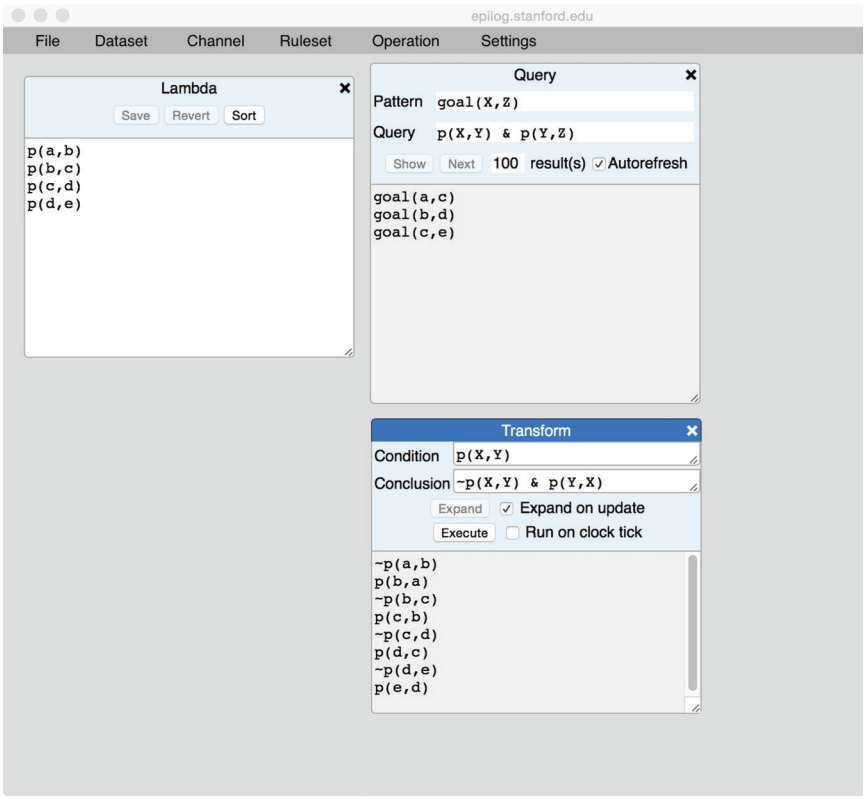

Рис. Б.20. Действие **Expand on update** + **Execute**

Наконец, флажок **Run on clock tick** (Запустить при тиканьи часов) позволяет запланировать выполнение правила трансформации каждый раз, когда тикают часы. В данном случае это приведет к тому, что порядок аргументов фактов в наборе данных будет колебаться туда-сюда. Этот случай не слишком интересен, но обновление по тику часов часто бывает полезно при моделировании динамических систем.

Б.6. Определения представлений

Представления в логическом программировании – это эффективно именованные запросы. Важным преимуществом этого является композиция. Как только у нас есть имя для представления, мы можем исполь-

зовать это имя при определении других представлений. Более того, мы можем использовать имя при определении самого представления, т. е. при определении рекурсивных представлений. Определения представлений выражаются путем добавления правил в набор правил. В Sierra доступ к наборам правил осуществляется через меню **Ruleset** (Набор правил).

Нажав на меню набора правил, мы видим только один вариант – **Library** (Библиотека). Это набор правил по умолчанию (в продвинутых версиях Sierra можно управлять несколькими наборами правил, но эта функция не включена в базовой версии, показанной здесь).

Давайте начнем с нажатия на **Library**. Откроется окно, показывающее содержимое набора правил, названного **Library**. Как и в случае с набором данных **Lambda**, он изначально пуст.

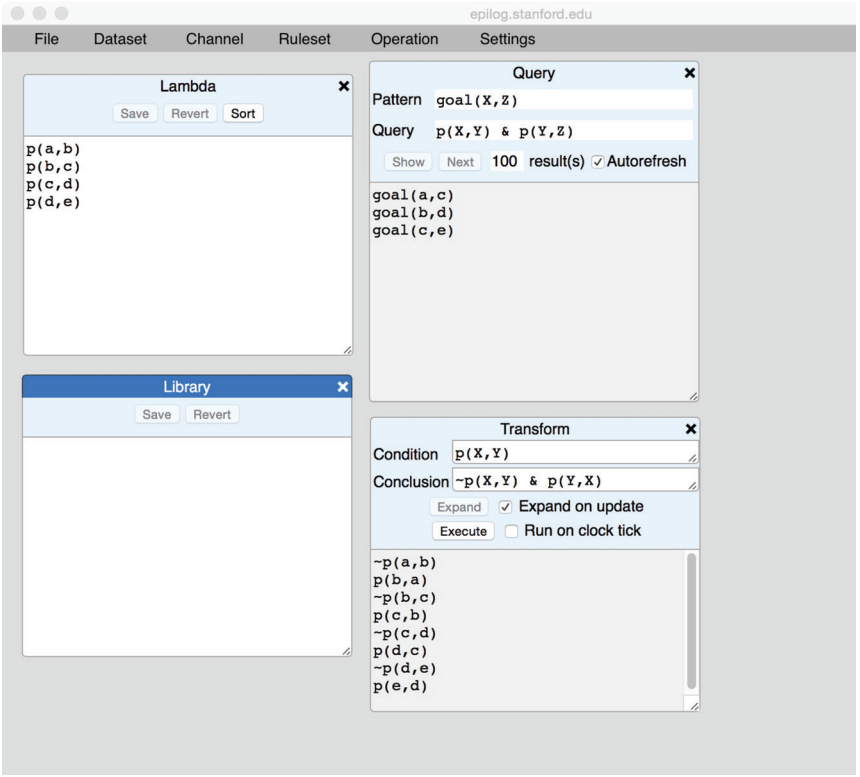

Рис. Б.21. Открыт набор правил **Library**

Поскольку это окно правил, мы можем вводить правила, набирая текст в окне. Здесь, например, мы определили отношение предков anc в терминах родительского отношения `p` (см. рис. Б.22).

Как и в случае с **Lambda**, нужно нажать кнопку **Save** (Сохранить), чтобы наше определение было записано в набор правил (см. рис. Б.23).

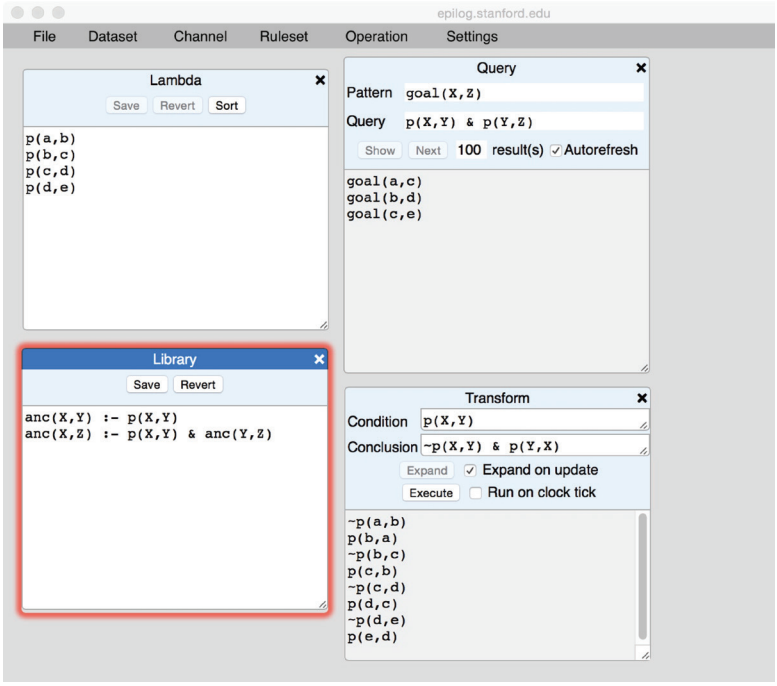

Рис. Б.22. Ввод нового правила

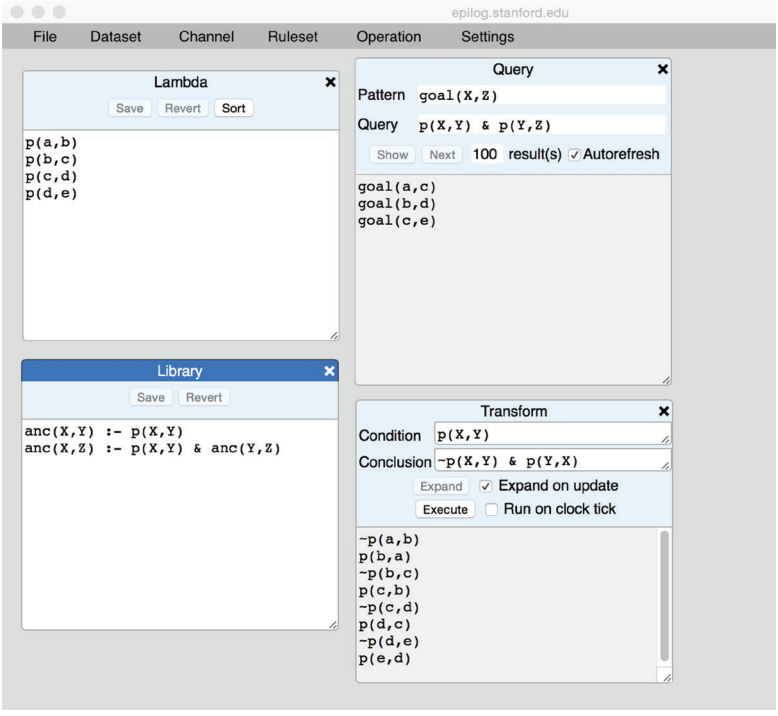

Рис. Б.23. Сохранение нового правила

После того как представление определено, возможно использовать его в запросах и преобразованиях. Например, мы можем открыть другое окно запроса и написать запрос с помощью anc. Однако существует более простой инструмент для этой цели, а именно **Compute** (Вычислить). Здесь мы нажали на кнопку **Compute** (в меню **Operation**) и получили пустое окно **Compute** (см. рис. Б.24).

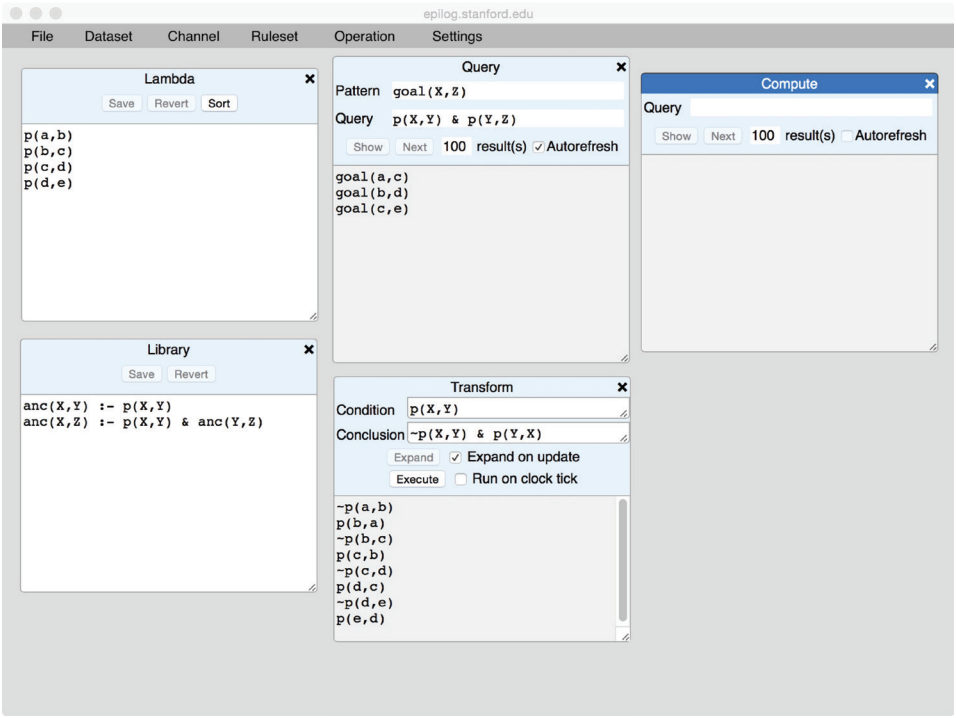

Рис. Б.24. Открыто окно **Compute**

Если мы введем anc(b,Z) в поле запроса и нажмем **Show** (Показать), получим список всех фактов, в которых b фигурирует в качестве первого аргумента (см. рис. Б.25).

Как и в случае с запросами, если мы отметим флажок **Autorefresh**, Sierra будет поддерживать отображение в актуальном состоянии по мере внесения изменений. Например, если мы добавим дополнительный факт в **Lambda**, набор ответов будет обновлен, как показано на рис. Б.26 (обратите внимание, что окна **Query** и **Transform** также были обновлены).

Как и в случае с запросами, можно запросить определенное количество ответов для показа и переходить от одного ответа к другому с помощью кнопки **Next** (Далее).

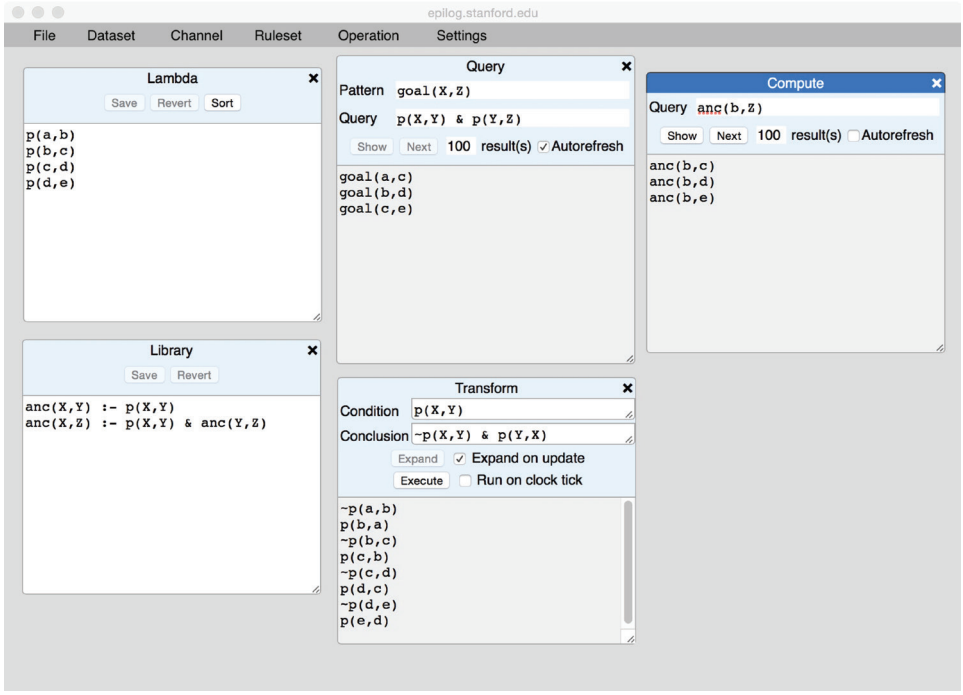

Рис. Б.25. В поле запроса введено **anc(b,Z)**

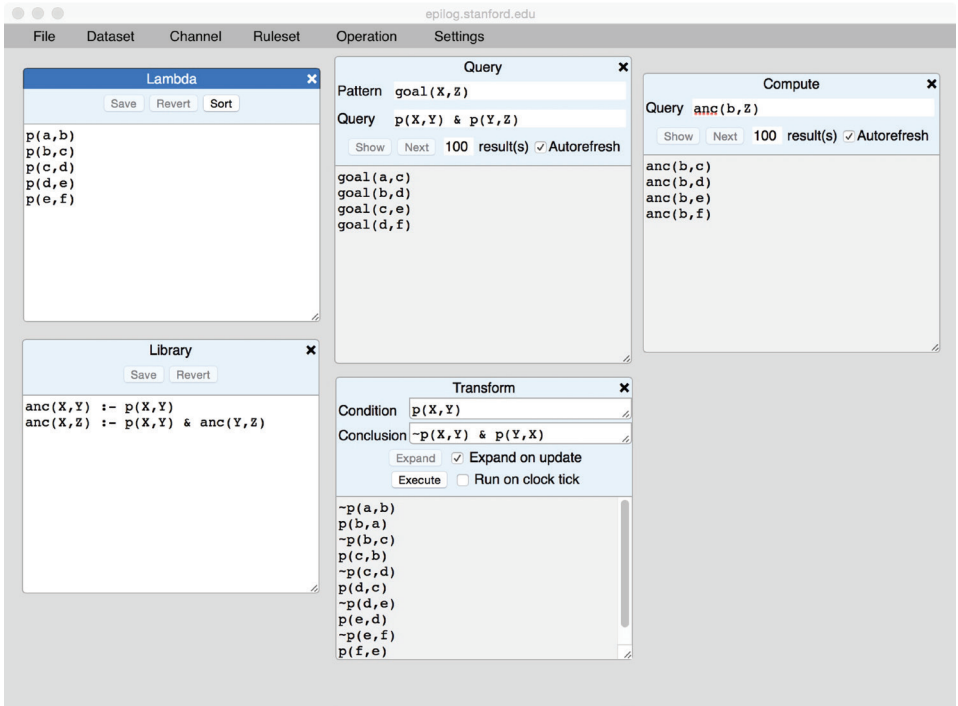

Рис. Б.26. Отмечен флажок **Autorefresh** и добавлен факт в **Lambda**

Б.7. Определения операций

Операции в логическом программировании являются эффективными именованными преобразованиями. Как и в случае с представлениями, их важным преимуществом является композиция. Если у нас есть именованная операция, мы можем использовать ее имя в определении других операций. Более того, мы можем использовать его и при определении самой операции, т. е. в определении рекурсивных операций.

Определения операций выражаются путем добавления правил в набор правил, в данном случае в **Library**. Здесь, например, мы определили операцию `purge` (см. рис. Б.27). При выполнении операции `purge(X)` Sierra удаляет все дочерние элементы `p`, все дочерние элементы дочерних элементов и т. д.

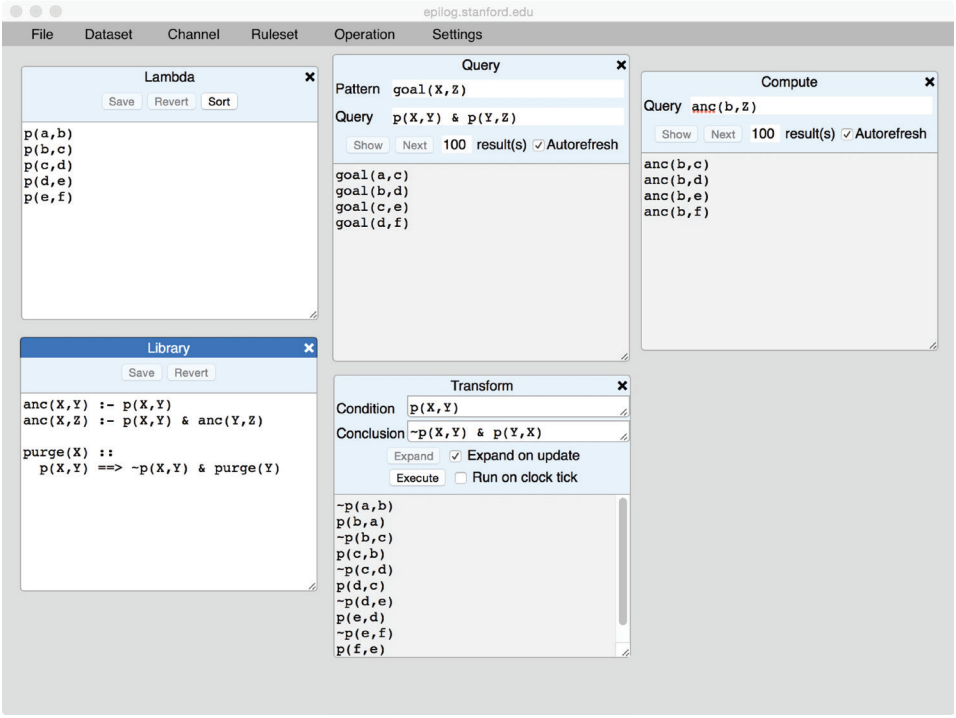

Рис. Б.27. Определение операции purge

Определив операцию, можно использовать ее в преобразованиях. Мы можем, например, открыть другое окно преобразования и записать purge в качестве вывода. Однако опять же для этой цели есть упрощенный инструмент, а именно **Execute** (Выполнить). Здесь мы выбрали **Execute** в меню **Operation** и получили пустое окно **Execute** (см. рис. Б.28).

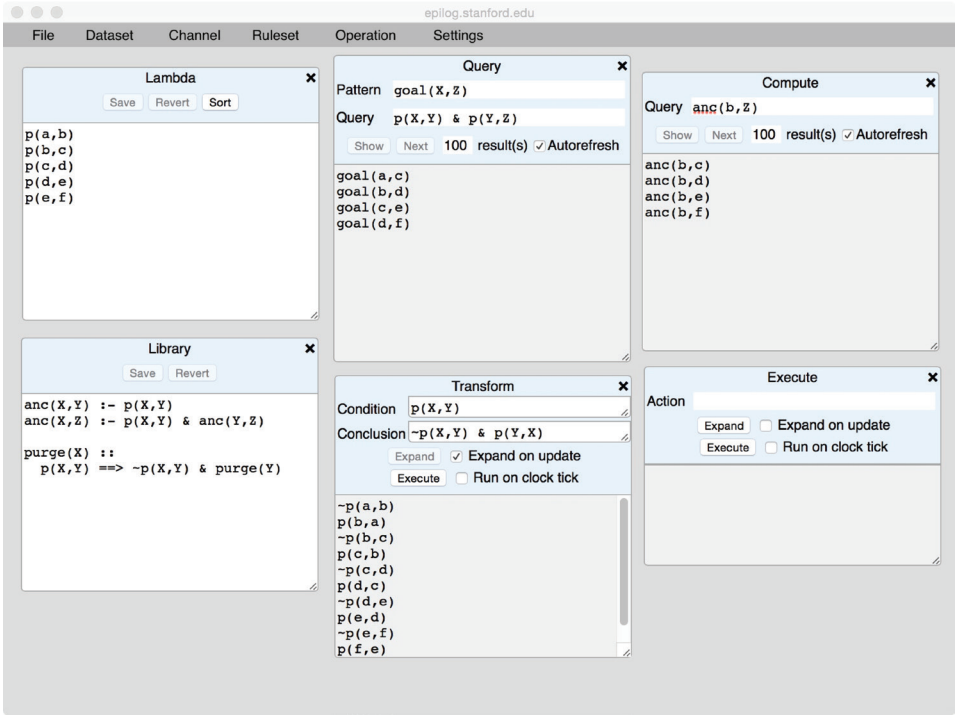

Рис. Б.28. Открыто окно **Execute**

Если мы введем purge(c) в поле **Action** (Действие) и нажмем **Expand**, мы увидим список всех фактов, которые будут удалены, если мы выполним `purge(c)` (см. рис. Б.29). Обратите внимание, что в случае выполнения операции эти факты будут удалены за один шаг, т. е. выполнение операции является «атомарным» действием.

Как и в случае с инструментом **Transform**, у нас есть возможности расширения при обновлении и выполнения по тику часов. Если в этот момент мы нажмем кнопку **Execute** (Выполнить), Sierra удалит указанные факты и обновит все окна соответствующим образом, как показано на рис. Б.30.

Хотя в данном примере это не показано, определения операций чрезвычайно полезны в качестве обработчиков событий для интерактивных пользовательских интерфейсов, например рабочих листов в браузере.

Б.8. Настройки

Меню **Settings** (Настройки) позволяет управлять механизмом вывода Sierra. Щелчок на **Queries** (Запросы) вызывает панель взаимодействия, которая позволяет указать число шагов вывода, выполняемых для отдельного запроса, прежде чем система прекратит работу (см. рис. Б.31).

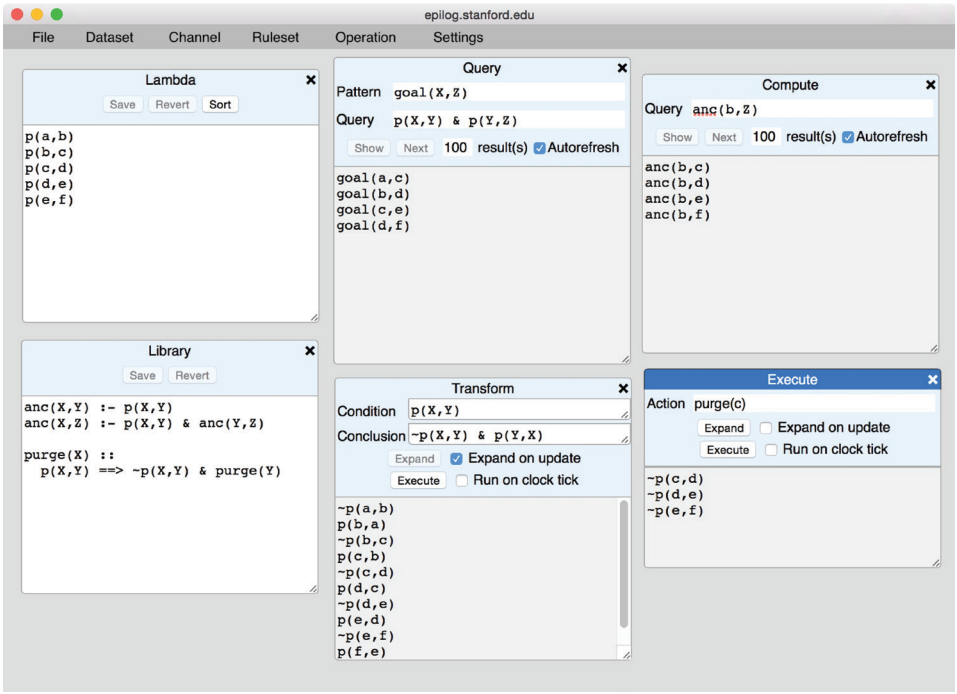

Рис. Б.29. Введено **purge(c)** и нажата кнопка **Expand**

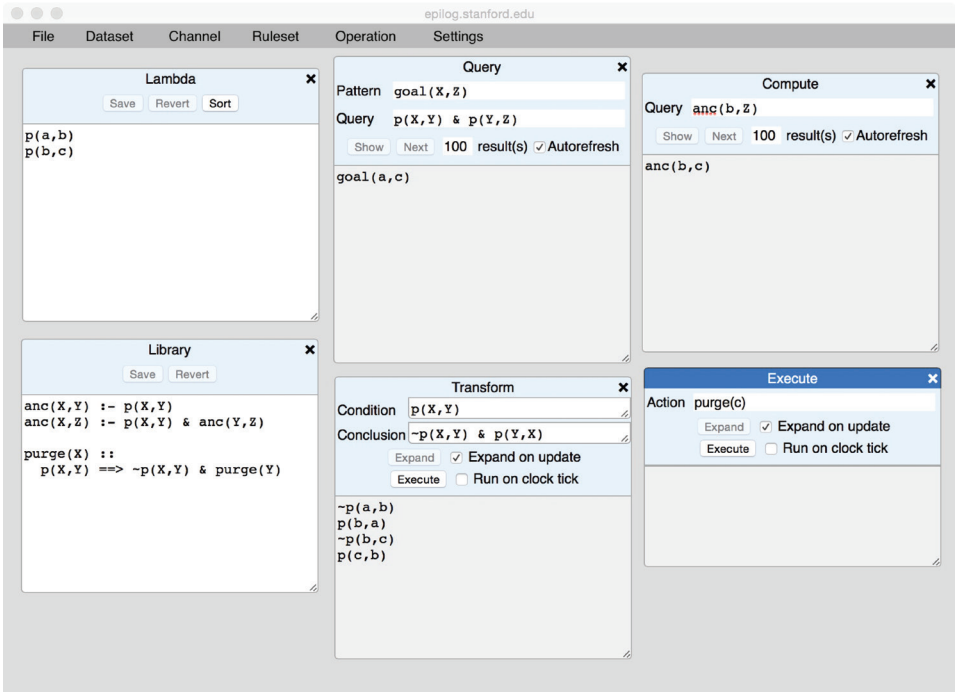

Рис. Б.30. Активирован флажок **Expand on update** и нажата кнопка **Execute**

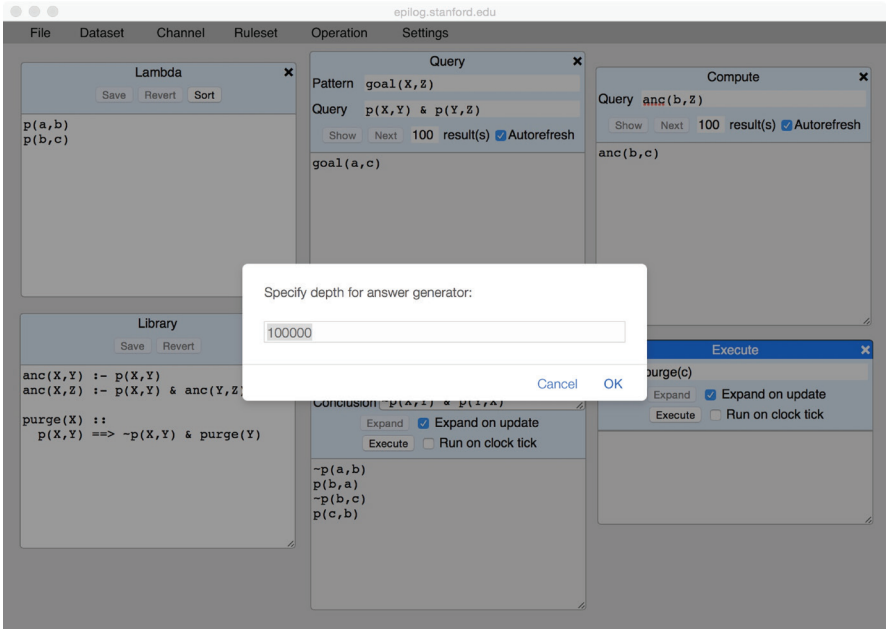

Рис. Б.31. Указано число шагов вывода

Нажатие на **Transitions** (Переходы) вызывает панель взаимодействия, которая позволяет указать глубину рекурсии при расширении определений операций (см. рис. Б.32).

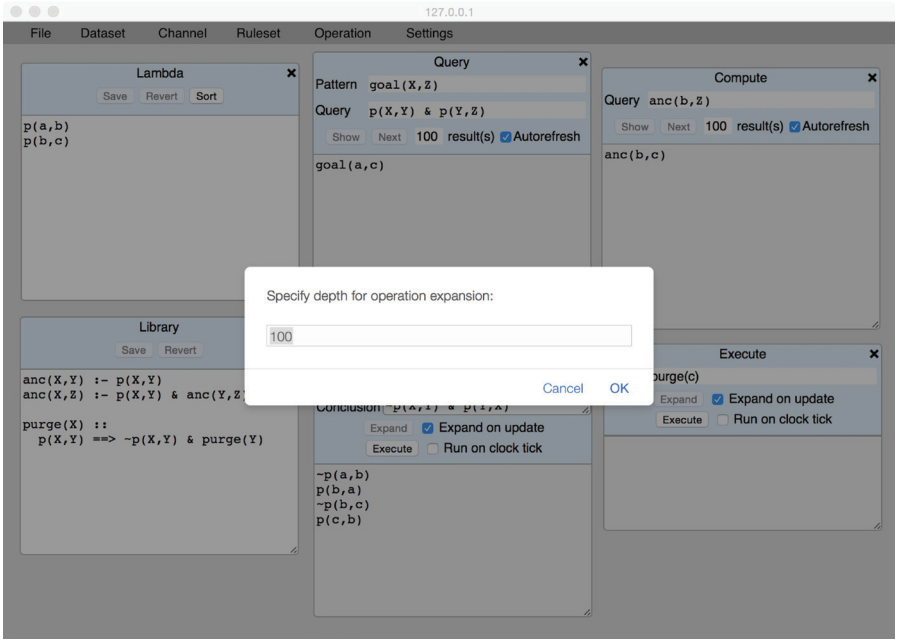

Рис. Б.32. Указана глубина рекурсии при расширении определений операций

При нажатии на кнопку **Timer** появляется окно, позволяющее установить таймер. Это позволит запустить операции в окнах **Transform** и **Execution**, где мы установили флажок **Run on clock tick** (см. рис. Б.33).

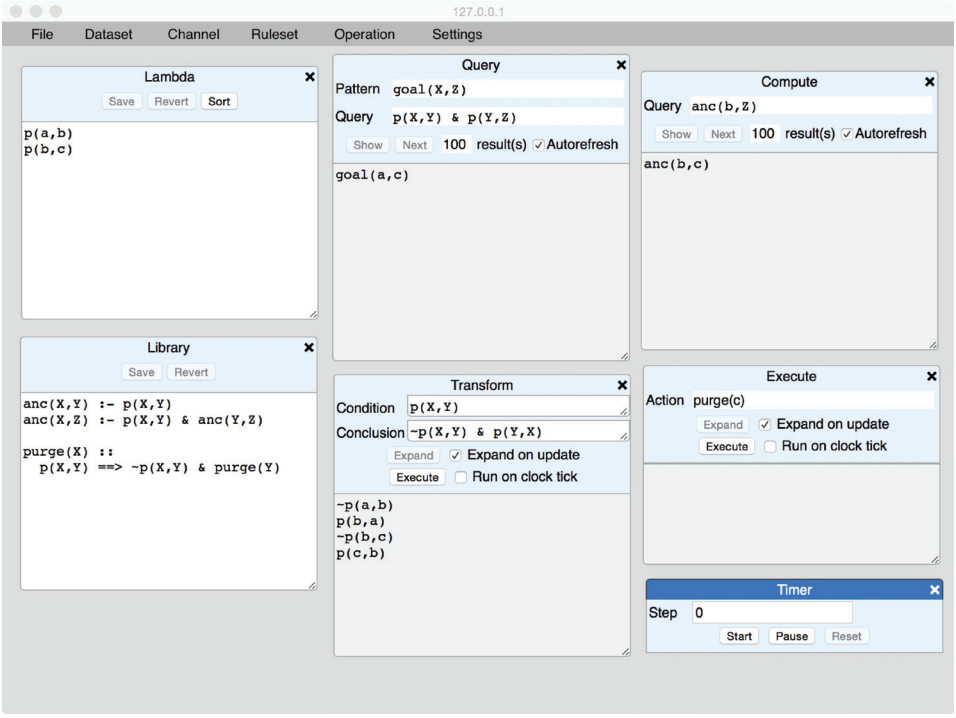

Рис. Б.33. Установка таймера

Б.9. Управление файлами

Опция **Load** (Загрузить) в меню **File** позволяет читать наборы данных и наборы правил из локальной файловой системы и загружать их в **Library**, **Lambda** или именованные наборы данных. Опции **Save** (Сохранить) позволяют нам записывать данные из любого окна в локальную файловую систему.

Опция **Save Configuration** (Сохранить конфигурацию) позволяет сохранить состояние Sierra в локальной файловой системе, включая данные, правила, настройки, открытые окна и т. д. Опция **Load configuration** (Загрузить конфигурацию) позволяет загрузить ранее сохраненный файл конфигурации. Эти операции чрезвычайно полезны при разработке приложений логического программирования. Мы можем остановить работу и продолжить ее с того места, где остановились. И можем обмениваться демонстрациями, передавая файлы конфигурации.

Б.10. Заключение

В дополнение к описанным здесь возможностям Sierra предоставляет инструменты для управления каналами связи, которые позволяют передавать информацию между Sierra и внешними источниками данных, а также обеспечивают взаимодействие между различными воплощениями Sierra, запущенными на одной и той же машине или на других машинах. К сожалению, детали являются довольно сложными, и поэтому мы их пропустили в этом простом введении.

Наконец, стоит отметить, что существует расширение Sierra, известное как Halle, которое предназначено для использования при разработке интерактивных, веб-ориентированных рабочих листов. В дополнение к описанным здесь возможностям Halle предоставляет инструменты для компоновки рабочих листов в режиме WYSIWYG, отображения этих рабочих листов в виде отдельных окон на одной веб-странице, подобной Sierra, и взаимодействия с рабочими листами. Это позволяет авторам просматривать и редактировать базовые данные и правила в среде, подобной Sierra.

Комментарии и жалобы направлять по адресу **genesereth@stanford.edu**.

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать
адрес (полностью), по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com.

Майкл Дженесерет
Винай К. Чаудри

Введение в логическое программирование

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Минц С. В.*
Корректор *Абросимова Л. А.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Печать цифровая.
Усл. печ. л. 15.6. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

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.

Майкл Дженесерет *– профессор факультета компьютерных наук и по совместительству профессор юридического факультета Стэнфордского университета. Он наиболее известен благодаря своей работе по вычислительной логике и применению этой работы в управлении предприятиями, вычислительном праве и общих играх. Дженесерет является одним из основателей Teknowledge, CommerceNet, Mergent Systems и Symbium.*

Винай К. Чаудри *является признанным экспертом в области искусственного интеллекта, включая представление знаний и рассуждения, ответы на вопросы, онтологии и приобретение знаний. В Стэнфорде его деятельность включает продвижение логического образования для средних школ, изучение методов быстрого получения формальных знаний и создание интеллектуальных учебников.*

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

ДМК ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-968-2