

THE EXPERT'S VOICE® IN PROGRAMMING LANGUAGES

Practical Common Lisp

"that book is dead sexy"

—Nathan Flap

"Two prehensile toes up!"

—Kenny Tiboni, sampling, loop-de-loop, reporting on behalf of his development team

Peter Seibel

apress®

1. Введение: почему Lisp?

Если вы считаете, что величайшее удовольствие в программировании доставляет код, делающий многое и выражающий ваши желания просто и ясно, тогда программирование на Common Lisp вероятно будет самым приятным из того, что можно сделать на компьютере. С Common Lisp вы сделаете больше и быстрее, чем с большинством других языков программирования.

Серьезное заявление. Могу ли я доказать это? Да, но не на нескольких страницах введения. Вам придётся познакомиться с Lisp поближе и убедиться в этом самим — поэтому читайте книгу до конца. А сейчас позвольте мне начать с нескольких смешных эпизодов из истории моего пути к языку Lisp. В следующей главе я объясню выгоды, которые вы получите от изучения Common Lisp.

Я — один из немногих Lisp-хакеров второго поколения. Мой отец начал заниматься компьютерами с написания на ассемблере операционной системы для машины, которую он использовал для сбора данных при подготовке его докторской диссертации по физике. После работы с компьютерами в разных лабораториях физики, к 80-м, отец полностью оставил физику и стал работать в большой фармацевтической компании. У этой компании был проект по созданию программы, моделирующей производственные процессы на химических заводах (если вы увеличите эту емкость, как это повлияет на годовой результат?). Старая команда писала всё на языке FORTRAN, использовала половину бюджета и почти всё отведённое время, но не могла продемонстрировать никаких результатов. Это было в 80-х, пик бума искусственного интеллекта (ИИ), Lisp так и витал в воздухе. Так что мой папа — в то время еще не поклонник языка Lisp — пошёл в университет Карнеги-Меллона, чтобы пообщаться с людьми, работавшими над тем, что впоследствии стало Common Lisp, и узнать, поможет ли Lisp его проекту.

Ребята из университета показали ему кое-что из своих разработок, и это его убедило. Отец, в свою очередь, убедил своих боссов позволить ему взять провальный проект и сделать его на Lisp. Год спустя, используя остатки бюджета, команда отца представила работающее приложение, обладающее возможностями, на реализацию которых старая команда уже и не надеялась. Мой папа объясняет, что причина успеха — в решении использовать Lisp.

Однако, это всего лишь первый эпизод. Может быть, мой отец ошибался в причине своего успеха. Или, может быть, Lisp был лучше других языков того времени. В настоящее время мы имеем кучу новых языков программирования, многие из которых переняли часть достоинств Lisp. Действительно ли я считаю, что использование языка Lisp может дать вам те же выгоды, что и моему отцу в 80-х? Читайте дальше.

Несмотря на все усилия моего отца, я не изучал Lisp в университете. После учёбы, которая не содержала много программирования на каком-либо языке, я был покорен Web и вернулся назад к компьютерам. Сначала я писал на Perl, изучив его достаточно, чтобы создать форум для сайта журнала Mother Jones, после этого я работал над большими (по тем временам) сайтами, такими, как, например, сайт компании Nike, запущенный к олимпийским играм 1996 года. После этого я перешёл на Java, будучи одним из первых разработчиков в WebLogic (теперь эта компания — часть BEA). После WebLogic я участвовал в другом стартапе, где был ведущим программистом по построению транзакционной системы обмена сообщениями на Java. Со временем мои основные интересы в программировании позволили мне использовать как популярные языки, такие, как C, C++ и Python, так и менее известные, такие, как Smalltalk, Eiffel и Beta.

Итак, я знал два языка вдоль и поперёк и был поверхностно знаком с несколькими другими языками. В конечном счёте я понял, что источником моего интереса к языкам программирования является идея, высказанная моим отцом, когда он рассказывал о Lisp, — идея о том, что разные языки программирования на самом деле различны, и, несмотря на

формальное равенство всех языков программирования по Тьюрингу, вы действительно можете быстрее достигнуть больших результатов, используя одни языки вместо других, и при этом получить больше удовольствия. Тем не менее, я пока не изучил сам Lisp. Так что я начал потихоньку учить его в свободное время. Меня воодушевляло то, насколько быстро я проходил путь от идеи к работающему коду.

Например, в одном из отпусков, имея около недели на опыты с Lisp, я решил попробовать написать версию программы, написанной мною на Java в начале программистской карьеры. Эта программа применяла генетические алгоритмы для игры в Го. Даже с моими зачаточными знаниями Common Lisp написание всего-лишь основных функций было намного продуктивнее, чем если бы я решил переписать всё на Java заново. Для написания программы на Java потребовалось несколько лет работы с этим языком.

Похожий эксперимент привёл к созданию библиотеки, о которой я расскажу в главе 24. В начале моей карьеры в WebLogic я написал библиотеку на Java для разбора java-классов (файлов *.class). Она работала, но код был запутан, и его трудно было изменить или добавить новую функциональность. В течение нескольких лет я пытался переписать библиотеку, думая, что смогу использовать мои новые знания в Java и не увязнуть в куче дублирующегося кода, но так и не смог. Когда же я попробовал написать её на Common Lisp, это заняло всего 2 дня, и я получил не просто библиотеку для разбора java-классов, но библиотеку для разбора любых двоичных файлов. Вы увидите, как она работает, в главе 24, и используете её в главе 25 для разбора тэгов ID3 в MP3-файлах.

Почему Lisp?

Сложно объяснить на нескольких страницах введения, почему пользователи языка любят какой-то конкретный язык, ещё сложнее объяснить, почему вы должны тратить своё время на его изучение. Личный пример не слишком убеждает. Может быть, я люблю Lisp, потому что какая-то цепь в моём мозгу замкнулась. Это может быть даже генетическим отклонением, так как мой отец похоже тоже имел его. Так что прежде, чем вы погрузитесь в изучение языка Lisp, вполне естественным кажется желание узнать, что это вам даст, какую выгоду принесёт.

Для некоторых языков выгода очевидна. Например, если вы хотите писать низкоуровневые программы для Unix, то должны выучить C. Или если вы хотите писать кросс-платформенные приложения, то должны использовать Java. И большое число компаний до сих пор использует C++, так что если вы хотите получить работу в одной из них, то должны знать C++.

Тем не менее, для большинства языков выгоду не так просто выделить. Мы имеем дело с субъективными оценками того, насколько язык удобно использовать. Защитники Perl любят говорить, что он "делает простые вещи простыми, а сложные - возможными" и радуются факту, озвученному в девизе Perl - "Есть более чем один способ сделать это". С другой стороны, фанаты языка Python думают, что Python – прозрачный и простой язык, и код на Python проще понять, потому что, как гласит их лозунг, "Есть лишь один способ сделать это".

Так почему же Common Lisp? Здесь нет такой очевидной выгоды, как для C, Java или C++ (конечно, если вы не являетесь счастливым обладателем Lisp-машины). Выгоды от использования Lisp заключены в переживаниях и впечатлениях от его использования. В остальной части книги я буду показывать отличительные черты языка, так что вы сможете по себе оценить, на что эти впечатления похожи. Сейчас я попытаюсь показать смысл философии Lisp.

В качестве девиза для Common Lisp лучше всего подходит похожее на дзенский коан описание "программируемый язык программирования". Хотя и несколько запутанный, данный девиз, тем не менее, выделяет суть преимущества, которое Lisp до сих пор имеет перед другими языками программирования. Больше, чем другие языки, Common Lisp следует философии: что хорошо

для разработчика языка, то хорошо для его пользователей. Программируя на Common Lisp, вы, скорее всего, никогда не обнаружите нехватки каких-то возможностей в языке, которые упростили бы программирование, потому что, как будет показано далее, вы можете просто добавить эти возможности в язык.

Следовательно, программы на Common Lisp стараются предоставить наиболее прозрачное отображение между вашими идеями о том, как программа должна работать, и кодом, который вы пишете. Ваши идеи не замутняются нагромождением кода и бесконечно повторяющимися выражениями. Это делает ваш код более управляемым, потому что вам больше не приходится бродить по нему всякий раз, когда вы хотите внести какие-то изменения. Даже систематические изменения в программе могут быть достигнуты относительно малыми изменениями исходного кода. Это также означает, что вы будете писать код быстрее; вы будете писать меньше кода и не будете терять время на поиск пути для выражения своих идей в ограничениях, накладываемых языком программирования.

Common Lisp — это также прекрасный язык для исследовательского программирования (прототипирования?), когда вам неизвестно достоверно, как ваша программа должна работать. Common Lisp предоставляет некоторые возможности, помогающие вам вести инкрементальную интерактивную разработку.

Интерактивный цикл read-eval-print, о котором я расскажу в следующей главе, позволяет вам непрерывно взаимодействовать с вашей программой во время её разработки. Пишете новую функцию. Тестируете её. Меняете её. Проверяете другие подходы к реализации. Вам не приходится останавливаться для длительной компиляции.

Другими особенностями, которые поддерживают непрерывный, интерактивный стиль программирования, являются динамическая типизация Lisp и система обработки условий в Lisp. Первое позволяет вам тратить меньше времени на убеждение компилятора в том, что вам можно запустить программу, и больше времени на её действительный запуск и работу с ней. Последнее позволяет интерактивно разрабатывать даже код обработки ошибок.

Другим следствием того, что Lisp — "программируемый язык" является то, что, кроме возможности вносить мелкие изменения в язык, которые позволяют легче писать программы, есть возможность без труда отражать в языке значительные, новые понятия, касающиеся общего устройства языков программирования. Например, оригинальная реализация Common Lisp Object System (CLOS) — объектной системы Common Lisp, была библиотекой, написанной на самом Common Lisp. Это позволило Lisp программистам получить реальный опыт работы с возможностями, которые она предоставляла, еще до того момента, когда библиотека была официально включена в состав языка.

Какая бы новая парадигма программирования не появилась, Common Lisp, скорее всего, без труда сможет впитать её без изменений в ядре языка. Например, один программист на Lisp недавно написал библиотеку AspectL, которая добавляет Common Lisp поддержку аспектно-ориентированного программирования (AOP). Если будущее за AOP, то Common Lisp сможет поддерживать его без изменений в базовом языке и без дополнительных препроцессоров и прекомпиляторов.

Как это началось?

Common Lisp — современный потомок языка программирования Lisp, придуманного Джоном Маккарти в 1956 году. Lisp был создан для "обработки символьных данных" и получил своё имя от одной вещи, в которой он был очень хорош: обработки списков (LISt Processing). Много воды утекло с тех пор, и теперь Common Lisp обогащён набором современных типов данных, которые вам только могут понадобиться, а также системой обработки ситуаций, которая, как вы увидите

в главе 19, предоставляет уровень гибкости, отсутствующий в системах обработки исключений таких языков, как C++, Java, Python; мощной системой объектно-ориентированного программирования; несколькими особенностями, которых нет ни в одном другом языке. Как такое возможно? Что, скажите, обусловило превращение Lisp в такой богатый язык?

Маккарти был (и есть) исследователем в области искусственного интеллекта, и многие особенности, которые он заложил в первую версию, сделали этот язык замечательным инструментом для программирования искусственного интеллекта. Во время бума ИИ в 80-е Lisp оставался излюбленным языком для решения сложных проблем, как то: автоматическое доказательство теорем, планирование и составление расписаний, компьютерное зрение. Это были проблемы, требующие сложных программ, для написания которых нужен был мощный язык, так что программисты ИИ сделали Lisp таковым. Помогла и Холодная война, т.к. Пентагон выделял деньги Управлению перспективных исследовательских программ (DARPA), часть этих денег попадала к людям, занимающимся моделированием крупных сражений, автоматическим планированием и интерфейсами на естественных языках. Эти люди также использовали Lisp и продолжали совершенствовать его, чтобы язык полностью удовлетворял их потребностям.

Те же силы, что развивали Lisp, также расширяли границы и в других направлениях — сложные проблемы ИИ требуют больших вычислительных ресурсов, как бы вы их ни решали, и если вы примените закон Мура в обратном порядке, то сможете себе представить, сколь скудными эти ресурсы были в 80-е. Так что разработчики должны были найти все возможные пути улучшения производительности их реализаций языка. В результате этих усилий современные реализации Common Lisp часто включают в себя сложные компиляторы в язык, понятный машине. Хотя сегодня, благодаря закону Мура, возможно получить высокую производительность даже интерпретируемых языков, это больше не является проблемой для Common Lisp. И, как я покажу в главе 32, используя специальные (дополнительные) объявления, с помощью хорошего компилятора можно получить вполне приличный машинный код, сравнимый с тем, который выдаст компилятор C.

80-е — это также эра Lisp-машин. Несколько компаний, самая известная из которых Symbolics, выпускали компьютеры, которые могли запускать непосредственно Lisp-код на своих чипах. Так Lisp стал языком системного программирования, который использовали для написания операционных систем, текстовых редакторов, компиляторов и много чего еще, что можно было запустить на Lisp-машине.

Фактически, к началу 80-х существовало множество Lisp-лабораторий и несколько компаний, каждая со своей реализацией Lisp, их было так много, что люди из DARPA стали высказывать свои опасения о разобщенности Lisp-сообщества. Чтобы достигнуть единства, группа Lisp-хакеров собралась вместе и начала процесс стандартизации нового языка, Common Lisp, который бы впитал в себя лучшие черты существующих диалектов. Их работа запечатлена в книге Common Lisp the Language Гая Стила (Guy Steele, Digital Press, 1984) (CLtL).

К 1986 году существовало несколько реализаций стандарта, призванного заменить разобщенные диалекты. В 1996 организация The American National Standards Institute (ANSI) выпустила стандарт, расширяющий Common Lisp на базе CLtL, добавив в него новую функциональность, такую, как CLOS и систему обработки условий. Но и это не было последним словом: как CLtL до этого, так и стандарт ANSI теперь целенаправленно позволяет разработчикам реализаций экспериментировать с тем, как лучше сделать те или иные вещи: реализация Lisp содержит богатую среду исполнения с доступом к графическому пользовательскому интерфейсу, многопоточности, сокетам TCP/IP и многому другому. В наши дни Common Lisp эволюционирует, как и большинство других языков с открытым кодом: люди, использующие его, пишут библиотеки, которые им необходимы, и часто делают их доступными для всего сообщества. В последние годы, в частности, замечается усиление активности в разработке для

Lisp библиотек с открытым кодом.

Так что, с одной стороны, Lisp — один из классических языков в информатике (Computer Science), базирующийся на идеях, проверенных временем. С другой стороны, Lisp — современный язык общего назначения, с дизайном, отражающим прагматический подход к решению сложных задач с максимальной надёжностью и эффективностью. Единственным недостатком "классического" наследия Лиспа является то, что многие все еще топчутся вокруг представлений о Лиспе, основанных на определенном диалекте этого языка, который они открыли для себя в середине прошлого столетия в то время, когда Маккарти разработал Лисп. Если кто-то говорит вам, что Lisp — только интерпретируемый язык, что он медленный, или что вы обязаны использовать рекурсию буквально для всего, спросите вашего оппонента, какой диалект Lisp'a имеется в виду, и носили ли люди клёш, когда он изучал Lisp.

Но я изучал Lisp раньше, и он не был тем, что вы описываете!

Если вы изучали Lisp в прошлом, то можете подумать, что тот Lisp не имеет ничего общего с Common Lisp. Хотя Common Lisp вытеснил большинство диалектов, от которых он был порождён, это не единственный сохранившийся диалект, и, в зависимости от того, где и когда вы встретились с Lisp, вы могли хорошо изучить один из этих, отличных от Common Lisp, диалектов.

Кроме Common Lisp, активное сообщество пользователей есть у диалекта Lisp общего назначения под названием Scheme. Common Lisp позаимствовал из Scheme несколько важных особенностей, но никогда не пытался заменить его.

Разработанный в Массачуссетском Технологическом Институте (MIT), Scheme был быстро принят в качестве языка для начальных курсов по вычислительной технике. Scheme изначально занимал отдельную нишу, в частности, проектировщики языка постарались сохранить ядро Scheme настолько малым, насколько это возможно. Это давало очевидные выгоды при использовании Scheme как языка для обучения, а также при исследованиях в области языков программирования, так давало возможность формального доказательства предположений о языке.

Существовало также ещё одно преимущество: язык легко можно было изучить по спецификации. Все эти преимущества достигнуты за счёт отсутствия многих удобных особенностей, стандартизированных в Common Lisp. Конкретные реализации Scheme могут предоставлять эти возможности, но такие отклонения от стандарта делают написание переносимого кода на Scheme более сложным, чем на Common Lisp.

В Scheme гораздо большее внимание, чем в Common Lisp, уделяется функциональному стилю программирования и использованию рекурсии. Если вы изучали Lisp в университете и остались с впечатлением, что это академический язык без возможности применения в реальной жизни, существует вероятность, что вы изучали именно Scheme. Нельзя сказать, что это правдивая характеристика Scheme, но это определение гораздо менее подходит для Common Lisp, который создавался для реальных инженерных задач, нежели для теоретизирования.

Также, если вы изучали Scheme, вас могут сбить с толку некоторые различия между Scheme и Common Lisp. Эти различия являются поводом непрекращающихся религиозных войн между горячими парнями, программирующими на этих диалектах. В данной книге я постараюсь указать на наиболее существенные различия.

Двумя другими распространёнными диалектами Lisp являются ELisp, язык расширений для редактора Emacs, и Autolisp, язык расширений для программы Autodesk AutoCAD. Хотя,

возможно, суммарный объём кода, написанного на этих диалектах, перекрывает весь остальной код, написанный на Lisp, оба эти диалекта могут использоваться только в рамках приложений, которые они расширяют. Кроме того, они являются устаревшими по сравнению и с Common Lisp, и с Scheme. Если Вы использовали один из этих диалектов, приготовьтесь к путешествию на Lisp-машине времени на несколько десятилетий вперёд.

Для кого эта книга?

Эта книга для вас, если вы интересуетесь Common Lisp, независимо от того, знаете ли вы его или просто хотите понять, из-за чего вокруг него разгорелась вся эта шумиха.

Если вы уже изучали Lisp, но не смогли перейти от академических упражнений к созданию реальных полезных программ, эта книга покажет вам путь для такого перехода. С другой стороны, вы не обязаны желать применять Lisp для того, чтобы получить пользу от данной книги.

Если вы упёртый прагматик, желающий знать достоинства Common Lisp перед другими языками, такими, как Perl, Python, Java, C или C#, эта книга даст вам несколько идей по этому поводу. Или, может быть, вам нет никакого дела до использования Lisp и вы уверены, что он ничуть не лучше языков, которые вы уже знаете, но вам надоели заявления какого-нибудь Lisp-программиста, что вы просто не поняли его как следует. Если так, то в данной книге вы найдёте краткое введение в Common Lisp. Если после чтения этой книги вы по-прежнему будете думать, что Common Lisp ничем не лучше, чем ваши любимые языки, у вас будут веские обоснованные аргументы.

В книге описывается не только синтаксис и семантика языка, но и реальные способы написания на нём полезных программ. В первой части книги я описываю сам язык и даю несколько практических примеров написания на нём реальных программ. Затем, после описания большей части языка, включая несколько областей, предложенных в других книгах для самостоятельного изучения, следует девять практических глав, в которых я помогу вам написать несколько программ среднего размера, выполняющих полезную работу: фильтрацию спама, разбор двоичных файлов, каталогизацию MP3, вещание MP3 по сети, создание веб-интерфейса к каталогу MP3 на сервере.

После окончания чтения книги вы будете знакомы с большинством важнейших возможностей языка и с тем, как их следует использовать. Вы будете иметь опыт использования Common Lisp для написания нетривиальных программ и будете готовы к дальнейшему самостоятельному изучению языка. И, хотя у каждого свой путь к Lisp, я надеюсь, данная книга поможет вам на этом пути. Итак, приступим!

2. Намылить, смыть, повторить: знакомство с REPL

В этой главе вы настроите среду программирования и напишете свои первые программы на Common Lisp. Мы воспользуемся лёгким в установке дистрибутивом **Lisp in a Box**, разработанным *Matthew Danish* и *Mikel Evins*, включающим в себя реализацию Common Lisp, мощным, прекрасно поддерживающим Lisp текстовым редактором **Emacs**, а также **SLIME** — средой разработки на Common Lisp, основанной на Emacs.

Этот набор предоставляет программисту современную среду разработки на Common Lisp, поддерживающую инкрементальный, интерактивный стиль разработки, характерный для программирования на этом языке. Среда SLIME даёт дополнительное преимущество в виде унифицированного пользовательского интерфейса, не зависящего от выбранных вами операционной системы и реализации Common Lisp. В своей книге я буду ориентироваться на среду Lisp in a Box, но те, кто хочет изучить другие среды разработки, например, графические интегрированные среды разработки (IDE - Integrated Development Environment), предоставляемые некоторыми коммерческими поставщиками, или среды, основанные на других текстовых редакторах, не должны испытывать больших трудностей в понимании.

Выбор реализации Lisp

Первое, что вам предстоит сделать — выбрать реализацию Lisp. Это может показаться немного странным тем, кто раньше занимался программированием на таких языках как Perl, Python, Visual Basic (VB), C# или Java. Разница между Common Lisp и этими языками заключается в том, что Common Lisp определяется своим стандартом: не существует ни единственной его реализации, контролируемой "великодушным диктатором" (как в случае с Perl и Python), ни канонической реализации, контролируемой одной компанией (как в случае с VB, C# или Java). Любой желающий может создать свою реализацию на основе стандарта. Кроме того, изменения в стандарт должны вноситься в соответствии с процессом, контролируемым Американским Национальным Институтом Стандартов (ANSI). Этот процесс организован таким образом, что "случайные лица", такие, как частные поставщики программных решений, не могут носить изменения в стандарт по своему усмотрению. Таким образом, стандарт Common Lisp — это договор между поставщиком Common Lisp и использующими Common Lisp разработчиками; этот договор подразумевает, что, если вы пишете программу, использующую возможности языка так, как это описано в стандарте, вы можете рассчитывать, что эта программа запустится на любой совместимой реализации Common Lisp.

С другой стороны, стандарт может описывать не всё из того, что вам может понадобиться в ваших программах. Более того, на некоторые аспекты языка спецификация намеренно отсутствует, чтобы дать возможность поэкспериментировать с различными способами их реализации, если при разработке стандарта не было достигнуто договорённости о наилучшем способе. Как видите, каждая реализация предоставляет пользователям как входящие в стандарт возможности, так и возможности, выходящие за его пределы. В зависимости от того, программированием какого рода вы собираетесь заняться, вы можете выбрать реализацию Common Lisp, поддерживающую именно те дополнительные возможности, которые вам больше всего понадобятся. С другой стороны, если вы предоставите другим разработчикам пользоваться вашим кодом на Lisp, например, разработанными вами библиотеками, вы, вероятно, захотите — конечно, в пределах возможного — писать переносимый код на Common Lisp. Для нужд написания кода, который должен быть переносимым, но, в тоже время, использовать возможности, не описанные в стандарте, Common Lisp предоставляет гибкий способ писать код, "зависящий" от возможностей текущей реализации. Вы увидите пример такого кода в главе 15,

когда мы будем разрабатывать простую библиотеку, "сглаживающую" некоторые различия в обработке разными реализациями Lisp имён файлов.

Сейчас, однако, наиболее важная характеристика реализации — её способность работать в вашей любимой операционной системе. Сотрудники компании Franz, занимающейся разработкой Allegro Common Lisp, выпустили пробную версию своего продукта, предназначенного для использования с этой книгой, и выполняющегося на GNU/Linux, Windows и OS X. У читателей, предпочитающих реализации с открытыми исходными текстами, есть несколько вариантов. **SBCL** - высококачественная открытая реализация, способная компилировать в машинный код и работать на множестве различных UNIX-систем, включая Linux и OS X. SBCL — "наследник" **CMUCL** — реализации Common Lisp, разработанной в университете **Carnegie Mellon**, и, как и CMUCL, является всеобщим достоянием (public domain, за исключением нескольких секций, покрываемых BSD-подобными (Berkley Software Distributions) лицензиями). CMUCL — тоже хороший выбор, однако SBCL, обычно, легче в установке и поддерживает 21-разрядный Unicode. **OpenMCL** будет отличным выбором для пользователей OS X: эта реализация способна компилировать в машинный код, поддерживать работу с потоками, а также прекрасно интегрируется с инструментальными комплектами Carbon и Coda. Кроме перечисленных, существуют и другие свободные и коммерческие реализации. Если вы захотите получить больше информации, в главе 32 вы найдёте список ресурсов.

Весь код на Lisp, приведённый в этой книге, должен работать на любой совместимой реализации Common Lisp, если явно не указано обратное, и SLIME будет "сглаживать" некоторые различия между реализациями, предоставляя общий интерфейс для взаимодействия с Lisp. Сообщения интерпретатора, приведённые в этой книге, сгенерированы **Allegro**, запущенном на **GNU/Linux**. В некоторых случаях другие реализации Lisp могут генерировать сообщения, незначительно отличающиеся от приведённых.

Введение в Lisp in a Box

Lisp in a Box спроектирован с целью быть "дружелюбным" к Лисперам-новичкам и предоставлять первоклассную среду разработки на Lisp с минимальными усилиями, и потому всё что вам нужно для работы - это взять соответствующий пакет для вашей операционной системы и выбранную вами реализацию Lisp с веб-сайта Lisp in a Box (см. главу 32) и далее следовать инструкциям по установке.

Так как Lisp in a Box использует Emacs в качестве текстового редактора, вы должны хоть немного уметь им пользоваться. Возможно, лучший способ начать работать с Emacs - это изучать его по встроенному учебнику(tutorial). Чтобы вызвать tutorial, выберете первый пункт меню Help – Emacs tutorial. Или же зажмите Ctrl и нажмете h, затем отпустите Ctrl и нажмете t. Большинство команд в Emacs доступно через комбинации клавиш, поэтому они будут встречаться довольно часто, и чтобы долго не описывать комбинации (например: "зажмите Ctrl и нажмете h, затем..."), в Emacs существует краткая форма записи комбинаций клавиш. Клавиши, которые должны быть нажаты вместе, пишутся вместе, разделяются тире, и называются связками; связки разделяются пробелами. С обозначает Ctrl, а M - Meta (Alt). Например вызов tutorial будет выглядеть таким образом: C-h t.

Tutorial также описывает много других полезных команд Emacs и вызывающих их комбинаций клавиш. У Emacs также есть расширенная онлайн документация, для просмотра которой используется специальный браузер – Info. Чтобы её вызвать нажмете C-h i. У Info также есть своя справка, которую можно вызвать, нажав клавишу h, находясь в браузере Info. Emacs предоставляет ещё несколько способов получить справку – это все сочетания клавиш, начинающиеся с C-h – полный список по C-h ?. В этом списке есть две полезные вещи: C-h k "объяснит" комбинацию клавиш, а C-h w – команду.

Ещё одна важная часть терминологии (для тех, кто отказался от работы с tutorial) - это буфер. Во время работы в Emacs, каждый файл, который Вы редактируете, представлен в отдельном буфере. Только один буфер может быть "текущим" в любой момент времени. В текущий буфер поступает весь ввод – всё, что Вы печатаете и любые команды, которые вызываете. Буферы также используются для представления взаимодействия с программами (например с Common Lisp). Есть одна простая вещь, которую вы должны знать – "переключение буферов", означающее смену текущего буфера, так что Вы можете редактировать определённый файл или взаимодействовать с определённой программой. Команда `switch-to-buffer`, привязанная к комбинации клавиш `C-x b`, запрашивает имя буфера (в нижней части окна Emacs). Во время ввода имени буфера, Вы можете пользоваться автодополнением по клавише `Tab`, которое по начальным символам завершает имя буфера или выводит список возможных вариантов. Просто нажав ввод, Вы переключитесь в буфер "по-умолчанию" (таким же образом и обратно). Вы также можете переключать буферы, выбирая нужный пункт в меню `Buffers`.

В определенных контекстах для переключения на определенные буферы могут быть доступны другие комбинации клавиш. Например, при редактировании исходных файлов Lisp сочетание клавиш `C-c C-z` переключает на буфер, в котором вы взаимодействуете с Lisp.

Освободите свой разум: Интерактивное программирование

При запуске Lisp in a Box, вы должны увидеть приглашение, которое может выглядеть примерно так :

```
CL-USER>
```

Это приглашение Lisp. Как и приглашение оболочки DOS или UNIX, приглашение Lisp — это место, куда вы можете печатать выражения, которые заставляют что-либо делать компьютер. Однако вместо того, чтобы считывать и выполнять строку команд оболочки, Lisp считывает Lisp выражения, вычисляет их согласно правилам Lisp и печатает результат. Потом он (lisp) повторяет свои действия со следующим введенным вами выражением. Вот вам бесконечный цикл: считывания, вычисления, и печати(вывода на экран), поэтому он называется *цикл-чтение-вычисление-печать* (по-английски *read-eval-print-loop*), или сокращённо REPL . Этот процесс может также называться *top-level*, *top-level listener*, или *Lisp listener*.

Через окружение, предоставленное REPL'ом, вы можете определять и переопределять элементы программ такие как переменные, функции, классы и методы; вычислять выражения Lisp; загружать файлы, содержащие исходные тексты Lisp или скомпилированные программы; компилировать целые файлы или отдельные функции; входить в отладчик; пошагово выполнять программы; и проверять состояние отдельных объектов Lisp;

Все эти возможности встроены в язык, и доступны через функции, определённые в стандарте языка. Если вы захотите, вы можете построить достаточно приемлемую среду разработки только из REPL и текстового редактора, который знает как правильно форматировать код Lisp. Но для истинного опыта Lisp программирования вам необходима среда разработки типа SLIME, которая бы позволяла вам взаимодействовать с Lisp как посредством REPL так и при редактировании исходных файлов. Например, вы ведь не захотите каждый раз копировать и вставлять куски кода из редактора в REPL или перезагружать весь файл только потому, что изменилось одно определение, ваше окружение должно позволять вам вычислять или компилировать как отдельные выражения так и целые файлы из вашего редактора.

Эксперименты в REPL

Для знакомства с REPL, вам необходимо выражение Lisp, которое может быть прочитано, вычислено и выведено на экран. Простейшее выражение Lisp - это число. Если вы наберете 10 в приглашении Lisp и нажмете ВВОД, то сможете увидеть что-то наподобие:

```
CL-USER> 10
10
```

Первая 10 - это то, что вы набрали. Считыватель Lisp, R в REPL, считывает текст "10" и создаёт объект Lisp, представляющий число 10. Этот объект - самовычисляемый объект, это означает, что такой объект при передаче в вычислитель, E в REPL, вычисляется сам в себя. Это значение подаётся на принтер, который напечатает объект "10" в отдельной строке. Хотя это и похоже на сизифов труд, можно получить что-то поинтереснее если дать интерпретатору Lisp пищу для размышлений. Например, вы можете набрать (+ 2 3) в приглашение Lisp.

```
CL-USER> (+ 2 3)
5
```

Все что в скобках - это список, в данном случае список из трех элементов, символ +, и числа 2 и 3. Lisp, в общем случае, вычисляет списки считая первый элемент именем функции и остальные - выражениями для вычисления и передачи в качестве аргументов этой функции. В нашем случае, символ + - название функции которая вычисляет сумму. 2 и 3 вычисляются сами в себя и передаются в функцию суммирования, которая возвращает 5. Значение 5 отправляется на устройство вывода, которое отображает его. Lisp может вычислять выражения и другими способами, но не будем сильно отдаляться от основной темы. В первую очередь вы должны написать . . .

"Здравствуй, Мир" в стиле Lisp

Нет законченной книги по программированию без программы "Здравствуй, мир"("hello, world."). После того как интерпретатор запущен, нет ничего проще чем набрать строку "Здравствуй, мир".

```
CL-USER> "Здравствуй, мир"
"Здравствуй, мир"
```

Это работает поскольку строки, также как и числа, имеют символьный синтаксис понимаемый *считывателем* Lisp и являются самовычисляемыми объектами: Lisp считывает строку в двойных кавычках и создает в памяти строковый объект, который при вычислении вычисляется сам в себя и потом печатается в том же символьном представлении. Кавычки не являются частью строкового объекта в памяти - это просто синтаксис, который позволяет считывателю определить что этот объект - строка. Принтер также выводит кавычки на вывод, потому что он пытается выводить объекты в таком же виде, в каком понимает их считыватель.

Однако, наш пример не может квалифицироваться как программа "Здравствуй мир". Это, скорее, значение "Здравствуй мир".

Вы можете сделать шаг к настоящей программе, напечатав код, который в качестве побочного эффекта выведет на стандартный вывод строку "Здравствуй, мир". Common Lisp предоставляет несколько путей для вывода данных, но самый гибкий - это функция FORMAT. FORMAT получает переменное количество параметров, но только два из них обязательны: указание, куда осуществлять вывод, и строка для вывода. В следующей главе Вы увидите, как строка может содержать встроенные директивы, которые позволяют вставлять в строку последующие параметры функции, а-ля printf или строку-% из Python. До тех пор, пока строка не содержит

символа `~`, она будет выводиться как есть. Если вы передадите `t` в качестве первого параметра, функция `FORMAT` направит свой вывод на стандартный вывод. Итак, выражение `FORMAT` для печати "Здравствуй, мир" выглядит примерно так:

```
CL-USER> (format t "Здравствуй, мир")
Здравствуй, мир
NIL
```

Стоит заметить, что результатом выражения `FORMAT` является `NIL` в строке после вывода "Здравствуй, мир". Этот `NIL` является результатом вычисления выражения `FORMAT`, напечатанного `REPL`. (`NIL` – это Lisp-версия `false` и/или `null`. Подробнее об этом рассказывается в главе 4.) В отличие от других выражений, рассмотренных ранее, нас больше интересует побочный эффект выражения `FORMAT` (в данном случае, печать на стандартный вывод), чем возвращаемое им значение. Но каждое выражение в Lisp вычисляется в некоторый результат.

Однако, до сих пор остается спорным, написали ли мы настоящую программу. Но вы ведь здесь. Вы видите восходящий стиль программирования, поддерживаемый `REPL`: вы можете экспериментировать с различными подходами и строить решения из уже протестированных частей. Теперь, когда у вас есть простое выражение, которое делает то, что вы хотите, нужно просто упаковать его в функцию. Функции являются одним из основных строительных материалов в Lisp и могут быть определены с помощью выражения `DEFUN` подобным образом:

```
CL-USER> (defun hello-world () (format t "hello, world"))
HELLO-WORLD
```

Выражение `hello-world`, следующее за `DEFUN`, является именем функции. В главе 4 мы рассмотрим, какие именно символы могут использоваться в именах, но сейчас будет достаточно сказать, что многие символы, такие как `<<-`, которые нелегальны в именах в других языках, абсолютно легальны в Common Lisp. Это стандартный стиль "Lisp – not to mention more in line with normal English typography" – формирование составных имен с помощью дефисов, как в `hello-world`, вместо использования знаков подчеркивания, как в `hello_world`, или использованием заглавных букв внутри имени, как `helloWorld`. Скобки `()` после имени отделяют список параметров, который в данном случае пуст, так как функция не принимает аргументов. Остальное – это тело функции.

На одном уровне это выражение, подобно всем другим, которые вы видели, всего лишь еще одно выражение для чтения, вычисления и печати, осуществляемых `REPL`. Возвращаемое значение в этом случае – это имя только что определенной функции. Но, подобно выражению `FORMAT`, это выражение более интересно своими побочными эффектами, нежели возвращаемым значением. Однако, в отличие от выражения `FORMAT`, побочные эффекты невидимы: после вычисления этого выражения создается новая функция, не принимающая аргументов, с телом `(format t "hello, world")` и ей дается имя `HELLO-WORLD`.

Теперь, после определения функции, вы можете вызвать ее следующим образом:

```
CL-USER> (hello-world)
hello, world
NIL
```

Вы можете видеть, что вывод в точности такой же, как при вычислении выражения `FORMAT` напрямую, включая значение `NIL`, напечатанное `REPL`. Функции в Common Lisp автоматически возвращают значение последнего вычисленного выражения.

Сохранение вашей работы

Вы могли бы утверждать, что это готовая программа "hello, world". Однако, остаётся одна проблема. Если вы выйдете из Lisp и перезапустите его, определение функции исчезнет. Написав такую изящную функцию, вы захотите сохранить вашу работу.

Это достаточно просто. вы просто должны создать файл, в котором сохраните определение. В Emacs вы можете создать новый файл, набрав C-x C-f и затем, когда Emacs выведет подсказку, введя имя файла, который вы хотите создать. Не особенно важно, где будет находиться этот файл. Обычно исходные файлы Common Lisp именуются с расширением .lisp, хотя некоторые люди предпочитают .cl вместо него.

Открыв файл, вы можете набирать определение функции, введённое ранее в области REPL. Обратите внимание, что после набора открывающей скобки и слова DEFUN, в нижней части окна Emacs SLIME подскажет вам предполагаемые аргументы. Точная форма зависит от используемой вами реализации Common Lisp, но вы вероятно увидите что-то вроде этого:

```
(defun name varlist &rest body)
```

Сообщение будет исчезать, когда вы будете начинать печатать каждый новый элемент, и снова появляться после ввода пробела. При вводе определения в файл вы можете захотеть разбить определение после списка параметров так, чтобы оно занимало две строки. Если вы нажмете Enter, а затем Tab, SLIME автоматически выровняет вторую строку соответствующим образом:

```
(defun hello-world ()  
  (format t "hello, world"))
```

SLIME также поможет вам в согласовании скобок – как только вы наберете закрывающую скобку, SLIME подсветит соответствующую открывающую скобку. Или вы можете просто набрать C-c C-q для вызова команды slime-close-parens-at-point, которая вставит столько закрывающих скобок, сколько нужно для согласования со всем открытыми скобками.

Теперь вы можете отправить это определение в вашу среду Lisp несколькими способами. Самый простой - это набрать C-c C-c, когда курсор находится где-нибудь внутри или сразу после формы DEFUN, что вызовет команду slime-compile-defun, которая, в свою очередь, пошлет определение в Lisp для вычисления и компиляции. Для того, чтобы убедиться, что это работает, вы можете сделать несколько изменений в hello-world, перекомпилировать ее, а затем вернуться назад в REPL, используя C-c C-z или C-x b, и вызвать ее снова. Например, вы можете сделать эту функцию более грамматически правильной.

```
(defun hello-world ()  
  (format t "Hello, world!"))
```

Теперь перекомпилируем ее с помощью C-c C-c и перейдем в REPL, набрав C-c C-z, чтобы попробовать новую версию.

```
CL-USER> (hello-world)  
Hello, world!  
NIL
```

Теперь вы возможно захотите сохранить файл, с которым работаете; находясь в буфере hello.lisp, наберите C-x C-s для вызова функции Emacs save-buffer.

Теперь, для того, чтобы попробовать перезагрузить эту функцию из файла с исходным кодом,

вы должны выйти из Lisp и перезапустить его. Для выхода вы можете использовать клавишную комбинацию SLIME: находясь в REPL, наберите запятую. Внизу окна Emacs вам будет предложена ввести команду. Наберите quit (или sayonara), а затем нажмите Enter. Произойдет выход из Lisp, а все окна, созданные SLIME (такие как буфер REPL), закроются. Теперь перезапустите SLIME, набрав M-x slime.

Просто ради интереса, вы можете попробовать вызвать hello-world.

```
CL-USER> (hello-world)
```

После этого возникнет новый буфер SLIME, содержимое которого будет начинаться с чего-то вроде этого:

```
attempt to call `HELLO-WORLD' which is an undefined function.
[Condition of type UNDEFINED-FUNCTION]
Restarts:
 0: [TRY-AGAIN] Try calling HELLO-WORLD again.
 1: [RETURN-VALUE] Return a value instead of calling HELLO-WORLD.
 2: [USE-VALUE] Try calling a function other than HELLO-WORLD.
 3: [STORE-VALUE] Setf the symbol-function of HELLO-WORLD and call it again.
 4: [ABORT] Abort handling SLIME request.
 5: [ABORT] Abort entirely from this process.
Backtrace:
 0: (SWANK::DEBUG-IN-EMACS #<UNDEFINED-FUNCTION @ #x716b082a>)
 1: ((FLET SWANK:SWANK-DEBUGGER-HOOK SWANK::DEBUG-IT))
 2: (SWANK:SWANK-DEBUGGER-HOOK #<UNDEFINED-FUNCTION @ #x716b082a> #<Function
SWANK-DEBUGGER-HOOK>)
 3: (ERROR #<UNDEFINED-FUNCTION @ #x716b082a>)
 4: (EVAL (HELLO-WORLD))
 5: (SWANK::EVAL-REGION "(hello-world)
" T)
```

Что же произошло? Просто вы попытались вызвать функцию, которая не существует. Но не смотря на такое количество выведенной информации, Lisp на самом деле обрабатывает такую ситуацию изящно. В отличие от Java или Python, Common Lisp не просто генерирует исключение и разворачивает стек. И он точно не завершается, оставив после себя образ памяти (dump core), только потому, что вы попытались вызвать несуществующую функцию. Вместо этого он перенесет вас в отладчик.

Во время работы с отладчиком вы все еще имеете полный доступ к Lisp, поэтому вы можете вычислять выражения для исследования состояния вашей программы и может быть даже для исправления каких-то вещей. Сейчас не стоит беспокоиться об этом; просто наберите q для выхода из отладчика и возвращения назад в REPL. Буфер отладчика исчезнет, а REPL выведет следующее:

```
CL-USER> (hello-world)
; Evaluation aborted
CL-USER>
```

Конечно, в отладчике можно сделать гораздо больше, чем просто выйти из него – в главе 19 мы увидим, например, как отладчик интегрируется с системой обработки ошибок. А сейчас, однако, важной вещью, которую нужно знать, является то, что вы всегда можете выйти из отладчика и вернуться обратно в REPL, набрав q.

Вернувшись в REPL вы можете попробовать снова. Ошибка произошла, потому что Lisp не знает определения hello-world. Поэтому вам нужно предоставить Lisp определение, сохраненное нами в

файле `hello.lisp`. Вы можете сделать это несколькими способами. Вы можете переключиться назад в буфер, содержащий файл (наберите `C-x b`, а затем введите `hello.lisp`) и перекомпилировать определение, как вы это делали ранее с помощью `C-c C-c`. Или вы можете загрузить файл целиком (что будет более удобным способом, если файл содержит множество определений) путем использования функции `LOAD` в `REPL` следующим образом:

```
CL-USER> (load "hello.lisp")
; Loading /home/peter/my-lisp-programs/hello.lisp
T
```

`T` означает, что загрузка всех определений произошла успешно. Загрузка файла с помощью `LOAD` в сущности эквивалентна набору каждого выражения этого файла в `REPL` в том порядке, в каком они находятся в файле, таким образом, после вызова `LOAD`, `hello-world` должен быть определен.

```
CL-USER> (hello-world)
Hello, world!
NIL
```

Еще один способ загрузки определений файла - предварительная компиляция файла с помощью `COMPILE-FILE`, а затем загрузка (с помощью `LOAD`) уже скомпилированного файла, называемого `FASL`-файлом, что является сокращением для `fast-load file` (быстро загружаемый файл). `COMPILE-FILE` возвращает имя `FASL`-файла, таким образом мы можем скомпилировать и загрузить файла из `REPL` следующим образом:

```
CL-USER> (load (compile-file "hello.lisp"))
;;; Compiling file hello.lisp
;;; Writing fasl file hello.fasl
;;; Fasl write complete
; Fast loading /home/peter/my-lisp-programs/hello.fasl
T
```

`SLIME` также предоставляет возможность загрузки и компиляции файлов без использования `REPL`. Когда вы находитесь в буфере с исходным кодом, вы можете использовать `C-c C-l` для загрузки файла с помощью `slime-load-file`. Emacs выведет запрос имени файла для загрузки с уже введенным именем текущего файла; вы можете просто нажать `Enter`. Или же вы можете набрать `C-c C-k` для компиляции и загрузки файла, представляемого текущим буфером. В некоторых реализациях `Common Lisp` компилирование кода таким образом выполнится немного быстрее; в других - нет, обычно потому что они всегда компилируют весь файл целиком.

Этого должно быть достаточно, чтобы дать вам почувствовать красоту того, как осуществляется программирование на `Lisp`. Конечно я пока не описал всех трюков и техник, но вы увидели важнейшие элементы – взаимодействие с `REPL`, загрузку и тестирование нового кода, настройку и отладку. Серьезные хакеры `Lisp` часто держат образ `Lisp` непрерывно запущенным многие дни, добавляя, переопределяя и тестируя части своих программ инкрементально.

Кроме того, даже если приложение, написанное на `Lisp`, уже развернуто, часто существует возможность обратиться к `REPL`. В главе 26 вы увидите как можно использовать `REPL` и `SLIME` для взаимодействия с `Lisp`, запустившим `Web-сервер`, в то же самое время, когда он продолжает отдавать `Web-страницы`. Возможно даже использовать `SLIME` для соединения с `Lisp`, запущенным на другой машине, что позволяет, например, отлаживать удаленный сервер так же, как локальный.

И даже более впечатляющий пример удаленной отладки произошел в миссии NASA "Deep Space

1" в 1998 году. Через полгода после запуска космического корабля, небольшой код на Lisp должен был управлять космическим кораблем в течении двух дней для проведения серии экспериментов. Однако, неуловимое состояние гонки (race condition) в коде не было выявлено при тестировании на земле и было обнаружено уже в космосе. Когда ошибка была выявлена в космосе (100 миллионов миль от Земли) команда смогла произвести диагностику и исправление работающего кода, что позволило завершить эксперимент. Один из программистов сказал об этом следующее:

Отладка программы, работающей на оборудовании стоимостью 100 миллионов долларов, которая находится в 100 миллионах миль от вас, является интересным опытом. REPL, работающий на космическом корабле, предоставляет бесценные возможности в нахождении и устранении проблем.

Вы пока не готовы отправлять какой бы то ни было код Lisp в дальний космос, но в следующей главе вы напишите программу, которая немного более интересна, чем "hello, world".

3. Практикум: Простая база данных

Очевидно, перед тем, как создавать настоящие программы на Lisp, вам необходимо изучить язык. Но давайте смотреть правде в глаза — вы можете подумать "Practical Common Lisp? Похоже на оксюморон. Зачем тратить силы на изучение деталей языка, если на нем невозможно сделать что-то дельное?". Итак, я начну с маленького примера на Common Lisp, который вы можете попробовать сами. В этой главе вы напишете простую базу данных для хранения CD-треков. В 27 главе вы будете использовать схожую технику при создании базы данных записей в формате MP3 для вашего потокового MP3-сервера. Фактически, можете считать это частью вашего программного проекта — в конце концов, для того, чтобы иметь сколько-нибудь MP3-записей для прослушивания, совсем не помешает знать, какие записи у вас есть, а какие нужно извлечь с диска.

В этой главе я пройду по языку Lisp достаточно для того, чтобы вы продвинулись до понимания, каким образом работает код на нём. Но я не буду вдаваться в детали. Вы можете не беспокоиться, если что-то здесь будет вам непонятно — в нескольких следующих главах все (и даже больше) используемые здесь конструкции Common Lisp будут описаны гораздо более систематически.

Одно замечание по терминологии: в этой главе я расскажу о некоторых операторах Lisp. В главе 4 вы узнаете, что Common Lisp предоставляет три разных типа операторов: функции, макросы и специальные операторы. Для целей этой главы вам необязательно понимать разницу. Однако я буду ссылаться на различные операторы как на функции, макросы или специальные операторы, в зависимости от того, чем они на самом деле являются, вместо того, чтобы попытаться скрыть эти детали за одним словом — оператор. Сейчас вы можете рассматривать функции, макросы и специальные операторы как более или менее эквивалентные сущности.

Также имейте в виду, что я не буду использовать все наиболее сложные техники Common Lisp для вашей первой после "Hello, world" программы. Целью этой главы является не то, как вы можете написать базу данных на Lisp; цель — дать вам понимание того, чем хорошо программирование на Lisp и видение того, что даже относительно простая программа на Lisp может иметь много возможностей.

CD и Записи

Для хранения данных о дорожке, которая должна быть перекодирована в MP3, и том, какой CD должен быть перекодирован в первую очередь, каждая запись в базе данных будет содержать название и имя исполнителя компакт диска, оценку того, насколько он нравится пользователю, и флаг, указывающий, был ли диск уже перекодирован. Итак, для начала вам необходим способ представления одной записи в базе данных (другими словами, одного CD). Common Lisp предоставляет для этого много различных структур данных, от простого четырёхэлементного списка до определяемого пользователем с помощью CLOS класса данных.

Для начала вы можете остановиться на простом варианте и использовать список. Вы можете создать его с помощью функции `LIST`, которая, соответственно, возвращает **список** из переданных аргументов.

```
CL-USER> (list 1 2 3)
(1 2 3)
```

Вы могли бы использовать четырёхпозиционный список, отображающий позицию в списке на соответствующее поле записи. Однако другая существующая разновидность списков, называемая *property list* (список свойств) или, сокращенно, *plist*, в нашем случае гораздо удобнее. *Plist* —

это такой список, в котором каждый нечетный элемент является *символом*, описывающим следующий (чётный) элемент списка. На этом этапе я не буду углубляться в подробности понятия *символ*; по своей природе это имя. Для символов, именующих поля в базе данных, мы можем использовать частный случай символов, называемый *символами-ключами* (*keyword symbol*). Ключ — это имя, начинающееся с двоеточия (:), например, :foo . Вот пример *plist*, использующего символы-ключи :a, :b и :c как имена свойств:

```
CL-USER> (list :a 1 :b 2 :c 3)
(:A 1 :B 2 :C 3)
```

Заметьте, вы можете создать список свойств той же функцией `LIST`, которой создавали прочие списки. Характер содержимого — вот что делает его списком свойств.

Причина, по которой использование *plist* является предпочтительным — наличие функции `GETF`, в которую передают *plist* и желаемый символ и получают следующее за символом значение. Это делает *plist* чем-то вроде упрощенной хэш-таблицы. В Lisp есть и "настоящие" хэш-таблицы, но для ваших текущих нужд достаточно *plist*, к тому же намного проще сохранять данные в такой форме в файл, это сильно пригодится позже.

```
CL-USER> (getf (list :a 1 :b 2 :c 3) :a)
1
CL-USER> (getf (list :a 1 :b 2 :c 3) :c)
3
```

Теперь, зная это, вам будет достаточно просто написать функцию `make-cd`, которая получит четыре поля в качестве аргументов и вернёт *plist*, представляющий CD.

```
(defun make-cd (title artist rating ripped)
  (list :title title :artist artist :rating rating :ripped ripped))
```

Слово `DEFUN` говорит нам, что это запись определяет новую функцию. Имя функции `make-cd`. После имени следует список параметров. Функция содержит четыре параметра — `title`, `artist`, `rating` и `ripped`. Всё, что следует за списком параметров — тело функции. В данном случае *тело* — лишь форма, просто вызов функции **LIST**. При вызове `make-cd` параметры, переданные при вызове, будут связаны с переменными в списке параметров из объявления функции. Например, для создания записи о CD *Roses* от Kathy Mattea вы можете вызвать `make-cd` примерно так:

```
CL-USER> (make-cd "Roses" "Kathy Mattea" 7 t)
(:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T)
```

Заполнение CD

Впрочем, создание одной записи — ещё не создание базы данных. Вам необходима более комплексная структура данных для хранения записей. Опять же, простоты ради, список представляется здесь вполне подходящим выбором. Также для простоты вы можете использовать глобальную переменную `*db*`, которую можно будет определить с помощью макроса **DEFVAR**. Звездочка (*) в имени переменной — это договоренность, принятая в языке Lisp при объявлении глобальных переменных.

```
(defvar *db* nil)
```

Вы можете использовать макрос **PUSH** для добавления элементов в `*db*`. Но, возможно,

неплохой идеей будет немного абстрагировать вещи и определить функцию 'add-record', которая будет добавлять записи в базу данных.

```
(defun add-record (cd) (push cd *db*))
```

Теперь вы можете использовать add-record вместе с make-cd для добавления CD в базу данных.

```
CL-USER> (add-record (make-cd "Roses" "Kathy Mattea" 7 t))
((:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
CL-USER> (add-record (make-cd "Fly" "Dixie Chicks" 8 t))
((:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
CL-USER> (add-record (make-cd "Home" "Dixie Chicks" 9 t))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

Всё, что REPL выводит после каждого вызова add-record — значения, возвращаемые последним выражением в теле функции, в нашем случае — PUSH. А PUSH возвращает новое значение изменяемой им переменной. Таким образом, после каждого добавления порции данных вы видите содержимое вашей базы данных.

Просмотр содержимого базы данных

Вы также можете посмотреть текущее значение *db* в любой момент, набрав *db* в REPL.

```
CL-USER> *db*
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

Правда, это не лучший способ просмотра данных. Вы можете написать функцию dump-db, которая выводит содержимое базы данных в более удобной форме, например, так:

```
TITLE: Home
ARTIST: Dixie Chicks
RATING: 9
RIPPED: T
TITLE: Fly
ARTIST: Dixie Chicks
RATING: 8
RIPPED: T
TITLE: Roses
ARTIST: Kathy Mattea
RATING: 7
RIPPED: T
```

Эта функция может выглядеть так:

```
(defun dump-db ()
  (dolist (cd *db*)
    (format t "~{~a:~10t~a~%~}~%" cd)))
```

Работа функции заключается в циклическом обходе всех элементов *db* с помощью макроса **DOLIST**, связывая на каждой итерации каждый элемент с переменной cd. Для вывода на экран

каждого значения `cd` используется функция `FORMAT`.

Следует признать, вызов `FORMAT` выглядит немного загадочно. Но в действительности `FORMAT` не особенно сложнее, чем функция `printf` из **C** или **Perl** или оператор `%` из **Python**. В главе 18 я расскажу о `FORMAT` более подробно. Теперь же давайте шаг за шагом рассмотрим, как работает этот вызов. Как было показано в гл. 2, `FORMAT` принимает по меньшей мере два аргумента, первый из которых — поток, в который `FORMAT` направляет свой вывод; `t` — сокращённое обозначение потока `*standard-output*`.

Второй аргумент `FORMAT` — формат строки; он может как содержать символьный текст, так и управляющие команды, контролирующие работу этой функции, например то, как она должна интерпретировать остальные аргументы. Команды, управляющие форматом вывода, начинаются со знака тильды (`~`) (так же, как управляющие команды начинаются с `%`). `FORMAT` может принимать довольно много таких команд, каждую со своим набором параметров. Однако сейчас я сфокусируюсь только на тех управляющих командах, которые необходимы для написания функции `dump-db`.

Команда `~a` служит для придания выводимым строкам некоторой эстетичности. Она принимает аргумент и возвращает его в удобочитаемой форме. Эта команда отобразит ключевые слова без предваряющего `:` и строки без кавычек. Например:

```
CL-USER> (format t "~a" "Dixie Chicks")
Dixie Chicks
NIL
```

или:

```
CL-USER> (format t "~a" :title)
TITLE
NIL
```

Команда `~t` предназначена для табулирования. Например, `~10t` указывает `FORMAT`, что необходимо выделить достаточно места для перемещения в десятый столбец перед выполнением команды `~a`. `~t` не принимает аргументов.

```
CL-USER> (format t "~a:~10t~a" :artist "Dixie Chicks")
ARTIST: Dixie Chicks
NIL
```

Теперь рассмотрим немного более сложные вещи. Когда `FORMAT` обнаруживает `~{`, следующим аргументом должен быть список. `FORMAT` циклично просматривает весь список, на каждой итерации выполняя команды между `~{` и `~}` и используя столько элементов списка, сколько нужно для вывода согласно этим командам. В функции `dump-db` `FORMAT` будет циклично просматривать список и на каждой итерации принимать одно ключевое слово и одно значение списка. Команда `~%` не принимает аргументов, но заставляет `FORMAT` выполнять переход на новую строку. После выполнения команды `~}` итерация заканчивается, и последняя `~%` заставляет `FORMAT` сделать ещё один переход на новую строку, чтобы записи, соответствующие каждому `CD`, были разделены. Формально, вы также можете использовать `FORMAT` для вывода именно базы данных, сократив тело функции `dump-db` до одной строки.

```
(defun dump-db ()
  (format t "~{~{~a:~10t~a~%}~%}" *db*))
```

Это может показаться довольно элегантным или, напротив, грубым приёмом, FIXME в зависимости от вашего мнения на этот счёт.

Улучшение взаимодействия с пользователем

Хотя функция `add-record` прекрасно выполняет свои обязанности, она слишком необычна для пользователя, незнакомого с Lisp. И если он захочет добавить в базу данных несколько записей, это может показаться ему довольно неудобным. В этом случае вы возможно захотите написать функцию, которая будет запрашивать у пользователя информацию о нескольких CD. В этом случае, вам нужен какой-то способ запросить эту информацию у пользователя и считать её. Для этого создадим следующую функцию:

```
(defun prompt-read (prompt)
  (format *query-io* "~a: " prompt)
  (force-output *query-io*)
  (read-line *query-io*))
```

Мы использовали уже знакомую нам функцию `FORMAT`, чтобы вывести приглашение. Заметим, что в строке, описывающей формат, отсутствует `<<~%`, поэтому перевода курсора на новую строку не происходит. Вызов **FORCE-OUTPUT** необходим в некоторых реализациях для уверенности в том, что Lisp не будет ожидать вывода новой строки перед выводом приглашения.

Теперь прочитаем одну строку текста с помощью (очень удачно названной!) функции **READ-LINE**. Переменная `*QUERY-IO*` является глобальной (о чем можно догадаться по наличию в её имени символов `*`), она содержит входной поток, связанный с терминалом. Значение, возвращаемое функцией **PROMPT-READ** — это значение последней ее формы, вызова `READ-LINE`, возвращающего прочитанную им строку (без завершающего символа новой строки).

Вы можете скомбинировать уже существующую функцию `make-cd` с `prompt-read`, чтобы построить функцию создания новой записи о CD из данных, которые `make-cd` по очереди получает для каждого значения.

```
(defun prompt-for-cd ()
  (make-cd
    (prompt-read "Title")
    (prompt-read "Artist")
    (prompt-read "Rating")
    (prompt-read "Ripped [y/n]")))
```

Это почти правильно, если не считать того, что функция `prompt-read` возвращает строку. Это хорошо подходит для полей `Title` и `Artist`, но значения полей `Rating` и `Ripped` — числовое и булево. В зависимости от того, насколько развитым вы хотите сделать пользовательский интерфейс, можете проверять подстроки произвольной длины, чтобы удостовериться в корректности введенных пользователем данных. Теперь давайте опробуем самый очевидный (хотя и не лучший) вариант: мы можем упаковать вызов `prompt-read`, запрашивающий у пользователя его оценку диска, в вызов специфичной для Lisp функции **PARSE-INTEGER**. Это можно сделать так:

```
(parse-integer (prompt-read "Rating"))
```

К сожалению, по умолчанию функция `PARSE-INTEGER` сообщает об ошибке, если ей не удастся разобрать число из введенной строки, или если в строке присутствует "нечисловой" мусор. Однако она может принимать дополнительный параметр `:junk-allowed`, который позволит нам ненадолго расслабиться.

```
(parse-integer (prompt-read "Rating") :junk-allowed t)
```

Остается ещё одна проблема — если `PARSE-INTEGER` не удастся выделить число среди "мусорных" данных, она вернёт не число, а `NIL`. Следуя нашему подходу "сделать просто, пусть даже не совсем правильно", мы в этом случае можем просто задать 0 и продолжить. Макрос **OR** здесь — как раз то, что нужно. Это то же самое, что и операция `||` в Perl, Python, Java и C; макрос принимает набор выражений, поочерёдно вычисляет их и возвращает первое истинное значение (либо `NIL`, если все они равны `NIL`). Таким образом, используем следующую запись:

```
(or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)
```

чтобы получить 0 в качестве значения по умолчанию.

Исправление кода для запроса состояния `Ripped` немного проще. Можно воспользоваться стандартной функцией Common Lisp **Y-OR-N-P**.

```
(y-or-n-p "Ripped [y/n]: ")
```

Фактически, этот вызов является самой отказоустойчивой частью `prompt-for-cd`, поскольку `Y-OR-N-P` будет повторно запрашивать у пользователя состояние флага `Ripped`, если он введет что-нибудь, начинающееся не с `y`, `Y`, `n` или `N`.

Собрав код вместе, получим достаточно надёжную функцию `prompt-for-cd`:

```
(defun prompt-for-cd ()  
  (make-cd  
    (prompt-read "Title")  
    (prompt-read "Artist")  
    (or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)  
    (y-or-n-p "Ripped [y/n]: ")))
```

Наконец, мы можем закончить интерфейс добавления CD, упаковав `prompt-for-cd` в функцию, циклично запрашивающую пользователя о новых данных. Воспользуемся простой формой макроса `LOOP`, выполняющего выражения в своём теле до тех пор, пока его выполнение не будет прервано вызовом `RETURN`. Например:

```
(defun add-cds ()  
  (loop (add-record (prompt-for-cd))  
        (if (not (y-or-n-p "Another? [y/n]: ")) (return))))
```

Теперь с помощью `add-cds` добавим в базу несколько новых дисков.

```
CL-USER> (add-cds)
Title: Rockin' the Suburbs
Artist: Ben Folds
Rating: 6
Ripped [y/n]: y
Another? [y/n]: y
Title: Give Us a Break
Artist: Limpopo
Rating: 10
Ripped [y/n]: y
Another? [y/n]: y
Title: Lyle Lovett
Artist: Lyle Lovett
Rating: 9
Ripped [y/n]: y
Another? [y/n]: n
NIL
```

Сохранение и загрузка базы данных

Хорошо иметь удобный способ добавления записей в базу данных. Но пользователю вряд ли понравится заново добавлять все записи после каждого перезапуска Lisp. К счастью, используя текущие структуры данных, используемые для представления информации, сохранить данные в файл и загрузить их позже — задача тривиальная. Далее приводится функция `save-db`, которая принимает в качестве параметра имя файла и сохраняет в него текущее состояние базы данных:

```
(defun save-db (filename)
  (with-open-file (out filename
                    :direction :output
                    :if-exists :supersede)
    (with-standard-io-syntax
      (print *db* out))))
```

Макрос **WITH-OPEN-FILE** открывает файл, связывает поток с переменной, выполняет набор инструкций и затем закрывает файл. Он также гарантирует, что файл обязательно закроется, даже если во время выполнения тела макроса что-то пойдет не так. Список, находящийся сразу после `WITH-OPEN-FILE`, является не вызовом функции, а частью синтаксиса, определяемого этим макросом. Он содержит имя переменной, хранящей файловый поток, в который в теле макроса `WITH-OPEN-FILE` будет вестись запись, значение, которое должно быть именем файла, и несколько параметров, управляющих режимом открытия файла. В нашем примере файл будет открыт для записи (задаётся параметром `:direction :output`), и, если файл с таким именем уже существует, его содержимое будет перезаписано (параметр `:if-exists :supersede`).

После того, как файл открыт, всё, что вам нужно — это печать содержимого базы данных с помощью `(print *db* out)`. В отличие от `FORMAT`, функция **PRINT** печатает объекты Lisp в форме, которую Lisp может прочитать. Макрос `WITH-STANDARD-IO-SYNTAX` гарантирует, что переменным, влияющим на поведение функции `PRINT`, присвоены стандартные значения. Используйте этот же макрос и при чтении данных из файла для гарантии совместимости операций записи и чтения.

Аргументом функции `save-db` должна являться строка, содержащая имя файла, в который пользователь хочет сохранить базу данных. Точный формат строки зависит от используемой операционной системы. Например, в Unix пользователь может вызвать функцию `save-db` таким образом:

```
CL-USER> (save-db "~/my-cds.db")
((:TITLE "Lyle Lovett" :ARTIST "Lyle Lovett" :RATING 9 :RIPPED T)
 (:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T)
 (:TITLE "Rockin' the Suburbs" :ARTIST "Ben Folds" :RATING 6 :RIPPED T)
 (:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 9 :RIPPED T))
```

В Windows имя файла может выглядеть так: `c:/my-cds.db`. Или так: `c:\my-cds.db`.

Вы можете открыть этот файл в любом текстовом редакторе и посмотреть, как выглядят записи. Вы должны увидеть что-то очень похожее на то, что печатает REPL, если вы наберёте `*db*`.

Функция загрузки базы данных из файла реализуется аналогично:

```
(defun load-db (filename)
  (with-open-file (in filename)
    (with-standard-io-syntax
      (setf *db* (read in))))))
```

В этот раз нет необходимости задавать `:direction` в параметрах `WITH-OPEN-FILE`, так как её значение по умолчанию — `:input`. И вместо печати вы используете функцию **READ** для чтения из потока `in`. Это тот же считыватель, что и в REPL, и он может прочитать любое выражение на Lisp, которое можно написать в строке приглашения REPL. Однако, в нашем случае, вы просто читаете и сохраняете выражение, не выполняя его. И снова, макрос `WITH-STANDARD-IO-SYNTAX` гарантирует, что `READ` использует тот же базовый синтаксис, что и функция `save-db`, когда она печатает данные с помощью `PRINT`.

Макрос **SETF** является главным оператором присваивания в Common Lisp. Он присваивает своему первому аргументу результат вычисления второго аргумента. Таким образом, в `load-db` переменная `*db*` будет содержать объект, прочитанный из файла, а именно, список списков, записанных функцией `save-db`. Обратите внимание на то, что `load-db` затирает то, что было в `*db*` до её вызова. Так что, если вы добавили записи, используя `add-records` или `add-cds`, и не сохранили их функцией `save-db`, эти записи будут потеряны.

Выполнение запросов к базе данных

Теперь, когда у вас есть способ сохранения и загрузки базы данных вместе с удобным интерфейсом для добавления новых записей, ничто не мешает вам собрать такую их коллекцию, когда вы уже не захотите просто распечатывать всю базу данных для того, чтобы просмотреть её содержимое. Вам нужно как-то выполнять запросы к базе данных. Может быть, вам понравится, например, следующий способ обращения к базе:

```
(select :artist "Dixie Chicks")
```

Наверно, в ответ на этот запрос вы захотите получить список всех записей, в которых исполнителем является Dixie Chicks. И снова оказалось, что выбор списка в качестве контейнера данных был очень удачным.

Функция **REMOVE-IF-NOT** принимает предикат и список в качестве параметров и возвращает список, содержащий только элементы исходного списка, удовлетворяющие предикату. Другими словами, она удаляет все элементы, не удовлетворяющие предикату. На самом деле, `REMOVE-IF-NOT` ничего не удаляет — она создает новый список, оставляя исходный список нетронутым.

Эта операция аналогична работе утилиты `grep`. Предикатом может быть любая функция, принимающая один аргумент и возвращающая логическое значение — `NIL` (ложь) и любое другое значение (истина).

Например, если вы хотите получить все чётные элементы из списка чисел, можете использовать `REMOVE-IF-NOT` таким образом:

```
CL-USER> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

В этом случае предикатом является функция **EVENP**, которая возвращает "истину", если её аргумент — чётное число. Нотация `#'` является сокращением выражения "Получить функцию с данным именем". Без `#'` Lisp обратится к `EVENP` как к имени переменной и попытается получить её значение, а не саму функцию.

Вы также можете передать в `REMOVE-IF-NOT` анонимную функцию. Например, если бы `EVENP` не существовало, вы могли бы так написать предыдущее выражение:

```
CL-USER> (remove-if-not #'(lambda (x) (= 0 (mod x 2)))) '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

В этом случае предикатом является анонимная функция

```
(lambda (x) (= 0 (mod x 2)))
```

которая проверяет, равен ли нулю остаток от деления аргумента на 2 (другими словами, является ли аргумент чётным). Если вы хотите извлечь только нечётные числа, используя анонимную функцию, вы можете написать следующее:

```
CL-USER> (remove-if-not #'(lambda (x) (= 1 (mod x 2)))) '(1 2 3 4 5 6 7 8 9 10))
(1 3 5 7 9)
```

Заметьте, что **lambda** не является именем функции — это слово показывает, что вы определяете анонимную функцию. Если не считать имени, `LAMBDA`-выражение выглядит очень похожим на `DEFUN`: после слова `lambda` следует список параметров, за которым идёт тело функции.

Чтобы выбрать все альбомы Dixie Chicks из базы данных, используя `REMOVE-IF-NOT`, вам нужна функция, возвращающая "истину", если поле в записи `artist` содержит значение "Dixie Chicks". Помните, мы выбрали *список свойств* в качестве представления записей базы данных, потому что функция `GETF` может извлекать из *списка свойств* именованные поля. Итак, полагая, что `cd` является именем переменной, хранящей одну запись базы данных, вы можете использовать выражение `(getf cd :artist)`, чтобы извлечь имя исполнителя. Функция **EQUAL** посимвольно сравнивает переданные ей строковые параметры. Таким образом, `(equal (getf cd :artist) "Dixie Chicks")` проверит, хранит ли поле `artist`, соответствующего данному CD, значение "Dixie Chicks". Всё, что вам нужно — упаковать это выражение в `LAMBDA`-форму, чтобы создать анонимную функцию и передать ее `REMOVE-IF-NOT`.

```
CL-USER> (remove-if-not
  #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks"))) *db*)
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

Предположим теперь, что вы хотите упаковать всё выражение в функцию, которая принимает имя исполнителя в качестве параметра. Вы можете записать это так:

```
(defun select-by-artist (artist)
  (remove-if-not
    #'(lambda (cd) (equal (getf cd :artist) artist))
    *db*))
```

Заметьте, что анонимная функция, содержащая код, который не будет выполнен, пока функция не вызвана в REMOVE-IF-NOT, тем не менее может ссылаться на переменную `artist`. В этом случае анонимная функция не просто избавляет вас от необходимости писать обычную функцию, — она позволяет вам написать функцию, наследующую часть её значений FIXME — содержимое поля `artist` — из контекста, в который она встроена.

Итак, мы покончили с функцией `select-by-artist`. Однако выборка по исполнителю — лишь одна разновидность запросов, которые вам захочется реализовать. Вы можете написать ещё несколько функций, таких, как `select-by-title`, `select-by-rating`, `select-by-title-and-artist`, и так далее. Но все они будут идентичными, за исключением содержимого анонимной функции. Вместо этого вы можете создать более универсальную функцию `select`, которая принимает функцию в качестве аргумента.

```
(defun select (selector-fn)
  (remove-if-not selector-fn *db*))
```

А что случилось с `#'`? Дело в том, что в этом случае вам не нужно, чтобы функция REMOVE-IF-NOT использовала функцию под названием `selector-fn`. Вы хотите, чтобы она использовала анонимную функцию, переданную в качестве аргумента функции `select` в переменной `selector-fn`. Однако, символ `#'` вернулся в вызов `select`:

```
CL-USER> (select #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks"))))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

Правда, это выглядит довольно грубо. К счастью, вы можете упаковать создание анонимной функции.

```
(defun artist-selector (artist)
  #'(lambda (cd) (equal (getf cd :artist) artist)))
```

`artist-selector` возвращает функцию, имеющую ссылку на переменную, которая перестанет существовать после выхода из `artist-selector`. Функция выглядит странно, но она работает именно так, как нам нужно — если вызвать `artist-selector` с аргументом "Dixie Chicks", мы получим анонимную функцию, которая ищет CD с полем `:artist`, содержащим "Dixie Chicks", и если вызвать её с аргументом "Lyle Lovett", то мы получим другую функцию, которая будет искать CD с полем `:artist`, содержащим "Lyle Lovett". Итак, мы можем переписать вызов `select` следующим образом:

```
CL-USER> (select (artist-selector "Dixie Chicks"))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

Теперь нам понадобится больше функций, чтобы генерировать выражения для выбора. Но так как вы не хотите писать `select-by-title`, `select-by-rating` и др., потому что они будут

во многом схожими, вы не станете создавать множество почти идентичных генераторов выражений для выбора значений для каждого из полей. Почему бы не написать генератор функции-выражения для выбора общего назначения — функцию, которая, в зависимости от передаваемых ей аргументов, будет генерировать выражение выбора для разных полей или, может быть, даже комбинации полей? Вы можете написать такую функцию, но сначала нам придётся пройти краткий курс для овладения средством, называемым *параметрами-ключами* (keyword parameters).

В функциях, что вы писали до этого, вы задавали простой список параметров, которые связывались с соответствующими аргументами в вызове функции. Например, следующая функция:

```
(defun foo (a b c) (list a b c))
```

имеет три параметра: *a*, *b* и *c*, и должна быть вызвана с тремя аргументами. Но иногда возникает необходимость в вызове функции, которая может вызываться с переменным числом аргументов. Параметры-ключи — один из способов это сделать. Версия *foo* с использованием параметров-ключей может выглядеть так:

```
(defun foo (&key a b c) (list a b c))
```

Единственное отличие — элемент **&key** в начале списка аргументов. Однако вызовы новой функции *foo* выглядят немного по-другому. Все нижеперечисленные варианты вызова *foo* допустимы, результат вызова помещён справа от `==>`.

```
(foo :a 1 :b 2 :c 3) ==> (1 2 3)
(foo :c 3 :b 2 :a 1) ==> (1 2 3)
(foo :a 1 :c 3) ==> (1 NIL 3)
(foo) ==> (NIL NIL NIL)
```

Как показывают эти примеры, значения переменных *a*, *b* и *c* привязаны к значениям, которые следуют за соответствующими ключевыми словами. И если какой-либо ключ в вызове отсутствует, соответствующая переменная устанавливается в *NIL*. Я не буду уточнять, как именно задаются ключевые параметры и как они соотносятся с другими типами параметров, но вам важно знать одну деталь.

Обычно, когда функция вызывается без аргумента для конкретного параметра-ключа, параметр будет иметь значение *NIL*. Но иногда нужно различать *NIL*, который был явно передан в качестве аргумента к параметру-ключу, и *NIL*, который задаётся по умолчанию. Чтобы сделать это, при задании параметра-ключа вы можете заменить обычное имя списком, состоящим из имени параметра, его значения по умолчанию и другого имени параметра, называемого параметром *supplied-p*. Этот параметр *supplied-p* будет содержать значения "истина" или "ложь", в зависимости от того, действительно ли для данного параметра-ключа в данном вызове функции был передан аргумент. Вот версия новой функции *foo*, которая использует эту возможность.

```
(defun foo (&key a (b 20) (c 30 c-p)) (list a b c c-p))
```

Результаты тех же вызовов теперь выглядят иначе:

```
(foo :a 1 :b 2 :c 3) ==> (1 2 3 T)
(foo :c 3 :b 2 :a 1) ==> (1 2 3 T)
(foo :a 1 :c 3) ==> (1 20 3 T)
(foo) ==> (NIL 20 30 NIL)
```

Основной генератор выражения выбора, который по причинам, которые, если вы знакомы с SQL, скоро станут очевидными, можно назвать **where**, является функцией, принимающей четыре параметра-ключа для соответствующих полей в наших записях CD и генерирующей выражение выбора, которое возвращает все записи о CD, совпадающие по значениями, задаваемым в where. Например, можно будет написать такое выражение:

```
(select (where :artist "Dixie Chicks"))
```

Или такое:

```
(select (where :rating 10 :ripped nil))
```

Функция выглядит так:

```
(defun where (&key title artist rating (ripped nil ripped-p))
  #'(lambda (cd)
    (and
      (if title (equal (getf cd :title) title) t)
      (if artist (equal (getf cd :artist) artist) t)
      (if rating (equal (getf cd :rating) rating) t)
      (if ripped-p (equal (getf cd :ripped) ripped) t))))
```

Эта функция возвращает анонимную функцию, возвращающую логическое И для одного условия в каждом поле записей о CD. Каждое условие проверяет, задан ли подходящий аргумент, и если задан, то сравнивает его значение со значением соответствующего поля в записи о CD, или возвращает t, обозначение истины в Lisp, если аргумент не был задан. Таким образом, выражение выбора возвратит t только для тех CD, описание которых совпало по значению с аргументами переданными where. Заметьте, что, чтобы задать ключ-параметр `ripped`, вам необходимо использовать список из трёх элементов, потому что вам нужно знать, действительно ли вызывающая функция передала ключ-параметр `:ripped nil`, означающее "Выбрать те CD, в поле `ripped` которых установлено значение `nil`", либо опустила его, что означает "Мне всё равно, какое значение установлено в поле `ripped`".

Обновление существующих записей — повторное использование where

Теперь, после того, как у вас есть достаточно универсальные функции `select` и `where`, очень логичной представляется реализация следующей возможности, которая необходима каждой базе данных, — возможности обновления отдельных записей. В SQL команда `update` используется для обновления набора записей, удовлетворяющих `FIXME` конкретному условию `where`. Эта модель кажется хорошей, особенно когда у вас уже есть генератор условий `where`. Фактически, функция `update` — применение некоторых идей, которые вы уже видели: использование передаваемого выражения выбора для указания записей, подлежащих обновлению, и использование аргументов-ключей для задания нового значения. Новая вещь здесь — использование функции **MAPCAR**, которая отображает список, в нашем случае это `*db*`, и возвращает новый список, содержащий результаты вызова функции для каждого элемента исходного списка.

```
(defun update (selector-fn &key title artist rating (ripped nil ripped-p))
  (setf *db*
    (mapcar
      #'(lambda (row)
        (when (funcall selector-fn row)
          (if title (setf (getf row :title) title))
          (if artist (setf (getf row :artist) artist))
          (if rating (setf (getf row :rating) rating))
          (if ripped-p (setf (getf row :ripped) ripped))))
        row) *db*)))
```

Ещё одна новинка в этой функции — приложение SETF к сложной форме вида (getf row :title). Я расскажу о SETF подробнее в главе 6, но сейчас вам просто нужно знать, что это общий оператор присваивания, который может использоваться для присваивания друг другу различных "вещей", а не только переменных. (То, что SETF и GETF имеют настолько похожие имена — просто совпадение. Между ними нет никакой особой взаимосвязи). Сейчас достаточно знать, что после выполнения (setf (getf row :title) title) у списка свойств, на который ссылается row, значением переменной, следующей за именем свойства :title, будет title. С помощью функции update, Если вы решите, что действительно любите творчество Dixie Chicks, и что все их альбомы должны быть оценены в 11 баллов, можете выполнить следующую форму:

```
CL-USER> (update (where :artist "Dixie Chicks") :rating 11)
NIL
```

Результат работы будет выглядеть так:

```
CL-USER> (select (where :artist "Dixie Chicks"))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T))
```

Добавить функцию удаления строк из базы данных еще проще.

```
(defun delete-rows (selector-fn)
  (setf *db* (remove-if selector-fn *db*)))
```

Функция **REMOVE-IF** является дополнением к REMOVE-IF-NOT; она возвращает список всех элементов, удалив те из них, что удовлетворяют предикату. Так же, как и REMOVE-IF-NOT, она, в действительности, не изменяет список, который был ей передан в качестве параметра, тем не менее, сохраняя результат обратно в *db*, delete-rows фактически изменяет содержимое базы данных.

Избавление от дублирующего кода и большой выигрыш

До сих пор весь код базы данных, обеспечивающий операции INSERT, SELECT, UPDATE и DELETE, если не считать интерфейс командной строки для добавления новых записей и распечатки содержимого базы, укладывался в немногим более пятидесяти строк. Целиком.

Всё еще существует некоторое раздражающее дублирование кода. И, оказывается, вы можете избавиться от этого дублирования, в то же время сделав код более гибким. Дублирование, о котором я говорю, находится в функции where. Тело функции where — набор условий для каждого поля, таких, как это:

```
(if title (equal (getf cd :title) title) t)
```

Сейчас это не так плохо, но, как и во многих случаях дублирования кода, за это всегда приходится платить одну цену: если вы хотите изменить работу этого кода, вам нужно изменять множество копий. И если вы изменили поля в CD, вам придётся добавить или удалить условия для `where`. `update` страдает точно таким же дублированием. Это, несомненно, плохо, так как весь смысл функции `where` заключается в динамической генерации куска кода, проверяющего нужные нам значения; почему она должна производить работу во время выполнения, каждый раз проверяя, было ли ей передано значение `title`?

Представьте, что вы попытались оптимизировать этот код и обнаружили, что много времени тратится на проверку того, заданы ли значения `title` и оставшиеся ключ-параметры. Если вы на самом деле хотите избавиться от этих проверок во время выполнения, вы можете просмотреть программу и найти все места, где вы вызываете `where`, и посмотреть, какие аргументы вы передаёте. Затем вы можете заменить каждый вызов `where` анонимной функцией, выполняющей только необходимое вычисления. Например, если вы нашли такой кусок кода:

```
(select (where :title "Give Us a Break" :ripped t))
```

вы можете заменить его на такой:

```
(select
  #'(lambda (cd)
    (and (equal (getf cd :title) "Give Us a Break")
         (equal (getf cd :ripped) t))))
```

Заметьте, что анонимная функция отличается от той, что возвращает `where`; мы не пытаемся сохранить вызов `where`, а обеспечиваем большую производительность функции выбора. Эта анонимная функция имеет условия только для нужных нам полей, и она не производит дополнительной работы, в отличие от функции, которую может вернуть `where`.

Вы можете представить себе, что значит пройти по всему исходному тексту исправить все вызовы `where` таким образом. И вы можете представить, насколько это болезненно. Если бы этого было достаточно, и это было бы очень важно, вероятно, стоило бы написать некоторого рода препроцессор, который бы конвертировал вызовы `where` в то, что вы бы написали вручную.

Средство Lisp, позволяющее делать это очень просто, называется системой макросов. Подчеркиваю, что макрос в Common Lisp не имеет, в сущности, ничего общего (кроме имени) с текстовыми макросами из C и C++. В то время, как препроцессор C оперирует текстовой подстановкой и не знает ничего о структуре C и C++, в Lisp макрос, в сущности, является генератором кода, который автоматически запускается для вас компилятором. Когда выражение на Lisp содержит вызов макроса, компилятор Lisp, вместо вычисления аргументов и передачи их в функцию, передает аргументы, не вычисляя их, в код макроса, который, в свою очередь, возвращает новое выражение на Lisp, которое затем вычисляется в месте исходного вызова макроса.

Я начну с простого и глупого примера и затем покажу, как вы можете заменить функцию `where` макросом `where`. Перед тем, как я напишу этот макрос-пример, мне необходимо представить вам одну новую функцию: **REVERSE** принимает аргумент в виде списка и возвращает новый список, который является обратным к исходному. Таким образом, `(reverse '(1 2 3))` вернёт `(3 2 1)`. Теперь попробуем создать макрос.

```
(defmacro backwards (expr)
  (reverse expr))
```

Главное синтаксическое отличие между функцией и макросом заключается в том, что макрос определяется ключевым словом **DEFMACRO**, а не **DEFUN**. После ключевого слова в определении макроса, подобно определению функции, следует имя, список параметров и тело с выражениями. Однако макросы действуют совершенно по-другому. Вы можете использовать макрос так:

```
CL-USER> (backwards ("hello, world" t format))
hello, world
NIL
```

Как это работает? Когда REPL начинает вычислять выражение `backwards`, он обнаруживает, что `backwards` — имя макроса. Поэтому он не вычисляет выражение `("hello, world" t format)`, что очень хорошо, так как это некорректная для Lisp структура. Далее он передаёт это список коду `backwards`. Код `backwards` передает список в функцию `REVERSE`, которая возвращает список `(format t "hello, world")`. Затем `backwards` передает это значение обратно REPL, который подставляет его на место исходного выражения.

Макрос `backwards`, таким образом, определяет новый язык, во многом похожий на Lisp — только задом наперёд — который вы можете вставлять в свой код в любой момент, просто обернув обратное выражение на Lisp в вызов макроса `backwards`. И в скомпилированной программе на Lisp этот новый язык покажет такую же производительность, как и обычный Lisp, потому что весь код в макросе — код, сгенерированный в новом выражении — выполняется во время компиляции. Другими словами, компилятор сгенерирует один и тот же код, независимо от того, напишете вы `(backwards ("hello, world" t format))` или `(format t "hello, world")`.

Итак, как это поможет решить проблему дублирующегося кода в `where`? Очень просто. Вы можете написать макрос, генерирующий совершенно такой же код, какой вы написали бы для каждого вызова `where`. И снова, лучший подход — это разрабатывать код снизу вверх. В оптимизированной вручную функции выбора `where` для каждого из заданных полей у вас было выражение в следующей форме:

```
(equal (getf cd field) value)
```

Давайте напишем функцию, которая, получив имя поля и некоторое значение, возвращает такое выражение. Так как выражение — это просто список, вы можете подумать, что возможно написать что-нибудь вроде:

```
(defun make-comparison-expr (field value) ; неправильно
  (list equal (list getf cd field) value))
```

Однако здесь имеется небольшой нюанс: как вы знаете, когда Lisp обнаруживает просто имя вроде `field` или `value`, а не первый элемент списка, он полагает, что это имя переменной, и пытается получить ее значение. Это нормально для `field` и `value`; это именно то, что нужно. Но он будет обращаться к `equal`, `getf` и `cd` таким же образом, а это в нашем случае нежелательно. Вы, однако, знаете также, как не позволить Lisp пытаться вычислить структуру: поместить перед ней одиночную кавычку (`'`). Таким образом, если вы напишете функцию `make-comparison-expr` вот так, она сделает то, что вам нужно:

```
(defun make-comparison-expr (field value)
  (list 'equal (list 'getf 'cd field) value))
```

Вы можете проверить её работу в REPL:

```
CL-USER> (make-comparison-expr :rating 10)
(EQUAL (GETF CD :RATING) 10)
CL-USER> (make-comparison-expr :title "Give Us a Break")
(EQUAL (GETF CD :TITLE) "Give Us a Break")
```

Но, оказывается, существует лучший способ сделать это. То, что вам действительно нужно, — это иметь возможность написать выражение, которое в большинстве случаев не вычисляется, и затем каким-либо образом выбирать некоторые выражения, которые вы хотите вычислить. И, конечно же, такой механизм существует. Обратная кавычка (```) перед выражением не позволяет вычислить выражение точно так же, как и прямая одиночная кавычка.

```
CL-USER> `(1 2 3)
(1 2 3)
CL-USER> '(1 2 3)
(1 2 3)
```

Однако в выражении с обратной кавычкой любое подвыражение, перед которым стоит запятая, вычисляется. Обратите внимание на влияние запятой во втором выражении:

```
`(1 2 (+ 1 2)) ==> (1 2 (+ 1 2))
`(1 2 , (+ 1 2)) ==> (1 2 3)
```

Используя обратную кавычку, вы можете переписать функцию `make-comparison-expr` следующим образом:

```
(defun make-comparison-expr (field value)
  `(equal (getf cd ,field) ,value))
```

Теперь, если вы посмотрите на оптимизированную ручную функцию выбора, вы увидите, что тело функции состоит из одного оператора сравнения для каждой пары поле/значение, обернутое в выражение `AND`. На мгновение положим, что вам нужно расположить аргументы таким образом, чтобы передать их макросу `where` единым списком. Вам понадобится функция, которая принимает аргументы этого списка попарно и сохраняет результаты выполнения вызова `make-comparison-expr` для каждой пары. Чтобы реализовать эту функцию, вы можете воспользоваться мощным макросом **LOOP**.

```
(defun make-comparisons-list (fields)
  (loop while fields
        collecting (make-comparison-expr (pop fields) (pop fields)))))
```

Полное описание макроса `LOOP` отложим до 22 главы, а сейчас заметим, что выражение `LOOP` выполняет именно то, что требуется: оно циклично проходит по всем элементам в списке `fields`, каждый раз возвращая по два элемента, передает их в `make-comparison-expr` и сохраняет возвращаемые результаты, чтобы их вернуть при выходе из цикла. Макрос **POP** выполняет операцию, обратную операции, выполняемой макросом `PUSH`, который вы использовали для добавления записей в `*db*`.

Теперь вам нужно просто обернуть список, возвращаемый функцией `make-comparison-list` в

AND и анонимную функцию, которую вы можете реализовать прямо в макросе where. Это просто: используйте обратную кавычку, чтобы создать шаблон, который будет заполнен значениями функции make-comparison-list.

```
(defmacro where (&rest clauses)
  `#' (lambda (cd) (and ,@(make-comparisons-list clauses)))))
```

Этот макрос использует вариацию , (а именно, ,@) перед вызовом make-comparison-list. Сочетание ,@ "вклеивает" значение следующего за ним выражения, которое должно возвращать список, во "внешний" список.

```
`(and ,(list 1 2 3)) ==> (AND (1 2 3))
`(and ,@(list 1 2 3)) ==> (AND 1 2 3)
```

Вы также можете использовать ,@ для "вклейки" элементов в середину списка:

```
`(and ,@(list 1 2 3) 4) ==> (AND 1 2 3 4)
```

Другая важная особенность макроса where — использование &rest в списке аргументов. Так же, как и &key, &rest изменяет способ разбора аргументов. Если в списке параметров обнаруживается &rest, функция или макрос могут принимать произвольное число аргументов, которые собираются в единый список, становящийся значением переменной, имя которой следует за &rest. Итак, если вы вызовете where так:

```
(where :title "Give Us a Break" :ripped t)
```

переменная clauses будет содержать список:

```
(:title "Give Us a Break" :ripped t)
```

Этот список передается функции make-comparisons-list, которая возвращает список выражений сравнения. С помощью функции **MACROEXPAND-1** вы можете точно видеть, какой код будет сгенерирован where. Если вы передадите в MACROEXPAND-1 форму, являющуюся вызовом макроса, она вызовет макрос с заданными аргументами и вернёт его развёрнутый вид. Итак, вы можете проверить предыдущий вызов where следующим образом:

```
CL-USER> (macroexpand-1 '(where :title "Give Us a Break" :ripped t))
#'(LAMBDA (CD)
  (AND (EQUAL (GETF CD :TITLE) "Give Us a Break")
    (EQUAL (GETF CD :RIPPED) T)))
T
```

Выглядит неплохо. Теперь попробуем испытать макрос в действии:

```
CL-USER> (select (where :title "Give Us a Break" :ripped t))
((:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T))
```

Работает. И макрос where с его двумя функциями-помощниками оказался на одну строку короче, чем старая функция where. И, что самое главное, where больше не привязана к конкретным полям наших записей о CD.

Об упаковке

Случилась интересная вещь. Вы избавились от дублирования и сделали код одновременно более производительным и универсальным. Так часто бывает, если правильно выбрать макрос. Это имеет смысл, потому что макрос — это ещё один механизм создания абстракций — абстракций на синтаксическом уровне, а абстракции — это, по определению, более короткий путь для выражения подразумеваемых сущностей. Сейчас код мини-базы данных, который относится к CD и полям, его описывающим, находится только в функциях `make-cd`, `prompt-for-cd` и `add-cd`. Фактически, наш новый макрос будет работать с любой базой данных, основанной на списке свойств.

Тем не менее, эта база данных всё ещё далека от завершения. Вероятно, вы думаете о добавлении множества возможностей, например, таких, как поддержка множества таблиц или более сложных запросов. В главе 27 мы создадим базу данных о записях MP3, которая будет содержать некоторые из этих возможностей.

Целью этой главы являлось быстрое введение в лишь малую часть возможностей Lisp и демонстрация того, как они используются для написания кода, чуть более интересного, чем "Hello, world". В следующей главе мы начнём более систематический обзор Lisp.

4. Синтаксис и семантика

После столь стремительного тура мы уgomонимся на несколько глав для получения более систематического взгляда на возможности, которые вы до этого использовали. Я начну с обзора базовых элементов синтаксиса и семантики Lisp, что, конечно же, означает, что я должен сначала ответить на неотложный вопрос...

Почему так много скобок?

Синтаксис Lisp немного отличается от синтаксиса языков, произошедших от Algol. Две наиболее очевидные черты — это обширное использование скобок и префиксная нотация. Именно эти черты отпугивают многих людей. Очернители Lisp склонны описывать его синтаксис как "непонятный" и "раздражающий". Название "Lisp", по их словам, должно обозначать "Множество Раздражающих Ненужных Скобок" (Lots of Irritating Superfluous Parentheses). С другой стороны, люди, использующие Lisp, склонны рассматривать синтаксис Lisp как одно из главных его достоинств. Как может быть то, что так не нравится одной группе, быть предметом восхищения другой?

Я не смогу действительно объяснить вам все состояние дел с синтаксисом Lisp, пока не расскажу немного подробнее о макросах Lisp. Но я могу начать с предыстории, которая наводит на мысль, что имеет смысл откинуть предрассудки и попытаться разобраться, что к чему: когда John McCarthy изобрел Lisp, он намеревался реализовать его в более Algol-подобном синтаксисе, который он называл M-выражения. Однако он так не и сделал этого. Причину он объясняет в своей статье "История Lisp".

Проект по точному определению M-выражений и их компиляции или, хотя бы, трансляции их в S-выражения не был ни завершен, ни явно заброшен. Он просто был отложен на неопределенное будущее, а тем временем появилось новое поколение программистов, которые предпочитали S-выражения любой Fortran- или Algol-подобной нотации, которая только может быть выдумана.

Другими словами, люди, которые действительно использовали Lisp на протяжении последних 45 лет, полюбили синтаксис и нашли, что он делает язык более мощным. По прочтении последующих глав вы начнете понимать почему.

Разделение черного ящика

Перед тем, как мы рассмотрим специфику синтаксиса и семантики Lisp, будет полезно уделить внимание тому, чтобы разобраться, как они определены и чем отличаются от множества других языков.

В большинстве языков программирования процессор языка (интерпретатор или компилятор) работает как черный ящик: вы засовываете последовательность символов, представляющих собой текст программы, в черный ящик и он (в зависимости от того, является ли он интерпретатором или компилятором) либо выполняет указанные инструкции, либо создает скомпилированную версию программы, которая выполняет инструкции после запуска.

Внутри черного ящика, конечно, процессоры языка обычно разделяются на подсистемы, каждая из которых ответственна за одну из частей задачи трансляции текста программы в последовательность инструкций или объектный код. Типичное разделение — это разбиение процессора на три фазы, каждая из которых предоставляет данные следующей: лексический анализатор разделяет поток знаков на лексемы и передает их синтаксическому анализатору, который строит дерево, представляющее выражения программы в соответствии с грамматикой языка. Это дерево (называемое абстрактным синтаксическим деревом, AST) далее передается

процедуре вычисления, которая либо напрямую интерпретирует его, либо компилирует его в какой-то другой язык, такой как машинный код. Так как языковой процессор является черным ящиком, то структуры данных, используемые процессором, такие как лексемы или абстрактные синтаксические деревья, интересуют только конструкторов реализации языка.

В Common Lisp фазы разбиты иначе, с последствиями как для конструкторов реализации, так и для определения языка. Вместо одного черного ящика, который осуществляет переход от текста программы к ее поведению за один шаг, Common Lisp определяет два черных ящика, первый из которых транслирует текст в объекты Lisp, а другой реализует семантику языка в терминах этих объектов. Первый ящик называется процедурой чтения, а второй – процедурой вычисления.

Каждый черный ящик определяет один уровень синтаксиса. Процедура чтения определяет, как строки знаков могут транслироваться в объекты, называемые s-выражениями. Так как синтаксис s-выражений включает синтаксис для списков произвольных объектов, включая другие списки, s-выражения могут представлять произвольные древовидные выражения (*tree expressions*), очень похожие на абстрактные синтаксические деревья, генерируемые синтаксическими анализаторами не-Lisp языков.

В свою очередь, процедура вычисления определяет синтаксис форм Lisp, которые могут быть построены из s-выражений. Не все s-выражения являются допустимыми формами Lisp также как и не все последовательности знаков являются допустимыми s-выражениями. Например, и `(foo 1 2)`, и `("foo" 1 2)` являются s-выражениями, но только первое может быть формой Lisp, так как список, который начинается со строки, не является формой Lisp.

Это разделение черного ящика имеет несколько следствий. Одно из них состоит в том, что вы можете использовать s-выражения, как вы видели в главе 3, в качестве внешнего формата для данных, не являющихся исходным кодом, используя **READ** для их чтения и **PRINT** для их записи. Другое следствие состоит в том, что так как семантика языка определена в терминах деревьев объектов, а не в терминах строк знаков, то генерировать код внутри языка становится легче, чем это можно было бы сделать, если бы код генерировался как текст. Генерирование кода полностью с нуля не намного легче: и построение списков, и построения строк являются примерно одинаковыми по сложности работами. Однако реальный выигрыш в том, что вы можете генерировать код, манипулируя существующими данными. Это является базой для макросов Lisp, которые я опишу гораздо подробнее в будущих главах. Сейчас я сфокусируюсь на двух уровнях синтаксиса, определенных Common Lisp: это синтаксис s-выражений, понимаемый процедурой чтения, и синтаксис форм Lisp, понимаемый процедурой вычисления.

S-выражения

Базовыми элементами s-выражения являются списки и атомы. Списки ограничиваются скобками и могут содержать любое число разделенных пробелами элементов. Все, что не список, является атомом. Элементами списков в свою очередь также являются s-выражения (другими словами, атомы или вложенные списки). Комментарии (которые, строго говоря, не являются s-выражениями) начинаются с точки с запятой, распространяются до конца строки, и трактуются как пробел.

И это почти все. Так как списки синтаксически просты, то те оставшиеся синтаксические правила, которые вам необходимо знать, касаются только различных типов атомов. В этой секции я опишу правила для большинства часто используемых типов атомов: чисел, строк и имен. После этого, я расскажу как s-выражения, составленные из этих элементов, могут быть вычислены как формы Lisp.

С числами все довольно очевидно: любая последовательность цифр (возможно, начинающаяся со знака (+ или -), содержащая десятичную точку или знак деления, и, возможно,

заканчивающаяся меткой показателя степени) трактуется как число. Например:

```
123 ; целое число "сто двадцать три"
3/7 ; отношение "три седьмых"
1.0 ; число с плавающей точкой "один" с точностью, заданной по умолчанию
1.0e0 ; другой способ записать то же самое число с плавающей точкой
1.0d0 ; число с плавающей точкой "один" двойной точности
1.0e-4 ; эквивалент с плавающей точкой числа "одна десяти тысячная"
+42 ; целое число "сорок два"
-42 ; целое отрицательное число "минус сорок два"
-1/4 ; отношение "минус одна четвертая"
-2/8 ; другой способ записать то же отношение
246/2 ; другой способ записать целое "сто двадцать три"
```

Эти различные формы представляют различные типы чисел: целые, рациональные, числа с плавающей точкой. Lisp также поддерживает комплексные числа, которые имеют свою собственную нотацию, и которые мы рассмотрим в главе 10.

Как показывают некоторые из этих примеров, вы можете задать одно и то же число множеством различных способов. Но независимо от того, как вы запишите их, все рациональные (целые и отношения) внутри Lisp представляются в "упрощенной" форме. Другими словами, объекты, которые представляют числа $-2/8$ и $246/2$, не отличаются от объектов, которые представляют числа $-1/4$ и 123 . Таким же образом, 1.0 и $1.0e0$ – просто два разных способа записать одно число. С другой стороны, 1.0 , $1.0d0$ и 1 могут представлять различные объекты, так как различные представления чисел с плавающей точкой и целых чисел являются различными типами. Мы рассмотрим детали характеристик различных типов чисел в главе 10.

Строковые литералы, как вы видели в предыдущей главе, заключаются в двойные кавычки. Внутри строки обратный слеш (\) экранирует следующий знак, что вызывает включение этого знака в строку "как есть". Только два знака должны быть экранированы в строке: двойная кавычка и сам обратный слеш. Все остальные знаки могут быть включены в строковый литерал без экранирования, не обращая внимания на их значение вне строки. Несколько примеров строковых литералов:

```
"foo" ; строка, содержащая знаки 'f', 'o' и 'o'.
"fo\" ; такая же строка.
"fo\\\" ; строка, содержащая знаки 'f', 'o', '\\' и 'o'.
"fo\"o\" ; строка, содержащая знаки 'f', 'o', '\"' и 'o'.
```

Имена, используемые в программах на Lisp, такие как **FORMAT**, `hello-world` и `*db*` представляются объектами, называемыми *символами*. Процедура чтения ничего не знает о том, как данное имя будет использоваться – является ли оно именем переменной, функции или чем-то еще. Она просто читает последовательность знаков и создает объект, представляющий имя. Почти любой знак может входить в имя. Однако, это не может быть пробельный знак, так как пробелом разделяются элементы списка. Цифры могут входить в имена, если имя целиком не сможет интерпретироваться как число. Схожим образом, имена могут содержать точки, но процедура чтения не может прочитать имя, состоящее только из точек. Существует десять знаков, которые не могут входить в имена, так как предназначены для других синтаксических целей: открывающая и закрывающая скобки, двойные и одинарные кавычки, обратный апостроф, запятая, двоеточие, точка с запятой, обратный слеш и вертикальная черта. Но даже эти знаки могут входить в имена, если их экранировать обратным слешем или окружить часть имени, содержащую знаки, которые нужно экранировать, с помощью вертикальных линий.

Две важные характерные черты того, каким образом процедура чтения переводит имена в символьные объекты, касаются того, как она трактует регистр букв в именах и как она

обеспечивает то, чтобы одинаковые имена всегда читались как одинаковые символы. Во время чтения имен процедура чтения конвертирует все незранированные знаки в именах в их эквивалент в верхнем регистре. Таким образом, процедура чтения прочитает `foo`, `Foo` и `FOO` как одинаковый символ: `FOO`. Однако, `\f\o\o` и `|foo|` оба будут прочитаны как `foo`, что будет отличным от символа `FOO` объектом. Это как раз и является причиной, почему при определении функции в REPL, он печатает имя функции, преобразованное к верхнему регистру. Сейчас стандартным стилем является написание кода в нижнем регистре, позволяя процедуре чтения преобразовывать имена к верхнему.

Чтобы быть уверенным в том, что одно и то же текстовое имя всегда читается как один и тот же символ, процедура чтения хранит все символы — после того, как она прочитала имя и преобразовала его к верхнему регистру, процедура чтения ищет в таблице, называемой *пакетом* (*package*), символ с таким же именем. Если она не может найти такой, то она создает новый символ и добавляет его к таблице. Иначе она возвращает символ, уже хранящийся в таблице. Таким образом, где бы одно и то же имя не появлялось в любых s-выражениях, оно будет представлено одним и тем же объектом.

Так как имена в Lisp могут содержать намного большее множество знаков, чем в языках, произошедших от Algol, в Lisp существуют определенные соглашения по именованию, такие как использование дефисов в именах наподобие `hello-world`. Другое важное соглашение состоит в том, что глобальным переменным дают имена, начинающиеся и заканчивающиеся знаком `*`. Подобным образом, константам дают имена, начинающиеся и заканчивающиеся знаком `+`. Также некоторые программисты называют очень низкоуровневые функции именами, начинающимися с `%` или даже `%%`. Имена, определенные в стандарте языка, используют только алфавитные знаки (A-Z), а также `*`, `+`, `-`, `/`, `1`, `2`, `<`, `=`, `>`, `&`.

Синтаксис для списков, чисел, строк и символов описывает большую часть Lisp программ. Другие правила описывают нотацию для векторных литералов, отдельных знаков, массивов, которые я опишу в главах 10 и 11, когда мы будем говорить об этих типах данных. Сейчас главным является понимание того, как комбинируются числа, строки и символы с разделенными скобками списками для построения s-выражений, представляющих произвольные деревья объектов. Несколько простых примеров:

```
x ; символ X
() ; пустой список
(1 2 3) ; список из трех элементов
("foo" "bar") ; список из двух строк
(x y z) ; список из трех символов
(x 1 "foo") ; список из символа, числа и строки
(+ (* 2 3) 4) ; список из символа, списка и числа
```

Еще одним чуть более сложным примером является четырехэлементный список, содержащий два символа, пустой список и другой список, в свою очередь содержащий два символа и строку:

```
(defun hello-world ()
  (format t "hello, world"))
```

S-выражения как формы Lisp

После того, как процедура чтения преобразовывает текст в s-выражения, эти s-выражения могут быть вычислены как код Lisp. Точнее некоторые из них могут — не каждое s-выражение, которое процедура чтения может прочитать, обязательно может быть вычислено как код Lisp. Правила вычислений Common Lisp определяют второй уровень синтаксиса, который определяет, какие s-выражения могут трактоваться как формы Lisp. Синтаксические правила на этом уровне очень

просты. Любой атом (не список или пустой список) является допустимой формой Lisp, а также любой список, который содержит символ в качестве своего первого элемента, также является допустимой формой Lisp.

Конечно, интересным является не синтаксис форм Lisp, а то, как эти формы вычисляются. Для целей дальнейшего обсуждения вам достаточно думать о процедуре вычисления как о функции, которая получает в качестве аргумента синтаксически правильную форму Lisp и возвращает значение, которое мы можем назвать *значением* (*value*) формы. Конечно, когда процедура вычисления является компилятором, это является небольшим упрощением – в этом случае процедура вычисления получает выражение и генерирует код, который, будучи запущенным, вычислит соответствующее значение. Но это упрощение позволит мне описать семантику Common Lisp в терминах того, как различные типы форм Lisp вычисляются с помощью этой воображаемой функции.

Простейшие формы Lisp, атомы, могут быть разделены на две категории: символы и все остальное. Символ, вычисляемый как форма, трактуется как имя переменной и вычисляется в ее текущее значение. Я обсужу в главе 6 как переменные получают свои значения впервые. Также следует заметить, что некоторые "переменные" являются старым программистским оксюмороном: "константными переменными". Например, символ **PI** именуется константную переменную, чье значение – число с плавающей точкой, являющееся наиболее близкой аппроксимацией математической константы π .

Все остальные атомы (числа и строки) являются типом объектов, который вы уже рассмотрели – это *самовычисляемые объекты* (*self-evaluating objects*). Это означает, что когда выражение передается в воображаемую функцию вычисления, оно просто возвращается. Вы видели примеры самовычисляемости объектов в главе 2, когда набирали 10 и "hello, world" в REPL.

Символы также могут быть самовычисляемыми в том смысле, что переменной, которую именуется такой символ, может быть присвоено значение самого этого символа. Две важные константы определены таким образом: **T** и **NIL**, стандартные истинное и ложное значения. Я обсужу их роль как логических выражений в секции "Правда, ложь и равенство".

Еще один класс самовычисляемых символов – это *символы-ключи* (*keyword symbols*) – символы, чьи имена начинаются с `:`. Когда процедура чтения обрабатывает такое имя, она автоматически определяет константную переменную с таким именем и таким символом в качестве значения.

Все становится гораздо интереснее при рассмотрении того, как вычисляются списки. Все допустимые формы списков начинаются с символа, но существуют три разновидности форм списков, которые вычисляются тремя различными способами. Для определения того, какую разновидность формы представляет из себя данный список, процедура вычисления должна определить чем является первый символ списка: именем функции, макросом или специальным оператором. Если символ еще не был определен (такое может быть в случае, если вы компилируете код, который содержит ссылки на функции, которые будут определены позднее) – предполагается, что он является именем функции. Я буду ссылаться на эти три разновидности форм как на *формы вызова функции* (*function call forms*), *формы макросов* (*macro forms*) и *специальные формы* (*special forms*).

Вызовы функций

Правило вычисления для форм вызова функции просто: вычисление элементов списка, начиная со второго, как форм Lisp и передача результатов в функцию, именованную первым элементом. Это правило явно добавляет несколько дополнительных синтаксических ограничений на форму вызова функции: все элементы списка после первого должны также быть правильными формами Lisp. Другими словами, базовый синтаксис формы вызова функции следующий (каждый

аргумент также является формой Lisp):

```
(function-name argument*)
```

Таким образом, следующее выражение вычисляется путем первоначального вычисления 1, затем 2, а затем передачи результатов вычислений в функцию +, которая возвращает 3:

```
(+ 1 2)
```

Более сложное выражение, такое как следующее, вычисляется схожим образом за исключением того, что вычисление аргументов (+ 1 2) и (- 3 4) влечет за собой вычисление аргументов этих форм и применение соответствующих функций к ним:

```
(* (+ 1 2) (- 3 4))
```

В итоге, значения 3 и -1 передаются в функцию *, которая возвращает -3.

Как показывают эти примеры, функции используются для многих вещей, которые требуют специального синтаксиса в других языках. Это помогает сохранять синтаксис Lisp регулярным.

Специальные операторы

Нужно сказать, что не все операции могут быть определены как функции. Так как все аргументы функции вычисляются перед ее вызовом, не существует возможности написать функцию, которая ведет себя как оператор **IF**, который вы использовали в главе 3. Для того, чтобы увидеть почему, рассмотрим такую форму:

```
(if x (format t "yes") (format t "no"))
```

Если **IF** является функцией, процедура вычисления будет вычислять аргументы выражения слева направо. Символ *x* будет вычислен как переменная, возвращающая свое значение; затем как вызов функции будет вычислена (format t "yes"), возвращающая **NIL** после печати "yes" на стандартный вывод; и затем будет вычислена (format t "no"), печатающая "no" и возвращающая **NIL**. Только после того, как эти три выражения будут вычислены, их результаты будут переданы в **IF**, слишком поздно для того, чтобы проконтролировать, какое из двух выражений **FORMAT** будет вычислено.

Для решения этой проблемы Common Lisp определяет небольшое количество так называемых специальных операторов (и один из них **IF**), которые делают те вещи, которые функции сделать не могут. Всего их 25, но только малая их часть напрямую используется в ежедневном программировании.

Если первый элемент списка является символом, именуемым специальным оператором, остальная часть выражения вычисляется в соответствии с правилом для этого оператора.

Правило для **IF** очень просто: вычисление первого выражения. Если оно вычисляется не в **NIL**, то вычисляется следующее выражение и возвращается его результат. Иначе возвращается значение вычисления третьего выражения или **NIL**, если третье выражение не задано. Другими словами, базовая форма выражения **IF** следующая:

```
(if test-form then-form [ else-form ])
```

test-form вычисляется всегда, а затем только одна из then-form и else-form.

Еще более простой специальный оператор – это **QUOTE**, который получает одно выражение как аргумент и просто возвращает его не вычисляя. Например, следующая форма вычисляется в список `(+ 1 2)`, а не значение 3:

```
(quote (+ 1 2))
```

Этот список не отличается ни от какого другого, вы можете манипулировать им также, как и любым другим, который вы можете создать с помощью функции **LIST**

QUOTE используется достаточно часто, поэтому для него в процедуру чтения был встроен специальный синтаксис. Вместо написания такого:

```
(quote (+ 1 2))
```

вы можете написать это:

```
'(+ 1 2)
```

Этот синтаксис является небольшим расширением синтаксиса s-выражений, понимаемым процедурой чтения. С этой точки зрения для процедуры вычисления оба этих выражения выглядят одинаково: список, чей первый элемент является символом **QUOTE**, а второй элемент – список `(+ 1 2)`.

В общем, специальные операторы реализуют возможности языка, которые требуют специальной обработки процедурой вычисления. Например, некоторые специальные операторы манипулируют окружением, в котором вычисляются другие формы. Один из них, который я обсужу детально в главе 6, – **LET**, который используется для создания новой *привязки переменной* (*variable binding*). Следующая форма вычисляется в 10, так как второй `x` вычисляется в окружении, где он именует переменную, связанную оператором **LET** со значением 10:

```
(let ((x 10)) x)
```

Макросы

В то время как специальные операторы расширяют синтаксис Common Lisp, выходя за пределы того, что может быть выражено простыми вызовами функций, множество специальных операторов ограничено стандартом языка. С другой стороны, макросы дают пользователям языка способ расширения его синтаксиса. Как вы увидели в главе 3, макрос – это функция, которая получает в качестве аргументов s-выражения и возвращает форму Lisp, которая затем вычисляется на месте формы макроса. Вычисление формы макроса происходит в две фазы: сначала элементы формы макроса передаются, не вычисляясь, в функцию макроса, а затем форма, возвращенная функцией макроса (называемая ее *раскрытием* (*expansion*)), вычисляется в соответствии с обычными правилами вычисления.

Очень важно понимать обе фазы вычисления форм макросов. Очень легко запутаться когда вы печатаете выражения в REPL, так как эти две фазы происходят одна за одной и значение второй фазы немедленно возвращается. Но, когда код Lisp компилируется, эти два фазы выполняются в разное время, и очень важно понимать, что и когда происходит. Например, когда вы компилируете весь файл с исходным кодом с помощью функции **COMPILE-FILE**, все формы макросов в файле рекурсивно раскрываются, пока код не станет содержать ничего кроме форм вызова функций и специальных форм. Этот не содержащий макросов код затем компилируется в файл FASL, который функция **LOAD** знает как загрузить. Скомпилированный код, однако, не

выполняется пока файл не будет загружен. Так как макросы генерируют свое расширение во время компиляции, они могут проделывать довольно большой объем работы, генерируя свои раскрытия, без платы за это во время загрузки файла или при вызове функций, определенных в этом файле.

Так как процедура вычисления не вычисляет элементы формы макроса перед передачей их в функцию макроса, они не обязательно должны быть правильными формами Lisp. Каждый макрос назначает смысл s-выражениям, используемым в *форме* этого макроса (macro form), посредством того, как он использует эти s-выражения для генерации своего расширения. Другими словами, каждый макрос определяет свой собственный локальный синтаксис. Например, макрос переворачивания списка задом наперед из главы 3 определяет синтаксис, в котором выражение является допустимой перевернутой формой если ее список, будучи перевернутым, является допустимой формой Lisp.

Я расскажу больше о макросах в этой книге. А сейчас вам важно понимать, что макросы, несмотря на то, что синтаксически похожи на вызовы функции, служат иной цели, предоставляя добавочный уровень к компилятору.

Истина, Ложь и Равенство

Оставшейся частью базовых знаний, которые вам необходимо получить, являются понятия истины, лжи и равенства объектов в Common Lisp. Понятия истины и лжи очень просты: символ **NIL** является единственным ложным значением, а все остальное является истиной. Символ **T** является каноническим истинным значением и может быть использован когда вам нужно вернуть не-**NIL** значение, но само значение не важно. Единственной хитростью является то, что **NIL** также является единственным объектом, который одновременно является и атомом и списком: вдобавок к представлению ложного значения он также используется для представления пустого списка. Эта равнозначность **NIL** и пустого списка встроена в процедуру чтения: если процедура чтения видит `()`, она считывает это как символ **NIL**. Обе записи полностью взаимозаменяемые. И так как **NIL**, как я уже упоминал раньше, является именем константной переменной, значением которой является символ **NIL**, то выражения `nil`, `()`, `'nil` и `'()` вычисляются в одинаковый объект: unquoted формы вычисляются как ссылки на константную переменную, чье значения – символ **NIL**, а quoted формы специальный оператор **QUOTE** вычисляет в символ напрямую. По этим же причинам, и `t` и `'t` будут вычислены в одинаковый объект: символ **T**.

Использование фраз, таких как "то же самое", конечно рождает вопрос о том, что для двух значений значит "то же самое". Как вы увидите в следующих главах, Common Lisp предоставляет ряд типо-зависимых предикатов равенства: `=` используется для сравнения чисел; `CHAR=` для сравнения знаков и т.д. В этой секции мы рассмотрим четыре "общих" ("generic") предиката равенства – функции, которым могут быть переданы два Lisp-объекта, и которые возвратят истину, если эти объекты эквивалентны, и ложь в противном случае. Вот они в порядке ослабления понятия "различности": **EQ**, **EQL**, **EQUAL**, и **EQUALP**.

EQ проверяет "идентичность объектов": она возвращает истинное значение если два объекта идентичны. К сожалению, понятие идентичности таких объектов, как числа и знаки, зависит от того, как эти типы данных реализованы в конкретной реализации Lisp. Таким образом, **EQ** может считать два числа или два знака с одинаковым значением, как эквивалентными, так и нет. Стандарт языка оставляет реализациям достаточную свободу действий в этом вопросе, что приводит к тому, что выражение `(eq 3 3)` может вполне законно вычисляться как в истинное, так и в ложное значение. Таким же образом `(eq x x)` может вычисляться как в истинное, так и в ложное значение в различных реализациях если значением `x` является число или знак.

Поэтому вы никогда не должны использовать **EQ** для сравнения значений, которые могут

оказаться числами или знаками. Может показаться, что она вполне предсказуемо работает для некоторых значений в конкретной реализации, но вы не можете гарантировать, что она будет работать таким же образом если вы смените реализацию. К тому же смена реализации может означать просто обновление вашей реализации до новой версии: если конструкторы вашей реализации изменили внутреннее представление чисел или знаков, то поведение **EQ** вполне могло измениться.

Поэтому, Common Lisp определяет **EQL**, работающую аналогично **EQ**, за исключением того, что она также гарантирует рассмотрение эквивалентными двух объектов одного класса, представляющих одинаковое числовое или знаковое (character) значение. Поэтому (eq1 1 1) гарантировано будет истиной. А (eq1 1 1.0) гарантировано будет ложью, так как целое значение 1 и значение с плавающей точкой 1.0 являются представителями различных классов.

Существуют два лагеря по отношению к вопросу где использовать **EQ** и где использовать **EQL**: сторонники "когда возможно всегда используйте **EQ**" убеждают вас использовать **EQ**, когда вы уверены, что не будете сравнивать числа или знаки так как: а) это способ указать, что вы не собираетесь сравнивать числа и знаки; б) это будет немного эффективней, так как **EQ** не нужно проверять, являются ли ее аргументы числами или знаками.

Сторонники "всегда используйте **EQL**" советуют вам никогда не использовать **EQ**, так как (а) потенциальный выигрыш в ясности теряется, так как каждый раз, когда кто-либо будет читать ваш код (включая вас) и увидит **EQ**, он должен будет остановиться и проверить, корректно ли эта функция используется (другими словами, проверить, что она никогда не вызывается для сравнения цифр или знаков) и (б) различие в эффективности между **EQ** и **EQL** очень мало по сравнению с производительностью в действительно узких местах.

Код в этой книге написан в стиле "всегда используйте **EQL**".

Другие два предиката равенства **EQUAL** и **EQUALP** являются общими в том смысле, что они могут оперировать всеми типами объектов, но они не настолько фундаментальные, как **EQ** или **EQL**. Каждый из них определяет несколько более слабое понятие "различности", чем **EQL**, позволяя другим объектам считаться эквивалентным. Нет ничего особенного в тех конкретных понятиях эквивалентности, что реализуют эти функции, за исключением того, что они оказались полезными Lisp-программистам прошлого. Если эти предикаты не подходят вам, вы всегда можете определить свой собственный предикат для сравнения объектов других типов нужным вам способом.

EQUAL ослабляет понятие "различности" между **EQL**, считая списки эквивалентными, если они рекурсивно, согласно тому же **EQUAL**, имеют одинаковую структуру и содержимое. **EQUAL** также считает строки эквивалентными, если они содержат одинаковые знаки. **EQUAL** также ослабляет понятие "различности" по сравнению с **EQL** для битовых векторов (bit vectors) и путей — двух типах, о которых я расскажу в следующих главах. Для всех остальных типов он аналогичен **EQL**.

EQUALP аналогична **EQUAL** за исключением еще большего ослабления понятия "различности". **EQUALP** считает две строки эквивалентными, если они имеют одинаковые знаки, игнорируя разницу в регистре. Два знака также считаются эквивалентными, если они отличаются только регистром. Числа эквивалентны по **EQUALP**, если они представляют одинаковое математическое значение. Например, (equalp 1 1.0) вернет истину. Списки, элементы которых попарно эквивалентны по **EQUALP**, считаются эквивалентными; подобным же образом массивы с элементами, эквивалентными по **EQUALP**, также считаются эквивалентными. Как и в случае с **EQUAL**, существует несколько других типов данных, которые я пока не рассмотрел, для которых **EQUALP** может рассмотреть два объекта эквивалентными, в то время как **EQL** и **EQUAL** будут считать их различными. Для всех остальных типов данных **EQUALP** аналогична

Форматирование кода Lisp

Хотя форматирование кода, строго говоря, не имеет ни синтаксического, ни семантического значения, хорошее форматирование важно для легкого чтения и написания кода. Ключевым моментов в форматировании кода Lisp является правильная расстановка отступов. Отступы должны отражать структуру кода так, чтобы вам не пришлось считать скобки для его понимания. Вообще, каждый новый уровень вложенности должен иметь больший отступ, а если нужен перенос строки, то элементы следующей строки имеют тот же уровень вложенности, что и предыдущей. Таким образом, вызов функции, который должен быть разбит на несколько строк может быть записан следующим образом:

```
(some-function arg-with-a-long-name
               another-arg-with-an-even-longer-name)
```

Расстановка отступов в макросах и специальных формах, которые реализуют структуры контроля, обычно немного отличается: элементы "тела" отступаются на два пробела относительно открывающей скобки формы. Таким образом:

```
(defun print-list (list)
  (dolist (i list)
    (format t "item: ~a~%" i)))
```

Однако, вам не нужно сильно беспокоиться на счет этих правил, так как хорошая среда Lisp, такая как SLIME, возьмет эту заботу на себя. Фактически, одним из преимуществ регулярного синтаксиса Lisp является то, что программному обеспечению, такому как текстовые редакторы, очень легко расставлять отступы. Так как расстановка отступов нужна для отражения структуры кода, а структура определяется скобками, легко позволить редактору расставить отступы вместо вас.

В SLIME нажатие Tab в начале каждой строки приводит к тому, что строка будет правильно выровнена; также вы можете перевыровнять целое выражение, поставив курсор на открывающую скобку и набрав C-M-q. Или вы можете перевыровнять все тело функции, набрав C-c M-q, находясь где угодно в теле функции.

На самом деле, опытный Lisp-программист предпочитает полагаться на текстовый редактор, который обработает отступы автоматически, не только для того, чтобы код выглядел красиво, но и для обнаружения опечаток: как только вы привыкните к тому, как отступы в коде расставляются правильно, потерянная скобка будет легко распознаваться по странной расстановке отступов вашим редактором. Например, предположим, что вы написали следующую функцию:

```
(defun foo ()
  (if (test)
      (do-one-thing)
      (do-another-thing)))
```

Теперь предположим, что вы случайно не закрыли левую скобку после test. Поскольку вы не обеспокоены подсчетом скобок, вы просто добавите дополнительную скобку в конец формы **DEFUN**, получив следующий код:

```
(defun foo ()
  (if (test
      (do-one-thing)
      (do-another-thing))))
```

Однако, если вы выравнивали код, нажимая Tab в начале каждой строки, вы не получите вышеприведенный код. Вместо него вы получите это:

```
(defun foo ()
  (if (test
      (do-one-thing)
      (do-another-thing))))
```

Выравнивание веток if и then, перенесенных под условие вместо того, чтобы находиться чуть правее if, немедленно говорит нам, что что-то не так.

Другое важное правило форматирования заключается в том, что закрывающие скобки всегда помещаются в той же строке, что и последний элемент списка, который они закрывают. Так что не пишите так:

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)
  )
)
```

правильный вариант:

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)))
```

Строка))) в конце может казаться некрасивой, но когда ваш код имеет правильные отступы скобки должны уходить на второй план. Не нужно привлекать к ним несвоевременное внимание, располагая их на нескольких строках.

И, наконец, комментарии должны предваряться от одной до четырех точек с запятой, в зависимости от контекста появления этого комментария:

```
;;; Четыре точки с запятой для комментария в начале файла
;;; Комментарий из трех точек с запятой обычно является параграфом комментариев,
;;; который предваряет большую секцию кода
(defun foo (x)
  (dotimes (i x)
    ;; Две точки с запятой показывают, что комментарий применен к последующему
код.
    ;; Заметьте, что этот комментарий имеет такой же отступ, как и последующий
код.
    (some-function-call)
    (another i) ; этот комментарий применим только к этой строке
    (and-another) ; а этот для этой строки
    (baz)))
```

Теперь вы готовы начать более детально рассматривать важнейшие строительные блоки программ Lisp: функции, переменные и макросы. Следующим шагом станут функции.

5. Функции

Кроме правил синтаксиса и семантики следующие три компонента составляют основу всех программ на Lisp – функции, переменные и макросы. Вы использовали их во время создания базы данных в главе 3, но я опустил много подробностей о том, как они работают, и как их лучше всего использовать. Я посвящу следующие главы этим вопросам, начав с функций, которые, также как и их аналоги в других языках программирования, обеспечивают основные возможности абстракции.

Большая часть самого Lisp состоит из функций. Более трех четвертей имен, указанных в стандарте, являются именами функций. Все базовые типы данных полностью определены в терминах функций, работающими с ними. Даже мощная объектная система языка Lisp построена на концептуальном развитии понятий функции и обобщенной функции, которые будут описаны в главе 16.

В конце концов, несмотря на важность макросов (The Lisp Way!), вся реальная функциональность обеспечивается функциями. Макросы выполняются во время компиляции и создают код программы. После того, как все макросы будут раскрыты, этот код полностью будет состоять из обращения к функциям и специальным операторам. Я не упоминаю, что макросы сами являются функциями, которые используются для генерации кода, а не для выполнения действий в программе.

Определение новых функций

Обычно функции определяются при помощи макроса DEFUN. Типовое использование DEFUN выглядит вот так:

```
(defun name (parameter*)  
  "Optional documentation string."  
  тело-функции*)
```

В качестве имени может использоваться любой символ. Как правило, имена функций содержат только буквы, цифры и знак минус, но, кроме того, разрешено использование других символов, и они используются в определенных случаях. Например, функции, которые преобразуют значения из одного типа в другой, иногда используют символ `->` в имени. Или функция, которая преобразует строку в виджет, может быть названа `string->widget`. Наиболее важное соглашение по именованию, затронутое в главе 2, заключается в том, что лучше создавать составные имена, используя знак минус вместо подчеркивания или использования заглавных букв внутри имени. Так что `frob-widget` лучше соответствует стилю Lisp, чем `frob_widget` или `frobWidget`.

Список параметров функции определяет переменные, которые будут использоваться для хранения аргументов, переданных при вызове функции. Если функция не принимает аргументов, то список пуст и записывается как `()`. Различают обязательные, необязательные, множественные, и именованные (keyword) параметры. Эти вопросы будут обсуждаться подробнее в следующем разделе.

За списком параметров может находиться строка, которая описывает назначение функции. После того, как функция определена, эта строка (строка документации) будет ассоциирована с именем функции и может быть позже получена с помощью функции `DOCUMENTATION`.

Тело DEFUN состоит из любого числа выражений Lisp. При вызове функции они вычисляются по порядку, и результат вычисления последнего выражения возвращается, как значение функции.

Для возврата из любой точки функции может использоваться специальный оператор RETURN-FROM, что я продемонстрирую через некоторое время.

В главе 2 мы написали функцию hello-world, которая выглядела вот так:

```
(defun hello-world () (format t "hello, world"))
```

Теперь вы можете проанализировать части этой функции. Она называется hello-world, список параметров пуст, потому что она не принимает аргументов, в ней нет строки документации, и ее тело состоит из одного выражения:

```
(format t "hello, world")
```

Вот пример немного более сложной функции:

```
(defun verbose-sum (x y)
  "Sum any two numbers after printing a message."
  (format t "Summing ~d and ~d.~%" x y)
  (+ x y))
```

Эта функция называется verbose-sum, получает два аргумента, которые связываются с параметрами x и y, имеет строку документации, и ее тело состоит из двух выражений. Значение, возвращенное вызовом функции +, становится значением функции verbose-sum.

Списки параметров функций

Это всё, больше нечего сказать об именах функций или о строках документации. Оставшуюся часть книги мы будем описывать то, что можно написать в теле функции, поэтому мы остаемся наедине со списками параметров функций.

Основное назначение списков параметров – объявление переменных, которые будут использоваться для хранения аргументов, переданных функции. Когда список параметров является простым списком имен переменных, как в verbose-sum, то параметры называются *обязательными*. Когда функция вызывается, она должна получить ровно по одному аргументу для каждого из обязательных параметров. Каждый параметр связывается с соответствующим аргументом. Если функция вызывается с меньшим или большим количеством аргументов, чем требуется, то Lisp сообщит об ошибке.

Однако, списки параметров в Common Lisp предоставляют более удобные способы отображения аргументов функции в параметры функции. В дополнение к обязательным параметрам функция может иметь *необязательные* параметры. Или функция может иметь один параметр, который будет связан со списком, содержащим все дополнительные аргументы. И в заключение, аргументы могут быть связаны с параметрами путем использования *ключевых слов* (keywords), а не путем соответствия позиции параметра и аргумента в списке. Таким образом, списки параметров Common Lisp предоставляют удобное решение для некоторых общих задач кодирования.

Необязательные параметры

В то время как многие функции, подобно verbose-sum, нуждаются только в обязательных параметрах, не все функции являются настолько простыми. Иногда функции должны иметь параметр, который будет использоваться только при некоторых вызовах, поскольку он имеет "правильное" значение по умолчанию. Таким примером может быть функция, которая создает

структуру данных, которая будет при необходимости расти. Поскольку, структура данных может расти, то не имеет значения, по большей части, какой начальный размер она имеет. Но пользователь функции, который имеет понятие о том, сколько данных будет помещено в данную структуру, может улучшить производительность программы путем указания нужного начального размера этой структуры. Однако, большинство пользователей данной функции, скорее всего, позволят выбрать наиболее подходящий размер автоматически. В Common Lisp вы можете предоставить этим пользователям одинаковые возможности с помощью необязательных параметров; пользователи, которые не хотят устанавливать значение сами, получают разумное значение по умолчанию, а остальные пользователи смогут подставить нужное значение.

Для определения функции с необязательными параметрами после списка обязательных параметров поместите символ `&optional`, за которым перечислите имена необязательных параметров. Простой пример использования выглядит так:

```
(defun foo (a b &optional c d)
  (list a b c d))
```

Когда функция будет вызвана, сначала аргументы связываются с обязательными параметрами. После того, как обязательные параметры получили переданные значения, и остались еще аргументы, то они будут присвоены необязательным параметрам. Если аргументы закончатся до того, как кончится список необязательных параметров, то оставшиеся параметры получат значение `NIL`. Таким образом, предыдущая функция будет выдавать следующие результаты:

```
(foo 1 2) ==> (1 2 NIL NIL)
(foo 1 2 3) ==> (1 2 3 NIL)
(foo 1 2 3 4) ==> (1 2 3 4)
```

Lisp все равно будет проверять количество аргументов, переданных функции (в нашем случае это число от 2 до 4-х, включительно), и будет выдавать ошибку, если функция вызвана с лишними аргументами, или их, наоборот, не хватает.

Конечно, вы можете захотеть использовать другие значения по умолчанию, отличные от `NIL`. Вы можете указать их, путем замены имени параметра на список, состоящий из имени и выражения. Это выражение будет вычислено только если пользователь не указал значения для необязательного параметра. Общепринятым является простое задание конкретного значения в качестве выражения.

```
(defun foo (a &optional (b 10))
  (list a b))
```

Эта функция требует указания одного аргумента, который будет присвоен параметру `a`. Второй параметр – `b`, получит либо значение второго аргумента, если он указан, либо число 10.

```
(foo 1 2) ==> (1 2)
(foo 1) ==> (1 10)
```

Однако, иногда, вам потребуется большая гибкость в выборе значения по умолчанию. Вы можете захотеть вычислять значение по умолчанию основываясь на других параметрах. И вы можете сделать это – выражение для значения по умолчанию может ссылаться на параметры, ранее перечисленные в списке параметров. Если вы пишете функцию, которая возвращает что-то типа описания прямоугольников, и вы хотите сделать ее удобной для использования с квадратами, то вы можете использовать такой вот список параметров:


```
(defun make-rectangle (width &optional (height width))
  ...)
```

что сделает параметр `height` равным параметру `width`, если только он не будет явно задан.

Иногда полезно будет знать, было ли значение необязательного параметра задано пользователем, или использовалось значение по умолчанию. Вместо того, чтобы писать код, который проверяет, является ли переданное значение равным значению по умолчанию (это все равно не будет работать, поскольку пользователь может явно задать значение, равное значению по умолчанию), вы можете добавить еще одно имя переменной к списку параметров после выражения для значения по умолчанию. Указанная переменная будет иметь истинное значение, если пользователь задал значение для аргумента, и `NIL` в противном случае. По соглашению, эти переменные называются также как и параметры, но с добавлением `"-supplied-p"` к концу имени. Например:

```
(defun foo (a b &optional (c 3 c-supplied-p))
  (list a b c c-supplied-p))
```

Выполнение этого кода приведет к следующим результатам:

```
(foo 1 2) ==> (1 2 3 NIL)
(foo 1 2 3) ==> (1 2 3 T)
(foo 1 2 4) ==> (1 2 4 T)
```

Остаточные (Rest) параметры

Необязательные параметры применяются только тогда, когда у вас есть отдельные параметры, для которых пользователь может указывать или не указывать значения. Но некоторые функции могут требовать изменяемого количества аргументов. Некоторые встроенные функции, которые вы уже видели, работают именно так. Функция `FORMAT` имеет два обязательных аргумента – поток вывода и управляющую строку. Но кроме этого, он требует переменное количество аргументов, зависящее от того, сколько значений он должен вставить в управляющую строку. Функция `+` также получает переменное количество аргументов – нет никаких причин ограничиваться складыванием только двух чисел, эта функция может вычислять сумму любого количества значений. (Она даже может работать вообще без аргументов, возвращая значение 0.) Следующие примеры являются допустимыми вызовами этих двух функций:

```
(format t "hello, world")
(format t "hello, ~a" name)
(format t "x: ~d y: ~d" x y)
(+)
(+ 1)
(+ 1 2)
(+ 1 2 3)
```

Очевидно, что вы можете написать функцию с переменным числом аргументов, просто описывая множество необязательных параметров. Но это будет невероятно мучительно – простое написание списка параметров может быть не очень хорошим делом, и это не связывает все параметры с их использованием в теле функции. Для того, чтобы сделать это правильно, вы должны иметь число необязательных параметров равным максимальному допустимому количеству аргументов при вызове функций. Это число зависит от реализации, но гарантируется, что оно будет равно минимум 50. В текущих реализациях оно варьируется от 4,096 до 536,870,911. Хех! Этот мозгодробительный подход явно не является хорошим стилем

написания программ.

Вместо этого, Lisp позволяет вам указать параметр, который примет все аргументы (этот параметр указывается после символа `&rest`). Если функция имеет параметр `&rest` (остаточный параметр), то любые аргументы, оставшиеся после связывания обязательных и необязательных параметров, будут собраны в список, который станет значением остаточного параметра `&rest`. Таким образом, список параметров для функций `FORMAT` и `+` будут выглядеть примерно так:

```
(defun format (stream string &rest values) ...)
(defun + (&rest numbers) ...)
```

Именованные параметры =

Необязательные и остаточные (`rest`) параметры дают вам достаточно гибкости, но ни один из них не помогает вам в следующей ситуации: предположим, что вы имеете функцию, которая получает четыре необязательных параметра. Теперь предположим, что пользователь захочет задать значение только для одного из параметров, и даже, что пользователь захочет задать значение только для некоторых, расположенных не последовательно, параметров.

Пользователи, которые хотят задать значение для первого параметра не имеют никаких проблем – они просто передадут один необязательный параметр, и пропустят оставшиеся. Но что делать пользователям, которые хотят указать значения для других параметров – разве это не та проблема, которую должно решить использование необязательных параметров?

Конечно, это она. Но проблема заключается в том, что необязательные параметры все равно являются позиционными – если пользователь хочет указать четвертый необязательный параметр, то первые три необязательных параметра превращаются для этого пользователя в обязательные. К счастью, существует еще один вид параметров – именованные (`keyword`) параметры, которые позволяют указывать пользователю, какие значения будут связаны с конкретными параметрами.

Для того, чтобы задать именованные параметры, необходимо после всех требуемых, необязательных и остаточных параметров, указать символ `&key` и затем перечислить любое количество спецификаторов именованных параметров. Вот пример функции, которая имеет только именованные параметры:

```
(defun foo (&key a b c)
  (list a b c))
```

Когда функция вызывается, каждый именованный параметр связывается со значением, которое указано после ключевого слова, имеющего то же имя, что и параметр. Вернемся к главе 4, в которой указывалось, что ключевые слова – это имена, которые начинаются с двоеточия, и которые автоматически определяются как константы, вычисляемые сами в себя `FIXME` (`self-evaluating`).

Если ключевое слово не указано в списке аргументов, то соответствующий параметр получает значение по умолчанию, точно также как и для необязательный параметр. Поскольку именованные аргументы имеют метку, то они могут быть указаны в любом порядке, если они следуют после обязательных аргументов. Например, `foo` может быть вызвана вот так:

```
(foo) ==> (NIL NIL NIL)
(foo :a 1) ==> (1 NIL NIL)
(foo :b 1) ==> (NIL 1 NIL)
(foo :c 1) ==> (NIL NIL 1)
(foo :a 1 :c 3) ==> (1 NIL 3)
(foo :a 1 :b 2 :c 3) ==> (1 2 3)
(foo :a 1 :c 3 :b 2) ==> (1 2 3)
```

Также как и для необязательных параметров, именованные параметры могут задавать выражение для вычисления значения по умолчанию и имя `supplied-p`-переменной. И для необязательных, и для именованных параметров, значение по умолчанию может ссылаться на параметры, указанные ранее в списке.

```
(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ a b)))
  (list a b c b-supplied-p))
```

```
(foo :a 1) ==> (1 0 1 NIL)
(foo :b 1) ==> (0 1 1 T)
(foo :b 1 :c 4) ==> (0 1 4 T)
(foo :a 2 :b 1 :c 4) ==> (2 1 4 T)
```

Также, если по некоторым причинам вы хотите, чтобы пользователь использовал имена аргументов, отличающиеся от имен параметров, то вы можете заменить имя параметра на список, содержащий имя, которое будет использоваться пользователем при вызове, и имя параметра. Следующее определение `foo`:

```
(defun foo (&key ( (:apple a) ) ( (:box b) 0) ( (:charlie c) 0 c-supplied-p) )
  (list a b c c-supplied-p))
```

позволяет пользователю вызывать функцию вот так:

```
(foo :apple 10 :box 20 :charlie 30) ==> (10 20 30 T)
```

Этот стиль особенно полезен, если вы хотите полностью отделить публичный интерфейс от деталей внутренней реализации, поскольку обычно внутри вы хотите использовать короткие имена переменных, и значащие имена в программном интерфейсе. Однако, обычно это используется не особо часто.

Совместное использование разных типов параметров

Это возможно, но редко используется, использовать все четыре вида параметров в одной функции. Когда используется более одного типа параметров, они должны быть объявлены в порядке, который мы уже обсуждали – сначала указываются имена требуемых параметров, затем - необязательных, затем - остаточных (`&rest`), и в заключение - именованных параметров. Но обычно в функциях, которые используют несколько типов параметров, комбинируют требуемые параметры с одним из других видов параметров, или возможно комбинируют необязательные и остаточные параметры. Два других сочетания – необязательных или остаточных параметров с именованными параметрами, могут привести к очень удивительному поведению функции.

Комбинация необязательных и именованных параметров приносят достаточно сюрпризов, так что вы скорее всего должны избегать их совместного использования. Проблема заключается в том, что если пользователь не задает значений для всех необязательных параметров, то эти параметры получают имена и значения именованных параметров. Например, эта функция

использует необязательные и именованные параметры:

```
(defun foo (x &optional y &key z)
  (list x y z))
```

Если она вызывается вот так, то все нормально:

```
(foo 1 2 :z 3) ==> (1 2 3)
```

И вот так, все работает нормально:

```
(foo 1) ==> (1 nil nil)
```

Но в этом случае, она выдает ошибку:

```
(foo 1 :z 3) ==> ERROR
```

Это происходит потому, что имя параметра `:z` берется как значение для необязательного параметра `y`, оставляя для обработки только аргумент 3. При этом, Lisp ожидает, что в этом месте встретится либо пара имя/значение, либо не будет ничего, и одиночное значение приведет к выдаче ошибки. Будет даже хуже, если функция будет иметь два необязательных параметра, так что использование функции как в последнем примере, приведет к тому, что значения `:z` и 3 будут присвоены двум необязательным параметрам, а именованный параметр `z` получит значение по умолчанию – `NIL`, без всякого указания, что что-то произошло неправильно.

В общем, если вы обнаружите, что вы пишете функцию, которая использует и необязательные и именованные параметры, то вам лучше просто исправить ее для использования только именованных параметров – этот подход более гибок, и вы всегда сможете добавить новые параметры не беспокоя пользователей вашей функции. Вы можете даже удалять именованные параметры, если никто не использует их. Использование именованных параметров помогает сделать код более легким для сопровождения и развития – если вам нужно изменить поведение функции, так что это изменение потребует ввода новых параметров, вы можете добавить именованные параметры без изменения, или даже перекомпиляции кода, который использует эту функцию.

Вы можете безопасно комбинировать остаточные и именованные параметры, но вначале поведение может показаться немного удивительным. Обычно, наличие либо остаточных, либо именованных параметров приведет к тому, что значения, оставшиеся после заполнения всех обязательных и необязательных параметров, будут обработаны определенным образом – либо собраны в список (для остаточных параметров), или присвоены соответствующим именованным параметрам. Если в списке параметров используются и остаточные и именованные параметры, то выполняются оба действия – все оставшиеся значения собираются в список, который присваивается параметру `&rest`, а также соответствующие значения присваиваются именованным параметрам. Так что имея следующую функцию:

```
(defun foo (&rest rest &key a b c)
  (list rest a b c))
```

вы получите следующие результаты:

```
(foo :a 1 :b 2 :c 3) ==> ((:A 1 :B 2 :C 3) 1 2 3)
```

Возврат значений из функции

Все функции, которые уже были написаны, так или иначе использовали обычное поведение, заключающееся в возврате значения последнего вычисленного выражения как собственного возвращаемого значения. Это самый употребительный способ возврата значений из функции.

Однако, иногда бывает нужно вернуть значение из середины функции, вырываясь таким образом из вложенных управляющих конструкций. В таком случае вы можете использовать специальный оператор `RETURN-FROM`, который предназначен для немедленного возвращения любого значения из функции.

Вы увидите в главе 20 что `RETURN-FROM` в самом деле не привязана к функциям; она используется для возврата из блока кода, определенного с помощью оператора `BLOCK`. Однако, `DEFUN` автоматически помещает тело функции в блок кода с тем же именем, что и имя функции. Так что, вычисление `RETURN-FROM` с именем функции и значением, которое вы хотите вернуть, приведет к немедленному выходу из функции с возвратом указанного значения. `RETURN-FROM` является специальным оператором, чьим первым аргументом является имя блока из которого необходимо выполнить возврат. Это имя не вычисляется, так что нет нужды его экранировать.

Следующая функция использует вложенные циклы для нахождения первой пары чисел, каждое из которых меньше чем 10, и чье произведение больше заданного аргумента, и она использует `RETURN-FROM` для возврата первой найденной пары чисел:

```
(defun foo (n)
  (dotimes (i 10)
    (dotimes (j 10)
      (when (> (* i j) n)
        (return-from foo (list i j))))))
```

Надо отметить, что необходимость указания имени функции из которой вы хотите вернуться, является не особо удобной – если вы измените имя функции, то вам нужно будет также изменить имя, использованное в операторе `RETURN-FROM`. Но следует отметить, что явное использование `RETURN-FROM` в Lisp происходит значительно реже, чем использование выражения `return` в C-подобных языках, поскольку все выражения Lisp, включая управляющие конструкции, такие как условные выражения и циклы, вычисляются в значения. Так что это не представляет особой сложности на практике.

Функции как данные, или Функции высшего порядка

В то время как основной способ использования функций – это вызов их с указанием имени, существуют ситуации, когда было бы полезно рассматривать функции как данные. Например, вы можете передать одну функцию в качестве аргумента другой функции, вы можете написать общую функцию сортировки, и предоставить пользователю возможность указания функции для сравнения двух элементов. Так что один и тот же алгоритм может быть использоваться с разными функциями сравнения. Аналогично, обратные вызовы (callbacks) и `FIXME` hooks зависят от возможности хранения ссылок на исполняемый код, который можно выполнить позже. Поскольку функции уже являются стандартным способом представления частей кода, имеет смысл разрешить рассмотрение функций как данных.

В Lisp функции являются просто другим типом объектов. Когда вы определяете функцию с помощью `DEFUN`, вы в действительности делаете две вещи: создаете новый объект-функцию, и даете ему имя. Кроме того, возможно, как вы увидели в главе 3, использовать `LAMBDA` для создания функции без имени. Действительное представление объекта-функции, независимо от

того, именованный он или нет, является неопределенным — в компилируемых вариантах Lisp, они вероятно состоят в основном из машинного кода. Единственными вещами которые вам надо знать — как получить эти объекты, и как выполнять их, если вы их получили.

Специальный оператор `FUNCTION` обеспечивает механизм получения объекта-функции. Он принимает единственный аргумент и возвращает функцию с этим именем. Имя не экранируется. Так что, если вы определили функцию `foo`, например вот так:

```
CL-USER> (defun foo (x) (* 2 x))
FOO
```

вы можете получить объект-функцию следующим образом:

```
CL-USER> (function foo)
#<Interpreted Function FOO>
```

В действительности, вы уже использовали `FUNCTION`, но это было замаскировано. Синтаксис `#'`, который вы использовали в главе 3, является синтаксической оберткой для `FUNCTION`, точно также как и `'` является оберткой для `QUOTE`. Так что вы можете получить объект-функцию вот так:

```
CL-USER> #'foo
#<Interpreted Function FOO>
```

После того, как вы получили объект-функцию, есть только одна вещь, которую вы можете сделать с ней — выполнить ее. Common Lisp предоставляет две функции для выполнения функции через объект-функцию: `FUNCALL` и `APPLY`. Они отличаются тем, как они получают аргументы, которые будут переданы вызываемой функции.

`FUNCALL` это функция, которая используется тогда, когда во время написания кода вы знаете количество аргументов, которые вы будете передавать функции. Первым аргументом `FUNCALL` является запускаемый объект-функция, а оставшиеся аргументы передаются данной функции. Так что следующие два выражения являются эквивалентными:

```
(foo 1 2 3) === (funcall #'foo 1 2 3)
```

Однако довольно мало смысла в использовании `FUNCALL` для вызова функции, чье имя вы знаете во время написания кода. В действительности, два предыдущих выражения скорее всего скомпилируются в один и тот же машинный код.

Следующая функция демонстрирует более реалистичное использование `FUNCALL`. Она принимает объект-функцию в качестве аргумента, и рисует простую текстовую диаграмму, значений, возвращенных функцией, вызываемой для значений от `min` до `max` с шагом `step`.

```
(defun plot (fn min max step)
  (loop for i from min to max by step do
    (loop repeat (funcall fn i) do (format t "~*"))
    (format t "~%")))
```

Выражение `FUNCALL` вычисляет значение функции для каждого значения `i`. Внутренний цикл использует это значение для определения того, сколько раз напечатать знак "звездочка".

Заметьте, что вы не используете `FUNCTION` или `#'` для получения значения `fn`; вы хотите, чтобы оно интерпретировалось как переменная, поскольку значение этой переменной является

объектом-функцией. Вы можете вызвать `plot` с любой функцией, которая берет один числовой аргумент, например, со встроенной функцией `EXP`, которая возвращает значение e , возведенное в степень переданного аргумента.

```
CL-USER> (plot #'exp 0 4 1/2)
*
*
**
****
*****
*****
*****
*****
*****
*****
```

NIL

Однако `FUNCALL` не особо полезен, когда список аргументов становится известен только во время выполнения. Например, для работы с функцией `plot` в других случаях, представьте, что вы получили список, содержащий объект-функцию, минимальное и максимальное значения, а также шаг изменения значений. Другими словами, список содержит значения, которые вы хотите передать как аргументы для `plot`. Предположим, что этот список находится в переменной `plot-data`. Вы можете вызвать `plot` с этими значениями вот так вот:

```
(plot
  (first plot-data)
  (second plot-data)
  (third plot-data)
  (fourth plot-data))
```

Это работает нормально, но достаточно раздражает необходимость явного доставания аргументов лишь для того, чтобы передать их функции `plot`.

Это как раз тот случай, когда на помощь приходит `APPLY`. Подобно `FUNCALL`, ее первым аргументом является объект-функция. Но после первого аргумента, вместо перечисления отдельных аргументов, она принимает список. Затем `APPLY` применяет функцию к значениям в списке. Это позволяет вам переписать предыдущий код следующим образом:

```
(apply #'plot plot-data)
```

Кроме того, `APPLY` может также принимать "свободные" аргументы, также как и обычные аргументы в списке. Таким образом, если `plot-data` содержит только значения для `min`, `max` и `step`, то вы все равно можете использовать `APPLY` для отображения функции `EXP` используя следующее выражение:

```
(apply #'plot #'exp plot-data)
```

APPLY не заботится о том, использует ли функция необязательные, остаточные или именованные объекты – список аргументов создается путем объединения всех аргументов, и результирующий список должен быть правильным списком аргументов для функции с достаточным количеством аргументов для обязательных параметров и соответствующими именованными параметрами.

Анонимные функции

После того, как вы начали писать или даже просто использовать функции, которые принимают в качестве аргументов другие функции, вы, скорее всего, открыли для себя тот момент, что иногда раздражает необходимость определять и давать имя функции, которая будет использоваться в одном месте, особенно, если вы никогда не будете вызывать ее по имени.

Когда кажется, что определение новых функций с помощью DEFUN является излишним, вы можете создать "анонимную" функцию, используя выражение LAMBDA. Как обсуждалось в главе 3, LAMBDA-выражение выглядит примерно так:

```
(lambda (parameters) body)
```

Можно представить себе, что LAMBDA-выражения – это специальный вид имен функций, где само имя напрямую описывает что эта функция делает. Это объясняет, почему вы можете использовать LAMBDA-выражение вместо имени функции с #1.

```
(funcall #'(lambda (x y) (+ x y)) 2 3) ==> 5
```

Вы даже можете использовать LAMBDA-выражение как "имя" функции в выражениях, вызывающих функцию. Если вы хотите, то вы можете переписать предыдущий пример с FUNCALL в следующем виде.

```
( (lambda (x y) (+ x y)) 2 3) ==> 5
```

Но обычно так никогда не пишут, это использовалось лишь для демонстрации, что LAMBDA-выражения разрешено и можно использовать везде, где могут использоваться обычные функции.

Анонимные функции могут быть очень полезными, когда вы хотите передать одну функцию в качестве аргумента другой, и она достаточно проста для записи на месте. Например, предположим, что вы хотите нарисовать график функции $2x$. Вы можете определить следующую функцию:

```
(defun double (x) (* 2 x))
```

которую затем передать `plot`.

```
CL-USER> (plot #'double 0 10 1)
**
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

Но легче и более понятно написать вот так:

[illegible]

Другим очень важным применением LAMBDA-выражений является их использование для создания замыканий (closures) – функций, которые захватывают часть среды выполнения, в которой они были созданы. Вы уже использовали замыкания в главе 3, но подробное описание о том, как замыкания работают и как они могут использоваться, больше относится к переменным, а не к функциям, так что я отложу это обсуждение до следующей главы.

6. Переменные

Следующим базовым строительным блоком, с которым нам нужно ознакомиться, — переменные. Common Lisp поддерживает два вида переменных: лексические и динамические. Эти два типа переменных примерно соответствуют "локальным" и "глобальным" переменным других языков. Однако, это соответствие лишь приблизительно. С одной стороны "локальные" переменные некоторых языков в действительности гораздо ближе к динамическим переменным Common Lisp. И, с другой, локальные переменные некоторых других языков имеют *лексическую область видимости* не предоставляя всех возможностей, предоставляемых лексическими переменными Common Lisp. В частности, не все языки, предоставляющие переменные, имеющие лексическую область видимости, поддерживают замыкания.

Чтобы сделать все еще более запутанным, многие формы, которые работают с переменными, могут использоваться как с лексическими, так и с динамическими переменными. Поэтому я начну с обсуждения некоторых аспектов переменных Lisp, которые применимы к обоим видам переменных, а затем рассмотрю специфические характеристики лексических и динамических переменных. Далее я обсужу оператор присваивания общего назначения Common Lisp, **SETF**, который используется для присваивания новых значений переменным и просто почти каждому месту, которое может содержать значение.

Основы переменных

Как и в других языках, в Common Lisp переменные являются именованными местами, которые могут содержать значения. Однако, в Common Lisp переменные не типизированы таким же образом, как в таких языках, как Java или C++. То есть вам не нужно описывать тип объектов, которые может содержать каждая переменная. Вместо этого переменная может содержать значения любого типа и сами значения содержат информацию о типе, которая может быть использована для проверки типов во время выполнения. Таким образом, Common Lisp является *динамически типизированным*: ошибки типов выявляются динамически. Например, если вы передадите не число в функцию `+`, Common Lisp сообщит вам об ошибке типов. С другой стороны, Common Lisp является *строго типизированным* языком в том смысле, что все ошибки типов будут обнаружены: нет способа представить объект в качестве экземпляра класса, которым он не является.

Все значения в Common Lisp, по крайней мере концептуально, являются ссылками на объекты. Поэтому присваивание переменной нового значения изменяет то, на *какой* объект ссылается переменная (то есть, куда ссылается переменная), но не оказывает никакого влияния на объект, на который переменная ссылалась ранее. Однако, если переменная содержит ссылку на изменяемый объект, вы можете использовать данную ссылку для изменения этого объекта, и это изменение будет видимо любому коду, который имеет ссылку на этот же объект.

Один из способов введения новой переменной вы уже использовали при определении параметров функции. Как вы видели в предыдущей главе, при определении функции с помощью **DEFUN** список параметров определяет переменные, которые будут содержать аргументы, переданные функции при вызове. Например, следующая функция определяет три переменные для хранения своих аргументов: `x`, `y` и `z`.

```
(defun foo (x y z) (+ x y z))
```

При каждом вызове функции Lisp создает новые *привязки* (*bindings*) для хранения аргументов, переданных при вызове этой функции. Привязка является проявлением переменной во время выполнения. Отдельная переменная — сущность, на которую вы можете сослаться в исходном

коде своей программы — может иметь множество различных привязок за время выполнения программы. Отдельная переменная даже может иметь множество привязок в одно и то же время: параметры рекурсивной функции, например, связываются заново (rebound) при каждом вызове этой функции.

Другой формой, позволяющей вводить новые переменные, является специальный оператор **LET**. Шаблон формы **LET** имеет следующий вид:

```
(let (variable*)
  body-form*)
```

где каждая *variable* является формой инициализации переменной. Каждая форма инициализации является либо списком, содержащим имя переменной и форму начального значения, либо, как сокращение для инициализации переменной в значение **NIL**, просто именем переменной. Следующая форма **LET**, например, связывает три переменные *x*, *y* и *z* с начальными значениями 10, 20 и **NIL**:

```
(let ((x 10) (y 20) z)
  ...)
```

При вычислении формы **LET** сначала вычисляются все формы начальных значений. Затем, перед выполнением форм тела, создаются и инициализируются в соответствующие начальные значения новые привязки. Внутри тела **LET** имена переменных ссылаются на только что вновь созданные привязки. После **LET** имена продолжают ссылаться на то, на что они ссылались перед **LET** (если они на что-то ссылались).

Значение последнего выражения тела возвращается как значение выражения **LET**. Как и параметры функций, переменные, вводимые **LET**, связываются заново (rebound) каждый раз, когда поток управления заходит в **LET**.

Область видимости (scope) параметров функций и переменных **LET** — область программы, где имя переменной может быть использовано для ссылки на привязку переменной — ограничивается формой, которая вводит переменную. Такая форма (определение функции или **LET**) называется *связывающей формой (binding form)*. Как вы скоро увидите, два типа переменных (лексические и динамические) используют два несколько отличающихся механизма области видимости, но в обоих случаях область видимости ограничена связывающей формой.

Если вы записываете вложенные связывающие формы, которые вводят переменные с одинаковыми именами, то привязки внутренних переменных скрывают внешние привязки. Например, при вызове следующей функции для параметра *x* создается привязка для хранения аргумента функции. Затем первая **LET** создает новую привязку с начальным значением 2, а внутренняя **LET** создает еще одну привязку с начальным значением 3. Комментарии справа указывают область видимости каждой привязки.

```
(defun foo (x)
  (format t "Параметр: ~a~%" x) ; |<----- x - аргумент
  (let ((x 2)) ; |
    (format t "Внешний LET: ~a~%" x) ; | |<---- x = 2
    (let ((x 3)) ; | |
      (format t "Внутренний LET: ~a~%" x)) ; | | |<-- x = 3
      (format t "Внешний LET: ~a~%" x)) ; | |
      (format t "Параметр: ~a~%" x)) ; |
```

Каждое обращение к *x* будет ссылаться на привязку с наименьшей окружающей областью

видимости. Как только поток управления покидает область видимости какой-то связывающей формы, привязка из непосредственно окружающего области видимости перестает скрываться и х ссылается уже на нее. Таким образом, результатом вызова `foo` будет следующий вывод:

```
CL-USER> (foo 1)
Параметр: 1
Внешний LET: 2
Внутренний LET: 3
Внешний LET: 2
Параметр: 1
NIL
```

В последующих главах я обсужу другие конструкции, которые также служат связывающими формами (вообще, любая конструкция, вводящая новые имена переменных, могущие использоваться только внутри этой конструкции, является связывающей формой).

Например, в главе 7 вы встретите цикл **DOTIMES**, простой цикл-счетчик. Он вводит переменную, которая содержит значение счетчика, увеличивающегося на каждой итерации цикла. Например, следующий цикл, печатающий числа от 0 до 9, связывает переменную `x`:

```
(dotimes (x 10) (format t "~d " x))
```

Еще одной связывающей формой является вариант **LET**, **LET ***. Различие состоит в том, что в **LET** имена переменных могут быть использованы только в теле **LET** (части **LET**, идущей после списка переменных), а в **LET *** формы начальных значений для каждой переменной могут ссылаться на переменные, введенные ранее в списке переменных. Таким образом, вы можете записать следующее:

```
(let* ((x 10)
      (y (+ x 10)))
      (list x y))
```

но не так:

```
(let ((x 10)
      (y (+ x 10)))
      (list x y))
```

Однако, вы можете добиться такого же результата при помощи вложенных **LET**.

```
(let ((x 10)
      (let ((y (+ x 10)))
          (list x y)))
```

Лексические переменные и замыкания

По умолчанию все связывающие формы в Common Lisp вводят переменные *лексической области видимости* (*lexically scoped*). На переменные лексической области видимости можно ссылаться только в коде, который текстуально находится внутри связывающей формы. Лексическая область видимости должна быть знакома каждому, кто программировал на Java, C, Perl или Python, так как все они предоставляют "локальные" переменные, имеющие лексическую область видимости. Программисты на Algol также должны чувствовать себя хорошо, так как этот язык первым ввел лексическую область видимости в 1960-х.

Однако, лексические переменные Common Lisp несколько искажают понятие лексической переменной, по крайней мере в сравнении с оригинальной моделью Algol. Это искажение проявляется при комбинировании лексической области видимости со вложенными функциями. По правилам лексической области видимости, только код, текстуально находящийся внутри связывающей формы, может ссылаться на лексическую переменную. Но что произойдет, когда анонимная функция содержит ссылку на лексическую переменную из окружающей области видимости? Например, в следующем выражении:

```
(let ((count 0)) #'(lambda () (setf count (1+ count))))
```

ссылка на `count` внутри формы **LAMBDA** допустима в соответствии с правилами лексической области видимости. Однако, анонимная функция, содержащая ссылку, будет возвращена как значение формы **LET**, и она может быть вызвана с помощью **FUNCALL** кодом, который *не* находится в области видимости **LET**. Так что же произойдет? Как выясняется, если `count` является лексической переменной, все работает. Привязка `count`, созданная когда поток управления зашел в форму **LET**, остается столько, сколько это необходимо, в данном случае до тех пор, пока что-то сохраняет ссылку на функциональный объект, возвращенный формой **LET**. Анонимная функция называется *замыканием* (*closure*), потому что она "замыкается вокруг" привязки, созданной **LET**.

Ключевым моментом для понимания замыканий является то, что захватывается не значение переменной, а привязка. Поэтому замыкание может не только иметь доступ ко значению переменной, вокруг которой оно "замкнуто", но и присваивать ей новые значения, которые будут сохраняться между вызовами замыкания. Например, вы можете захватить замыкание, созданное предыдущим выражением, в глобальную переменную следующим образом:

```
(defparameter *fn* (let ((count 0)) #'(lambda () (setf count (1+ count)))))
```

Затем, при каждом его вызове значение счетчика будет увеличиваться на единицу.

```
CL-USER> (funcall *fn*)
1
CL-USER> (funcall *fn*)
2
CL-USER> (funcall *fn*)
3
```

Отдельное замыкание может "замыкаться вокруг" нескольких привязок переменных просто ссылаясь на них. Также множество замыканий могут захватывать одну и ту же привязку. Например, следующее выражение возвращает список трех замыканий, первое из которых увеличивает значение привязки `count`, вокруг которой оно "замкнуто", второе – уменьшает его, а третье – возвращает текущее значение:

```
(let ((count 0))
  (list
    #'(lambda () (incf count))
    #'(lambda () (decf count))
    #'(lambda () count)))
```

Динамические (специальные) переменные

Привязки с лексической областью видимости помогают поддерживать код понятным путем ограничения области видимости, в которой, буквально говоря, данное имя имеет смысл. Вот

почему большинство современных языков программирования используют лексическую область видимости для локальных переменных. Однако, иногда вам действительно может понадобиться глобальная переменная – переменная, к которой вы можете обратиться из любой части своей программы. Хотя неразборчивое использование глобальных переменных может привести к "спагетти-коду" также быстро как и неумеренное использование `goto`, глобальные переменные имеют разумное использование и существуют в том или ином виде почти в каждом языке программирования. И, как вы сейчас увидите, глобальные переменные Lisp, динамические переменные, одновременно и более удобны, и более гибки.

Common Lisp предоставляет два способа создания глобальных переменных: **DEFVAR** и **DEFPARAMETER**. Обе формы принимают имя переменной, начальное значение и опциональную строку документации. После создания переменной с помощью **DEFVAR** или **DEFPARAMETER** имя может быть использовано где угодно для ссылки на текущую привязку этой глобальной переменной. Как вы заметили в предыдущих главах, по соглашению глобальные переменные именуются именами, начинающимися и заканчивающимися *. Далее в этой главе вы увидите почему очень важно следовать этому соглашению по именованию. Примеры **DEFVAR** и **DEFPARAMETER** выглядят следующим образом:

```
(defvar *count* 0
  "Число уже созданных виджетов.")
(defparameter *gap-tolerance* 0.001
  "Допустимое отклонение интервала между виджетами.")
```

Различие между этими двумя формами состоит в том, что **DEFPARAMETER** всегда присваивает начальное значение названной переменной, а **DEFVAR** делает это только если переменная не определена. Форма **DEFVAR** также может использоваться без начального значения для определения глобальной переменной без установки ее значения. Такая переменная называется *несвязанной* (*unbound*).

На деле вам следует использовать **DEFVAR** для определения переменных, которые будут содержать данные, которые вы хотите сохранять даже при изменениях исходного кода, использующего эту переменную. Например, представьте, что две переменные, определенные ранее, являются частью приложения управления "фабрикой виджетов". Правильным будет определить переменную `*count*` с помощью **DEFVAR**, так как число уже созданных виджетов не становится недействительным лишь потому, что мы сделали некоторые изменения в коде создания виджетов.

С другой стороны, переменная `*gap-tolerance*` вероятно влияет некоторым образом на поведение самого кода создания виджетов. Если вы решите, что вам нужно меньшее или большее допустимое отклонение и, следовательно, измените значение в форме **DEFPARAMETER**, вы захотите, чтобы изменение вступило в силу при перекомпиляции и перезагрузке файла.

После определения переменной с помощью **DEFVAR** или **DEFPARAMETER** вы можете ссылаться на нее откуда угодно. Например, вы можете определить следующую функцию для увеличения числа созданных виджетов:

```
(defun increment-widget-count () (incf *count*))
```

Преимуществом глобальных переменных является то, что вам не нужно передавать их в функции. Большинство языков программирования хранят потоки стандартного ввода и вывода в глобальных переменных именно по этой причине: вы никогда не знаете, когда именно вы захотите напечатать что-либо на стандартный вывод, и вы не хотите, чтобы каждая функция принимала и передавала далее аргументы, содержащие эти потоки, только потому, что

нижележащему коду они могут понадобиться.

Однако, поскольку значение, такое как поток стандартного вывода, хранится в глобальной переменной и вы написали код, ссылающийся на эту глобальную переменную, порой является заманчивым попытаться временно изменить поведение этого кода путем изменения значения переменной.

Например, представьте, что вы работаете над программой, содержащей некоторые низкоуровневые функции журналирования, печатающие в поток, хранящийся в глобальной переменной `*standard-output*`. Теперь представьте, что в какой-то части программы вы хотите перенаправить весь вывод, генерируемый этими функциями в файл. Вы можете открыть файл и присвоить полученный поток переменной `*standard-output*`. После этого вышеупомянутые функции будут слать свой вывод в этот файл.

Это работает замечательно, пока вы не забудете восстановить исходное значение `*standard-output*` после завершения действий. Если вы забудете восстановить `*standard-output*`, весь остальной код программы, использующий `*standard-output*`, также будет слать свой вывод в файл.

Но похоже, что то, что вам действительно нужно, это способ обернуть часть кода во что-то говорящее: "Весь нижележащий код (все функции, которые он вызывает, все функции, которые вызывают эти функции, и так далее до функций самого низкого уровня) должны использовать это значение для глобальной переменной `*standard-output*`". А затем, по завершении работы функции верхнего уровня, старое значение `*standard-output*` должно быть автоматически восстановлено.

Оказывается, что это именно те возможности, что предоставляет вам другой вид переменных Common Lisp: динамические переменные. Когда вы связываете динамическую переменную, например с **LET**-переменной или с параметром функции, привязка, создаваемая во время входа в связывающую форму, заменяет глобальную привязку на все время выполнения связывающей формы. В отличие от лексических привязок, к которым можно обращаться только из кода, находящегося в лексической области видимости связывающей формы, к динамическим привязкам можно обращаться из любого кода, вызываемого во время выполнения связывающей формы. И оказывается, что все глобальные переменные на самом деле являются динамическими.

Таким образом, если вы хотите временно переопределить `*standard-output*`, это можно сделать просто пересвязав ее, например, с помощью **LET**.

```
(let ((*standard-output* *some-other-stream*))  
  (stuff))
```

В любом коде, который выполняется в результате вызова `stuff`, ссылки на `*standard-output*` будут использовать привязку, установленную с помощью **LET**. А после того как `stuff` завершится и поток управления покинет **LET**, новая привязка `*standard-output*` исчезнет и последующие обращения к `*standard-output*` будут видеть привязку, бывшую до **LET**. В любой момент времени самая последняя установленная привязка скрывает все остальные. Можно представить, что каждая новая привязка данной динамической переменной помещается в стек привязок этой переменной и ссылки на эту переменную всегда используют последнюю установленную привязку. После выхода из связывающей формы созданные в ней привязки убираются из стека, делая видимыми предыдущие привязки.

Простой пример показывает как это работает:

```
(defvar *x* 10)
(defun foo () (format t "X: ~d~%" *x*))
```

DEFVAR создает глобальную привязку переменной `*x*` со значением 10. Обращение к `*x*` в `foo` ищет текущую привязку динамически. Если вы вызовете `foo` на верхнем уровне (from the top level), глобальная привязка, созданная **DEFVAR**, будет единственной доступной привязкой, поэтому будет напечатано 10.

```
CL-USER> (foo)
X: 10
NIL
```

Но вы можете использовать **LET** для создания новой привязки, которая временно скроет глобальную привязку, и `foo` напечатает другое значение.

```
CL-USER> (let ((*x* 20)) (foo))
X: 20
NIL
```

Теперь снова вызовем `foo` без **LET**, она опять будет видеть глобальную привязку.

```
CL-USER> (foo)
X: 10
NIL
```

Теперь определим новую функцию.

```
(defun bar ()
  (foo)
  (let ((*x* 20)) (foo))
  (foo))
```

Обратите внимание, что средний вызов `foo` находится внутри **LET**, которая связывает `*x*` с новым значением 20. При вызове `bar` вы получите следующий результат:

```
CL-USER> (bar)
X: 10
X: 20
X: 10
NIL
```

Как вы можете заметить, первый вызов `foo` видит глобальную привязку со значением 10. Средний вызов видит новую привязку со значением 20. А после **LET**, `foo` снова видит глобальную привязку.

Как и с лексической привязкой, присваивание нового значения влияет только на текущую привязку. Чтобы увидеть это, вы можете переопределить `foo`, добавив присваивание значения переменной `*x*`.

```
(defun foo ()
  (format t "Перед присваиванием~18tX: ~d~%" *x*)
  (setf *x* (+ 1 *x*)))
  (format t "После присваивания~18tX: ~d~%" *x*))
```


Теперь foo печатает значение *x*, увеличивает его на единицу, а затем печатает его снова. Если вы просто запустите foo, вы увидите следующее:

```
CL-USER> (foo)
Перед присваиванием X: 10
После присваивания X: 11
NIL
```

Ничего удивительного. Теперь запустим bar.

```
CL-USER> (bar)
Перед присваиванием X: 11
После присваивания X: 12
Перед присваиванием X: 20
После присваивания X: 21
Перед присваиванием X: 12
После присваивания X: 13
NIL
```

Обратите внимание, начальное значение *x* равно 11: предыдущий вызов foo действительно изменил глобальное значение. Первый вызов foo из bar увеличивает глобальную привязку до 12. Средний вызов не видит глобальную привязку из-за **LET**. А затем последний вызов снова может видеть глобальную привязку и увеличивает ее с 12 до 13.

Так как это работает? Как **LET** знает, когда связывает *x*, что подразумевается создание динамической привязки вместо обычной лексической? Она знает, так как имя было объявлено *специальным*. Имя каждой переменной, определенной с помощью **DEFVAR** и **DEFPARAMETER** автоматически глобально объявляется специальным. Это означает, что когда бы вы не использовали это имя в связывающей форме (в форме **LET**, или как параметр функции, или в любой другой конструкции, которая создает новую привязку переменной, вновь создаваемая привязка будет динамической. Вот почему *соглашение* *по* *именованию* так важно: будет не очень хорошо, если вы используете имя, о котором вы думаете как о лексической переменной, а эта переменная окажется глобальной специальной. С одной стороны, код, который вы вызываете, сможет изменить значение этой связи; с другой, вы сами можете скрыть связь, установленную кодом, находящимся выше по стеку. Если вы всегда будете именовать глобальные переменные, используя соглашение по именованию *, вы никогда случайно не воспользуетесь динамической связью, желая создать лексическую.

Также возможно локально объявить имя специальным. Если в связывающей форме вы объявите имя специальным, привязка, созданная для этой переменной, будет динамической, а не лексической. Другой код может локально определить имя специальным, чтобы обращаться к динамической привязке. Однако, локальные специальные переменные используются относительно редко, поэтому вам не стоит беспокоиться о них.

Динамические привязки делает глобальные переменные гораздо более гибкими, но важно понимать, что они позволяют осуществлять незаметные действия на расстоянии. Связывание глобальной переменной имеет два дальнедействующих эффекта: оно может изменить поведение нижележащего кода, а также открывает нижележащему коду возможность присваивания нового значения привязке, установленной выше по стеку. Вы должны использовать динамические переменные только в том случае, если вам нужно получить преимущества от одного или обоих из этих эффектов.

Константы

Еще одним видом переменных, вообще не упомянутых ранее, являются оксюморонические "константные переменные". Все константы являются глобальными и определяются с помощью **DEFCONSTANT**. Базовая форма **DEFCONSTANT** подобна **DEFPARAMETER**.

```
(defconstant name initial-value-form [ documentation-string ])
```

Как и в случае с **DEFPARAMETER**, **DEFCONSTANT** оказывает глобальный эффект на используемое имя: после этого имя может быть использовано только для обращения к константе; оно не может быть использовано как параметр функции или быть пересвязано с помощью любой другой связывающей формы. Поэтому многие программисты на Lisp следуют соглашению по именованию и используют для констант имена начинающиеся и заканчивающиеся +. Этому соглашению следуют немного в меньшей степени, чем соглашению для глобальных динамических имен, но оно является хорошей идеей по сходным причинам.

Еще на что нужно обратить внимание по поводу **DEFCONSTANT** это то, что в то время, как язык позволяет вам переопределять константы путем перевычисления **DEFCONSTANT** с другой формой начального значения, не определено то, что именно произойдет после такого переопределения. На практике большинство реализаций требуют, чтобы вы перевычислили любой код, ссылающийся на константу, чтобы изменение вступило в силу, так как старое значение могло быть встроено (inlined). Следовательно, правильным будет использовать **DEFCONSTANT** для определения только тех вещей, которые *действительно* являются константами, такие как значение NIL. Для вещей, которые вам может когда-нибудь понадобится изменить, следует использовать **DEFPARAMETER**.

Присваивание

После создания привязки вы можете совершать с ней два действия: получить текущее значение и установить ей новое значение. Как вы видели в главе 4, символ вычисляется в значение переменной, которую он именуется, поэтому вы можете получить текущее значение просто обратившись к переменной. Для присваивания нового значения привязке используйте макрос **SETF**, являющийся в Common Lisp оператором присваивания общего назначения. Базовая форма **SETF** следующая:

```
(setf place value)
```

Так как **SETF** является макросом, он может оценить форму "места", которому он осуществляет присваивание и расширяться (expand) в соответствующие низкоуровневые операции, осуществляющие необходимые действия. Когда "место" является переменной, этот макрос расширяется в вызов специального оператора **SETQ**, который, как специальный оператор, имеет доступ и к лексическим, и к динамическим привязкам. Например, для присваивания значения 10 переменной x вы можете написать это:

```
(setf x 10)
```

Как я рассказывал ранее, присваивание нового значения привязке не оказывает никакого влияния на остальные привязки этой переменной. И оно не оказывает никакого влияния на значение, которое хранилось в привязке до присваивания. Таким образом, **SETF** в следующей функции:

```
(defun foo (x) (setf x 10))
```

не окажет никакого влияния на любое значение вне foo. Привязка, которая создается при вызове foo, устанавливается в 10, незамедлительно заменяя то значение, что было передано в качестве

аргумента. В частности, следующая форма:

```
(let ((y 20))
  (foo y)
  (print y))
```

напечатает 20, а не 10, так как именно оно является значением `y`, которое передается `foo`, где уже является значением переменной `x` перед тем, как **SETF** дает `x` новое значение.

SETF также может осуществить последовательное присваивание множеству "мест". Например, вместо следующего:

```
(setf x 1)
(setf y 2)
```

вы можете записать следующее:

```
(setf x 1 y 2)
```

SETF возвращает присвоенное значение, поэтому вы можете вкладывать вызовы **SETF** как в следующем примере, который присваивает и `x`, и `y` одинаковое случайное значение:

```
(setf x (setf y (random 10)))
```

Обобщенное присваивание

Привязки переменных, конечно, не являются единственными "местами", которые могут содержать значения. Common Lisp поддерживает составные структуры данных, такие как массивы, хэш-таблицы, списки, а также определенные пользователем структуры данных — такие структуры состоят из множества "мест", могущих содержать значения.

Я опишу эти структуры данных в последующих главах, но так как мы рассматриваем присваивание, вы должны знать, что **SETF** может присвоить значение любому "месту". Когда я буду описывать различные составные структуры данных, я буду указывать, какие функции могут использоваться как "места, обрабатываемые **SETF**" ("**SETFable** places"). Кратко же можно сказать, что если вам нужно присвоить значение "месту", почти наверняка следует использовать **SETF**. Возможно даже расширить **SETF** для того, чтобы он мог осуществлять присваивание определенным пользователем "местам", хотя я не описываю такие возможности.

В этом отношении **SETF** не отличается от оператора присваивания = языков, произошедших от С. В этих языках оператор = присваивает новые значения переменным, элементам массивов, полям классов. В языках, таких как Perl и Python, которые поддерживают хэш-таблицы как встроенные типы данных, = может также устанавливать значения элементов хэш-таблицы. Таблица 6-1 резюмирует различные способы, которыми используется = в этих языках.

Таблица 6-1. Присваивание с помощью = в других языках программирования

Присваивание ...	Java, C, C++	Perl	Python
... переменной	<code>x = 10;</code>	<code>\$x = 10;</code>	<code>x = 10</code>
... элементу массива	<code>a[0] = 10;</code>	<code>\$a[0] = 10;</code>	<code>a[0] = 10</code>
... элементу хэш-таблицы	—	<code>\$hash{'key'} = 10;</code>	<code>hash['key'] = 10</code>
... полю объекта	<code>o.field = 10;</code>	<code>\$o->{'field'} = 10;</code>	<code>o.field = 10</code>

SETF работает сходным образом: первый "аргумент" **SETF** является "местом" для хранения значения, а второй предоставляет само значения. Как и с оператором `=` в этих языках, вы используете одинаковую форму и для выражения "места", и для получения значения. Таким образом, эквиваленты вышеприведенных в таблице 6-1 присваиваний для Lisp следующие (**AREF** — функция доступа к массиву, **GETHASH** осуществляет операцию поиска в хэш-таблице, а `field` может быть функцией, которая обращается к слоту под именем `field` определенного пользователем объекта):

```
Простая переменная: (setf x 10)
Массив: (setf (aref a 0) 10)
Хэш-таблица: (setf (gethash 'key hash) 10)
Слот с именем 'field': (setf (field o) 10)
```

Обратите внимание, что присваивание с помощью **SETF** "месту", которое является частью большего объекта, имеет ту же семантику, что и присваивание переменной: "место" модифицируется без оказания какого-либо влияния на объект, которых хранился там до этого. И вновь, это подобно тому, как ведет себя `=` в Java, Perl и Python.

Другие способы изменения "мест"

В то время как все присваивания можно выразить с помощью **SETF**, некоторые образцы, включающие присваивания нового значения, основанного на текущем значении, являются достаточно общими для того, чтобы получить свои собственные операторы. Например, вы можете увеличить число с помощью **SETF** следующим образом:

```
(setf x (+ x 1))
```

или уменьшить его так:

```
(setf x (- x 1))
```

Но это слегка утомительно по сравнению с стилем C: `++x` и `--x`. Вместо этого вы можете использовать макросы **INCF** и **DECF**, которые увеличивают и уменьшают "место" на определенную величину, по умолчанию 1.

```
(incf x) == (setf x (+ x 1))
(decf x) == (setf x (- x 1))
(incf x 10) == (setf x (+ x 10))
```

INCF и **DECF** являются примерами определенного вида макросов, называемых *модифицирующими макросами* (*modify macros*). Модифицирующие макросы являются макросами, построенными поверх **SETF**, которые модифицируют "места" путем присваивания нового значения, основанного на их текущем значении. Главным преимуществом таких макросов является то, что они более краткие, чем аналогичные операции, записанные с помощью **SETF**. Вдобавок, модифицирующие макросы определены таким образом, что делает их безопасными при использовании с "местами", когда выражение "места" должно быть вычислено лишь единожды. Несколько надуманным примером является следующее выражение, которое увеличивает значение произвольного элемента массива:

```
(incf (aref *array* (random (length *array*))))
```

Наивный перевод этого примера в выражение, использующее **SETF**, может выглядеть

следующим образом:

```
(setf (aref *array* (random (length *array*)))  
      (1+ (aref *array* (random (length *array*))))))
```

Однако, это не работает, так как два последовательных вызова **RANDOM** не обязательно вернут одинаковое значение: это выражение вероятно получит значение одного элемента массива, увеличит его, а затем сохранит его как новое значение другого элемента массива. Однако, выражение **INCF** сделает все правильно, так как знает, как правильно разобрать это выражение:

```
(aref *array* (random (length *array*)))
```

чтобы извлечь те части, которые возможно могут иметь побочные эффекты, и гарантировать, что они будут вычисляться лишь один раз. В этом случае, выражение **INCF** вероятно расширится в нечто более или менее подобное этому:

```
(let ((tmp (random (length *array*))))  
  (setf (aref *array* tmp) (1+ (aref *array* tmp))))
```

Вообще, модифицирующие макросы гарантируют однократное вычисление слева направо своих аргументов, а также подформ формы места (place form).

Макрос **PUSH**, который вы использовали в примере с базой данных для добавления элементов в переменную **db**, является еще одним модифицирующим макросом. Более подробно о его работе и работе **POP** и **PUSHNEW** будет сказано в главе 12, где я буду говорить о том, как представляются (represented) списки в Lisp.

И наконец, два слегка эзотерических, но полезных модифицирующих макроса – **ROTATEF** и **SHIFTF**. **ROTATEF** циклически сдвигает значение между "местами". Например, если вы имеете две переменные, *a* и *b*, этот вызов:

```
(rotatef a b)
```

обменяет значения двух переменных и вернет **NIL**. Так как *a* и *b* являются переменными и вам не нужно беспокоиться о побочных эффектах, предыдущее выражение **ROTATEF** эквивалентно следующему:

```
(let ((tmp a)) (setf a b b tmp) nil)
```

С другими видами "мест" эквивалентное выражение с использованием **SETF** может быть более сложным.

SHIFTF подобен **ROTATEF** за исключением того, что вместо циклического сдвига значений, он просто сдвигает их влево: последний аргумент предоставляет значение, которое перемещается в предпоследний аргумент и так далее. Исходное значение первого аргумента просто возвращается. Таким образом, следующее:

```
(shiftf a b 10)
```

эквивалентно (и снова, так как вам не нужно беспокоиться о побочных эффектах) следующему:

```
(let ((tmp a)) (setf a b b 10) tmp)
```

И **ROTATEF**, и **SHIFTF** оба могут использоваться с любым числом аргументов и, как все модифицирующие макросы, гарантируют однократное их вычисление слева направо.

С базовыми знаниями функций и переменных Common Lisp в своем арсенале вы готовы перейти к следующей особенности, которая еще больше отличает Lisp от других языков программирования: макросы.

7. Макросы: Стандартные управляющие конструкции

В то время, как многие из идей, появившиеся в Лиспе, от выражений условия до сборки мусора, были добавлены в другие языки, есть одна особенность языка, которая продолжает делать Common Lisp стоящим особняком от всех, это его система макросов. К несчастью, слово "макрос" описывает множество вещей в компьютерных науках, к которым макросы Common Lisp имеют неявное и метафорическое отношение FIXME. Это приводит к бесконечным недопониманиям, когда адепты Лиспа пытаются объяснить другим, насколько макросы замечательны.

Чтобы понять макросы Лиспа, необходимо подойти к делу со свободной головой, без предубеждений, основанных на других вещах, которые также оказались названными словом "макросы". Итак, давайте начнём нашу тему с шага назад и обзора различных путей, которыми создаются расширения в языках.

Всем программистам должна быть привычна идея о том, что определение языка может включать стандартную библиотеку функций, которая строится на "ядре" языка – библиотеку, которая могла бы быть написана посредством языка любым программистом, если бы она не была определена как часть стандартной библиотеки. Стандартная библиотека языка Си, например, может быть написана почти полностью на переносимом Си. Аналогично, большая часть всё растущего набора классов и интерфейсов, которые поставляются в стандартном наборе Java Development Kit (JDK), написаны на "чистом" Java.

Одним из преимуществ определения языков в терминах "ядро плюс стандартная библиотека", это лёгкость в понимании и воплощении. Однако реальная выгода заключается в выразительности – многое из того, про что вы думаете как про "язык", на самом деле просто библиотека – язык легко расширять. Если в Си нет функции для той или иной необходимой вам задачи, вы можете её написать и теперь у вас есть слегка улучшенная версия Си. Точно так же в языках, таких как Java или Smalltalk, где почти все интересные части "языка" определены в терминах классов, определяя новый класс, вы расширяете язык, делая его более подходящим для написания программ, делающих то, что вам надо.

В то время, как Common Lisp поддерживает оба этих метода расширения языка, макросы дают Лиспу ещё один путь. Как было упомянуто кратко в Главе 4, каждый макрос определяет свой собственный синтаксис – определение того, как s-выражения, которые ему передаются, будут превращены в Лисп-формы. С помощью макросов, как части ядра языка, возможно создавать новые синтаксические управляющие конструкции, такие как WHEN, DOLIST и LOOP, а так же формы определений вроде DEFUN и DEFPARAMETER, как часть "стандартной библиотеки", вместо встраивания их в ядро. Это имеет свои последствия для реализации языка, но как программиста на Лисп, вас будет больше заботить то, что это даёт вам ещё один способ расширения языка, делая его языком, лучше подходящим для выражения решений ваших собственных программистских проблем.

В данный момент может показаться, что преимущества от наличия ещё одного пути расширения языка будет легко понять. Но по некоторой причине большое количество программистов, которые фактически не использовали макросы Лиспа и которые не задумываются о создании новых функциональных абстракций или определений новых иерархий классов для решения своих задач, панически боятся самой мысли о том, что они будут иметь возможность описания новых синтаксических абстракций. Наиболее общей причиной "макрофобии", похоже является плохой опыт от использования других "макросистем". Простой страх перед неизвестным несомненно так же играет определенную роль. Чтобы избежать макрофобических реакций, я буду постепенно вводить в предмет через обсуждение нескольких стандартных макросов, конструкций контроля,

определённых в Common Lisp. Это те вещи, которые должны были быть встроены в ядро языка, если бы в Лиспе не было макросов. Когда вы используете их, вам не надо беспокоиться, что они сделаны в виде макросов, но они представляют из себя хороший пример того, что вы можете сделать с помощью макросов. В следующей главе я покажу вам, как вы можете определять свои собственные макросы.

WHEN и UNLESS

Как вы уже видели, наиболее базовую форму условного выражения – *если x, делай y; иначе делай z* – представляет специальный оператор IF, который имеет следующую базовую форму:

```
(if condition then-form [else-form])
```

condition вычисляется и, если его значение не NIL, тогда *then-form* выполняется и полученное значение возвращается. Иначе выполняется *else-form*, если она есть, и её значение возвращается. Если *condition* даёт NIL и нет *else-form*, тогда IF возвращает NIL.

```
(if (> 2 3) "Yup" "Nope") ==> "Nope"  
(if (> 2 3) "Yup") ==> NIL  
(if (> 3 2) "Yup" "Nope") ==> "Yup"
```

Однако, IF не является вообще-то такой уж замечательной синтаксической конструкцией, потому что *then-form* и *else-form* каждая ограничена одной лисп-формой. Это значит, что если вы хотите выполнить последовательность действий в каком-либо из этих случаев, вам надо обернуть их в какой-то другой синтаксис. Например, предположим, в середине программы спам-фильтра, вы захотите сохранить в файле сообщение, как спам, и обновить базу данных по спаму, если сообщение - спам. Вы не можете написать так:

```
(if (spam-p current-message)  
    (file-in-spam-folder current-message)  
    (update-spam-database current-message))
```

потому что вызов `update-spam-database` будет принят за случай *else*, а не как часть ветви *then*. Другой специальный оператор PROGN, выполняет любое число форм по порядку и возвращает значение последней формы. Так что вы могли бы получить желаемое, записав всё следующим образом:

```
(if (spam-p current-message)  
    (progn  
      (file-in-spam-folder current-message)  
      (update-spam-database current-message)))
```

Это не так уж и ужасно. Однако, учитывая количество раз, когда вам придётся использовать эту идиому, не трудно представить себе, что через некоторое время это станет утомительно. "Почему", вы можете спросить у себя, "Лисп не предоставляет возможность выразить то, что на самом деле мне надо, скажем 'Если x верно, делай то, то и ещё вот это'?" Другими словами, через некоторое время, вы заметите повторяемость сочетания IF плюс PROGN и захотите как-то абстрагироваться от этих деталей, вместо того, чтобы каждый раз заново переписывать их.

Это как раз то, что предоставляют макросы. В данном случае, Коммон Лисп поставляется со стандартным макросом WHEN, с которым всё можно написать так:


```
(when (spam-p current-message)
      (file-in-spam-folder current-message)
      (update-spam-database current-message))
```

Но если бы он не был встроен в стандартную библиотеку, вы могли бы определить WHEN самостоятельно, как макрос вот так, используя запись с обратной кавычкой, которую я обсуждал в Главе 3:

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

Сопутствующим макросу WHEN является UNLESS, который оборачивает условие, выполняя формы из тела, только если условие ложно. Другими словами:

```
(defmacro unless (condition &rest body)
  `(if (not ,condition) (progn ,@body)))
```

Конечно, это довольно тривиальные макросы. Тут нет никакой страшной чёрной магии; они просто абстрагируют некоторые детали языковой бухгалтерии, позволяя вам выражать свои намерения немного более ясно. Но их тривиальность имеет важное значение: так как система макросов встроена в язык, вы можете писать тривиальные макросы вроде WHEN и UNLESS, которые дают вам небольшую, но реальную выгоду в ясности, которая затем умножается в тысячу раз когда вы используете их. В Главах 24, 26 и 31 вы увидите, как макросы могут быть использованы для серьёзных вещей, создавая целый предметно-ориентированный (domain-specific), встроенный язык. Но сначала, давайте закончим наше обсуждение стандартных макросов управления.

COND

Другой раз, когда непосредственное IF выражение может оказаться ужасным, это когда у вас есть условное выражение с множественными ветвлениями: *если A, делай X, иначе, если B, делай Y; иначе делай Z*. Нет никакой логической проблемы в написании такой цепочки условных выражений с IF, но получится не очень красиво.

```
(if a
    (do-x)
    (if b
        (do-y)
        (do-z)))
```

И это будет выглядеть ещё более ужасно, если вам понадобится включить множество форм для *then* случаев, привлекая несколько PROGN. Так что ничего удивительного, что Коммон Лисп предоставляет макрос для выражения условия с множеством ветвлений: COND. Вот базовый вид:

```
(cond
  (test-1 form*)
  .
  .
  .
  (test-N form*))
```

Каждый элемент в теле представляет одну ветвь условия и состоит из списка, содержащего форму условия и ноль или более форм для выполнения, если выбрана эта ветвь. Условия

вычисляются в том порядке, в каком расположены ветви до тех пор, пока одно из них не даст истину. В этой точке, оставшиеся формы из ветви выполняются и значение последней формы ветви возвращается как результат работы всего COND. Если ветвь не содержит форм после условия, то возвращается само значение условия. По соглашению, ветвь представляющая последний случай *else* в цепочке *if/else-if* записывается с условием T. Подойдёт любое не-NIL значение, но T служит дорожным знаком при чтении кода. Таким образом вы можете записать предыдущее вложенное IF выражение, используя COND, вот так:

```
(cond (a (do-x))
      (b (do-y))
      (t (do-z)))
```

AND, OR и NOT

При написании условий в IF, WHEN, UNLESS и COND формах, три оператора оказываются очень полезны, это булевы логические операторы AND, OR и NOT.

NOT - это функция, которая строго говоря не относится к этой главе, но она очень тесно связана с AND и OR. Она берёт свой аргумент и обращает его значение истинности, возвращая T, если аргумент NIL и NIL в ином случае.

AND и OR, однако являются макросами. Они представляют логические конъюнкцию и дизъюнкцию произвольного числа подформ и определены как макросы, так что они оптимальны в выполнении. Это значит, что они вычисляют ровно столько своих подформ, в порядке слева направо, сколько необходимо для конечного значения. То есть AND останавливается и возвращает NIL сразу же, как только одна из подформ выдаст NIL. Если все подформ выдают не-NIL результат, она возвращает значение последней подформы. OR, с другой стороны, останавливается, как только одна из подформ выдаст не-NIL и возвращает полученное значение. Если ни одна из подформ не выдаст истину, OR возвращает NIL. Вот несколько примеров:

```
(not nil) ==> T
(not (= 1 1)) ==> NIL
(and (= 1 2) (= 3 3)) ==> NIL
(or (= 1 2) (= 3 3)) ==> T
```

Циклы

Циклические конструкции представляют собой важный тип управляющих конструкций (FIXME в оригинале сказано наоборот). Циклические средства в Коммон Лисп, в дополнение к мощности и гибкости, являются интересным уроком по программированию в стиле "получить всё и сразу", который позволяют макросы.

Как оказалось, ни один из 25 специальных операторов Лиспа не поддерживает напрямую структуру циклов. Все циклические конструкции контроля в Лиспе - это макросы, построенные на двух специальных операторах, которые представляют собой примитивное *goto* средство. Как многие хорошие абстракции, синтаксические или нет, циклические макросы в Лиспе построены как набор слоёв абстракций, начиная с основы, которой являются те два специальных оператора.

В самом низу (оставляя в стороне специальные операторы) находится наиболее общая конструкция контроля DO. Хотя и очень мощный, DO страдает, как и многие абстракции общего назначения, от чрезмерности для простых ситуаций. Так что Лисп предоставляет два других макроса DOLIST and DOTIMES, которые менее гибки, чем DO, но лучше поддерживают наиболее распространённые случаи цикла по элементам списка или цикла с подсчётом. Хотя реализация может реализовать эти макросы как ей угодно, обычно они реализованы как макросы, которые

раскрываются в соответствующий DO цикл. Таким образом DO предоставляет базовую структурную конструкцию цикла поверх нижележащих примитивов, представленных специальными операторами Коммон Лиспа, а DOLIST и DOTIMES представляют две лёгких в использовании, хотя и менее общие конструкции. И, как вы увидите в следующей главе, вы можете строить свои собственные конструкции цикла поверх DO в ситуациях, где DOLIST и DOTIMES вам не подходят.

Наконец, макрос LOOP представляет собой полномасштабный мини-язык для выражения циклических конструкций на не Лиспо-, а англо-подобном (или как минимум Алголо-подобном) языке. Некоторые хакеры Лиспа любят LOOP; другие ненавидят его. Фанаты LOOP любят его за то, что он предоставляет краткий способ выразить определённые, обычно необходимые циклические конструкции. Его недоброжелатели не любят его, потому что он недостаточно похож на остальной Лисп. Однако, к какому бы лагерю вы не примкнули, это замечательный пример возможностей макросов добавлять новые конструкции в язык.

DOLIST и DOTIMES

Я начну с лёгких для использования DOLIST и DOTIMES макросов.

DOLIST проходит по всем элементам списка, выполняя тело цикла с переменной, содержащей последовательно элементы списка. Вот базовый скелет (оставляя некоторые эзотерические опции):

```
(dolist (var list-form)
  body-form*)
```

Когда цикл стартует, *list-form* выполняется один раз, чтобы создать список. Затем тело цикла выполняется для каждого элемента в списке, с переменной *var*, содержащей значение элемента. Например:

```
CL-USER> (dolist (x '(1 2 3)) (print x))
1
2
3
NIL
```

Использованная таким образом, форма DOLIST, в целом, возвращает NIL.

Если вы хотите прервать цикл DOLIST до окончания списка, можете использовать RETURN.

```
CL-USER> (dolist (x '(1 2 3)) (print x) (if (evenp x) (return))))
1
2
NIL
```

DOTIMES - это конструкция цикла верхнего уровня для циклов с подсчётом. Основной вид более-менее такой же, как у DOLIST.

```
(dotimes (var count-form)
  body-form*)
```

count-form должна выдать целое число. Каждый раз, в процессе цикла, *var* содержит последовательные целые от 0 до на единицу меньшего, чем то число. Например:

```
CL-USER> (dotimes (i 4) (print i))
0
1
2
3
NIL
```

Так же, как и с DOLIST, вы можете использовать RETURN, чтобы прервать цикл раньше.

Так как тела обоих DOLIST и DOTIMES циклов могут содержать любые типы выражений, вы так же можете делать циклы вложенными. Например, чтобы напечатать таблицу умножения от 1×1 до $20 \times 20 = 400$, вы можете написать такую пару вложенных циклов DOTIMES:

```
(dotimes (x 20)
  (dotimes (y 20)
    (format t "~3d " (* (1+ x) (1+ y))))
  (format t "~%"))
```

DO

Хотя DOLIST и DOTIMES удобны и легки в использовании, они недостаточно гибки, чтобы использоваться для любых циклов. Например, что если вы захотите менять на каждом шаге несколько переменных параллельно? Или использовать произвольное выражение для проверки окончания цикла? Если ни DOLIST, ни DOTIMES не подходят для ваших целей, у вас всё ещё есть доступ к наиболее общему циклу DO.

Там, где DOLIST и DOTIMES предоставляют только одну переменную цикла, DO позволяет вам держать любое число переменных и даёт вам полный контроль над тем, как они будут изменяться на каждом шаге цикла. Вы так же определяете проверку, которая говорит когда циклу завершиться и может предоставлять форму для вычисления в конце цикла, чтобы сформировать возвращаемое значение для всего DO в целом. Базовый шаблон выглядит так:

```
(do (variable-definition*)
    (end-test-form result-form*)
    statement*)
```

Каждое *variable-definition* (определение переменной) вводит переменную, которая будет в поле видимости тела цикла. Полная форма одного определения переменной, это список, содержащий три элемента.

```
(var init-form step-form)
```

init-form будет выполнена в начале цикла и полученное значение присвоено переменной *var*. Перед каждой последующей итерацией цикла, *step-form* будет выполнена и её значение присвоено *var*. Форма *step-form* необязательна; если её не будет, переменная останется с тем же значением от итерации к итерации, пока вы прямо не назначите ей новое значение в теле цикла. Так же, как и с присвоением переменным значений в LET, если форма *init-form* не задана, переменной присваивается NIL. Так же, как и в LET, вы можете использовать просто имя переменной, вместо списка, содержащего только имя.

В начале каждой итерации, после того, как все переменные цикла получили свои новые значения, выполняется форма *end-test-form*. До тех пор, пока она вычисляется в NIL, итерация происходит, выполняя *statement* по порядку.

Когда вычисление формы *end-test-form* выдаст истину, будет вычислена форма *result-forms* и значение, полученное в результате, будет возвращено как значение всего выражения *DO*.

На каждом шаге итерации, шаговые формы для переменных вычисляются прежде придания новых значений этим переменным. Это значит, что вы можете ссылаться на любую другую переменную внутри шаговых форм. Таким образом, цикл выглядит так:

```
(do ((n 0 (1+ n))
    (cur 0 next)
    (next 1 (+ cur next)))
    ((= 10 n) cur))
```

шаговые формы $(1+ n)$, *next*, и $(+ cur next)$ все вычисляются, используя старое значение *n*, *cur* и *next*. Только после вычисления всех шаговых форм, переменные получают свои новые значения. (Математически образованные читатели могут заметить, что это частично эффективный способ подсчёта одиннадцатого числа Фибоначчи.)

Этот пример также иллюстрирует ещё одну характеристику *DO* – так как вы можете изменять на каждом шаге несколько переменных, вам зачастую не понадобится тело цикла вообще. В другой раз, вы можете обойтись без результирующей формы, в частности, если вы используете цикл, как конструкцию контроля. Эта гибкость, однако, является причиной по которой выражение *DO* может стать плохо читаемым. Что, собственно, все эти скобки здесь делают? Лучший способ понять *DO* выражение, это держать в голове основной шаблон.

```
(do (variable-definition*)
    (end-test-form result-form*)
    statement*)
```

Шесть скобок в этом шаблоне, это только те, которые требуются самим *DO*. Вам нужна одна пара для закрытия описания переменных, одна пара для закрытия теста окончания и результирующей формы и одна пара для закрытия всего выражения. Другие формы внутри *DO* могут потребовать своих собственных пар скобок – определения переменных, это обычно списки, например. К тому же форма проверки окончания, это зачастую функция. Однако скелет *DO* цикла всегда будет одним и тем же. Вот несколько примеров цикла *DO* со скелетом, отмеченным жирным шрифтом:

```
**(**do **(**(i 0 (1+ i))**)**
  **(**(>= i 4)**)**
(print i)**)**
```

(DELETEME - не знаю как выделить жирным скобки в коде)

Заметьте, что форма для результата пропущена. Это, однако, не особо распространённое использование *DO*, так как такой цикл гораздо проще написать используя *DOTIMES*.

```
(dotimes (i 4) (print i))
```

Другой пример, в котором отсутствует тело цикла для вычисления чисел Фибоначчи:

```
**(**do **(**(n 0 (1+ n))
  (cur 0 next)
  (next 1 (+ cur next))**)**
**(**(= 10 n) cur**)**)**
```

(DELETEME - не знаю как выделить жирным скобки в коде)

Наконец, следующий пример демонстрирует цикл DO, в котором нет привязанных переменных. Он крутится, пока текущее время меньше, чем значение глобальной переменной, печатая "Waiting" каждую минуту. Заметьте, что даже без переменных цикла, вы всё равно нуждаетесь в пустом списке для списка переменных.

```
**(**do **(**)**  
  **(**(> (get-universal-time) *some-future-date*)**)**  
  (format t "Waiting~%")  
  (sleep 60)**)**
```

(DELETEME - не знаю как выделить жирным скобки в коде)

Всемогущий LOOP

Для простых случаев у вас есть DOLIST и DOTIMES. И если они не удовлетворяют вашим нуждам, вы можете вернуться к совершенно общему DO. Чего ещё можно хотеть?

Однако оказывается, что удобные идиомы для циклов появляются снова и снова, такие, как циклы по элементам различных структур с данными: списков, векторов, хэш-таблиц и пакетов, либо накопление значений разными способами в процессе цикла: собирание, подсчёт, суммирование, минимизация или максимизация. Если вы нуждаетесь в цикле, который бы делал одну из этих вещей (или несколько одновременно), макрос LOOP может предоставить вам простой путь это выразить.

Макрос LOOP на самом деле бывает двух видов: простой и расширенный. Простая версия проста как только можно: бесконечный цикл без связанных с ним переменных. Скелет выглядит так:

```
(loop  
  body-form*)
```

Формы внутри тела выполняются каждый раз в процессе цикла, который будет длиться вечно, пока вы не используете RETURN, чтобы прервать его. Например, вы могли бы записать предыдущий DO цикл с простым LOOP.

```
(loop  
  (when (> (get-universal-time) *some-future-date*)  
    (return))  
  (format t "Waiting~%")  
  (sleep 60))
```

Расширенный LOOP – несколько иной зверь. Он отличается использованием специальных ключевых слов цикла, которые представляют язык специального назначения для выражения циклических идиом. FIXME(Это не по русски:) Это ничего не значит, что не все Лисповоды любят язык расширенного LOOP. Как минимум один из создателей Коммон Лиспа ненавидит его. Критики LOOP жалуются, что его синтаксис совсем не лисповский. (другими словами, в нём недостаточно скобок). Любители LOOP замечают, что в этом то и весь смысл: сложные циклические конструкции достаточно тяжелы для восприятия и без заворачивания их в туманный синтаксис DO. Лучше, говорят они, иметь немного более наглядный синтаксис, который давал бы вам какие-то подсказки о том, что происходит.

Вот, например, характерный DO цикл, который собирает числа от 1 до 10 в список:

```
(do ((nums nil) (i 1 (1+ i)))
    (> i 10) (nreverse nums))
(push i nums)) ==> (1 2 3 4 5 6 7 8 9 10)
```

У опытного Лиспера не будет никаких проблем понять этот код – это просто вопрос понимания основной формы DO цикла и распознавания PUSH/NREVERSE идиомы для построения списка. Но это не совсем прозрачно. Версия с LOOP, с другой стороны, почти понятна как предложение на английском. (цикл по *i* от 1 до 10, собирая *i*)

```
(loop for i from 1 to 10 collecting i) ==> (1 2 3 4 5 6 7 8 9 10)
```

Далее, ещё несколько примеров простого использования LOOP. Вот сумма квадратов первых десяти чисел:

```
(loop for x from 1 to 10 summing (expt x 2)) ==> 385
```

Это – подсчёт числа гласных в строке:

```
(loop for x across "the quick brown fox jumps over the lazy dog"
      counting (find x "aeiou")) ==> 11
```

Вот вычисление одиннадцатого числа Фибоначчи, аналогично использованному ранее циклу DO:

```
(loop for i below 10
      and a = 0 then b
      and b = 1 then (+ b a)
      finally (return a))
```

Символы across, and, below, collecting, counting, finally, for, from, summing, then и to являются некоторыми из ключевых слов цикла, чьё присутствие обозначает, что перед нами расширенная версия LOOP.

Я приберегу подробности о LOOP для Главы 22, однако сейчас стоит заметить, что это ещё один пример того, как макросы могут быть использованы для расширения основы языка. В то время как LOOP предоставляет свой собственный язык для выражения циклических конструкций, он никак не отрезает вас от остального Лиспа. Ключевые слова LOOP разбираются в соответствии с его грамматикой, но остальной код внутри LOOP, это обычный Лисп-код.

И так же стоит отметить ещё раз, что хотя макрос LOOP гораздо более сложный, чем WHEN или UNLESS, он просто ещё один макрос. Если бы он не был включён в стандартную библиотеку, вы могли бы сделать это сами или взять стороннюю библиотеку, которая это сделает.

На этом я завершу наш тур в основные макросы, конструкции контроля. Теперь вы готовы взглянуть поближе на то, как определять свои собственные макросы.

8. Макросы: Создание собственных макросов

Теперь пора начать писать свои собственные макросы. Стандартные макросы, описанные мною в предыдущей главе, должны были дать вам некоторое представление о том, что вы можете сделать при помощи макросов, но это было только начало. Поддержка макросов в Common Lisp не является чем-то большим, чем поддержка функций в С, и поэтому каждый программист на Lisp может создать свои собственные варианты стандартных конструкций контроля точно так же, как каждый программист на С может написать простые варианты функций из стандартной библиотеки С. Макросы являются частью языка, которая позволяет вам создавать абстракции поверх основного языка и стандартной библиотеки, что приближает вас к возможности непосредственного выражения того, что вы хотите выразить.

Возможно, самым большим препятствием для правильного понимания макросов является, как это ни парадоксально, то, что они так хорошо интегрированы в язык. Во многих отношениях они кажутся просто странной разновидностью функций — они написаны на Лисп, они принимают аргументы и возвращают результаты, и они позволяют вам абстрагироваться от отвлекающих деталей. Тем не менее, несмотря на эти многочисленные сходства, макросы работают на другом, по сравнению с функциями, уровне и создают совершенно иной вид абстракции.

Как только вы поймете разницу между макросами и функциями, тесная интеграция макросов в язык станет огромным благом. И в то же время для новых лисперов это часто является источником путаницы. Следующая история, не являющаяся подлинной в историческом или техническом смысле, попытается уменьшить ваше замешательство, направляя ваши мысли касательно работы макросов в правильное русло.

История Мака: обычная такая история

Давным давно, много лет назад, была компания Lisp программистов. Это было так давно, что Lisp не имел макросов. Каждый раз все то, что не могло быть определено с помощью функций или сделано с помощью специальных операторов, должно было быть написано в полном объеме, что было очень тяжело. К сожалению программисты в этой компании были хоть и блестящи, но очень ленивы. Нередко в середине процесса написания своих программ, когда скука от написания больших объемов кода становилась слишком велика, они вместо кода писали комментарии, описывающие требуемый в этом месте программы код. К еще большему сожалению, ведь они были ленивы, программисты также ненавидели возвращаться назад и действительно писать код, описанный в комментариях. Вскоре компания получила большую кучу кода, которую никто не мог запустить, потому что он был полон комментариев с описанием того, что еще предстоит написать.

В отчаянье большой босс нанял младшего (junior) программиста, Мака, чьей работой стал поиск комментариев, написание требуемого кода и вставка его в программу на место комментариев. Мак никогда не запускал программы, ведь они не были завершены и поэтому он попросту не мог этого сделать. Но даже если бы они были завершены, Мак не знал, какие данные необходимо подать на их вход. Поэтому он просто писал свой код, основываясь на содержимом комментариев, и посылал его назад создавшему комментарий программисту.

С помощью Мака все программы вскоре были закончены и компания сделала массу денег продавая их: так много денег, что смогла удвоить количество программистов. Но по какой-то причине никто не думал нанимать кого-то в помощь Маку; вскоре он один помогал нескольким дюжинам программистов. Чтобы избежать траты всего своего времени на поиск комментариев в исходном коде, Мак внес небольшие изменения в используемый программистами компилятор. Теперь, когда компилятор встречал комментарий, он слал Маку электронное письмо с ним и ждал от Мака ответа с заменяющим комментарий кодом. К сожалению, даже с этими

изменениями Маку было тяжело справляться со всей работой. Он работал тщательно, как только мог, но иногда, особенно когда записи не были ясны, он допускал ошибки.

Однако программисты обнаружили, что чем точнее они пишут свои комментарии, тем больше вероятность того, что Мак вернет правильный код. Однажды один из программистов, встретив затруднение с описанием в словах нужного кода, включил в один из комментариев программу на Lisp, которая генерировала нужный код. Такой комментарий был удобен Маку: он просто запустил программу и послал результат компилятору.

Следующая инновация появилась, когда программист вставил в самый верх одной из своих программ комментарий, содержащий определение функции и пояснение, гласившее: "Мак, не пиши здесь никакого кода, но сохрани эту функцию на будущее; я собираюсь использовать ее в некоторых своих комментариях." Другие комментарии в этой программе гласили следующее: "Мак, замени этот комментарий на результат выполнения той функции с символами *x* и *y* как аргументами."

Этот метод распространился так быстро что в течение нескольких дней большинство программ стало содержать дюжины комментариев с описанием функций, которые использовались только кодом в других комментариях. Чтобы облегчить Маку различие комментариев, содержащих только определения и не требующих немедленного ответа, программисты отмечали их стандартным предисловием: "Definition for Mac, Read Only" (Определение для Мака, только для чтения). Это (как мы помним, программисты были очень ленивы) быстро сократилось до "DEF. MAC. R/O", а потом до "DEFMACRO".

Очень скоро в комментариях для Мака вообще не осталось английского. Целыми днями он читал и отвечал на электронные письма от компилятора, содержащие DEFMACRO комментарии и вызывал функции, описанные в DEFMACRO. Так как Lisp программы в комментариях осуществляли всю реальную работу, то работа с электронными письмами перестала быть проблемой. У Мака внезапно стало много свободного времени, и он сидел в своем кабинете и грезил о белых песчаных пляжах, чистой голубой океанской воде и напитках с маленькими бумажными зонтиками.

Несколько месяцев спустя программисты осознали что Мака уже довольно давно никто не видел. Придя в его кабинет, они обнаружили на всем тонкий слой пыли, стол был усыпан брошюрами о различных тропических местах, а компьютер был выключен. Но компилятор продолжал работать! Как ему это удавалось? Выяснилось, что Мак сделал последнее изменение в компиляторе: вместо отправления электронного письма с комментарием Маку компилятор теперь сохранял функции, описанные с помощью DEFMACRO комментариев, и запускал при вызове их из других комментариев. Программисты решили, что нет оснований говорить большим боссам, что Мак больше не приходит на работу. Так происходит и по сей день: Мак получает зарплату и время от времени шлет программистам открытки то из одной тропической страны, то из другой.

Время раскрытия макросов против времени выполнения

Ключом к пониманию макросов является полное понимание разницы между кодом, генерирующим код (макросами), и кодом, который в конечном счете выполняет программу (все остальное). Когда вы пишете макросы, вы пишете программы, которые будут использоваться компилятором для генерации кода, который затем будет скомпилирован. Только после того, как все макросы будут полностью раскрыты, а полученный код скомпилирован, программа сможет быть запущена. Время, когда выполняются макросы, называется *временем раскрытия макросов*; оно отлично от *времени выполнения*, когда выполняется обычный код, включая код, сгенерированный макросами.

Очень важно полностью понимать это различие, так как код, работающий во время раскрытия

макросов, запускается в окружении, сильно отличающемся от окружения кода, работающего во время выполнения. А именно, во время раскрытия макросов не существует способа получить доступ к данным, которые будут существовать во время выполнения. Подобно Маку, который не мог запускать программы, над которыми он работал, так как не знал, что является корректным входом для них, код, работающий во время раскрытия макросов, может работать только с данными, являющимися неотъемлемой частью исходного кода. Для примера предположим, что следующий исходный код появляется где-то в программе:

```
(defun foo (x)
  (when (> x 10) (print 'big)))
```

Обычно вы бы думали о `x` как о переменной, которая будет содержать аргумент, переданный при вызове `foo`. Но во время раскрытия макросов (например когда компилятор выполняет макрос **WHEN**) единственными доступными данными является исходный код. Так как программа пока не выполняется, нет вызова `foo` и, следовательно, нет значения, ассоциированного с `x`. Вместо этого значения, которые компилятор передает в **WHEN**, являются списками Lisp, представляющими исходный код, а именно `(> x 10)` и `(print 'big)`. Предположим, что **WHEN** определен, как вы видели в предыдущей главе, подобным образом:

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

При компиляции кода `foo` макрос **WHEN** будет запущен с этими двумя формами в качестве аргументов. Параметр `condition` будет связан с формой `(> x 10)`, а форма `(print 'big)` будет собрана (will be collected) в список (и будет его единственным элементом), который станет значением параметра **&rest** `body`. Выражение квазичитирования затем сгенерирует следующий код:

```
(if (> x 10) (progn (print 'big)))
```

подставляя значение `condition`, а также вклеивая значение `body` в **PROGN**.

Когда Lisp интерпретируется, а не компилируется, разница между временем раскрытия макросов и временем выполнения менее очевидна, так как они "переплетены" во времени (temporally intertwined). Также стандарт языка не специфицирует в точности того, как интерпретатор должен обрабатывать макросы: он может раскрывать все макросы в интерпретируемой форме, а затем интерпретировать полученный код, или же он может начать с непосредственно интерпретирования формы и раскрывать макросы при их встрече. В обоих случаях макросам всегда передаются невычисленные объекты Lisp, представляющие подформы формы макроса, и задачей макроса все также является генерирование кода, который затем осуществит какие-то действия, а не непосредственное осуществление этих действий.

DEFMACRO

Как вы видели в главе 3, макросы на самом деле определяются с помощью форм **DEFMACRO**, что означает, разумеется, "DEFine MACRO", а не "Definition for Mac". Базовый шаблон **DEFMACRO** очень похож на шаблон **DEFUN**.

```
(defmacro name (parameter*)
  "Optional documentation string."
  body-form*)
```

Подобно функциям, макрос состоит из имени, списка параметров, необязательной строки

документации и тела, состоящего из выражений Lisp . Однако, как я только что говорил, работой макроса не является осуществление какого-то действия напрямую, — его работой является генерирование кода, который затем сделает то, что вам нужно.

Макросы могут использовать всю мощь Lisp при генерировании своих раскрытий, поэтому в этой главе я смогу дать лишь обзор того, что вы можете делать с помощью макросов. Однако я могу описать общий процесс написания макросов, который подходит для всех типов макросов, от самых простых до наиболее сложных.

Задачей макроса является преобразование формы макроса (другими словами, формы Lisp, первым элементом которой является имя макроса) в код, который осуществляет определенные действия. Иногда вы пишете макрос начиная с того кода, который вы бы хотели иметь возможность писать, то есть с примера формы макроса. В другой раз вы решаете написать макрос после того, как вы использовали какой-то образец кода несколько раз и понимаете, что можете сделать ваш код чище путем абстрагирования этого образца.

Несмотря на то, с какого конца вы начинаете, вы должны представлять противоположный конец перед тем, как сможете начать создавать макрос: вы должны знать и то, откуда движетесь, и то, куда, до того как сможете рассчитывать написать код, осуществляющий это автоматически. Таким образом, первым шагом в написании макроса является написание по крайней мере одного примера вызова макроса и кода, в который этот вызов должен раскрыться.

После того, как у вас есть пример вызова и его желаемое раскрытие, вы готовы ко второму шагу: фактическому написанию кода макроса. Для простых макросов это будет тривиальным делом написания шаблона-квазигитирования с параметрами макроса, вставленными на нужные места. Сложные макросы сами будут значительными программами, использующими вспомогательные функции и структуры данных.

После того, как вы написали код, преобразующий пример вызова в соответствующее раскрытие, вам нужно убедиться в том, что у абстракции, предоставляемой макросом, нет "протечек" деталей реализации. Предоставляемые макросами "дырявые" абстракции будут работать хорошо только для определенных аргументов или будут взаимодействовать с кодом вызывающего окружения нежелательными способами. Как оказывается, макросы могут "протекать" лишь небольшим количеством способов, все из которых легко избежать, если вы знаете, как выявлять их. Я обсужу как это делается в секции "Устранение протечек".

Подводя итог можно сказать, что шаги по написанию макросов следующие:

1. Написание примера вызова макроса, а затем кода, в который он должен быть раскрыт (или в обратном порядке).
2. Написание кода, генерирующего написанный вручную код раскрытия по аргументам в примере вызова.
3. Проверка того, что предоставляемая макросом абстракция не "протекает".

Пример макроса: **do-primes**

Для того, чтобы увидеть, как этот трехшаговый процесс осуществляется, вы напишете макрос `do-primes`, который предоставляет конструкцию итерирования, подобную **DOTIMES** и **DOLIST**, за исключением того, что вместо итерирования по целым числам или элементам списка итерирование будет производиться по последовательным простым числам. Этот пример не является примером чрезвычайно полезного макроса, он — всего лишь средство демонстрации вышеописанного процесса.

Прежде всего вам нужны две вспомогательные функции: одна для проверки того, является ли данное число простым, и вторая, возвращающая следующее простое число, большее или равное ее аргументу. В обоих случаях вы можете использовать простой, но неэффективный метод "грубой силы".

```
(defun primep (number)
  (when (> number 1)
    (loop for fac from 2 to (isqrt number) never (zerop (mod number fac)))))
(defun next-prime (number)
  (loop for n from number when (primep n) return n))
```

Теперь вы можете написать макрос. Следуя процедуре, очерченной выше, вам нужен по крайней мере один пример вызова макроса и кода, в который он должен быть раскрыт. Предположим, что вы начали с мысли о том, что хотите иметь возможность написать следующее:

```
(do-primes (p 0 19)
  (format t "~d " p))
```

для выражения цикла, который выполняет тело для каждого простого числа, большего либо равного 0 и меньшего либо равного 19, используя переменную `p` для хранения очередного простого числа. Имеет смысл смоделировать этот макрос с помощью стандартных макросов **DOLIST** и **DOTIMES**; макрос, следующий образцу существующих макросов, легче понять и использовать, нежели макросы, которые вводят неоправданно новый синтаксис.

Без использования макроса `do-primes` вы можете написать такой цикл путем использования **DO** (и двух вспомогательных функций, определенных ранее) следующим образом:

```
(do ((p (next-prime 0) (next-prime (1+ p))))
  ((> p 19))
  (format t "~d " p))
```

Теперь вы готовы к написанию кода макроса, который будет выполнять необходимое преобразование.

Макропараметры

Так как аргументы, передаваемые в макрос, являются объектами Lisp, представляющими исходный код вызова макроса, первым шагом любого макроса является извлечение тех частей этих объектов, которые нужны для вычисления раскрытия. Для макросов, которые просто подставляют свои аргументы напрямую в шаблон, этот шаг тривиален: подходит простое определение правильных параметров для захвата нужных аргументов.

Но, кажется, такого подхода недостаточно для `do-primes`. Первый аргумент вызова `do-primes` является списком, содержащим имя переменной цикла, `p`; нижнюю границу, 0; верхнюю границу, 19. Но, если вы посмотрите на раскрытие, список, как целое, не встречается в нем: эти три элемента разделены и вставлены в различные места.

Вы можете определить `do-primes` с двумя параметрами, первый для захвата этого списка и параметр **&rest** для захвата форм тела цикла, а затем разобрать первый список вручную подобным образом:

```
(defmacro do-primes (var-and-range &rest body)
  (let ((var (first var-and-range))
        (start (second var-and-range))
        (end (third var-and-range)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
        ((> ,var ,end))
        ,@body)))
```

Очень скоро я объясню как тело макроса генерирует правильное раскрытие; сейчас же вам следует отметить, что переменные `var`, `start` и `end`, каждая содержит значение, извлеченное из `var-and-range`, и эти значения затем подставляются в выражение квазицитирования, генерирующее раскрытие `do-primes`.

Однако, вам не нужно разбирать `var-and-range` вручную, так как список параметров макроса является так называемым списком *деструктурируемых* параметров. Деструктурирование, как и говорит название, осуществляет разбор некоторой структуры, в нашем случае списочной структуры форм, переданных макросу.

Внутри списка деструктурируемых параметров простое имя параметра может быть заменено вложенным списком параметров. Параметры в таком списке будут получать свои значения из элементов выражения, которое было бы связано с параметром, замененным этим списком. Например, вы можете заменить `var-and-range` списком `(var start end)` и три элемента списка будут автоматически деструктурированы в эти три параметра.

Другой особенностью списка параметров макросов является то, что вы можете использовать **&body** как синоним **&rest**. Семантически **&body** и **&rest** эквиваленты, но множество сред разработки будут использовать факт наличия параметра **&body** для изменения того, как они будут выравнивать код использования макроса, поэтому обычно параметры **&body** используются для захвата списка форм, которые составляют тело макроса.

Таким образом, вы можете улучшить определение макроса `do-primes` и дать подсказку как людям, читающим ваш код, так и вашим инструментам разработки, об его предназначении следующим образом

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
      ((> ,var ,end))
      ,@body))
```

В стремлении к краткости список деструктурируемых параметров также предоставляет вам автоматическую проверку ошибок: при определении таким образом `do-primes`, Lisp будет способен определять вызовы, в которых первый аргумент не является трехэлементным списком, и выдавать вам разумные сообщения об ошибках как когда вы вызываете функцию со слишком малым или, наоборот, слишком большим числом аргументов. Также, среды разработки, такие как SLIME, указывающие вам какие аргументы ожидаются, как только вы напечатаете имя функции или макроса, при использовании вами списка деструктурируемых параметров будут способны более конкретно указать синтаксис вызова макроса. С исходным определением SLIME будет подсказывать вам, что `do-primes` вызывается подобным образом:

```
(do-primes var-and-range &rest body)
```

С новым же описанием она сможет указать вам, что вызов должен выглядеть следующим образом:

```
(do-primes (var start end) &body body)
```

Списки деструктурируемых параметров могут содержать параметры **&optional**, **&key** и **&rest**, а также вложенные деструктурируемые списки. Однако все эти возможности не нужны вам для написания `do-primes`.

Генерация раскрытия

Так как `do-primes` является довольно простым макросом, после деструктурирования аргументов все, что вам остается сделать — это подставить их в шаблон для получения раскрытия.

Для простых макросов, наподобие `do-primes`, лучшим вариантом является использование специального синтаксиса квазицитирования. Коротко говоря, выражения квазицитирования подобны выражениям цитирования, за исключением того, что вы можете "раскавычить" определенные подвыражения, предваряя их запятой, за которой возможно следует знак "at" (@). Без этого знака "at" запятая вызывает включение как есть значения следующего за ней подвыражения. Со знаком "at" значение, которое должно быть списком, "вклеивается" в окружающий список.

Другой пригодный способ думать о синтаксисе квазицитирования как об очень кратком способе написания кода, генерирующего списки. Такое представление о нем имеет преимущество, так как является очень близким к тому, что на самом деле происходит "под капотом": когда процедура чтения считывает выражение квазицитирования, она преобразует его в код, который генерирует соответствующую списковую структуру. Например, ``(a b)` вероятно будет прочитано как `(list a 'b)`. Стандарт языка не указывает, какой в точности код процедура чтения должна выдавать, пока она генерирует правильные списковые структуры.

Таблица 8-1 показывает некоторые примеры выражений квазицитирования вместе с эквивалентным создающим списки кодом, а также результаты, которые вы получите при вычислении как выражений квазицитирования, так и эквивалентного кода:

Таблица 8-1. Примеры квазицитирования		
Синтаксис квазицитирования	Эквивалентный создающий списки код	Результат
<code>`(a (+ 1 2) c)</code>	<code>(list 'a '(+ 1 2) 'c)</code>	<code>(a (+ 1 2) c)</code>
<code>`(a ,(+ 1 2) c)</code>	<code>(list 'a (+ 1 2) 'c)</code>	<code>(a 3 c)</code>
<code>`(a (list 1 2) c)</code>	<code>(list 'a '(list 1 2) 'c)</code>	<code>(a (list 1 2) c)</code>
<code>`(a ,(list 1 2) c)</code>	<code>(list 'a (list 1 2) 'c)</code>	<code>(a (1 2) c)</code>
<code>`(a ,@(list 1 2) c)</code>	<code>(append (list 'a) (list 1 2) (list 'c))</code>	<code>(a 1 2 c)</code>

Важно заметить, что нотация квазицитирования является просто удобством. Но это большое удобство. Для оценки того, насколько оно велико, сравните версию `do-primes` с квазицитированием со следующей версией, которая явно использует создающий списки код:

```
(defmacro do-primes-a ((var start end) &body body)
  (append '(do)
    (list (list (list var
                    (list 'next-prime start)
                    (list 'next-prime (list '1+ var))))))
    (list (list (list '> var end)))
    body))
```

Как вы очень скоро увидите, текущая реализация `do-primes` не обрабатывает корректно некоторые граничные случаи. Но первое, что вы должны проверить, — это то, что она по крайней мере работает для исходного примера. Вы можете сделать это двумя способами. Во-первых, вы можете косвенно протестировать свою реализацию просто воспользовавшись ею (подразумевая, что если итоговое поведение корректно, то и раскрытие также корректно). Например, вы можете напечатать исходный пример использования `do-primes` в REPL и увидеть, что он и в самом деле напечатает правильную последовательность простых чисел.

```
CL-USER> (do-primes (p 0 19) (format t "~d " p))
2 3 5 7 11 13 17 19
NIL
```

Или же вы можете проверить макрос напрямую, посмотрев на раскрытие определенного вызова. Функция **MACROEXPAND-1** получает любое выражение Lisp в качестве аргумента и возвращает результат осуществления одного шага раскрытия макроса. Так как **MACROEXPAND-1** является функцией, для дословной передачи ей формы макроса вы должны зацитировать эту форму. Теперь вы можете воспользоваться **MACROEXPAND-1** для просмотра раскрытия предыдущего вызова.

```
CL-USER> (macroexpand-1 '(do-primes (p 0 19) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
  ((> P 19))
  (FORMAT T "~d " P))
T
```

Также, для большего удобства, в SLIME вы можете проверить раскрытие макроса поместив курсор на открывающую скобку формы макроса в вашем исходном коде и набрав C-c RET для вызова функции Emacs `slime-macroexpand-1`, которая передаст форму макроса в **MACROEXPAND-1** и напечатает результат во временном буфере.

Теперь вы можете видеть, что результат раскрытия макроса совпадает с исходным написанным вручную раскрытием, и поэтому кажется, что `do-primes` работает.

Устранение протечек

В своем эссе "Закон дырявых абстракций" Джоэл Спольски придумал термин "дырявой абстракции" для описания такой абстракции, через которую "протекают" детали, абстрагирование от которых предполагается. Так как написание макроса — это способ создания абстракции, вам следует убедиться, что ваш макрос излишне не "протекает"

Как оказывается, внутренние детали реализации могут "протекать" через макросы тремя способами. К счастью, довольно легко сказать, имеет ли место одна из трех этих возможностей, и устранить их.

Текущее определение страдает от одной из трех возможных "протечек" макросов, а именно, оно вычисляет подформу `end` слишком много раз. Предположим, что вы вызвали `do-primes` с таким выражением, как `(random 100)`, на месте параметра `end` вместо использования числового литерала, такого, как `19`.

```
(do-primes (p 0 (random 100))
  (format t "~d " p))
```

Предполагаемым поведением здесь является итерирование по простым числам от нуля до какого-то случайного простого числа, возвращенного `(random 100)`. Однако, это не то, что делает

текущая реализация, как это показывает **MACROEXPAND-1**.

```
CL-USER> (macroexpand-1 '(do-primes (p 0 (random 100)) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
  ((> P (RANDOM 100)))
  (FORMAT T "~d " P))
T
```

При запуске кода раскрытия **RANDOM** будет вызываться при каждой проверке условия окончания цикла. Таким образом, вместо итерирования, пока *p* не станет больше, чем изначально выбранное случайное число, этот цикл будет осуществляться пока не случится, что выбранное в очередной раз случайное число окажется меньше текущего значения *p*. Хотя общее число итераций по-прежнему случайно, оно будет подчиняться вероятностному распределению, отличному от равномерного распределения результатов **RANDOM**.

Это является "протечкой" абстракции, так как для корректного использования макроса, его пользователь должен быть осведомлен о том, что форма *end* будет вычисляться более одного раза. Одним из способов устранения этой "протечки" является простое специфицирование ее как поведения *do-primes*. Но это не достаточно удовлетворительно: при реализации макросов вам следует пытаться соблюдать Правило Наименьшего Удивления. К тому же программисты обычно ожидают, что формы, которые они передают макросам, будут вычисляться не большее число раз, чем это действительно необходимо. Более того, так как *do-primes* построена на основе модели стандартных макросов **DOTIMES** и **DOLIST**, которые вычисляют однократно все свои формы, кроме форм тела, то большинство программистов будут ожидать от *do-primes* подобного поведения.

Вы можете исправить множественное вычисление достаточно легко: вам просто следует сгенерировать код, который вычисляет *end* однократно и сохраняет результат в переменную для дальнейшего использования. Вспомним, что в цикле **DO** переменные с формой инициализации и без формы вычисления последующих значений не изменяются от итерации к итерации. Поэтому вы можете исправить проблему множественных вычислений следующим определением:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((ending-value ,end)
        (,var (next-prime ,start) (next-prime (1+ ,var))))
    ((> ,var ending-value))
    ,@body))
```

К сожалению данное исправление вводит две новые "протечки" в предоставляемую нашим макросом абстракцию.

Одна из этих "протечек" подобна проблеме множественных вычислений, которую мы только что исправили. Так как формы инициализации переменных цикла **DO** вычисляются в том порядке, в каком переменные определены, то когда раскрытие макроса вычисляется, выражение, переданное как *end*, будет вычислено перед выражением, переданным как *start*, то есть в обратном порядке от того, как они идут в вызове макроса. Эта "протечка" не вызывает никаких проблем пока *start* и *end* являются литералами вроде 0 и 19. Но, если они являются формами, которые могут иметь побочные эффекты, вычисление их в неправильном порядке снова нарушает Правило Наименьшего Удивления.

Эта "протечка" устраняется тривиально путем изменения порядка определения двух переменных.


```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (ending-value ,end))
      (> ,var ending-value))
    ,@body))
```

Последняя "протечка", которую нам нужно устранить, была создана использованием имени переменной `ending-value`. Проблема заключается в том, что имя, которое должно быть полностью внутренней деталью реализации макроса, может вступить во взаимодействие с кодом, переданным макросу, или с контекстом, в котором макрос вызывается. Следующий, кажущийся вполне допустимым, вызов `do-primes` не работает корректно из-за данной "протечки":

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

То же касается и следующего вызова:

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

И снова **MACROEXPAND-1** может вам показать, в чем проблема. Первый вызов расширяется в следующее:

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
    (ending-value 10))
  (> ending-value ending-value))
  (print ending-value))
```

Некоторые реализации Lisp могут отвергнуть такой код из-за того, что `ending-value` используется дважды в качестве имен переменных одного и того-же цикла **DO**. Если же этого не произойдет, то код заикнется, так как `ending-value` никогда не станет больше себя самого.

Второй проблемный вызов расширяется следующим образом:

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
      (ending-value 10))
    (> p ending-value))
    (incf ending-value p))
  ending-value)
```

В этом случае сгенерированный код полностью допустим, но его поведение совсем не то, что нужно вам. Так как привязка `ending-value`, установленная с помощью **LET** снаружи цикла перекрывается переменной с таким же именем внутри **DO**, то форма `(incf ending-value p)` увеличивает переменную цикла `ending-value` вместо внешней переменной с таким же именем, создавая другой вечный цикл.

Очевидно, что то, что нам нужно для устранения этой "протечки" — это символ, который никогда не будет использоваться снаружи кода, сгенерированного макросом. Вы можете попытаться использовать действительно маловероятный символ, но это все равно не даст вам никаких гарантий. Вы можете также защитить себя в некоторой степени путем использования пакетов, описанных в главе 21. Но существует лучшее решение.

Функция **GENSYM** возвращает уникальный символ при каждом своем вызове. Такой символ никогда до этого не был прочитан процедурой чтения Lisp и, так как он не интернирован (isn't interned) ни в один пакет, никогда не будет прочитан ею. Поэтому, вместо использования литеральных имен наподобие `ending-value`, вы можете генерировать новый символ при каждом раскрытии `do-primes`.

```
(defmacro do-primes ((var start end) &body body)
  (let ((ending-value-name (gensym)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (,ending-value-name ,end))
      (> ,var ,ending-value-name))
      ,@body)))
```

Обратите внимание, что код, вызывающий **GENSYM** не является частью раскрытия; он запускается как часть процедуры раскрытия макроса и поэтому создает новый символ при каждом раскрытии макроса. Это может казаться несколько странным сначала: `ending-value-name` является переменной, чье значение является именем другой переменной. Но на самом деле тут нет никаких отличий от параметра `var`, чье значение также является именем переменной. Единственная разница состоит в том, что значение `var` было создано процедурой чтения, когда форма макроса была прочитана, а значение `ending-value-name` было сгенерировано программно при запуске кода макроса.

С таким определением две ранее проблемные формы расширяются в код, который работает так, как вам нужно. Первая форма:

```
(do-primes (ending-value 0 10)
  (print ending-value))
```

расширяется в следующее:

```
(do ((ending-value (next-prime 0) (next-prime (1+ ending-value)))
    (#:g2141 10))
    (> ending-value #:g2141))
  (print ending-value))
```

Теперь переменная, используемая для хранения конечного значения является сгенерированным функцией `gensym` символом, `#:g2141`. Имя идентификатора, `G2141`, было сгенерировано с помощью **GENSYM**, но важно не это; важно то, что идентификатор хранит значение объекта. Сгенерированные таким образом символы печатаются в обычном синтаксисе для неинтернированных символов: с начальным `#:`.

Вторая ранее проблемная форма:

```
(let ((ending-value 0))
  (do-primes (p 0 10)
    (incf ending-value p))
  ending-value)
```

после замены `do-primes` его раскрытием будет выглядеть подобным образом:

```
(let ((ending-value 0))
  (do ((p (next-prime 0) (next-prime (1+ p)))
      (#:gensym 10))
      ((> p #:gensym 10))
      (incf ending-value p))
  ending-value)
```

И снова, тут нет никакой "протечки", так как переменная `ending-value`, связанная окружающей цикл `do-primes` формой **LET**, больше не перекрывается никакими переменными, вводимыми в код раскрытия.

Не все литеральные имена, используемые в раскрытии макросов, обязательно вызовут проблему; когда вы приобретете больше опыта работы с различными связывающими формами, вы сможете определять, приведет ли использование данного имени в определенном месте к "протечке" в предоставляемой макросом абстракции. Но нет никаких реальных проблем в использовании сгенерированных имен везде для уверенности.

Этим исправлением мы устранили все "протечки" в реализации `do-primes`. После получения некоторого опыта в написании макросов, вы научитесь писать макросы с заранее устраненными "протечками" такого рода. На самом деле это довольно просто, если вы будете следовать следующим правилам:

- * Если только нет определенной причины сделать иначе, включайте все подформы в раскрытие на такие позиции, чтобы они выполнялись в том же порядке, в каком они идут в вызове макроса.
- * Если только нет определенной причины сделать иначе, убедитесь, что все подформы вычисляются лишь единожды, путем создания в раскрытии переменных для содержания значений вычисления форм аргументов и последующего использования этих переменных везде в раскрытии, где нужны значения этих форм.
- * Используйте **GENSYM** во время раскрытия макросов для создания имен переменных, используемых в раскрытии.

Макросы, создающие макросы

Конечно же, нет никаких причин, по которым вы должны получать преимущества от использования макросов только при написании функций. Задачей макросов является абстрагирование общих синтаксических образцов, а некоторые образцы появляются снова и снова и при написании макросов, поэтому и тут можно получить преимущества от абстрагирования.

На самом деле, вы уже видели один такой образец: многие макросы, как и последняя версия `do-primes`, начинаются с **LET**, который вводит несколько переменных, содержащих сгенерированные символы для использования в раскрытии макроса. Так как это общий образец, почему бы нам не абстрагировать его с помощью его собственного макроса?

В этой секции вы напишете макрос `with-gensyms`, который делает именно это. Другими словами, вы напишете макрос, создающий макрос: макрос, который генерирует код, который генерирует код. В то время как сложные макросы, создающие макросы, могут слегка сбивать с толку, пока вы не привыкнете к легкому умозрительному обращению с различными уровнями кода, `with-gensyms` довольно прямолинеен и послужит полезным и, в то же время, не требующим непомерных умственных усилий упражнением.

Предположим, вы хотите иметь возможность написать подобное:

```
(defmacro do-primes ((var start end) &body body)
  (with-gensyms (ending-value-name)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
        (,ending-value-name ,end))
      (> ,var ,ending-value-name))
      ,@body)))
```

и получить `do-primes`, эквивалентный его предыдущей версии. Другими словами, `with-gensyms` должен раскрываться в **LET**, которая связывает каждую перечисленную переменную, `ending-value-name` в данном случае, со сгенерированным символом. Достаточно просто написать это с помощью простого шаблона-квазитирования.

```
(defmacro with-gensyms ((&rest names) &body body)
  `(let ,(loop for n in names collect `(,n (gensym)))
    ,@body))
```

Обратите внимание, как мы можем использовать запятую для подстановки значения выражения **LOOP**. Этот цикл генерирует список связывающих форм, каждая из которых состоит из списка, содержащего одно из переданных `with-gensyms` имен, а также литеральный код `(gensym)`. Вы можете проверить, какой код сгенерирует выражение **LOOP** в REPL, заменив `names` списком символов.

```
CL-USER> (loop for n in '(a b c) collect `(,n (gensym)))
((A (GENSYM)) (B (GENSYM)) (C (GENSYM)))
```

После списка связывающих форм в качестве тела **LET** вклеивается аргумент `body` `with-gensyms`. Таким образом, из кода, который вы оборачиваете в `with-gensyms`, вы можете ссылаться на любое из имен переменных из списка переменных, переданного `with-gensyms`.

Если вы воспользуетесь `macro-expand` для формы `with-gensyms` в новом определении `do-primes`, то вы получите подобное:

```
(let ((ending-value-name (gensym)))
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
      (,ending-value-name ,end))
    (> ,var ,ending-value-name))
    ,@body))
```

Выглядит неплохо. Хотя этот макрос довольно прост, очень важно ясно понимать то, когда различные макросы раскрываются: когда вы компилируете **DEFMACRO** `do-primes`, форма `with-gensyms` раскрывается в код, который вы только что видели. Таким образом, скомпилированная версия `do-primes` в точности такая же, как если бы вы написали внешний **LET** вручную. Когда вы компилируете функцию, которая использует `do-primes`, то для генерации расширения `do-primes` запускается код, сгенерированный `with-gensyms`, но сам `with-gensyms` при компиляции формы `do-primes` не нужен, так как он уже был раскрыт при компиляции `do-primes`.

Другой классический макрос, создающий макросы: **ONCE-ONLY**

Другим классическим макросом, создающим макросы, является `once-only`, который используется для генерации кода, вычисляющего определенные аргументы макроса только единожды и в определенном порядке. Используя `once-only` вы можете написать `do-primes` почти таким же простым способом, как исходную "протекающую" версию, следующим образом:

```
(defmacro do-primes ((var start end) &body body)
  (once-only (start end)
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
      ((> ,var ,end))
      ,@body)))
```

Однако, реализация `once-only` несколько запутанна для обычного пошагового объяснения, так как зависит от множества уровней квазицитирования и "раскавычивания". Если вы действительно хотите попрактиковаться в понимании макросов, вы можете попытаться разобраться, как он работает. Макрос выглядит следующим образом:

```
(defmacro once-only ((&rest names) &body body)
  (let ((gensyms (loop for n in names collect (gensym))))
    `(let ((,@(loop for g in gensyms collect `(,g (gensym))))
      `(let ((,@(loop for g in gensyms for n in names collect ``(,,g ,,n)))
        ,(let ((,@(loop for n in names for g in gensyms collect `(,n ,g)))
          ,@body))))))
```

Не только простые макросы

Конечно я могу рассказать о макросах намного больше. Все макросы, которые вы до сих пор видели, были довольно простыми примерами, избавляющими вас от небольшого количества работы по набору текста, но не предоставляющими радикально новых способов выражения мыслей. В последующих главах вы увидите примеры макросов, позволяющих вам выражать мысли способами, практически не возможными без макросов. И вы начнете прямо со следующей главы, в которой вы создадите простой, но эффективный фреймворк для модульного тестирования.

Практикум: Каркас для юнит-тестирования.

В этой главе вы вернётесь к написанию кода и разработаете простой каркас для юнит-тестирования Lisp. Это даст вам возможность использовать в реальном коде некоторые возможности языка, о которых вы узнали после главы 3, включая макросы и динамические переменные.

Вашей главной целью при проектировании каркаса для тестирования будут лёгкость добавления новых тестов, запуска различных наборов тестов и отслеживания проваленных тестов. Вы сосредоточите усилия на проектировании каркаса, который можно использовать при интерактивной разработке.

Главная особенность автоматизированного тестирования состоит в том, что каркас отвечает за проверку, все ли тесты выполнены успешно. Вам не требуется тратить время на то, чтобы пробираться сквозь результаты, сверяя их с ожидаемыми — компьютер может сделать это гораздо быстрее и аккуратнее вас. Как следствие, каждый тест должен быть выражением, которое вырабатывает логическое значение — истина или ложь, тест выполнен успешно или провалился. К примеру, если вы тестируете встроенную функцию `+`, следующие выражения являются вполне разумными тестами :

```
(= (+ 1 2) 3)
(= (+ 1 2 3) 6)
(= (+ -1 -3) -4)
```

Функции с побочными эффектами необходимо тестировать слегка по-другому — вам придётся вызвать функцию и затем проверить наличие ожидаемых побочных эффектов . Но в любом случае каждый тест сводится к логическому выражению: сработало или не сработало.

Два первых подхода

Если бы вы тестировали вручную, вы бы вводили эти выражения в REPL и проверяли бы, что они возвращают `T`. Но вам нужен каркас, который позволяет с лёгкостью организовывать и запускать эти тесты в любое время. Если вы хотите начать с самой простой работающей версии, вы можете просто написать функцию, которая вычисляет все тесты и возвращает `T` в случае успешного прохождения всех тестов (используя `AND` для этого).

```
(defun test-+ ()
  (and
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

Для запуска тестов просто вызовите `test-+`.

```
CL-USER> (test-+)
T
```

Пока функция возвращает `T`, вы знаете, что тесты проходят. Такой способ организации тестов также весьма выразителен — вам не нужно писать много кода, обслуживающего тестирование. Однако при первом же проваливающемся тесте вы заметите, что отчёт о тестировании оставляет желать лучшего: если `test-+` возвращает `NIL`, вы знаете, что какой-то тест провалился, но не имеете понятия, какой именно.

Давайте попробуем другой простой (можно даже сказать — глупый) подход: чтобы проверить, что случилось с каждым тестом, будете писать так:

```
(defun test-+ ()
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

Теперь каждый тест будет сообщать результат отдельно. Часть `~:[FAIL~;pass~]` форматной строки `FORMAT` печатает `FAIL` если первый аргумент ложен и `pass` — если истинен. Теперь запуск `test-+` покажет подробности происходящего.

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
NIL
```

В этот раз отчёт выглядит гораздо лучше, но сам код ужасен. Повторяющиеся вызовы `FORMAT` и утомительное дублирование тестовых выражений напрашиваются на рефакторинг. Дублирование выражений особо раздражает, потому что если вы опечатаетесь, то и результаты тестирования будут промаркированы неверно.

Другая проблема состоит в том, что вы не получаете единого ответа, прошли ли все тесты успешно. Для трёх тестов достаточно легко проверить, что вывод не содержит строчек `FAIL`, но при наличии сотен тестов это начнёт надоедать.

Рефакторинг

Что вам действительно нужно — это способ писать тесты так элегантно, как в первой функции `test-+`, которая возвращает `T` или `NIL`, но также отчитывается о результатах индивидуальных тестов, так, как во второй версии. Поскольку вторая версия близка по функциональности к тому, что вам нужно, лучшее, что вы можете сделать — проверить, можно ли исключить из неё раздражающее дублирование.

Простейший способ избавиться от повторяющихся похожих вызовов `FORMAT` — создать новую функцию.

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form))
```

Теперь вы можете писать `test-+`, вызывая `report-result` вместо `FORMAT`. Не слишком упрощает жизнь, но по крайней мере если вы решите изменить вид выдаваемых результатов, то вам придётся менять код только в одном месте.

```
(defun test-+ ()
  (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

Следующее, что нужно сделать — избавиться от дублирования тестового выражения, с присущим дублированию риском неправильной маркировки результата тестирования. Что вам нужно — это возможность обработать тестовое выражение одновременно как код (для получения результата теста) и как данные (для использования в качестве метки теста).

Использование кода, как данных — это безошибочный признак того, что вам нужен макрос. Или, если посмотреть на это с другой стороны, вам нужен способ автоматизировать подверженное ошибкам написание вызовов `report-result`. Неплохо было бы написать что-то, похожее на

```
(check (= (+ 1 2) 3))
```

и чтобы это означало следующее:

```
(report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
```

Написание макроса для выполнения этой трансформации тривиально.

```
(defmacro check (form)
  `(report-result ,form ',form))
```

Теперь вы можете изменить `test-+`, чтобы использовать `check`.

```
(defun test-+ ()
  (check (= (+ 1 2) 3))
  (check (= (+ 1 2 3) 6))
  (check (= (+ -1 -3) -4)))
```

Раз уж вы устраняете дублирование, почему бы не избавиться от повторяющихся вызовов `check`? Можно заставить `check` принимать произвольное количество аргументов и заворачивать каждый из них в вызов `report-result`.

```
(defmacro check (&body forms)
  `(progn
    ,@(loop for f in forms collect `(report-result ,f ',f))))
```

Это определение использует общепринятую идиому — оборачивание набора форм в вызов `PROGN`, чтобы сделать их единой формой. Заметьте, как можно использовать `,@` для `FIXME` вклеивания результата выражения, которое возвращает список выражений, которые сами по себе созданы с помощью шаблона, созданного обратной кавычкой.

С новой версией `check` можно написать новую версию `test-+` следующим образом:

```
(defun test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

что эквивалентно следующему коду:

```
(defun test-+ ()
  (progn
    (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
    (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
    (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4))))
```

Благодаря макросу `check`, этот вариант столь же краток, как первая версия `test-+`, но раскрывается в код, который делает то же самое, что вторая версия. Кроме того, вы можете внести любые изменения в поведение `test-+`, изменяя только `check`.

Чиним возвращаемое значение

Вы можете начать с изменения `test+` таким образом, чтобы его возвращаемое значение показывало, все ли тесты завершились успешно. Поскольку `check` отвечает за генерацию кода, который запускает тексты, вам нужно изменить его так, чтобы генерируемый код подсчитывал результаты тестов.

Для начала можно внести небольшое изменение в `report-result`, чтобы он возвращал результат выполняемого им теста.

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form)
  result)
```

Теперь, когда `report-result` возвращает значение теста, кажется, что вы можете просто изменить `PROGN` на `AND`. К сожалению, `AND` не будет работать так, как вам хочется в этом случае, из-за своего `FIXME short-circuit`: как только один из тестов провалится, `AND` пропустит остальные. С другой стороны, если бы вы имели конструкцию, которая действует как `AND`, но не `FIXME short-circuit`, вы могли бы её использовать на месте `PROGN`. Common Lisp не предоставляет такой конструкции, но это не мешает вам использовать её: вы с легкостью написать макрос, предоставляющий такую конструкцию.

Оставляя тесты в стороне на минуту, вам нужен макрос (назовём его `combine-results`), который позволит вам сказать

```
(combine-results
  (foo)
  (bar)
  (baz))
```

и это будет значить

```
(let ((result t))
  (unless (foo) (setf result nil))
  (unless (bar) (setf result nil))
  (unless (baz) (setf result nil))
  result)
```

Единственный нетривиальный момент в написании этого макроса - это введение переменной (`result` в предыдущем кусочке кода) в раскрытие макроса. Как вы видели в предыдущей главе, использование обычных имён для переменных в раскрытом макросе может заставить протекать абстракцию, так что вам нужно будет создать уникальное имя, что делается с помощью `with-gensyms`. Вы можете определить `combine-results` так:

```
(defmacro combine-results (&body forms)
  (with-gensyms (result)
    `(let ((,result t))
      ,@(loop for f in forms collect `(unless ,f (setf ,result nil)))
      ,result)))
```

Теперь вы можете исправить `check`, просто заменив `PROGN` на `combine-results`.

```
(defmacro check (&body forms)
  `(combine-results
    ,@(loop for f in forms collect `(report-result ,f ',f))))
```

С этой версией `check test-+` должен выдавать результаты своих трёх тестов и затем возвращать `T`, показывая, что все тесты завершились успешно .

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
T
```

Если вы измените один из тестов так, чтобы он проваливался , возвращаемое значение изменится на `NIL`.

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
FAIL ... (= (+ -1 -3) -5)
NIL
```

Улучшение отчёта

Пока вы тестируете только одну функцию, результаты тестирования обозримы. Если какой-то тест проваливается, всё, что вам нужно сделать - это найти его в конструкции `check` и понять, почему он не срабатывает. Но если вы пишете много тестов, вы, возможно, захотите структурировать их, а не запихивать все больше и больше тестов в одну функцию. Например, предположим, что вы хотите добавить несколько тестов для функции `*`. Вы могли бы написать новую функцию тестирования.

```
(defun test-* ()
  (check
    (= (* 2 2) 4)
    (= (* 3 5) 15)))
```

Теперь у вас есть две тестовые функции, так что вы возможно захотите написать ещё одну функцию, которая запускает все тесты. Это достаточно легко.

```
(defun test-arithmetic ()
  (combine-results
    (test-+)
    (test-*)))
```

В этой функции вы используете `combine-results` вместо `check`, потому что и `test-+`, и `test-*` сами позаботятся о выводе результатов своих тестов. Когда вы запустите `test-arithmetic`, вы получите следующий результат:

```
CL-USER> (test-arithmetic)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
pass ... (= (* 2 2) 4)
pass ... (= (* 3 5) 15)
T
```

Теперь представьте, что один из тестов провалился, и вам нужно найти проблему. Для пяти тестов и двух тестовых функций это будет не так сложно. Но представьте себе, что у вас 500 тестов, разнесённых по 20 функциям. Неплохо было бы, чтобы результаты сообщали вам, в какой функции находится каждый тест.

Поскольку код, который печатает результаты тестов, собран в `report-result`, вам нужен способ передать в неё информацию о том, в какой тестовой функции вы находитесь. Вы можете добавить параметр, сообщающий это, в `report-result`, но `check`, который генерирует вызовы `report-result`, не знает, из какой функции он вызван, что означает, что вам придётся изменить вызовы `check`, передавая аргумент, который он будет передавать дальше, в `report-result`.

Это в точности та проблема, для решения которой были придуманы динамические переменные. Если вы создадите динамическую переменную, которая привязывается к имени тестовой функции, то `report-result` сможет использовать её, а `check` может ничего о ней не знать.

Для начала определим переменную на верхнем уровне.

```
(defvar *test-name* nil)
```

Теперь слегка изменим `report-result`, чтобы включить `*test-name*` в вывод `FORMAT`.

```
(format t "~:[FAIL~;pass~] ... ~a: ~a%" result *test-name* form)
```

После этих изменений тестовые функции всё ещё работают, но выдают следующие результаты из-за того, что `*test-name*` нигде не привязывается к значению, отличному от начального:

```
CL-USER> (test-arithmetic)
pass ... NIL: (= (+ 1 2) 3)
pass ... NIL: (= (+ 1 2 3) 6)
pass ... NIL: (= (+ -1 -3) -4)
pass ... NIL: (= (* 2 2) 4)
pass ... NIL: (= (* 3 5) 15)
T
```

Для того, чтобы правильно выдавать имена тестовых функций в выводе, вам нужно изменить их.

```
(defun test-+ ()
  (let ((*test-name* 'test-+))
    (check
      (= (+ 1 2) 3)
      (= (+ 1 2 3) 6)
      (= (+ -1 -3) -4))))
(defun test-* ()
  (let ((*test-name* 'test-*))
    (check
      (= (* 2 2) 4)
      (= (* 3 5) 15))))
```

Теперь результаты правильно помечены именами тестовых функций.

```
CL-USER> (test-arithmetic)
pass ... TEST-+: (= (+ 1 2) 3)
pass ... TEST-+: (= (+ 1 2 3) 6)
pass ... TEST-+: (= (+ -1 -3) -4)
pass ... TEST-*: (= (* 2 2) 4)
pass ... TEST-*: (= (* 3 5) 15)
T
```

Выявление абстракций

При внесении изменений в тестовые функции, вы снова получили дублирующийся код. Тестовые функции не только дважды включают своё имя — первый раз при определении, второй раз при связывании с глобальной переменной `*test-name*` — но обе они начинаются совершенно одинаково (вся разница — имя функции). Вы могли бы попытаться избавиться от дублирования просто потому, что это некрасиво. Но если рассмотреть причину, вызвавшую дублирование, более подробно, то можно извлечь довольно важный урок по использованию макросов.

Причина, по которой обе функции начинаются одинаково, в том, что они обе предназначены для тестирования. Дублирование возникает из-за того, что тестовая функция — это только одна половина абстракции. Эта абстракция существует в вашей голове, но в коде нет возможности сказать "это — тестовая функция" другим способом, кроме как написанием соответствующего паттерна.

К сожалению, неполные абстракции — плохие помощники при написании программ. Полуабстракция, описанная в коде соответствующим паттерном, гарантирует вам массовое дублирование кода со всеми сопутствующими проблемами поддержки этого кода в дальнейшем. Более того, так как подобные абстракции целиком существуют только в наших мыслях, у нас нет никакой возможности убедиться, что разные программисты (или даже один и тот же — но в разное время) одинаково понимают одну и ту же абстракцию. Дабы полностью абстрагировать идею, вам нужно как-то выразить фразу "это — тестовая функция" соответствующим паттерном. Другими словами, вам нужен макрос.

Так как паттерн, который вы пытаетесь написать, представляет собой вызов `DEFUN` — и ещё немного кода — вам нужен макрос, раскрывающийся в вызов `DEFUN`. Вы будете использовать этот макрос вместо `DEFUN` для определения тестовых функций, так что имеет смысл назвать его `deftest`.

```
(defmacro deftest (name parameters &body body)
  `(defun ,name ,parameters
    (let ((*test-name* ',name))
      ,@body)))
```

Используя этот макрос, вы можете переписать `test-+` следующим образом:

```
(deftest test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

Иерархия тестов

Теперь, когда у вас есть полноценные тестовые функции, может возникнуть вопрос — должна ли функция `test-arithmetic` также быть тестовой? Казалось бы — если вы определите её с помощью `deftest`, то её связывание с `*test-name*` скроет связывания `test-+` и `test-*` — и это отразится на выводе результатов тестов.

Но представьте, что у вас есть тысяча (или даже больше) тестов, которые нужно как-то упорядочить. На самом нижнем уровне находятся такие функции как `test-+` и `test-*`, прямо выполняющих проверку. При наличии тысяч тестов их потребуется каким-либо образом упорядочить. Такие функции как `test-arithmetic` могут группировать схожие тестовые функции в наборы тестов. Допустим, что некоторые низкоуровневые тестовые функции могут использоваться в разных наборах тестов. Тогда вполне возможна такая ситуация, что тест будет пройден в одном контексте и провалится в другом. Если это случится, вам наверняка захочется узнать несколько больше, чем просто имя провалившегося теста.

Если вы определите `test-arithmetic` посредством `deftest`, сделав небольшие изменения при связывании с `*test-name*`, то сможете получить отчёты с более подробным описанием контекста выполнившегося теста:

```
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
```

Поскольку процесс определения тестовых функций описан отдельным паттерном, изменить отчёт можно и не меняя код самих тестовых функций. Сделать так, чтобы `*test-name*` хранил список имён тестовых функций вместо имени последней вызванной функции, очень просто. Вам нужно всего лишь изменить связывание:

```
(let ((*test-name* ',name))
```

на такое:

```
(let ((*test-name* (append *test-name* (list ',name)))))
```

Так как `APPEND` возвращает новый список, составленный из его аргументов, это версия будет связывать `*test-name*` со списком, содержащим старое значение `*test-name*`, с новым именем, добавленным в конец списка. После выхода из функции, старое значение `*test-name*` восстанавливается.

Теперь вы можете переопределить `test-arithmetic` используя `deftest` вместо `DEFUN`.

```
(deftest test-arithmetic ()  
  (combine-results  
    (test-+)  
    (test-*)))
```

В результате вы получите именно то, что хотели:

```
CL-USER> (test-arithmetic)  
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)  
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)  
pass ... (TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)  
pass ... (TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)  
pass ... (TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)  
T
```

С ростом количества тестов, вы можете добавлять новые уровни — и пока они будут определяться через `deftest`, вывод результата будет корректен. Так, если вы определите таким образом `test-math`:

```
(deftest test-math ()  
  (test-arithmetic))
```

то получите вот такой результат:

```
CL-USER> (test-math)  
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)  
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)  
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)  
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)  
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)  
T
```

Подведение итогов

Вы могли бы продолжить работу над этим каркасом, добавляя новые возможности — но как каркас для написания тестов без особого напряжения и с возможностью использовать REPL, это очень неплохое начало. Ниже код приведён полностью, все 26 строк:

```

(defvar *test-name* nil)
(defmacro deftest (name parameters &body body)
  "Define a test function. Within a test function we can call
  other test functions or use 'check' to run individual test
  cases."
  `(defun ,name ,parameters
    (let ((*test-name* (append *test-name* (list ',name))))
      ,@body)))
(defmacro check (&body forms)
  "Run each expression in 'forms' as a test case."
  `(combine-results
    ,@(loop for f in forms collect `(report-result ,f ',f))))
(defmacro combine-results (&body forms)
  "Combine the results (as booleans) of evaluating 'forms' in order."
  (with-gensyms (result)
    `(let ((,result t))
      ,@(loop for f in forms collect `(unless ,f (setf ,result nil)))
      ,result)))
(defun report-result (result form)
  "Report the results of a single test case. Called by 'check'."
  (format t "~:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
  result)

```

Этот пример прекрасно иллюстрирует обычный ход программирования на языке Lisp, так что давайте рассмотрим процесс его написания ещё раз.

Вы начали с постановки задачи — вычислить совокупность булевых выражений и узнать — все ли они возвращают true. Простое AND работало и синтаксически было абсолютно верно — но вывод результатов оставлял желать лучшего. Тогда вы написали немного действительно глуповатого кода, битком набитого повторениями и способствующими ошибкам выражениями, чтобы всё работало так, как вы хотели.

Естественно, что вы решили попробовать привести вторую версию программы к более ясному и красивому виду. Вы начали со стандартного приёма — выделения части кода в отдельную функцию — `report-result`. Увы, но использование `report-result` утомительно и чревато ошибками — тестовое выражение приходится писать дважды. Тогда вы написали макрос `check` для автоматически корректного вызова `report-result`.

В процессе написания макроса `check`, вы добавили возможность оборачивать несколько вызовов `report-result` в один вызов `check`, сделав новую версию `test+` столь же краткой, как и первоначальную с `AND`.

Следующей задачей было исправить `check` таким образом, чтобы генерируемый этим макросом код возвращал `t` или `nil` в зависимости от того, все ли тесты прошли удачно. Прежде чем переименовывать `check`, вы предположили, что у вас есть `FIXME` non-short-circuit `AND` конструкция. В этом случае правка `check` — тривиальное дело. Вы обнаружили, что хотя такой конструкции и нет, написать её самим совсем не трудно. После написания `combine-results` исправить `check` было элементарным делом.

Затем, всё, что оставалось, это сделать более удобным отчёт. Начав с исправления тестовых функций, вы представили их как особый вид функций — и в результате написали макрос `deftest`, выделив паттерн, отличающий тестовые функции от всех прочих.

Наконец, с помощью макроса `deftest`, разделившего определение тестовой функции от лежащей в её основе структуры, вы получили возможность улучшить вывод результатов не меняя при этом сами тестовые функции.

Теперь, имея представления об основах — функциях, переменных и макросах; получив немного опыта по их практическому применению; вы готовы начать изучение богатой стандартной библиотеки функций и типов данных языка Common Lisp.

10. Числа, знаки и строки

В то время как функции, переменные, макросы и 25 специальных операторов составляют основные блоки самого языка, строительными блоками ваших программ будут структуры данных, которые вы будете использовать. Как заметил Фред Брукс (Fred Brooks) в своей книге "Мифический человек-месяц (The Mythical Man-Month)", "представление – это сущность программирования."

Common Lisp предоставляет поддержку для основных типов, обычно существующих в современных языках программирования: чисел (целых, с плавающей запятой и комплексных чисел), знаков (characters), строк, массивов (включая многомерные массивы), списков, хеш-таблиц (hash tables), потоков ввода-вывода, и переносимое представление имен файлов. В Lisp функции также являются типом данных – они могут быть сохранены в переменных, переданы как аргументы, использованы в качестве возвращаемых значений, а также созданы во время выполнения.

Эти встроенные типы являются только началом. Они определены в стандарте языка, так что программисты могут рассчитывать на то, что они будут доступны, а также они могут быть реализованы более эффективно за счет более сильной интеграции с конкретной реализацией. Но, как вы увидите в следующих главах, Common Lisp также предоставляет вам возможности для определения новых типов данных, определения операций для них, и интеграции их со встроенными типами.

Однако сейчас вы начнете изучать встроенные типы данных. Поскольку Lisp является языком высокого уровня, детальная информация о реализации различных типов данных хорошо скрыта. С вашей точки зрения, как пользователя языка, встроенные типы данных определяются функциями, которые работают с ними. Так что, для изучения типа данных, вы просто должны изучить функции, которые используют эти типы. В дополнение к этому, большинство встроенных типов имеет специальный синтаксис, который понимает процедура чтения Lisp и который использует процедура печати Lisp. Поэтому, например, вы можете записывать строки как "foo"; числа как 123, 1/23, и 1.23; а списки как (a b c). Я буду описывать синтаксис для различных видов объектов тогда, когда я буду описывать функции для работы с этими типами данных.

В этой главе я опишу встроенные "скалярные" типы данных: числа, знаки и строки. С технической точки зрения, строки не являются настоящими скалярами: строка – последовательность букв и вы можете получить доступ к отдельным буквам и работать со строками с помощью функций, работающих с последовательностями. Но я опишу строки в этой главе, поскольку большая часть строко-ориентированных функций работает с ними как с едиными объектами, а также поскольку имеется тесная связь между некоторыми строковыми функциями и их аналогами, работающими с знаками.

Числа

Математика, как сказала Барби – очень тяжела. Common Lisp не сделает математику более легкой, но он делает работу с ней более простой, чем многие другие языки программирования. В этом нет ничего удивительно, учитывая математическое наследство. Lisp был спроектирован математиками как средство для изучения математических функций. Одним из основных продуктов в составе проекта MAC университета MIT была система символьной алгебры Macsyma, написанная на Maclisp, одним из непосредственных предков Common Lisp. Кроме того, Lisp использовался как язык для обучения в таких заведениях как MIT, даже когда профессора компьютерных объясняли, что $10/4 = 5/2$, что привело к поддержке Lisp'ом целочисленных дробей (exact ratios). И много раз Lisp состязался с FORTRAN в области

высокопроизводительных вычислений.

Одной из причин, почему Lisp является отличным языком для работы с математикой, является то, что поведение его чисел больше всего похоже на математическое, нежели та, жалкая пародия, которую легко реализовать на уровне оборудования. Например, целые числа в Common Lisp могут быть произвольной величины, а не ограничены размером машинного слова. И деление двух целых чисел приводит к получению целочисленной дроби, а не усеченному значению. А поскольку дроби, представляются как пары произвольных чисел, они могут представлять числа с произвольной точностью.

С другой стороны, для высокопроизводительных вычислений, вы можете захотеть пожертвовать точностью в погоне за скоростью, которая может быть достигнута за счет использования операций с плавающей точкой, поддерживаемых оборудованием. Так что Common Lisp также предлагает несколько типов чисел с плавающей точкой, которые преобразуются реализацией в соответствующие аппаратные представления, поддерживаемые конкретной платформой. Числа с плавающей точкой, также используются для представления результатов вычислений, чье математическое значение может быть иррациональным числом.

Кроме того, Common Lisp имеет поддержку комплексных чисел – чисел, которые получаются в результате таких операций, как вычисление квадратного корня и логарифмов отрицательных чисел. Стандарт Common Lisp имеет даже больше возможностей и позволяет указывать FIXME principal values and branch cuts для иррациональных и трансцендентальных функций для комплексных чисел.

Запись чисел

Вы можете записывать числа разными способами; вы видели это на примерах в главе 4. Однако, очень важно помнить о разнице между процедурой чтения Lisp и процедурой вычисления – процедура чтения отвечает за преобразование текста в объекты Lisp, а процедура вычисления, работает только с этими объектами. Для конкретного числа с конкретным типом, может быть множество способов текстовой записи, все из которых преобразуются процедурой чтения в один и тот же объект. Например, вы можете записать целое число 10 как 10, 20/2, #xA, или еще дюжиной способов, но процедура чтения преобразует все эти числа в один и тот же объект. Когда числа выводятся на печать, например в FIXME REPL – они печатаются в канонической текстовой форме, которая может отличаться от той формы, которая использовалась для задания числа. Например:

```
CL-USER> 10
10
CL-USER> 20/2
10
CL-USER> #xA
10
```

Целые числа записываются в следующем виде: необязательный знак числа (+ или -), за которым следует одна или несколько цифр. Дроби записываются как необязательный знак числа, за ним последовательность цифр, представляющих числитель, знак косая черта (/), и другая последовательность цифр, представляющая знаменатель. Все рациональные числа "канонизируются" во время чтения – вот поэтому 10 и 20/2 представляют одно и то же число, также как и 3/4 и 6/8. Рациональные числа печатаются в "упрощенной" форме – целые числа печатаются с использованием синтаксиса для целых чисел, а дроби печатаются с числителем, и знаменателем, сокращенным до минимальных значений.

Также возможна запись числа с основанием, отличным от 10. Если число предваряется #B или

#b, то число считывается как двоичное, в котором разрешены только цифры 0 и 1. Строки #o или #o используются для восьмеричных чисел (допустимые цифры – 0–7), а #X или #x используются для шестнадцатеричных (допустимые цифры 0–F или 0–f). Вы можете записывать числа с использованием других оснований (от 2 до 36) с указанием префикса #nR, где n определяет основание (всегда записывается в десятичном виде). В качестве дополнительных "цифр" используются буквы A–Z или a–z. Заметьте, что знак основания применяется ко всему числу – невозможно написать дробь, где числитель использует одно основание исчисления, а знаменатель – другое. Вы также можете записывать целые значения (но не дроби!) в виде десятичных цифр, завершаемых десятичной точкой. Ниже приводятся некоторые примеры рациональных чисел, с их каноническим, десятичным представлением:

```
123 ==> 123
+123 ==> 123
-123 ==> -123
123. ==> 123
2/3 ==> 2/3
-2/3 ==> -2/3
4/6 ==> 2/3
6/3 ==> 2
#b10101 ==> 21
#b1010/1011 ==> 10/11
#o777 ==> 511
#xDADA ==> 56026
#36rABCDEFGHIJKLMNOPQRSTUVWXYZ ==> 8337503854730415241050377135811259267835
```

Числа с плавающей точкой вы также можете записывать разными способами. В отличие от рациональных чисел, синтаксис, используемый для записи может влиять на тип считываемого числа. Common Lisp имеет четыре подтипа для чисел с плавающей точкой: FIXME short, single, double и long. Каждый подтип может использовать разное количество бит для представления, что означает, что каждый тип может представлять значения разных диапазонов и с разной точностью. Большее количество бит дает более широкий диапазон и большую точность.

Основным форматом для чисел с плавающей точкой является следующий: необязательный знак числа, за которым следует непустая последовательность десятичных цифр, возможно с указанием десятичной точки. За этой последовательностью может следовать маркер экспоненты для "компьютеризированной научной нотации." Маркер экспоненты состоит из единственной буквы, за которой следует необязательный знак числа и последовательность цифр, которая интерпретируется как степень десяти на которую должно быть умножено число, указанное до маркера экспоненты. Буква в маркере экспоненты играет двойную роль: она обозначает начало маркера, и показывает что для числа должно использоваться представление с плавающей точкой. Маркеры экспоненты в виде букв s, f, d, l (и их заглавные эквиваленты) обозначают использование short, single, double и long подтипов. Буква e показывает, что должно использоваться представление по умолчанию (первоначально подтип single).

Числа без маркера экспоненты считываются с использованием представления по умолчанию и должны содержать десятичную точку, и как минимум одну цифру после нее, чтобы быть отличимы от целых чисел. Цифры в числах с плавающей запятой всегда рассматриваются как имеющие основание исчисления 10 – синтаксис с префиксами #B, #X, #O, и #R могут использоваться только с рациональными числами. Ниже приведены примеры чисел с плавающей точкой, и их соответствующее каноническое представление:

```
1.0 ==> 1.0
1e0 ==> 1.0
1d0 ==> 1.0d0
123.0 ==> 123.0
123e0 ==> 123.0
0.123 ==> 0.123
.123 ==> 0.123
123e-3 ==> 0.123
123E-3 ==> 0.123
0.123e20 ==> 1.23e+19
123d23 ==> 1.23d+25
```

В заключение, комплексные числа имеют отличающийся синтаксис, а именно: префикс #C или #c, за которым следует список из двух чисел, представляющих реальную и мнимую часть комплексного числа. В действительности существует пять видов комплексных чисел, поскольку реальная и мнимая части могут либо быть рациональными числами или числами с плавающей точкой.

Но вы можете записывать их как хотите – если в комплексном числе одна часть записана как рациональное число, а вторая – как число с плавающей точкой, то рациональное число преобразуется в число с плавающей точкой. Аналогично, если обе части являются числами с плавающей точкой, но с разным представлением, то число с типом с меньшей точностью, будет преобразовано в число с типом с большей точностью.

Однако не существует комплексных чисел с рациональной реальной частью и мнимой частью равной нулю, поскольку такие числа с математической точки зрения являются рациональными и они будут представлены соответствующими рациональными значениями. То же самое должно быть сделано и для чисел с компонентами, состоящими из чисел с плавающей точкой, но для этих комплексных типов, число с нулевой мнимой частью всегда является объектом, отличающимся от числа с плавающей точкой, представляющего реальную часть. Вот некоторые примеры чисел, записанных с использованием синтаксиса для комплексных чисел:

```
#c(2 1) ==> #c(2 1)
#c(2/3 3/4) ==> #c(2/3 3/4)
#c(2 1.0) ==> #c(2.0 1.0)
#c(2.0 1.0d0) ==> #c(2.0d0 1.0d0)
#c(1/2 1.0) ==> #c(0.5 1.0)
#c(3 0) ==> 3
#c(3.0 0.0) ==> #c(3.0 0.0)
#c(1/2 0) ==> 1/2
#c(-6/3 0) ==> -2
```

Базовые математические операции

Базовые математические операции – сложение, вычитание, умножение и деление, реализуются всеми типами чисел Lisp с помощью функций `functions` `+`, `-`, `*` и `/`. Вызов любой из этих функций с количеством аргументов больше двух, эквивалентен вызову той же самой функции для первых двух аргументов, и последующим вызовом функции для результата и оставшихся аргументов. Например, `(+ 1 2 3)` эквивалентно `(+ (+ 1 2) 3)`. При вызове с одним аргументом, `+` и `*` возвращают само значение; `-` возвращает отрицание значения, а `/` возвращает значение обратное аргументу.

```
(+ 1 2) ==> 3
(+ 1 2 3) ==> 6
(+ 10.0 3.0) ==> 13.0
(+ #c(1 2) #c(3 4)) ==> #c(4 6)
(- 5 4) ==> 1
(- 2) ==> -2
(- 10 3 5) ==> 2
(* 2 3) ==> 6
(* 2 3 4) ==> 24
(/ 10 5) ==> 2
(/ 10 5 2) ==> 1
(/ 2 3) ==> 2/3
(/ 4) ==> 1/4
```

Если все аргументы имеют один тип (рациональный, с плавающей точкой или комплексный), то тип результата будет тем же, за исключением случая, когда операция с комплексными приводит к получению результата с нулевой мнимой частью, и тогда результат будет иметь рациональный тип. Однако комплексные числа и числа с плавающей точкой являются "заразными" – если все аргументы являются рациональными числами, а одно или больше чисел являются числами с плавающей точкой, то все остальные аргументы преобразуются в соответствующие числа с плавающей точкой с типом, соответствующим "наибольшему" типу аргументов с плавающей точкой. Числа с плавающей точкой с "меньшим" представлением, также преобразуются в тип с "большим" представлением. То же самое происходит и с комплексными числами – все обычные числа преобразуются в комплексные числа.

```
(+ 1 2.0) ==> 3.0
(/ 2 3.0) ==> 0.6666667
(+ #c(1 2) 3) ==> #c(4 2)
(+ #c(1 2) 3/2) ==> #c(5/2 2)
(+ #c(1 1) #c(2 -1)) ==> 3
```

Поскольку / при делении не отбрасывает остаток, то Common Lisp предлагает четыре вида функций для усечения и округления для вещественных чисел (рациональных или с плавающей точкой) в целые числа: FLOOR усекает число в сторону отрицательной бесконечности, возвращая наибольшее целое, меньшее, или равное аргументу. CEILING усекает число в сторону положительной бесконечности, возвращая наименьшее целое большее или равное аргументу. TRUNCATE усекает число в сторону нуля, ведя себя как FLOOR для положительных аргументов, и как CEILING для отрицательных. А ROUND округляет число до ближайшего целого. Если аргумент находится ровно между двумя целыми числами, то округление происходит в сторону ближайшего четного числа.

К этой же теме можно отнести и две функции – MOD и REM, которые возвращают модуль и остаток деления с усечением чисел. Эти две функции соотносятся с FLOOR и TRUNCATE следующими отношениями:

```
(+ (* (floor (/ x y)) y) (mod x y)) == x
(+ (* (truncate (/ x y)) y) (rem x y)) == x
```

Таким образом, для положительных частных они будут эквивалентны, но для отрицательных, они будут давать разные результаты.)

Функции 1+ и 1– могут использоваться как сокращения для добавления и вычитания единицы из числа. Заметьте, что они отличаются от макросов INCF и DECF. 1+ и 1– являются функциями, которые возвращают значения, а INCF и DECF изменяют заданное значение. Следующие

примеры показывают соответствие между INCF/DECF, 1+/1- и +/-:

```
(incf x) === (setf x (1+ x)) === (setf x (+ x 1))  
(decf x) === (setf x (1- x)) === (setf x (- x 1))  
(incf x 10) === (setf x (+ x 10))  
(decf x 10) === (setf x (- x 10))
```

Сравнение чисел

Функция `=` является предикатом для проверки равенства чисел. Она сравнивает значения математически, игнорируя разницу в типах. Таким образом, `=` будет считать равными математически равные значения разных типов, в то время как общий предикат равенства `EQL` будет считать их не равными, из-за разницы типов. (При этом общий предикат равенства `EQUALP` использует `=` для сравнения чисел.) Если эта функция была вызвана с более чем одним аргументом, то она вернет истинное значение только если они все имеют одно и тоже значение. Так что:

```
(= 1 1) ==> T  
(= 10 20/2) ==> T  
(= 1 1.0 #c(1.0 0.0) #c(1 0)) ==> T
```

В противоположность этому, функция `/=`, вернет истинное значение, если все ее аргументы имеют разное значение.

```
(/= 1 1) ==> NIL  
(/= 1 2) ==> T  
(/= 1 2 3) ==> T  
(/= 1 2 3 1) ==> NIL  
(/= 1 2 3 1.0) ==> NIL
```

Функции `<`, `>`, `<=` и `>=` сравнивают порядок рациональных чисел и чисел с плавающей точкой (другими словами, вещественных чисел). Подобно `=` и `/=`, эти функции могут быть вызваны с более чем двумя аргументами, и в этом случае каждый аргумент сравнивается с аргументом, находящимся справа от него.

```
(< 2 3) ==> T  
(> 2 3) ==> NIL  
(> 3 2) ==> T  
< 2 3 4) ==> T  
< 2 3 3) ==> NIL  
<= 2 3 3) ==> T  
<= 2 3 3 4) ==> T  
<= 2 3 4 3) ==> NIL
```

Для выбора наименьшего или наибольшего числа из нескольких, вы можете использовать функцию `MIN` или `MAX`, которые принимают любое число вещественных аргументов, и возвращают минимальное или максимальное значение.

```
(max 10 11) ==> 11  
(min -12 -10) ==> -12  
(max -1 2 -3) ==> 2
```

Другими полезными функциями являются `ZEROP`, `MINUSP` и `PLUSP`, которые проверяют, является ли одиночное вещественное число равным, меньшим или большим чем ноль. Два

других предиката, `EVENP` и `ODDP`, проверяют является ли число четным или нечетным. Суффикс `P` в именах этих функций соответствует стандарту наименования предикатных функций, которые проверяют некоторое условие и возвращают логическое значение.

Высшая математика

Функции которые вы уже увидели являются началом списка встроенных математических функций. Lisp также поддерживает логарифмы: `LOG`; экспоненты: `EXP` и `EXPT`; основные тригонометрические функции: `SIN`, `COS` и `TAN`; их противоположности: `ASIN`, `ACOS` и `ATAN`; гиперболические функции: `SINH`, `COSH` и `TANH`; и их противоположности: `ASINH`, `ACOSH` и `ATANH`. Также имеются функции для доступа к отдельным битам целых чисел и для извлечения частей комплексных чисел. Для получения полного списка функций смотрите в любой справочник по Common Lisp.

Знаки (Characters)

В Common Lisp знаки (characters) являются отдельным типом объектов, а не числами. Это то, что и должно быть – знаки не являются числами, и языки, которые рассматривают их как числа, столкнулись с проблемами, когда изменится кодировка знаков, например, с 8-битного ASCII на 21-битный Unicode. Поскольку стандарт Common Lisp не требует конкретного представления для знаков, некоторые из существующих реализаций Lisp используют Unicode как "родную" кодировку знаков, несмотря на то, что Unicode только задумывался в то время, когда проводилась стандартизация Common Lisp.

Синтаксис чтения для объектов-знаков очень простой: префикс `#\` за которым следует нужный знак. Так что, `#\x` обозначает знак `x`. После `#\` может использоваться любой знак, включая специальные знаки, такие как `"`, `(` и пробел. Однако, поскольку запись пробелов и аналогичных знаков не особенно хорошо выглядит (для человека), то для некоторых знаков существует альтернативный синтаксис, состоящий из `#\`, за которым следует название знака. Список поддерживаемых имен зависит от набора знаков и реализации Lisp, но все реализации поддерживают имена `Space` и `Newline`. Так что вы должны писать `#\Space` вместо `#\` , хотя последний вариант и допустим с технической точки зрения. Другими полу-стандартными именами (которые реализации должны использовать, если набор знаков содержит соответствующие знаки) являются `Tab`, `Page`, `Rubout`, `Linefeed`, `Return` и `Backspace`.

Сравнение знаков

Основной операцией которую вы можете выполнить со знаками, отличной от помещения их в строки (что будет описано дальше в этой главе), является их сравнение с другими знаками. Поскольку знаки не являются числами, то вы не можете использовать такие функции как `<` и `>`. Вместо этого два набора функций обеспечивают операции над знаками, аналогичные операциям сравнения чисел; одни функции учитывают регистр знаков, а другие – нет.

Чувствительным к регистру аналогом численной функции `=` является функция `CHAR=`. Также как и `=`, `CHAR=` может получать любое количество аргументов, и возвращает истину, если они все являются одним и тем же знаком. Нечувствительная к регистру функция называется `CHAR=` `EQUAL`.

Остальные функции сравнения знаков, следуют той же схеме наименования: чувствительные к регистру функции именуются путем добавления `CHAR` к имени аналогичной функции для сравнения чисел; а нечувствительные к регистру функции добавляют к `CHAR` название операции, отделенное знаком минус. Однако заметьте, что `<=` и `>=` "именуются" логическими эквивалентами операций `NOT-GREATERP` и `NOT-LESSP`, а не более подробными названиями вида

LESSP-OR-EQUALP и GREATERP-OR-EQUALP. Также как и их численные аналоги, все эти функции могут принимать один или больше аргументов. Таблица 10-1 показывает отношение между функциями сравнения для чисел и знаков.

Таблица 10-1. Функции сравнения знаков

Numeric Analog	Case-Sensitive	Case-Insensitive
=	CHAR=	CHAR-EQUAL
/=	CHAR/=	CHAR-NOT-EQUAL
<	CHAR<	CHAR-LESSP
>	CHAR>	CHAR-GREATERP
<=	CHAR<=	CHAR-NOT-GREATERP
>=	CHAR>=	CHAR-NOT-LESSP

Другие функции работающие со знаками, среди прочего предоставляют операции по проверке является ли знак буквой или цифрой, проверке регистра знака, получению соответствующего знака в другом регистре, а также по преобразованию между численным значением и соответствующим знаком, и наоборот. Для получения полной информации об этих функциях, смотрите справочник по Common Lisp.

Строки

Как упоминалось ранее, строки в Common Lisp на самом деле являются составными типами данных, а именно – одномерными массивами букв. Соответственно, я опишу много вещей, которые можно сделать со строками, когда в следующей главе я буду описывать функции для работы с последовательностями, подтипом которых и являются строки. Но строки также имеют свой собственный синтаксис и библиотеку функций для выполнения операций, специфичных для строк. Я буду обсуждать данные особенности в этой главе, а все остальное будет обсуждаться в главе 11.

Как вы уже видели, строки записываются заключенными в двойные кавычки. Вы можете включить в строку любой знак, поддерживаемую набором знаков, за исключением двойной кавычки (") и обратного слэша (\). Вы можете включить и эти два знака, если вы замаскируете их с помощью обратного слэша. В действительности, знак обратный слэш всегда маскирует следующий знак, независимо от того, чем он является, хотя нет необходимости использовать его для чего-то отличного от " и самого себя. Таблица Table 10-2 показывает как различные записи строк будут считываться процедурой чтения Lisp.

Таблица 10-2. Представление строк

Literal	Contents	Comment
"foobar"	foobar	Обычная строка.
"foo\"bar"	foo"bar	Обратный слэш маскирует кавычку.
"foo\\bar"	foo\bar	Первый обратный слэш маскирует второй.
"\"foobar\""	"foobar"	Знаки обратные слэш маскируют кавычки.
"foo\bar"	foobar	Обратный слэш "маскирует" знак b

Заметьте, что FIXME REPL обычно печатает строки в форме, пригодной для считывания, добавляя охватывающие знаки кавычек, и нужные маскирующие знаки. Так что если вы хотите видеть содержимое строки, то вы должны использовать какую-либо функцию, такую как

FORMAT, спроектированную для печати данных в форме, удобной для человека. Например, вот что вы увидите, если напечатаете строку, содержащую знак кавычки в REPL:

```
CL-USER> "foo\"bar"  
"foo\"bar"
```

С другой стороны, FORMAT выведет содержимое строки:

```
CL-USER> (format t "foo\"bar")  
foo"bar  
NIL
```

Сравнение строк

Вы можете сравнивать строки используя набор функций, который использует то же самое соглашение о наименовании, что и функции для сравнения знаков, с той разницей, что они используют в качестве префикса слово STRING вместо CHAR (смотрите таблицу 10-3).

Table 10-3. Функции сравнения строк

Numeric Analog	Case-Sensitive	Case-Insensitive
=	STRING=	STRING-EQUAL
/=	STRING/=	STRING-NOT-EQUAL
<	STRING<	STRING-LESSP
>	STRING>	STRING-GREATERP
<=	STRING<=	STRING-NOT-GREATERP
>=	STRING>=	STRING-NOT-LESSP

Однако в отличие от сравнения знаков и чисел, функции сравнения строк могут сравнивать только две строки. Это происходит потому, что эти функции также получают именованные аргументы, которые позволяют вам ограничить области сравнения одной или обеих строк. Аргументы: :start1, :end1, :start2 и :end2 указывают начальный (включительно) и конечный (исключительно FIXME exclusive) индексы подстрок в первой и второй строке. Таким образом, следующий код:

```
(string= "foobarbaz" "quuxbarfoo" :start1 3 :end1 6 :start2 4 :end2 7)
```

сравнивает подстроки "bar" в двух аргументах, и возвращает истинное значение. Аргументы :end1 и :end2 могут иметь значение NIL (или именованные аргументы могут быть полностью убраны) для указания, что соответствующие подстроки берутся до конца строк.

Функции сравнения, которые возвращают истинное значение, когда их аргументы отличаются друг от друга – это все функции за исключением STRING= и STRING-EQUAL, возвращают индекс позиции в первой строке, где строки начинают отличаться.

```
(string/= "lisp" "lissome") ==> 3
```

Если первая строка является префиксом второй строки, то возвращаемое значение будет равно длине первой строки, что на единицу больше чем самый большой допустимый индекс в этой строке.

```
(string< "lisp" "lisper") ==> 4
```

При сравнении подстрок, результатом все равно будет являться индекс в полной строке. Например, следующий код сравнивает подстроки "bar" и "baz", но возвращает 5, поскольку это индекс знака `r` в первой строке:

```
(string< "foobar" "abaz" :start1 3 :start2 1) ==> 5 ; N.B. not 2
```

Другие строковые функции позволяют вам преобразовывать регистр знаков в строке, а также удалять (`trim`) знаки с одного, или обоих концов строки. И, как я уже отмечал перед этим, поскольку строки на самом деле являются последовательностями, все функции работы с последовательностями, которые я буду описывать в следующей главе, могут быть использованы для работы со строками. Например, вы можете узнать длину строки с помощью функции `LENGTH` и можете получить отдельный знак строки с помощью функции доступа к элементу последовательности – `ELT`, или функции доступа к элементу массива – `AREF`. Или вы можете использовать функцию доступа к элементу строки – `CHAR`. Но эти функции являются предметом обсуждения следующей главы, так что перейдем к ней.

Коллекции

Подобно большинству языков программирования, Common Lisp предоставляет стандартные типы данных, которые собирают множество значений в один объект. Каждый язык решает проблему коллекций немного по-разному, но базовые типы коллекций обычно сводятся к наличию массивов с целочисленными индексами, и таблицам, которые могут использоваться для отображения произвольных (более или менее) ключей в значения. Первые часто называются массивами, списками или кортежами (записями, FIXME tuple), а вторые – хэш-таблицами, ассоциативными массивами, картами и словарями.

Конечно Lisp известен своими списками, и большинство книжек по Lisp следуют FIXME ontogeny-recapitulates-phylogeny принципу знакомства с языком, и начинают обсуждение коллекций Lisp со списков.

Здесь следует перевести "Онтогенез повторяет филогенез" Это правило было сформулировано немецким зоологом Эрнстом Геккелем. Сказав это, он имел в виду, что развитие зародыша (онтогенез) повторяет эволюцию видов (филогенез).

Автор по отношению к обучению Lisp очевидно имел в виду то, что так как списки были первой основной структурой данных Lisp и лишь затем стали добавляться другие, то большинство книг описывают структуры данных Lisp в такой же последовательности, то есть сначала списки, а потом другие структуры данных. (dream.designer)

Однако, этот подход часто ведет к тому, что читатели считают, что единственным типом коллекций в Lisp является список. Ухудшает положение еще и то, что списки Lisp являются такими гибкими структурами, что возможно их использование для таких вещей, для которых в других языках используются массивы и хэш-таблицы. Но было бы ошибкой слишком сильно сосредотачиваться на списках; хотя списки и являются ключевой структурой данных для представления кода на Lisp в виде данных Lisp, во многих ситуациях использование других структур данных может быть более предпочтительным.

Чтобы списки не заслоняли картину, в этой главе я сосредоточусь на остальных типах коллекций, используемых в Common Lisp: векторах и хэш-таблицах. Однако вектора и списки имеют достаточно много общих признаков, так что Common Lisp рассматривает их как подтипы более общей абстракции – последовательности. Таким образом, вы можете использовать многие функции, описанные в этой главе, как для векторов, так и для списков.

Вектора

Вектора Common Lisp являются базовой коллекцией с доступом по целочисленному индексу, и имеет две разновидности. Вектора с фиксированным размером похожи на массивы в языках, подобных Java: простая надстройка над областью памяти, которая хранит элементы вектора. С другой стороны, вектора с изменяемым размером, более похожи на вектора в Perl или Ruby, списки в Python, или на класс ArrayList в Java: они прячут детали реализации хранилища данных, позволяя векторам менять размер по мере добавления или удаления элементов.

Вы можете создать вектор фиксированной длины, содержащий конкретные значения, с помощью функции VECTOR, которая принимает любое количество аргументов, и возвращает заново созданный вектор фиксированного размера, содержащий переданные значения.

```
(vector) ==> #()
(vector 1) ==> #(1)
(vector 1 2) ==> #(1 2)
```

Синтаксис #(...) является способом записи векторов, используемый процедурами записи и

чтения Lisp. Это позволяет вам сохранять и восстанавливать вектора путем их вывода на печать и последующего считывания. Вы можете использовать `#(...)` для записи векторов в вашем коде, но поскольку эффект изменения таких объектов не определен, то вы всегда должны использовать `VECTOR`, или более общую функцию `MAKE-ARRAY` для создания векторов, которые вы планируете изменять.

`MAKE-ARRAY` является более общей функцией, чем `VECTOR`, поскольку вы можете использовать ее как для создания массивов любой размерности, так и для создания векторов фиксированной и изменяемой длины. Единственным обязательным аргументом `MAKE-ARRAY` является список, содержащий размерности массива. Поскольку, вектор – одномерный массив, то список будет содержать только одно число – размер вектора. Для удобства, `MAKE-ARRAY` может также принимать простое число вместо списка из одного элемента. Без предоставления дополнительных аргументов, `MAKE-ARRAY` создаст вектор с неинициализированными элементами, которые должны быть заданы до осуществления доступа к ним. Для создания вектора, с присвоением всем элементам определенного значения, вы можете использовать аргумент `:initial-element`. Таким образом, чтобы создать вектор из пяти элементов, которые все равны `NIL`, вы должны написать следующее:

```
(make-array 5 :initial-element nil) ==> #(NIL NIL NIL NIL NIL)
```

`MAKE-ARRAY` может также использоваться для создания векторов переменного размера. Вектор с изменяемым размером, является более сложным, чем вектор фиксированного размера; в добавление к отслеживанию памяти, используемой для хранения элементов и количества доступных ячеек, вектор с изменяемым размером также отслеживает число элементов, сохраненных в векторе. Это число хранится в указателе заполнения вектора (vector's fill pointer), так названного, поскольку это индекс следующей позиции, которая будет заполнена, когда вы добавите элемент в вектор.

Чтобы создать вектор с указателем заполнения, вы должны передать `MAKE-ARRAY` аргумент `:fill-pointer`. Например, следующий вызов `MAKE-ARRAY` создаст вектор с местом для пяти элементов; но он будет выглядеть пустым, поскольку указатель заполнения равен нулю:

```
(make-array 5 :fill-pointer 0) ==> #()
```

Для того, чтобы добавить элемент в конец вектора, вы можете использовать функцию `VECTOR-PUSH`. Она добавляет элемент в точку, указываемую указателем заполнения, и затем увеличивает его на единицу, возвращая индекс ячейки, куда был добавлен новый элемент. Функция `VECTOR-POP` возвращает последний добавленный элемент, уменьшая указатель заполнения на единицу.

```
(defparameter *x* (make-array 5 :fill-pointer 0))
(vector-push 'a *x*) ==> 0
*x* ==> #(A)
(vector-push 'b *x*) ==> 1
*x* ==> #(A B)
(vector-push 'c *x*) ==> 2
*x* ==> #(A B C)
(vector-pop *x*) ==> C
*x* ==> #(A B)
(vector-pop *x*) ==> B
*x* ==> #(A)
(vector-pop *x*) ==> A
*x* ==> #()
```

Однако, даже вектор с указателем заполнения, не является настоящим вектором с изменяемыми размерами. Вектор `*x*` может хранить максимум пять элементов. Для того, чтобы создать вектор с изменяемым размером, вам необходимо передать `MAKE-ARRAY` другой именованный аргумент: `:adjustable`.

```
(make-array 5 :fill-pointer 0 :adjustable t) ==> #()
```

Этот вызов приведет к созданию вектора, чей размер может изменяться по мере необходимости. Для добавления элементов в такой вектор, вам нужно использовать функцию `VECTOR-PUSH-EXTEND`, которая работает также как и `VECTOR-PUSH`, за тем исключением, что она автоматически увеличит массив, если вы пытаетесь добавить элемент в уже заполненный вектор – вектор, чей указатель заполнения равен размеру выделенной памяти.

Подтипы векторов

Все вектора, с которыми мы уже встречались, были векторами общего назначения, которые могут хранить объекты любого типа. Кроме того, возможно создание специализированных векторов, которые предназначены для хранения элементов определенного типа. Одной из причин использования специализированных векторов является то, что они могут использовать меньше места, и обеспечивать более быстрый доступ к своим элементам, по сравнению с векторами общего назначения. Однако, давайте сосредоточимся на некоторых специализированных векторах, которые сами являются важными типами данных.

С одним из них мы уже встречались: строки – это вектора, предназначенные для хранения знаков. Строки достаточно важны, чтобы они имели собственный синтаксис чтения/записи (двойные кавычки) и набор отдельных функций, которые мы обсуждали в предыдущей главе. Но, поскольку они также являются векторами, все функции работающие с векторами и которые мы будем обсуждать в следующих разделах, могут также использоваться для работы со строками. Эти функции дополняют библиотеку функций работы со строками новыми функциями для таких вещей, как поиск подстроки в строке, нахождение позиции знака в строке и т.п.

Строки, такие как `"foo"`, подобны векторам, записанным с использованием синтаксиса `#()` – их размер фиксирован, и они не должны изменяться. Однако вы можете использовать функцию `MAKE-ARRAY` для создания строк с изменяемым размером, просто добавляя еще один именованный аргумент – `:element-type`. Этот аргумент принимает описание типа. Я не буду тут описывать типы, которые вы можете использовать; сейчас достаточно знать, что вы можете создать строку, путем передачи символа `CHARACTER` в качестве аргумента `:element-type`. Заметьте, что вам необходимо экранировать символ, чтобы он не считался именем переменной. Например, чтобы создать пустую строку, с изменяемым размером, вы можете написать вот так:

```
(make-array 5 :fill-pointer 0 :adjustable t :element-type 'character) ""
```

Битовые вектора (специализированные вектора, чьи элементы могут иметь значение ноль или один) также отличаются от обычных векторов. Они также имеют специальный синтаксис чтения/записи, который выглядит вот так `#*00001111`, а также, достаточно большой набор функций (которые я не буду тут описывать) для выполнения битовых операций, таких как выполнение `"и"` для двух битовых массивов. Для создания такого вектора, вам нужно передать `:element-type` символ `BIT`.

Вектора как последовательности

Как уже упоминалось ранее, вектора и списки являются подтипами абстрактного типа "последовательность". Все функции, которые будут обсуждаться в следующих разделах,

работают с последовательностями; в дополнение к тому, что они применимы к векторам (и специализированным, и общего назначения), они также могут использоваться и со списками.

Две самые простые функции для работы с последовательностями являются `LENGTH`, которая возвращает длину последовательности, и `ELT`, которая позволяет осуществить доступ к отдельным элементам, используя целочисленный индекс. `LENGTH` получает последовательность в качестве единственного аргумента, и возвращает число элементов в этой последовательности. Для векторов с указателем заполнения, это число будет равно значению указателя. `ELT` (сокращение слова элемент), получает два аргумента – последовательность и числовой индекс между нулем (включительно) и длиной последовательности ($n-1$) и возвращает соответствующий элемент. `ELT` выдаст ошибку, если индекс находится за границами последовательности. Подобно `LENGTH`, `ELT` рассматривает вектор с указателем заполнения, как имеющий длину, указанную этим указателем.

```
(defparameter *x* (vector 1 2 3))
(length *x*) ==> 3
(elt *x* 0) ==> 1
(elt *x* 1) ==> 2
(elt *x* 2) ==> 3
(elt *x* 3) ==> error
```

`ELT` возвращает ячейку, для которой можно выполнить `SETF`, так что вы можете установить значение отдельного элемента, с помощью примерно такого кода:

```
(setf (elt *x* 0) 10)
*x* ==> #(10 2 3)
```

Функции для работы с элементами последовательностями

Хотя в теории, все операции над последовательностями могут быть сведены к комбинациям `LENGTH`, `ELT`, и `SETF` на результат `ELT`, но Common Lisp все равно предоставляет большую библиотеку функций для работы с последовательностями.

Одна группа функций позволит вам выполнить некоторые операции, такие как нахождение или удаление определенных элементов, без явного написания циклов. Краткая сводка этих функций приводится в таблице 11-1.

Table 11-1. Базовые функции для работы с последовательностями

Название	Обязательные аргументы	Возвращаемое значение
<code>COUNT</code>	Объект и последовательность	Число вхождений в последовательности
<code>FIND</code>	Объект и последовательность	Объект или <code>NIL</code>
<code>POSITION</code>	Объект и последовательность	Индекс ячейки в последовательности или <code>NIL</code>
<code>REMOVE</code>	Удаляемый объект и последовательность	Последовательность, из которой удалены указанные объекты
<code>SUBSTITUTE</code>	Новый объект, заменяемый объект и последовательность	Последовательность, в которой указанные объекты заменены на новые

Вот несколько простых примеров использования этих функций:

```
(count 1 #(1 2 1 2 3 1 2 3 4)) ==> 3
(remove 1 #(1 2 1 2 3 1 2 3 4)) ==> #(2 2 3 2 3 4)
(remove 1 '(1 2 1 2 3 1 2 3 4)) ==> (2 2 3 2 3 4)
(remove #\a "foobarbaz") ==> "foobrbz"
(substitute 10 1 #(1 2 1 2 3 1 2 3 4)) ==> #(10 2 10 2 3 10 2 3 4)
(substitute 10 1 '(1 2 1 2 3 1 2 3 4)) ==> (10 2 10 2 3 10 2 3 4)
(substitute #\x #\b "foobarbaz") ==> "fooxarxaz"
(find 1 #(1 2 1 2 3 1 2 3 4)) ==> 1
(find 10 #(1 2 1 2 3 1 2 3 4)) ==> NIL
(position 1 #(1 2 1 2 3 1 2 3 4)) ==> 0
```

Заметьте, что REMOVE и SUBSTITUTE всегда возвращают последовательность того-же типа, что и переданный аргумент.

Вы можете изменить поведение этих функций используя различные именованные аргументы. Например, по умолчанию, эти функции ищут в последовательности точно такой же объект, что и переданный в качестве аргумента. Вы можете изменить это поведение двумя способами: во первых, вы можете использовать именованный аргумент `:test` для указания функции, которая принимает два аргумента, и возвращает логическое значение. Если этот аргумент указан, то он будет использоваться для сравнения каждого элемента, вместо стандартной проверки на равенство с помощью `EQL`. Во вторых, используя именованный параметр `:key` вы можете передать функцию одного аргумента, которая будет вызвана для каждого элемента последовательности для извлечения значения, которое затем будет сравниваться с переданным объектом. Однако заметьте, что функции (например `FIND`), возвращающие элементы последовательности, все равно будут возвращать элементы, а не значения, извлеченные из этих элементов.

```
(count "foo" #("foo" "bar" "baz") :test #'string=) ==> 1
(find 'c #((a 10) (b 20) (c 30) (d 40)) :key #'first) ==> (C 30)
```

Для ограничения действия этих функций в рамках только определенных пределов, вы можете указать граничные индексы, используя именованные аргументы `:start` и `:end`. Передача `NIL` в качестве значения `:end` (или его полное отсутствие) равносильно указанию длины последовательности.

Если указывается неравный `NIL` аргумент `:from-end`, то элементы последовательности проверяются в обратном порядке. Простое указание `:from-end` может затронуть результаты `FIND` и `POSITION`. Например:

```
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first) ==> (A 10)
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first :from-end t) ==> (A 30)
```

Однако использование `:from-end` может влиять на работу REMOVE и SUBSTITUTE при использовании с другим именованным параметром — `:count`, которые используется для указания количества заменяемых или удаляемых элементов. Если вы указываете `:count` меньший, чем количество совпадающих элементов, то результат будет зависеть от того, с какого конца последовательности вы начинаете обработку:

```
(remove #\a "foobarbaz" :count 1) ==> "foobrbaz"
(remove #\a "foobarbaz" :count 1 :from-end t) ==> "foobarbz"
```

И хотя `:from-end` не может изменить результат функции COUNT, его использование может влиять на порядок элементов, передаваемых функциям, указанным в параметрах `:test` и `:key`,

которые возможно могут использовать FIXME побочные эффекты (side effects). Например:

```
CL-USER> (defparameter *v* #((a 10) (b 20) (a 30) (b 40)))
*v*
CL-USER> (defun verbose-first (x) (format t "Looking at ~s~%" x) (first x))
VERBOSE-FIRST
CL-USER> (count 'a *v* :key #'verbose-first)
Looking at (A 10)
Looking at (B 20)
Looking at (A 30)
Looking at (B 40)
2
CL-USER> (count 'a *v* :key #'verbose-first :from-end t)
Looking at (B 40)
Looking at (A 30)
Looking at (B 20)
Looking at (A 10)
2
```

В таблице 11-2 приведены описания всех стандартных аргументов.

Table 11-2. Стандартные именованные аргументы функций работы с последовательностями

Аргумент	Описание	Значение по умолчанию
:test	Функция двух аргументов, используемая для сравнения элементов (или значений, извлеченных функцией :key) с указанным объектом.	EQL
:key	Функция одного аргумента, используемая для извлечения значения из элемента последовательности. NIL указывает на использование самого элемента.	NIL
:start	Начальный индекс (включительно) обрабатываемой последовательности.	0
:end	Конечный индекс (n-1) обрабатываемой последовательности. NIL указывает на конец последовательности.	NIL
:from-end	Если имеет истинное значение, то последовательность будет обрабатываться в обратном порядке, от конца к началу.	NIL
:count	Число, указывающее количество удаляемых или заменяемых элементов, или NIL для всех элементов (только для REMOVE и SUBSTITUTE).	NIL

Аналогичные функции высшего порядка

Для каждой из функций, которая была описана выше, Common Lisp также предоставляет два набора функций высшего порядка, которые вместо аргумента, используемого для сравнения, получают функцию, которая вызывается для каждого элемента последовательности. Первый набор функций имеет те же имена, что и функции из базового набора, но с добавлением суффикса -IF. Эти функции подсчитывают, ищут, удаляют и заменяют элементы, для которых аргумент-функция возвращает истинное значение. Другой набор функций, отличается тем, что использует суффикс -IF-NOT, и выполняет те же операции, но для элементов, для которых функция не возвращает истинного значения.


```
(count-if #'evenp #(1 2 3 4 5)) ==> 2
(count-if-not #'evenp #(1 2 3 4 5)) ==> 3
(position-if #'digit-char-p "abcd0001") ==> 4
(remove-if-not #'(lambda (x) (char= (elt x 0) #\f))
  #("foo" "bar" "baz" "foom")) ==> #("foo" "foom")
```

В соответствии со стандартом языка, функции с суффиксом `-IF-NOT` являются устаревшими. Однако, это требование само считается неразумным. Если стандарт будет пересматриваться, то скорее будет удалено это требование, а не функции с суффиксом `-IF-NOT`. Для некоторых вещей, `REMOVE-IF-NOT` может использоваться чаще, чем `REMOVE-IF`. За исключением своего отрицательно звучащего имени, в действительности `REMOVE-IF-NOT` является положительной функцией – она возвращает элементы, которые соответствуют предикату.

Оба варианта функций принимают те же именованные аргументы, что и базовые функции, за исключением аргумента `:test`, которые не нужен, поскольку главный аргумент сам является функцией. При указании аргумента `:key`, функции передается значение, извлеченное функцией аргумента `:key`, а не сам элемент.

```
(count-if #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first) ==> 2
(count-if-not #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first) ==> 3
(remove-if-not #'alpha-char-p
  #("foo" "bar" "1baz")) :key #'(lambda (x) (elt x 0))) ==> #("foo" "bar")
```

Семейство функций `REMOVE` также имеет четвертый вариант, функцию `REMOVE-DUPPLICATES`, которая имеет один аргумент – последовательность, из которой удаляются все, кроме одного экземпляра, каждого дублированного элемента. Она может принимать те же самые именованные аргументы что и `REMOVE`, за исключением `:count`, поскольку она всегда удаляет все дубликаты.

```
(remove-duplicates #(1 2 1 2 3 1 2 3 4)) ==> #(1 2 3 4)
```

Работа с последовательностью целиком

Некоторое количество функций выполняют операции над последовательностями целиком. Это приводит к тому, что они проще чем функции, которые я описывал ранее. Например, `COPY-SEQ` и `REVERSE` получают по одному аргументу – последовательности, и возвращают новую последовательность того же самого типа. Последовательность, возвращенная `COPY-SEQ` содержит те же самые элементы, что и последовательность, аргумент этой функции, в то время как последовательность возвращенная `REVERSE` содержит те же самые элементы, но в обратном порядке. Заметьте, что ни одна из этих функций не копирует сами элементы, новым объектом является только возвращаемая последовательность.

Функция `CONCATENATE` создает новую последовательность, содержащую объединение любого числа последовательностей. Однако, в отличие от `REVERSE` и `COPY-SEQ`, которые просто возвращают последовательность того же типа, что и переданный аргумент, функции `CONCATENATE` должно быть явно указано, какой тип последовательности необходимо создать в том случае, если ее аргументы имеют разные типы. Первым аргументом функции является описание типа, подобный параметру `:element-type` функции `MAKE-ARRAY`. В этом случае, вероятнее всего вы будете использовать следующие символы для указания типа: `VECTOR`, `LIST` или `STRING`. Например:

```
(concatenate 'vector #(1 2 3) '(4 5 6)) ==> #(1 2 3 4 5 6)
(concatenate 'list #(1 2 3) '(4 5 6)) ==> (1 2 3 4 5 6)
(concatenate 'string "abc" '("#{d} #{e} #{f}")) ==> "abcdef"
```

Сортировка и слияние

Функции `SORT` и `STABLE-SORT` обеспечивают два метода сортировки последовательности. Они обе получают последовательность и функцию двух аргументов, и возвращают отсортированную последовательность.

```
(sort (vector "foo" "bar" "baz") #'string<) ==> #("bar" "baz" "foo")
```

Разница между этими функциями заключается в том, что `STABLE-SORT` гарантирует, что она не будет изменять порядок элементов, которые считаются одинаковыми, в то время как `SORT` гарантирует только, что результат будет отсортирован, так что некоторые одинаковые элементы могут быть поменяны местами.

Обе эти функции являются примерами так называемых деструктивных функций. Деструктивным функциям допускается (обычно в целях эффективности) модификация переданных аргументов тем или иным способом. Отсюда следует две вещи: во первых, вы всегда должны что-то делать с возвращаемым значением этих функций (например присвоить его переменной, или передать его другой функции), и во вторых, если вы хотите что-то еще делать с передаваемым в деструктивную функцию аргументом, то вы должны передать его копию, а не оригинальное значение. Я расскажу о деструктивных функциях более подробно в следующей главе.

Обычно вы не беспокоитесь о несортированной версии последовательности, так что имеет смысл позволить `SORT` и `STABLE-SORT` менять последовательность в процессе ее сортировки. Но это значит, что вы должны запомнить, что вы должны писать:

```
(setf my-sequence (sort my-sequence #'string<))
```

ВМЕСТО:

```
(sort my-sequence #'string<)
```

Обе эти функции принимают именованный аргумент `:key`, который также как и аргумент `:key` в других функциях работы с последовательностями, должен быть функцией, и используется для извлечения значений, которые будут передаваться предикату сортировки вместо оригинальных элементов. Извлеченные значения используются только для определения порядка элементов; возвращенная последовательность будет содержать сами элементы, а не извлеченные значения.

Функция `MERGE` принимает две последовательности и функцию-предикат, и возвращает последовательность, полученную путем слияния двух последовательностей в соответствии с предикатом. Она относится к функциям сортировки, так что, если каждая последовательность уже была отсортирована с использованием того же самого предиката, то и полученная последовательность также будет отсортирована. Также как и функции сортировки, `MERGE` принимает аргумент `:key`. Подобно `CONCATENATE`, и по тем же причинам, первым аргументом `MERGE` должно быть описание типа последовательности, которая будет получена в результате работы.

```
(merge 'vector #(1 3 5) #(2 4 6) #'<) ==> #(1 2 3 4 5 6)
(merge 'list #(1 3 5) #(2 4 6) #'<) ==> (1 2 3 4 5 6)
```

Работа с частями последовательностей

Другой набор функций позволит вам работать с частями последовательностей. Наиболее часто употребляемой функцией является SUBSEQ, которая выделяет часть последовательности начиная с определенного индекса и заканчивая другим индексом или концом последовательности. Например:

```
(subseq "foobarbaz" 3) ==> "barbaz"
(subseq "foobarbaz" 3 6) ==> "bar"
```

Для результата SUBSEQ также можно выполнить SETF, но оно не сможет увеличить или уменьшить последовательность; если часть последовательности и новое значение имеют разные длины, то более короткое из них определяет то, как много ???знаков будет изменено.

```
(defparameter *x* (copy-seq "foobarbaz"))
(setf (subseq *x* 3 6) "xxx") ; subsequence and new value are same length
*x* ==> "foxxxxbaz"
(setf (subseq *x* 3 6) "abcd") ; new value too long, extra character ignored.
*x* ==> "fooabcbaz"
(setf (subseq *x* 3 6) "xx") ; new value too short, only two characters changed
*x* ==> "fooxxcbaz"
```

Вы можете использовать функцию FILL для заполнения нескольких значений последовательность одним и тем же значением. Обязательными аргументами являются последовательность и значение, которым нужно заполнить элементы. По умолчанию, заполняется вся последовательность; вы можете использовать именованные аргументы :start и :end для ограничения границ заполнения.

Если вам нужно найти одну последовательность внутри другой, то вы можете использовать функцию SEARCH, которая работает также как и функция POSITION, но первым аргументом является последовательность, а не единичное значение.

```
(position #\b "foobarbaz") ==> 3
(search "bar" "foobarbaz") ==> 3
```

С другой стороны, для того, чтобы найти позицию в которой две последовательности с общим префиксом начинают различаться, вы можете использовать функцию MISMATCH. Она получает две последовательности и возвращает индекс первой пары неравных элементов.

```
(mismatch "foobarbaz" "foom") ==> 3
```

Эта функция возвращает NIL если строки совпадают. MISMATCH может также принимать стандартные именованные аргументы: аргумент :key для указания функции для извлечения сравниваемых значений; аргумент :test для указания функции сравнения; и аргументы :start1, :end1, :start2 и :end2 для указания границ действия внутри последовательностей. Также, указание :from-end со значением T приводит к тому, что поиск осуществляется в обратном порядке, заставляя MISMATCH вернуть индекс позиции в первой последовательности, где начинается общий суффикс последовательностей.

```
(mismatch "foobar" "bar" :from-end t) ==> 3
```

Предикаты для последовательностей

Другими полезными функциями являются `EVERY`, `SOME`, `NOTANY` и `NOTEVERY`, которые пробегают по элементам последовательности выполняя заданный предикат. Первым аргументом всех этих функций является предикат, а остальные аргументы – последовательности. Предикат должен получать столько аргументов, сколько последовательностей будет передано функциям. Элементы последовательностей передаются предикату (по одному элементу за раз) пока не закончатся элементы, или не будет выполнено условие завершения: `EVERY` завершается, возвращая ложное значение, сразу как это значение будет возвращено предикатом. Если предикат всегда возвращает истинное значение, то функция также вернет истинное значение. `SOME` возвращает первое не `NIL` значение, возвращенное предикатом, или возвращает ложное значение, если предикат никогда не вернул истинного значения. `NOTANY` возвращает ложное значение, если предикат возвращает истинное значение, или ложное, если этого не произошло. А `NOTEVERY` возвращает истинное значение сразу, как только предикат возвращает ложное значение, или ложное, если предикат всегда возвращал истинное. Вот примеры проверок для одной последовательности:

```
(every #'evenp #(1 2 3 4 5)) ==> NIL
(some #'evenp #(1 2 3 4 5)) ==> T
(notany #'evenp #(1 2 3 4 5)) ==> NIL
(notevery #'evenp #(1 2 3 4 5)) ==> T
```

В эти вызовы выполняют попарное сравнение последовательностей:

```
(every #'> #(1 2 3 4) #(5 4 3 2)) ==> NIL
(some #'> #(1 2 3 4) #(5 4 3 2)) ==> T
(notany #'> #(1 2 3 4) #(5 4 3 2)) ==> NIL
(notevery #'> #(1 2 3 4) #(5 4 3 2)) ==> T
```

Функции отображения последовательностей

В заключение, последними из функций работы с последовательностями, будут рассмотрены функции отображения (`mapping`). Функция `MAP`, подобно функциям-предикатам для последовательностей, получает функцию нескольких аргументов, и несколько последовательностей. Но вместо логического значения, `MAP` возвращает новую последовательность, содержащую результаты применения функции к элементам последовательности. Также как для `CONCATENATE` и `MERGE`, `MAP` необходимо сообщить тип создаваемой последовательности.

```
(map 'vector #'* #(1 2 3 4 5) #(10 9 8 7 6)) ==> #(10 18 24 28 30)
```

Функция `MAP-INTO` похожа на `MAP` за исключением того, что вместо создания новой последовательности заданного типа, она помещает результаты в последовательность, заданную в качестве первого аргумента. Эта последовательность может иметь такой же тип, как одна из последовательностей, предоставляющих данные для функции. Например, для суммирования нескольких векторов – `a`, `b` и `c` – в один, вы должны написать:

```
(map-into a #' + a b c)
```

Если последовательности имеют разную длину, то `MAP-INTO` изменяет столько элементов, сколько присутствует в самой короткой последовательности, включая ту, в которую помещаются результаты. Однако, если последовательность будет отображаться в вектор с указателем заполнения, то число изменяемых элементов будет определяться не указателем заполнения, а размером вектора. После вызова `MAP-INTO`, вектор заполнения будет установлен равным

количеству измененных элементов. Однако MAP-INTO не будет изменять размер векторов, которые допускают такую операцию.

Последней функцией работы с последовательностями, которую мы рассмотрим, является REDUCE, которая реализует другой вид отображения: она выполняет отображение для одной последовательности, применяя функцию двух аргументов сначала к первым двум элементам последовательности, а после первого вызова, последовательно применяя ее к полученному результату и следующим элементам. Таким образом, следующее выражение сложит числа от единицы до десяти:

```
(reduce #' + #(1 2 3 4 5 6 7 8 9 10)) ==> 55
```

REDUCE является очень полезной функцией – когда вам нужно создать из последовательности одно значение, есть вероятность, что вы сможете сделать это с помощью REDUCE, и она часто приводит к лаконичной записи того, что вы хотите сделать. Например, для нахождения максимального значения в последовательности вы можете просто написать (reduce #'max numbers). REDUCE также принимает полный набор стандартных именованных аргументов (:key, :from-end, :start и :end), а также один, уникальный для REDUCE (:initial-value). Этот аргумент указывает значение, которое будет логически помещено до первого элемента последовательности (или после последнего, если вы также зададите :from-end истинное значение).

Хэш-таблицы

Другой коллекцией общего назначения предоставляемой Common Lisp является хэш-таблица. В то время как вектора являются структурами данных с доступом по целочисленному индексу, хэш-таблицы позволяют вам использовать в качестве индексов (ключей) любые объекты. Когда вы добавляете объекты в хэш-таблицу, вы сохраняете их с определенным ключом. Позднее вы можете использовать тот же самый ключ для доступа к значению. Или вы можете связать новое значение с тем же самым ключом – каждый ключ отображается в единственное значение.

Без указания дополнительных аргументов MAKE-HASH-TABLE создает хэш-таблицу, которая сравнивает ключи с использованием функции EQL. Это нормально до тех пор, пока вы не захотите использовать в качестве ключей строки, поскольку две строки с одинаковым содержимым, не обязательно равны в терминах EQL. В таком случае вы захотите использовать для сравнения функцию EQUAL, что вы можете сделать передав функции MAKE-HASH-TABLE символ EQUAL в качестве именованного аргумента :test. Кроме того, для аргумента :test можно использовать еще два символа: EQ и EQUALP. Конечно, эти символы являются именами стандартных функций сравнения объектов, которые я обсуждал в главе 4. Однако в отличие от аргумента :test, передаваемого функциям работы с последовательностями, аргумент :test функции MAKE-HASH-TABLE не может использовать произвольную функцию – допустимы только значения EQ, EQL, EQUAL и EQUALP. Это происходит потому что хэш-таблицы в действительности нуждаются в двух функциях – функции сравнения и функции хэширования, которая вычисляет численный хэш-код из ключа способом, совместимым с тем как функция сравнения однозначно сравнивает два ключа. Хотя стандарт языка предоставляет хэш-таблицы, которые используют только стандартные функции сравнения, большинство реализаций обеспечивают некоторые механизмы для создания более тонко настраиваемых хэш-таблиц.

Функция GETHASH обеспечивает доступ к элементам хэш-таблиц. Она принимает два аргумента: ключ и хэш-таблицу, и возвращает значение, если оно найдено, или NIL в противном случае. Например:

```
(defparameter *h* (make-hash-table))
(gethash 'foo *h*) ==> NIL
(setf (gethash 'foo *h*) 'quux)
(gethash 'foo *h*) ==> QUUX
```

Поскольку GETHASH возвращает NIL если ключ не присутствует в таблице, то нет никакого способа узнать отсутствует ли ключ в таблице, или для данного ключа соответствует значение NIL. Функция GETHASH решает эту проблему за счет использования способа, который мы еще не обсуждали – возврат нескольких значений. В действительности GETHASH возвращает два значения: главное значение – значение для указанного ключа или NIL. Дополнительное значение имеет логический тип и указывает, присутствует ли значение в хэш-таблице. Из-за способа реализации возврата множественных значений, дополнительные значения просто отбрасываются, если только пользователь не обрабатывает эту ситуацию специальным образом, используя средства, которые "видят" множественные значения.

Я буду подробно обсуждать возврат множественных значений в главе 20, но сейчас я дам вам лишь общее представление о том, как использовать макрос MULTIPLE-VALUE-BIND для получения дополнительных значений, которые возвращает GETHASH. Макрос MULTIPLE-VALUE-BIND создает привязки переменных, как это делает LET, заполняя их множеством значений, возвращенных вызываемой функцией.

Следующие примеры показывают как вы можете использовать MULTIPLE-VALUE-BIND; связываемые переменные содержат значение и признак его наличия в таблице:

```
(defun show-value (key hash-table)
  (multiple-value-bind (value present) (gethash key hash-table)
    (if present
      (format nil "Значение ~a присутствует в таблице." value)
      (format nil "Значение равно ~a, поскольку ключ не найден." value))))
(setf (gethash 'bar *h*) nil) ; создает ключ со значением NIL
(show-value 'foo *h*) ==> "Значение QUUX присутствует в таблице."
(show-value 'bar *h*) ==> "Значение NIL присутствует в таблице."
(show-value 'baz *h*) ==> "Значение равно NIL , поскольку ключ не найден."
```

Поскольку установка значения в NIL оставляет ключ в таблице, вам понадобится другая функция для полного удаления пары ключ/значение. Функция REMHASH получает такие же аргументы как и GETHASH, и удаляет указанную запись. Вы также можете полностью очистить хэш-таблицу с помощью функции CLRHASH.

Функции для работы с записями в хэш-таблицах

Common Lisp предоставляет разные способы для работы с записями в хэш-таблицах. Простейшим из них является использование функции MAPHASH. Также как и функция MAP, функция MAPHASH принимает в качестве аргументов функцию двух аргументов и хэш-таблицу, и выполняет указанную функцию для каждой пары ключ/значение. Например, для вывода всех пар ключ/значение, вы можете использовать такой вызов MAPHASH:

```
(maphash #'(lambda (k v) (format t "~a => ~a~%" k v)) *h*)
```

Последствия добавления или удаления записей в хэш-таблице во время прохода по ее записям стандартом не указываются (и скорее всего это будет плохой практикой), за исключением двух случаев: вы можете использовать SETF вместе с GETHASH для изменения значения текущей записи, и вы можете использовать REMHASH для удаления текущей записи. Например, для удаления всех записей, чье значение меньше чем десять, вы можете записать вот так:

```
(maphash #'(lambda (k v) (when (< v 10) (remhash k *h*))) *h*)
```

Другим способом итерации по элементам хэш-таблицы является использование макроса LOOP, который будет описан в главе 22. Код использующий LOOP, реализующий то же, что и предыдущий пример, будет выглядеть вот так:

```
(loop for k being the hash-keys in *h* using (hash-value v)
      do (format t "~a => ~a~%" k v))
```

Я могу рассказать много больше о коллекциях, не являющихся списками, поддерживаемых Common Lisp. Например, я совсем не рассказал про многомерные массивы или про библиотеку функций для работы с битовыми массивами. Однако то, что я рассказал в этой главе должно быть достаточно для обычной работы. Теперь наступает время для того, чтобы взглянуть на структуру данных, давшую имя языку Lisp – списки.

12. Они назвали его Lisp не спроста: обработка списков

Списки в языке Lisp играют важную роль как по историческим, так и по практическим причинам. Изначально списки были основным составным типом данных в языке Lisp, и в течении десятилетий они были единственным составным типом данных. В наши дни, программист на Common Lisp может использовать как типы vector, hash table, самостоятельно определённые типы и структуры, так и списки.

Списки остаются в языке потому, что они являются прекрасным решением для определённого рода проблем. Одна из них - проблема представления кода, как данных для трансформации и генерации кода в макросах - является специфичной для языка Lisp, и это объясняет то, как другие языки обходятся без списков в стиле Lisp. Вообще говоря, списки - прекрасная структура данных для представления любых неоднородных и/или иерархических данных. Кроме того они достаточно легковесны и поддерживают функциональный стиль программирования - ещё одна важная часть наследия Lisp.

Итак, вы должны понять списки в их собственных терминах; после того как вы достигнете хорошего понимания как списки работают, вы сможете лучше оценивать стоит ли их использовать или нет.

Списков нет

Мальчик с ложкой: Не пытайся согнуть список. Это невозможно. Вместо этого... попытайся понять истину.

Нео: Какую истину?

Мальчик с ложкой: Списков нет.

Нео: Списков нет?

Мальчик с ложкой: И тогда ты поймёшь: это не списки, которые сгибаются; это только ты сам.

Ключом к пониманию списков, является осознание того, что они, по большей части, иллюзия, построенная на основе объектов более примитивных типов данных. Эти простые объекты - пары значений, называемые cons-ячейкой, от имени функции CONS, используемой для их создания.

CONS принимает 2 аргумента и возвращает новую cons-ячейку, содержащую 2 значения. Эти значения могут быть ссылками на объект любого типа. Если второе значение не NIL и не другая cons-ячейка, то ячейка печатается как два значения в скобках, разделённые точкой (так называемая точечная пара).

```
(cons 1 2) ==> (1 . 2)
```

Два значения в cons-ячейке называются CAR и CDR - по имени функций, используемых для доступа к ним. На заре времён эти имена были мнемониками, по крайней мере для людей, работающих над первой реализацией Lisp на IBM 704. Но даже тогда они были всего лишь позаимствованы из мнемоник ассемблера, используемых для реализации этих операций. Тем не менее, это не так уж плохо, что названия этих функций несколько бессмысленны, применительно к конкретной cons-ячейке лучше думать о них просто как о произвольной паре значений без какого-либо особого смысла. Итак:


```
(car (cons 1 2)) ==> 1
(cdr (cons 1 2)) ==> 2
```

Оба CAR и CDR могут меняться функцией SETF: если дана cons-ячейка, то возможно присвоить значение обеим её составляющим.

```
(defparameter *cons* (cons 1 2))
*cons* ==> (1 . 2)
(setf (car *cons*) 10) ==> 10
*cons* ==> (10 . 2)
(setf (cdr *cons*) 20) ==> 20
*cons* ==> (10 . 20)
```

Т.к. значения в cons-ячейке могут быть ссылками на любой объект, вы можете строить большие структуры, связывая cons-ячейки между собой. Списки строятся связыванием cons-ячеек в цепочку. Элементы списка содержатся в CAR cons-ячеек, а связи со следующими cons-ячейками содержатся в их CDR. Последняя ячейка в цепочке имеет CDR со значением NIL, которое - как я говорил в [главе 4](#) <psl:синтаксиссемантика>. - представляет собой как пустой список, так и булево значение ложь.

Такая компоновка отнюдь не уникальна для Lisp, она называется однонаправленный список. Несколько языков, не входящих в семейство Lisp, предоставляют расширенную поддержку для этого скромного типа данных.

Таким образом, когда я говорю, что какое-то значение является списком, я имею в виду, что оно либо NIL, либо ссылка на cons-ячейку. CAR cons-ячейки является первым элементом списка, а CDR является ссылкой на другой список, который в свою очередь также является cons-ячейкой, содержащий остальные элементы, или значением NIL. Lisp-печаталка понимает это соглашение и печатает такие цепочки cons-ячеек как списки в скобках, нежели как точечные пары.

```
(cons 1 nil) ==> (1)
(cons 1 (cons 2 nil)) ==> (1 2)
(cons 1 (cons 2 (cons 3 nil))) ==> (1 2 3)
```

Когда мы говорим о структурах из cons-ячеек, нам могут помочь несколько диаграмм. Эти диаграммы изображают cons-ячейки как пары блоков:

{{one-cons-cell.png}}

Блок слева представляет CAR, а блок справа - CDR. Значения, хранимые в ячейках также представлены в виде отдельных блоков или в виде стрелки выходящей из блока, для представления ссылки на значение. Например, список (1 2 3), который состоит из трёх cons-ячеек, связанных вместе с помощью их CDR, может быть изображён так:

{{list-1-2-3.png}}

Однако, в основном при работе со списками вы не обязаны иметь дело с отдельными ячейками, т.к. существуют функции, которые создают списки и используются для манипулирования ими. Например, функция LIST строит cons-ячейки и связывает их вместе, следующие LISP-выражения эквивалентны CONS-выражениям, приведённым выше:

```
(list 1) ==> (1)
(list 1 2) ==> (1 2)
(list 1 2 3) ==> (1 2 3)
```

Аналогично, когда вы думаете в терминах списков, вы не должны использовать бессмысленные имена CAR и CDR; вы должны использовать FIRST и REST - синонимы для CAR и CDR при рассмотрении cons-ячеек, как списков.

```
(defparameter *list* (list 1 2 3 4))
(first *list*) ==> 1
(rest *list*) ==> (2 3 4)
(first (rest *list*)) ==> 2
```

Как и cons-ячейки, списки могут содержать значения любых типов.

```
(list "foo" (list 1 2) 10) ==> ("foo" (1 2) 10)
```

Структура списка при этом выглядит так:

{{mixed-list.png}}

Т.к. элементами списка могут быть другие списки, то с помощью списков можно представлять деревья произвольной глубины и сложности. Таким образом, они идеально подходят для представления гетерогенных иерархических данных. Например, XML-процессоры, основанные на Lisp, как правило представляют XML-документы в виде списков. Другим очевидным примером данных с древовидной структурой является сам код на языке Lisp. В главах 30 и 31 вы напишете библиотеку для генерации HTML-кода, которая использует списки списков для представления кода, который необходимо сгенерировать. В следующей главе я расскажу больше об использовании cons-ячеек для представления других типов данных.

Common Lisp предоставляет обширную библиотеку функций для работы со списками. В разделах [Функции для работы со списками](#) <#Функции для работы со списками> и [Отображение](#) <#Отображение> вы узнаете о наиболее важных из них. Данные функции будет проще понять в контексте идей, взятых из парадигмы функционального программирования.

Функциональное программирование и списки

Суть функционального программирования в том, что программы состоят исключительно из функций без побочных эффектов, которые вычисляют свои значения исключительно на основе своих аргументов. Преимущество функционального стиля программирования в том, что он облегчает понимание программы. Устранение побочных эффектов ведёт к устранению практически всех возможностей удалённого воздействия. Т.к. результат функции определяется её аргументами, поведение функции легче понять и протестировать. Например, когда вы видите выражение `(+ 3 4)`, вы знаете, что результат целиком задаётся определением функции `+` и переданными аргументами 3 и 4. Вам не надо беспокоиться о том, что могло произойти в программе до этого, т.к. нет ничего, что могло бы изменить результат вычисления выражения.

Функции, работающие с числами, естественным образом являются функциональными, т.к. числа неизменяемы. Списки же, как вы видели ранее, могут меняться, при применении функции `SETF` над ячейками списка. Тем не менее, списки могут рассматриваться как функциональные типы данных, если считать, что их значение определяется элементами, которые они содержат. Так, любой список вида `(1 2 3 4)` функционально эквивалентен любому другому списку, содержащему эти четыре значения, независимо от того, какие cons-ячейки используются для

представления списка. И любая функция, которая принимает списки в качестве аргументов и возвращает значение, основываясь исключительно на содержании списка, могут считаться функциональными. Например, если функции REVERSE передать список (1 2 3 4), она всегда вернёт (4 3 2 1). Различные вызовы REVERSE с функционально-эквивалентными аргументами вернёт функционально-эквивалентные результаты. Другой аспект функционального программирования, который я рассматриваю в разделе "Отображение", это использование функций высших порядков: функций, которые используют другие функции, как данные, принимая их в качестве аргументов или возвращая в качестве результата.

Большинство функций Common Lisp для работы со списками написаны в функциональном стиле. Позднее мы обсудим как совмещать функциональный подход к программированию с другими подходами, но сначала вы должны понять некоторые тонкости функционального подхода применительно к спискам.

Т.к. большинство функций для работы со списками написаны в функциональном стиле, они могут возвращать результаты, которые имеют общие cons-ячейки с их аргументами. Возьмём конкретный пример: функция APPEND принимает любое число списком в качестве аргументов и возвращает новый список, содержащий все элементы списков-аргументов, например:

```
(append (list 1 2) (list 3 4)) ==> (1 2 3 4)
```

С точки зрения функционального подхода, задача функции APPEND - вернуть список (1 2 3 4) не изменяя ни одну из cons-ячеек в списках-аргументах (1 2) и (3 4). Очевидно, что это можно сделать создав новый список, состоящий из четырёх новых cons-ячеек. Однако в этом есть лишняя работа. Вместо этого APPEND на самом деле создаёт только две новые cons-ячейки, чтобы хранить значений 1 и 2, соединяя их вместе и делая ссылку из CDR второй ячейки на первый элемент последнего аргумента - списка (3 4). После этого функция возвращает cons-ячейку содержащую 1. Ни одна из входных cons-ячеек не была изменена, и результатом, как и требовалось, является список (1 2 3 4). Единственная хитрость в том, что результат, возвращаемый функцией APPEND имеет общие cons-ячейки со списком (3 4). Структура результата выглядит так:

{{after-append.png}}

В общем случае, функция APPEND должна копировать все кроме последнего аргумента-списка, и она может вернуть результат, которые имеет общие структуры с последним аргументом.

Другие функции также используют полезную особенность списков иметь общие структуры. Некоторые, как и APPEND, всегда возвращают результаты, которые имеют общие структуры со своими аргументами. Другим функциям позволено возвращать результаты с общими структурами, но это зависит от их реализации.

"Разрушающие" операции

Если бы Common Lisp был строго функциональным языком, больше не о чем было бы говорить. Однако, т.к. после создания cons-ячейки есть возможность изменить её значение применив функцию SETF над её CAR и CDR, мы должны обсудить как стыкуются побочные эффекты и общие структуры.

Из-за функционального наследия языка Lisp, операции, которые изменяют существующие объекты называются разрушающими - в функциональном программировании изменение состояния объекта "разрушает" его, т.к. объект больше не представляет того же значения, что до применения функции. Однако использование одного и того же термина для обозначения всего класса операций, изменяющих состояние существующих объектов, ведёт к некоторой путанице,

т.к. существует 2 сильно отличающихся класса таких операций: операции-для-побочных-эффектов и утилизирующие операции.

Операции-для-побочных-эффектов это те, которые используются ради их эффектов. В этом смысле, всякое использование SETF является разрушающим, как и использование функций, которые вызывают SETF чтобы изменить состояние объектов, например VECTOR-PUSH или VECTOR-POP. Но это несколько некорректно объявлять операции разрушающими, т.к. они не предназначены для написания программ в функциональном стиле, т.е. они не могут быть описаны с использованием терминов функциональной парадигмы. Тем не менее, если вы смешиваете нефункциональные операции-для-побочных-эффектов с функциями возвращающими результаты с общими структурами, то надо быть внимательным, чтобы случайно не изменить эти общие структуры. Например, имеем:

```
(defparameter *list-1* (list 1 2))  
(defparameter *list-2* (list 3 4))  
(defparameter *list-3* (append *list-1* *list-2*))
```

После вычисления у вас 3 списка, но list-3 и list-2 имеют общую структуру, как показано на предыдущей диаграмме.

```
*list-1* ==> (1 2)  
*list-2* ==> (3 4)  
*list-3* ==> (1 2 3 4)
```

Посмотрим, что случится если мы изменим list-2:

```
(setf (first *list-2*) 0) ==> 0  
*list-2* ==> (0 4) ; как и ожидалось  
*list-3* ==> (1 2 0 4) ; а вот этого возможно вы и не хотели
```

Из-за наличия общих структур, изменения в списке list-2 привели к изменению списка list-3: первая cons-ячейка в list-2 является также третьей ячейкой в list-3. Изменение значения FIRST списка list-2 изменили значение CAR в cons-ячейке, изменив оба списка.

Совсем по-другому обстоит дело с утилизирующими операциями, которые предназначены для запуска в функциональном коде. Они используют побочные эффекты лишь для оптимизации. В частности, они повторно используют некоторые cons-ячейки своих аргументов для получения результатов. Но в отличие от таких функций, как APPEND, которые используют cons-ячейки, включая их без изменений в результирующий список, утилизирующие операции используют cons-ячейки как сырьё, изменяя их CAR и CDR для получения желаемого результата. Таким образом утилизирующие операции спокойно могут быть использованы, когда их аргументы не пригодятся после их выполнения.

Чтобы понять как работают функции утилизации сравним неразрушающую операцию REVERSE, которая возвращает инвертированный аргумент, с функцией NREVERSE, которая является утилизирующей функцией и делает то же самое. Т.к. REVERSE не меняет своих аргументов, она создаёт новую cons-ячейку для каждого элемента списка-аргумента. Но предположим, вы напишите:

```
(setf *list* (reverse *list*))
```

Присвоив результат вычисления переменной *list*, вы удалили ссылку на начальное значение *list*. Если на cons-ячейки в начальном списке больше никто не ссылается, то они теперь

доступны для сборки мусора. Тем не менее в большинстве реализаций Lisp более эффективным является повторно использовать эти ячейки, чем создавать новые, а старые превращать в мусор.

NREVERSE именно это и делает. N - сокращение для "не создающая", в том смысле, что она не создаёт новых cons-ячеек. Конкретные побочные эффекты NREVERSE явно не описываются, она может проводить любые модификации над CAR и CDR любых cons-ячеек аргумента, но типичная реализация может быть такой: проход по списку с изменением CDR каждой cons-ячейки, так чтобы она указывала на предыдущую cons-ячейку, в конечном счёте результатом будет cons-ячейка, которая была последней в списке аргументе, а теперь является головой этого списка. Никакие новые cons-ячейки при этом не создаются и сборка мусора не производится.

Большинство утилизирующих функций, таких как NREVERSE, имеют своих неразрушающих двойников, которые вычисляют тот же результат. В общем случае утилизирующие функции имеют такое же имя, как их недеструктивные двойники с подставленной первой буквой N. Но есть и исключения, например часто используемые: NCONC - утилизирующая версия APPEND и DELETE, DELETE-IF, DELETE-IF-NOT, DELETE-DUPLICATED - версии семейства функций REMOVE.

В общем случае, вы можете использовать утилизирующие функции таким же образом, как их недеструктивные аналоги, учитывая, что они безопасны, если их аргументы не будут использованы после выполнения функций. Побочные эффекты большинства утилизирующих функций не достаточно строго описаны, чтобы на них можно было полагаться.

Однако ещё большую неразбериху вносит небольшой набор утилизирующих функций со строго определёнными побочными эффектами, на которые можно положиться. В этот набор входят NCONC, утилизирующая версия APPEND, и NSUBSTITUTE и её -IF и -IF-NOT варианты - утилизирующие версии группы функций SUBSTITUTE.

Как и APPEND, NCONC возвращает соединение своих аргументов, но строит такой результат следующим образом: для каждого непустого аргумента-списка, NCONC устанавливает в CDR его последней cons-ячейки ссылку на первую cons-ячейку следующего непустого аргумента-списка. После этого она возвращает первый список, который теперь является головой результата-соединения. Таким образом:

```
(defparameter *x* (list 1 2 3))
(nconc *x* (list 4 5 6)) ==> (1 2 3 4 5 6)
*x* ==> (1 2 3 4 5 6)
```

На функцию NSUBSTITUTE и её варианты можно положиться в следующем её поведении: она пробегает по списковой структуре аргумента-списка и устанавливает с помощью функции SETF новое значения в CAR его cons-ячеек. После этого она возвращает переданный ей аргумент-список, который теперь имеет то же значение, как если бы был вычислен с помощью SUBSTITUTE.

Ключевой момент, который необходимо запомнить о NCONC и NSUBSTITUTE состоит в том, что они являются исключением из правила не полагаться на побочные эффекты утилизирующих функций. Вполне допустимым и даже желательным, является игнорирование возможности полагаться на побочные эффекты этих функций и использование их как и любых других утилизирующих функций только ради возвращаемых ими значений.

Комбинирование утилизации с общими структурами

Хотя вы можете спокойно использовать утилизирующие функции, если их аргументы не будут использованы позднее, стоит заметить, что каждая утилизирующая функция как заряженное

ружье направленное на собственную ногу, в том смысле, что если вы случайно примените утилизирующую функцию к аргументу, который используете позднее, то наверняка потеряете пару пальцев.

Всё усложняется тем, что функции использующие общие структуры и утилизирующие функции работают над схожими задачами. Недеструктивные списочные функции возвращают списки, которые имеют общие структуры со своими аргументами, основываясь на предположении, что cons-ячейки этих общих структур никогда не поменяются, а утилизирующие функции работают с нарушением этого допущения. Другими словами, использование общих структур основано на предположении, что вам безразлично какие cons-ячейки составляют результат, утилизирующие же функции требуют, чтобы вы точно знали, откуда берутся cons-ячейки.

На практике утилизирующие функции используются в нескольких определённых случаях. Наиболее частый случай - построение списка, который будет возвращён из функции добавлением элементов в его конец (как правило с использованием функции PUSH), а потом инвертирование данного списка. При это список храниться в локальной для данной функции переменной.

Это является эффективным способом построения списка, потому что каждый вызов PUSH создаёт только одну cons-ячейку, а вызов NREVERSE быстро пробегает по списку, переназначая CDR ячеек. Т.к. список создаётся в локальной переменной внутри функции, то нет никакой возможности, что какой-то код за пределами функции имеет общие структуры с данным списком. Вот пример функции, которая использует такой приём для построения списка, который содержит числа от 0 до n :

```
(defun upto (max)
  (let ((result nil))
    (dotimes (i max)
      (push i result))
    (nreverse result)))
(upto 10) ==> (0 1 2 3 4 5 6 7 8 9)
```

Другой часто встречающийся случай применения утилизирующих функций - немедленное переприсваивание значения, возвращаемого утилизирующей функцией переменной, содержащей потенциально утилизированное значение. Например, вы часто будете такие видеть выражения с использованием функций DELETE (утилизирующей версии REMOVE):

```
(setf foo (delete nil foo))
```

Эта конструкция присваивает переменной foo её старое значение с удалёнными элементами, равными NIL. Однако, здесь утилизирующие функции должны применяться осмотрительно: если foo имеет общие структуры с другими списками, то использование DELETE вместо REMOVE может разрушить структуры этих других списков. Например, возьмём два списка list-2 и list-3, рассмотренные ранее, они разделяют свои последние 2 cons-ячейки.

```
*list-2* ==> (0 4)
*list-3* ==> (1 2 0 4)
```

Вы можете удалить 4 из list-3 так:

```
(setf *list-3* (delete 4 *list-3*)) ==> (1 2 0)
```

Но DELETE скорее всего произведёт удаление тем, что установит CDR третьей ячейки списка в NIL, отсоединив таким образом четвёртую ячейку от списка. Т.к. третья ячейка в list-3

является также первой ячейкой в `list-2`, этот код изменить также и `list-2`:

```
*list-2* ==> (0)
```

Если вы используете `REMOVE` вместо `DELETE`, то результат будет построен с помощью создания новых ячеек, не изменяя ячеек `list-3`. В этом случае `list-2` не будет изменён.

Эти два случая использования вместе составляют около 80% от общего числа случаев использования утилизирующих функций. Другие случаи использования возможны, но требуют внимательного отслеживания того возвращают ли функции списки с общими структурами или нет.

В общем случае, при работе со списками лучше писать код в функциональном стиле - ваши функции должны полагаться на содержание их аргументов-списков и не должны менять их. Следование этому правилу, разумеется, исключает использование разрушающих операций, утилизирующих или нет - не имеет значения. После того, как вы получите работающий код и профилирование покажет, что вам нужна оптимизация, вы можете заменить неразрушающие операции со списками на их утилизирующие варианты, но только при условии, что аргументы данных функций не используются где-то ещё.

Важный момент, на который следует обратить внимание: функции сортировки списков `SORT`, `STABLE-SORT` и `MERGE` из главы 11 также являются утилизирующими функциями когда они применяются к спискам. Эти функции не имеют неразрушающих аналогов, так что если вам необходимо отсортировать списки не разрушая их, вы должны передать в сортирующую функцию копию списка, сделанную с помощью `COPY-LIST`. В любом случае, вы должны сохранить результат функции, т.к. исходный аргумент-список будет разрушен. Например:

```
CL-USER> (defparameter *list* (list 4 3 2 1))
*LIST*
CL-USER> (sort *list* #'<)
(1 2 3 4) ; кажется то, что надо
CL-USER> *list*
(4) ; упс!
```

Функции для работы со списками

Теперь вы готовы взглянуть на библиотеку функций для работы со списками, которую предоставляет Common Lisp.

Вы уже видели базовые функции для извлечения элементов списка: `FIRST` и `REST`. Хотя вы можете получить любой элемент списка комбинируя вызовы `REST` (для продвижения по списку) и `FIRST` (для выделения элемента), это может быть утомительным занятием. Поэтому Lisp предоставляет функции от `SECOND` до `TENTH` извлекающие соответствующие элементы списка. Более общая функция `NTN` принимает два аргумента: индекс и список, и возвращает *n*-ый (начиная с нуля) элемент списка. Также существует функция `NTNCDR`, принимающая индекс и список и возвращающая результат *n*-кратного применения `REST` к списку. Таким образом, `(nthcdr 0 ...)` просто возвращает исходный список, а `(nthcdr 1 ...)` эквивалентно вызову `REST`. Имейте в виду, что ни одна из этих функций не является более производительной по сравнению с эквивалентной комбинацией `FIRST` и `REST`, т.к. нет иного способа получить *n*-ый элемент списка без *n*-кратного вызова `CDR`.

28 составных `CAR/CDR` функций составляют ещё одно семейство, которое вы можете время от времени использовать. Имя каждой функции получается подстановкой до 4 букв `A` или `D` между `C` и `R`, каждая `A` представляет вызов `CAR`, а каждая `D` - `CDR`. Таким образом:

```
(caar list) === (car (car list))  
(cadr list) === (car (cdr list))  
(cadadr list) === (car (cdr (car (cdr list))))
```

Обратите внимание, что многие из этих функций имеют смысл только применительно к спискам, содержащим другие списки. Например, CAAR возвращает CAR от CAR исходного списка, таким образом, этот список должен иметь своим первым элементом список. Другими словами, эти функции для работы скорее с деревьями нежели со списками:

```
(caar (list 1 2 3)) ==> ошибка  
(caar (list (list 1 2) 3)) ==> 1  
(cadr (list (list 1 2) (list 3 4))) ==> (3 4)  
(caadr (list (list 1 2) (list 3 4))) ==> 3
```

В настоящее время, эти функции не используются так часто, как раньше. Даже упёртые Lisp-хакеры старой закалки стремятся избегать этих длинных комбинаций. Однако они присутствуют в старом Lisp-коде, поэтому необходимо представлять, как они работают.

Функции FIRST-TENTH, CAR, CADR и т.д. могут также быть использованы как аргумент к SETF, если вы пишете в нефункциональном стиле.

Другие функции для работы со списками	
Функция	Описание
LAST	Возвращает последнюю cons-ячейку в списке. Если вызывается с целочисленным аргументом n, возвращает n ячеек.
BUTLAST	Возвращает копию списка без последней cons-ячейки. Если вызывается с целочисленным аргументом n, исключает последние n ячеек.
NBUTLAST	Утилизирующая версия BUTLAST; может изменять переданный список-аргумент, не имеет строго заданных побочных эффектов.
LDIFF	Возвращает копию списка до заданной cons-ячейки.
TAILP	Возвращает TRUE если переданный объект является cons-ячейкой, которая является частью списка.
LIST*	Строит список, содержащий все переданные аргументы кроме последнего, после этого присваивает CDR последней cons-ячейки списка последнему аргументу. Т.е. смесь LIST и APPEND.
MAKE-LIST	Строит список из n элементов. Начальные элементы списка NIL или значение заданное аргументом :initial-element.
REVAPPEND	Комбинация REVERSE и APPEND; инвертирует первый аргумент как REVERSE и добавляет второй аргумент. Не имеет строго заданных побочных эффектов.
NRECONC	Утилизирующая версия предыдущей функции; инвертирует первый аргумент как это делает NREVERSE и добавляет второй аргумент. Не имеет строгих побочных эффектов.
CONSP	Предикат для тестирования является ли объект cons-ячейкой.
ATOM	Предикат для тестирования является ли объект не cons-ячейкой.
LISTP	Предикат для тестирования является ли объект cons-ячейкой или NIL
NULL	Предикат для тестирования является ли объект NIL. Функционально эквивалентен функции NOT, но стилистически лучше использовать NULL при тестировании является ли список NIL, NOT для проверки булевого выражения FALSE

Отображение

Другой важный аспект функционально стиля программирования - это использование функций высших порядков, т.е. функций, которые принимают функции в качестве аргументов или возвращают функции. Вы видели несколько примеров функций высших порядков, таких как MAP, в предыдущей главе. Хотя MAP может использоваться как со списками, так и с векторами (т.е. с любым типом последовательностей), Common Lisp также предоставляет 6 функций отображения специально для списков. Разница между этими шестью функциями в том как они строят свой результат и в том применяют ли они переданные функции к элементам списка или к cons-ячейкам списка.

MAPCAR функция наиболее похожая на MAP. Т.к. она всегда возвращает список, она не требует уточнения типа результата, как MAP. Вместо этого, первый аргумент MAPCAR - функция, которую необходимо применить, а последующие аргументы - списки, чьи элементы будут поставлять аргументы для этой функции. Другими словами MAPCAR ведёт себя, как MAP: функция применяется к элементам аргументов-списков, беря от каждого списка по элементу. Результат каждого вызова функции собирается в новый список. Например:

```
(mapcar #'(lambda (x) (* 2 x)) (list 1 2 3)) ==> (2 4 6)
(mapcar #'(lambda (x) (+ 10 x)) (list 1 2 3)) ==> (11 12 13)
```

MAPLIST работает как MAPCAR, только вместо того, чтобы передавать функции элементы списка, она передаёт cons-ячейки. Таким образом, функция имеет доступ не только к значению каждого элемента списка (через функцию CAR, применённую к cons-ячейке), но и к хвосту списка (через CDR).

MAPCAN и MAPCON работают как MAPCAR и MAPLIST, разница состоит в том, как они строят свой результат. В то время как MAPCAR и MAPLIST строят новый список, содержащий результаты вызова функций, MAPCAN и MAPCON строят результат склеиванием результатов (которые должны быть списками), как это делает NCONC. Таким образом, каждый вызов функции может предоставлять любое количество элементов, для включения в результирующий список. MAPCAN, как MAPCAR, передаёт элементы списка отображающей функции, а MAPCON, как MAPLIST, передаёт cons-ячейки.

Наконец, функции MAPC и MAPL - управляющие структуры, замаскированные под функции, они просто возвращают свой первый аргумент-список, так что они используются только из-за побочных эффектов передаваемой функции. MAPC действует как MAPCAR и MAPCAN, а MAPL как MAPLIST и MAPCON.

Другие структуры

Хотя cons-ячейки и списки обычно трактуются как синонимы, это не всегда так - как я говорил ранее, вы можете использовать списки списков для представления деревьев. Так же как функции из этой главы позволяют вам интерпретировать структуры, построенные из cons-ячеек, как списки, другие функции позволяют использовать cons-ячейки для представления деревьев, множеств, и двух типов отображений ключ-значение. Мы рассмотрим большинство этих функций [в следующей главе](#) <psl:нетолькосписки>.

13. Не только списки: другие применения cons-ячеек

Как вы увидели в предыдущей главе, списочный тип данных является иллюзией, созданной множеством функций, которые манипулируют cons-ячейками. Common Lisp также предоставляет функции, позволяющие вам обращаться со структурами данных, созданными из cons-ячеек, как с деревьями, множествами, и таблицами поиска (*lookup tables*). В этой главе я дам вам краткий обзор некоторых из этих структур данных и оперирующих с ними функций. Как и в случае функций, работающих со списками, многие из них будут полезны когда вы начнете писать более сложные макросы, где потребуется обращаться с кодом Lisp как с данными.

Деревья

Рассматривать структуры созданные из cons-ячеек как деревья так же естественно, как рассматривать их как списки. Ведь что такое список списков, как не другое представление дерева? Разница между функцией, которая обращается с группой cons-ячеек как со списком, и функцией, обращающейся с той же группой cons-ячеек, как с деревом, определяется тем, какие ячейки обходят эти функции при поиске значений для дерева или списка. Те cons-ячейки, которые обходит функция работы со списком, называются *списочной структурой*, и располагаются начиная от первой cons-ячейки и затем следуя ссылкам по CDR, пока не достигнут NIL. Элементами списка являются объекты, на которые ссылаются CAR'ы cons-ячеек списочной структуры. Если cons-ячейка в списочной структуре имеет CAR, который ссылается на другую cons-ячейку, то та ячейка, на которую ведет ссылка, считается головой и элементом внешнего списка. *Древовидная структура*, с другой стороны, при обходе следует ссылкам как по CAR, так и по CDR, пока они указывают на другие cons-ячейки. Таким образом значения в древовидной структуре это атомы - те FIXME значения-не-cons-ячейки, на которые ссылаются CAR'ы или CDR'ы cons-ячеек в древовидной структуре.

Например, следующая стрелочная диаграмма показывает cons-ячейки, составляющие список из списков: ((1 2) (3 4) (5 6)). Списочная структура включает в себя только три cons-ячейки внутри пунктирного блока, тогда как древовидная структура включает все ячейки.

{{ pcl:chapter13:ch13-1.png }}

Чтобы увидеть разницу между функцией, работающей со списком, и функцией, работающей с деревом, вы можете рассмотреть как функции COPY-LIST и COPY-TREE будут копировать эту группу cons-ячеек. COPY-LIST, как функция, работающая со списком, копирует cons-ячейки, которые составляют списочную структуру. Другими словами, она создает новые cons-ячейки соответственно для каждой из cons-ячеек пунктирного блока. CAR'ы каждой новой ячейки ссылаются на тот же объект, что и CAR'ы оригинальных cons-ячеек в списочной структуре. Таким образом, COPY-LIST не копирует подсписки (1, 2), (3 4), или (5 6), как показано на этой диаграмме:

{{ pcl:chapter13:ch13-2.png }}

COPY-TREE, с другой стороны, создает новые cons-ячейки для каждой из cons-ячеек на диаграмме и соединяет их вместе в одну структуру, как показано на следующей диаграмме:

{{ pcl:chapter13:ch13-3.png }}

Там где cons-ячейки в оригинале ссылаются на атомарное значение, соответствующие им cons-ячейки в копии будут ссылаться на то же значение. Таким образом, единственными объектами, на которые ссылаются как оригинальное дерево, так и его копия, созданная COPY-TREE, будут

числа 5, 6, и символ NIL.

Еще одна функция, которая обходит как CAR'ы так и CDR'ы cons-ячеек дерева это TREE-EQUAL, которая сравнивает два дерева, и считает их равными, если их структуры имеют одинаковую форму и если их листья равны относительно EQL (или если они удовлетворяют условию, задаваемому через именованный аргумент :test).

Прочими ориентированными на деревья функциями являются работающие с деревьями аналоги функций для последовательностей SUBSTITUTE и NSUBSTITUTE, и их -IF и -IF-NOT варианты. Функция SUBST, как и SUBSTITUTE, принимает новый элемент, старый элемент и дерево (в отличие от последовательности), вместе с именованными аргументами :key и :test, и возвращает новое дерево с той же формой, что и исходное, но все вхождения старого элемента заменяются новым. Например:

```
CL-USER> (subst 10 1 '(1 2 (3 2 1) ((1 1) (2 2))))  
(10 2 (3 2 10) ((10 10) (2 2)))
```

SUBST-IF аналогична SUBSTITUTE-IF. Но вместо старого элемента, она принимает одноаргументную функцию, которая вызывается с аргументом в виде каждого атомарного значения в дереве, и всякий раз, когда она возвращает истину, текущая позиция в новом дереве заменяется новым значением. SUBST-IF-NOT действует также, за исключением того, что заменяются те значения, где функция возвращает NIL. NSUBST, NSUBST-IF, и NSUBST-IF-NOT - это утилизирующие аналоги соответствующих версии SUBST-функций. Как и с другими утилизирующими функциями, вам следует использовать эти функции только как временную FIXME (drop-in - небезопасную?) замену их недеструктивных аналогов, в ситуациях, где вы уверены, что нет опасности повреждения разделяемой структуры. В частности, вы должны продолжать сохранять возвращаемые значения этих функции пока у вас нет гарантии, что результат будет равен по предикату EQ оригинальному дереву.

Множества

Множества также могут быть реализованы посредством cons-ячеек. Фактически, вы можете обращаться с любым списком как с множеством - Common Lisp предоставляет несколько функций для выполнения теоретико-множественных операций над списками. Тем не менее вам следует помнить, что из-за особенности устройства списков эти операции становятся тем менее и менее эффективными, чем больше становится множество.

Впрочем, используя встроенные функции для работы с множествами легко писать оперирующий с множествами код. И для небольших множеств они могут быть более эффективными, чем их альтернативы. Если профилирование показывает вам, что производительность этих функций является узким местом вашего кода, вы всегда можете заменить множества на основе списков множествами, основанными на хэш-таблицах или битовых векторах.

Для создания множества вы можете использовать функцию ADJOIN. ADJOIN принимает в качестве аргументов элемент и список, представляющий множество, и возвращает список, содержащий этот элемент вместе со всеми элементами исходного множества. Для того, чтобы определить, присутствует ли элемент в множестве, функция просматривает список, и если элемент не найден, то ADJOIN создает новую cons-ячейку, содержащую этот элемент, ссылающуюся на исходный список, и возвращает ее. В противном случае она возвращает исходный список.

ADJOIN также принимает именованные аргументы :key и :test, которые используются при определении - присутствует ли элемент в исходном списке. Подобно CONS, ADJOIN не воздействует на исходный список - если вы хотите модифицировать определенный список, вам

нужно присвоить значение, возвращаемое ADJOIN, тому FIXME месту где находился исходный список. Деструктивный макрос PUSHNEW сделает это для вас автоматически.

```
CL-USER> (defparameter *set* ())
*SET*
CL-USER> (adjoin 1 *set*)
(1)
CL-USER> *set*
NIL
CL-USER> (setf *set* (adjoin 1 *set*))
(1)
CL-USER> (pushnew 2 *set*)
(2 1)
CL-USER> *set*
(2 1)
CL-USER> (pushnew 2 *set*)
(2 1)
```

Вы можете проверить, принадлежит ли данный элемент множеству с помощью функции MEMBER и родственных ей функций MEMBER-IF и MEMBER-IF-NOT. Эти функции похожи на функции для работы с последовательностями - FIND, FIND-IF, и FIND-IF-NOT, за исключением того, что они используются только со списками. Вместо того, чтобы вернуть элемент, когда он присутствует в множестве, они возвращают cons-ячейку содержащую элемент - другими словами, подсписок начинающийся с заданного элемента. Если искомый элемент отсутствует в списке, все три функции возвращают NIL.

Оставшиеся ориентированные на множества функции предоставляют операции с группами элементов: INTERSECTION, UNION, SET-DIFFERENCE, и SET-EXCLUSIVE-OR. Каждая из этих функций принимает два списка и именованные аргументы :key и :test и возвращает новый список, представляющий множество, полученное выполнением соответствующей операции над двумя списками. INTERSECTION возвращает список, содержащий все аргументы из обоих списков. UNION возвращает список, содержащий один экземпляр каждого уникального элемента из двух своих аргументов. SET-DIFFERENCE возвращает список, содержащий все элементы из первого аргумента, которые не встречаются во втором аргументе. И SET-EXCLUSIVE-OR возвращает список, содержащий элементы, находящиеся только в одном либо в другом списках, переданных в качестве аргументов, но не в обоих одновременно. Каждая из этих функций также имеет утилизирующий аналог, имя которого получается добавлением N в качестве префикса.

В заключение, функция SUBSETP принимает в качестве аргументов два списка и обычные именованные аргументы :key и :test и возвращает истину, если первый список является подмножеством второго - то есть, если каждый элемент первого списка также присутствует во втором списке. Порядок элементов в списках значения не имеет.

```
CL-USER> (subsetp '(3 2 1) '(1 2 3 4))
T
CL-USER> (subsetp '(1 2 3 4) '(3 2 1))
NIL
```

Таблицы поиска: ассоциативные списки и списки свойств

Помимо деревьев и множеств вы можете создавать таблицы, которые отображают ключи на значения вне cons-ячеек. Обычно используются две разновидности основанных на cons-ячейках таблиц поиска, об обеих я вскользь упоминал в предыдущих главах. Это *ассоциативные списки*

(*association lists* или *alists*) и *списки свойств* (*property lists* или *plists*). Вам не стоит использовать списки свойств или ассоциативные списки для больших таблиц - для них нужно использовать хэш-таблицы, но стоит знать, как работать с ними обоими, так как для небольших таблиц они могут быть более эффективны, чем хэш-таблицы, и еще потому, что у них есть несколько полезных собственных свойств. Ассоциативный список - это структура данных, которая отображает ключи на значения, а также поддерживает обратный поиск, находя ключ по заданному значению. Ассоциативные списки также поддерживают возможность добавления отображений ключ/значение, которые скрывают существующие отображения таким образом, что скрывающие отображения могут быть позже удалены и первоначальные отображения снова станут видимы.

Если смотреть глубже, то на самом деле ассоциативный список - это просто список, чьи элементы сами являются cons-ячейками. Каждый элемент можно представлять как пару ключ/значение с ключом в CAR cons-ячейки и значением в CDR. К примеру, следующая стрелочная диаграмма представляет ассоциативный список, состоящий из отображения символа А в номер 1, В в номер 2, и С в номер 3:

```
{{ pcl:chapter13:ch13-4.png }}
```

До тех пор, пока значение CDR является списком, cons-ячейки представляющие пары ключ/значение будут *точечными парами* в терминах s-выражений. Ассоциативный список, представленный на предыдущей диаграмме, к примеру, будет напечатан вот так:

```
((A . 1) (B . 2) (C . 3))
```

Главная процедура поиска для ассоциативных списков это ASSOC, которая принимает ключ и ассоциативный список в качестве аргументов, и возвращает первую cons-ячейку, чей CAR соответствует ключу или является NIL, если совпадения не найдено.

```
CL-USER> (assoc 'a '((a . 1) (b . 2) (c . 3)))  
(A . 1)  
CL-USER> (assoc 'c '((a . 1) (b . 2) (c . 3)))  
(C . 3)  
CL-USER> (assoc 'd '((a . 1) (b . 2) (c . 3)))  
NIL
```

Чтобы получить значение, соответствующее заданному ключу, вам следует просто передать результат ASSOC CDR'y.

```
CL-USER> (cdr (assoc 'a '((a . 1) (b . 2) (c . 3))))  
1
```

По умолчанию заданный ключ сравнивается с ключами в ассоциативном списке, используя предикат EQL, но вы можете изменить его с помощью стандартной комбинации из именованных аргументов :key и :test. Например, если вы хотите использовать строковые ключи, то можете написать так:

```
CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)) :test #'string=)  
("a" . 1)
```

Без явного задания в качестве :test предиката STRING= ASSOC вероятно вернул бы NIL, потому что две строки с одинаковым содержимым необязательно равны относительно EQL.

```
CL-USER> (assoc "a" '(("a" . 1) ("b" . 2) ("c" . 3)))  
NIL
```

Поскольку при поиске ASSOC просматривает список с начала, одна пара ключ/значение в ассоциативном списке может скрывать другие пары с тем же ключом, но находящиеся дальше в списке.

```
CL-USER> (assoc 'a '((a . 10) (a . 1) (b . 2) (c . 3)))  
(A . 10)
```

Вы можете добавить пару в начало списка с помощью функции CONS, как здесь:

```
(cons (cons 'new-key 'new-value) alist)
```

Однако для удобства Common Lisp предоставляет функцию ACONS, которая позволяет вам написать так:

```
(acons 'new-key 'new-value alist)
```

Подобно CONS, ACONS является функцией и, следовательно, не может модифицировать место, откуда был передан исходный ассоциативный список. Если вы хотите модифицировать ассоциативный список, вам нужно написать так:

```
(setf alist (acons 'new-key 'new-value alist))
```

или так:

```
(push (cons 'new-key 'new-value) alist)
```

Очевидно, время затраченное на поиск в ассоциативном списке при использовании ASSOC, является функцией от того, насколько глубоко в списке находится соответствующая пара. В худшем случае для определения, что никакая пара не соответствует искомой, ASSOC требуется просмотреть каждый элемент списка. Тем не менее, поскольку основной механизм работы ассоциативных списков довольно прост, то на небольших таблицах ассоциативный список может превзойти в производительности хэш-таблицу. Также ассоциативные списки могут дать вам большую гибкость при выполнении поиска. Ранее я отмечал, что ASSOC принимает именованные аргументы :key и :test. Если они не соответствуют вашим требованиям, вы можете использовать функции ASSOC-IF и ASSOC-IF-NOT, которые возвращают первую пару ключ/значение, чей CAR удовлетворяет (или не удовлетворяет, в случае ASSOC-IF-NOT) предикату, передаваемому вместо ключа. Еще три функции - RASSOC, RASSOC-IF, и RASSOC-IF-NOT действуют так же, как и соответствующие аналогичные ASSOC-функции, за исключением того, что они используют значение в CDR каждого элемента как ключ, совершая обратный поиск.

Функция COPY-ALIST похожа на COPY-TREE за исключением того, что вместо копирования всей древовидной структуры, она копирует только те cons-ячейки, которые составляют списочную структуру, плюс те cons-ячейки, на которые ссылаются CAR'ы этих ячеек. Другими словами, исходный ассоциативный список и его копия будут оба содержать одинаковые объекты как в виде ключей, так и значений, даже если ключи или значения будут состоять из cons-ячеек.

Наконец вы можете создать ассоциативный список из двух различных списков ключей и значений с помощью функции PAIRLIS. Получившийся ассоциативный список может содержать пары

либо в том порядке, в каком они были в исходных списках, либо в обратном порядке. Например, вы можете получить такой результат:

```
CL-USER> (pairlis '(a b c) '(1 2 3))  
((C . 3) (B . 2) (A . 1))
```

Или вы можете получить следующее:

```
CL-USER> (pairlis '(a b c) '(1 2 3))  
((A . 1) (B . 2) (C . 3))
```

Другой разновидностью таблицы поиска является список свойств (property list или сокращенно plist), который вы использовали для представления строк базы данных в Главе 3. Структурно список свойств есть просто обычный список с ключами и значениями в виде чередующихся величин. К примеру, список свойств отображающий A, B, и C, на 1, 2, и 3 это просто список (A 1 B 2 C 3). На стрелочной диаграмме он выглядит так:

{{ pcl:chapter13:ch13-5.png }}

Впрочем списки свойств является менее гибкими чем ассоциативные списки. В действительности список свойств поддерживает только одну фундаментальную операцию поиска, функцию GETF, которая принимает список свойств и ключ и возвращает связанное с ключом значение или NIL, если ключ не найден. GETF также принимает необязательный третий аргумент, который возвращается вместо NIL, если ключ не найден.

В отличие от ASSOC, которая использует EQ как проверочный предикат по умолчанию и позволяет использовать другой проверочный предикат в виде именованного аргумента :test, GETF всегда использует EQ для проверки, совпадает ли переданный ей ключ с ключами списка свойств. Следовательно, вам никогда не следует использовать числа или знаки в качестве ключей в списке свойств; как вы видели в Главе 4, поведение EQ для этих типов данных фактически не определено. На практике, ключи в списке свойств почти всегда являются символами, с тех пор как списки свойств были впервые изобретены для реализации "свойств" символов, то есть произвольных отображений между именами и значениями.

Вы можете использовать SETF вместе с GETF для установки переменной с заданным ключом. SETF также обращается с GETF немного особым образом, при котором первый аргумент GETF считается модифицируемым. Таким образом вы можете вызвать SETF поверх GETF для добавления новых пар ключ/значение к существующему списку свойств.

```
CL-USER> (defparameter *plist* ())  
*PLIST*  
CL-USER> *plist*  
NIL  
CL-USER> (setf (getf *plist* :a) 1)  
1  
CL-USER> *plist*  
(:A 1)  
CL-USER> (setf (getf *plist* :a) 2)  
2  
CL-USER> *plist*  
(:A 2)
```

Чтобы удалить пару ключ/значение из списка свойств, вы можете использовать макрос REMF, который присваивает месту, переданному в качестве своего первого аргумента, список свойств, содержащий все пары ключ/значение, за исключением заданной. Он возвращает истину если

заданный ключ был найден.

```
CL-USER> (remf *plist* :a)
T
CL-USER> *plist*
NIL
```

Как и GETF, REMF всегда использует EQ для сравнения заданного ключа с ключами в списке свойств.

Поскольку списки свойств часто используются в ситуациях, когда вы хотите извлечь несколько значений свойств из одного и того же списка, Common Lisp предоставляет функцию GET-PROPERTIES, которая делает более эффективным извлечение нескольких значений из одного списка свойств. Она принимает список свойств и список ключей для поиска, и возвращает, в виде множества значений (*multiple values*), первый найденный ключ, соответствующее ему значение, и голову списка, начинающегося с этого ключа. Это позволяет вам обработать список свойств, извлекая из него нужные свойства, без продолжительного повторного поиска с начала списка. К примеру, следующая функция эффективно обрабатывает, используя гипотетическую функцию process-property, все пары ключ/значение в списке свойств для заданного списка ключей:

```
(defun process-properties (plist keys)
  (loop while plist do
    (multiple-value-bind (key value tail) (get-properties plist keys)
      (when key (process-property key value))
      (setf plist (cddr tail)))))
```

Последней особенностью списков свойств является их отношение к символам: каждый символический объект имеет связанный список свойств, который может быть использован для хранения информации о символе. Список свойств может быть получен с помощью функции SYMBOL-PLIST. Тем не менее, обычно вам редко необходим весь список свойств; чаще вы будете использовать функцию GET, которая принимает символ и ключ и выполняет роль сокращенной записи для GETF с аргументами в виде того же ключа и списка символов, возвращаемого SYMBOL-PLIST.

```
(get 'symbol 'key) == (getf (symbol-plist 'symbol) 'key)
```

Как с GETF, к возвращаемому значению GET можно применить SETF, так что вы можете присоединить произвольную информацию к символу, как здесь:

```
(setf (get 'some-symbol 'my-key) "information")
```

Чтобы удалить свойство из списка свойств символа, вы можете использовать либо REMF поверх SYMBOL-PLIST или удобную функцию REMPROP

```
(remprop 'symbol 'key) == (remf (symbol-plist 'symbol) 'key)
```

Возможность связывать произвольную информацию с именами довольно удобна, если вы занимаетесь какого-либо рода символическим программированием. К примеру, один макрос, который вы напишете в Главе 24, будет связывать информацию с именами, которые другие экземпляры того же макроса будут извлекать и использовать при генерации собственных расширений.

DESTRUCTURING-BIND

Последний инструмент для разделки и нарезки списков, о котором я должен рассказать, поскольку он понадобится вам в дальнейших главах - это макрос `DESTRUCTURING-BIND`. Этот макрос предоставляет способ *FIXME* *деструктурировать* произвольные списки, подобно тому, как списки параметров макроса могут разбирать на части свои списки аргументов. Основной скелет `DESTRUCTURING-BIND` таков:

```
(destructuring-bind (parameter*) list
  body-form*)
```

Список параметров может включать любые из типов параметров, поддерживаемых в списках параметров макросов, таких как `&optional`, `&rest`, и `&key`. Как и в списке параметров макроса, любой параметр может быть заменен на вложенный деструктурирующий список параметров, который разделяет список на составные части, иначе список целиком был бы связан с замененным параметром. Форма *list* вычисляется один раз и возвращает список, который затем деструктурируется и соответствующие значения связываются с переменными в списке параметров. Затем после по порядку вычисляются все *body-form* с учетом значений связанных переменных. Вот несколько простых примеров:

```
(destructuring-bind (x y z) (list 1 2 3)
  (list :x x :y y :z z)) ==> (:X 1 :Y 2 :Z 3)
(destructuring-bind (x y z) (list 1 (list 2 20) 3)
  (list :x x :y y :z z)) ==> (:X 1 :Y (2 20) :Z 3)
(destructuring-bind (x (y1 y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) ==> (:X 1 :Y1 2 :Y2 20 :Z 3)
(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2 20) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) ==> (:X 1 :Y1 2 :Y2 20 :Z 3)
(destructuring-bind (x (y1 &optional y2) z) (list 1 (list 2) 3)
  (list :x x :y1 y1 :y2 y2 :z z)) ==> (:X 1 :Y1 2 :Y2 NIL :Z 3)
(destructuring-bind (&key x y z) (list :x 1 :y 2 :z 3)
  (list :x x :y y :z z)) ==> (:X 1 :Y 2 :Z 3)
(destructuring-bind (&key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z)) ==> (:X 3 :Y 2 :Z 1)
```

Единственный вид параметра, который вы можете использовать как с `DESTRUCTURING-BIND`, так и в списках параметров макросов, о котором я не упомянул в Главе 8, это параметр `&whole`. Если он указан, то располагается первым в списке параметров, и связывается со всей формой списка целиком. После параметра `&whole`, другие параметры могут появляться как обычно, и извлекать определенные части списка, как если бы параметр `&whole` отсутствовал. Пример использования `&whole` вместе с `DESTRUCTURING-BIND` выглядит так:

```
(destructuring-bind (&whole whole &key x y z) (list :z 1 :y 2 :x 3)
  (list :x x :y y :z z :whole whole))
==> (:X 3 :Y 2 :Z 1 :WHOLE (:Z 1 :Y 2 :X 3))
```

Вы будете использовать параметр `&whole` в одном из макросов, составляющих часть библиотеки для генерации HTML, которую вы разработаете в Главе 31. Однако мне нужно рассмотреть еще несколько вопросов, прежде чем вы приступите к ней. После двух глав довольно лисповских тем про `cons`-ячейки, вы можете перейти к более скучным вопросам о том, как работать с файлами и именами файлов.

14. Файлы и файловый ввод/вывод

Common Lisp предоставляет мощную библиотеку для работы с файловыми операциями. В этой главе я остановлюсь на нескольких основных операциях, относящихся к работе с файлами: чтение и запись файлов, а так же отображение структуры файловой системы. Возможности Common Lisp, предназначенные для данных операций, аналогичны тем, что предоставляют другие языки программирования. Для чтения и записи файлов Common Lisp предоставляет абстракцию потока ввода/вывода, а для манипулирования файловыми объектами в независимом от операционной системы формате - абстракцию файловых путей. Кроме того, Common Lisp предоставляет некоторое количество уникальных возможностей, например, такую как чтение и запись s-выражений.

Чтение данных из файлов

Основная операция работы с файловым вводом/выводом - чтение содержимого. Для того, чтобы получить поток, из которого вы можете прочитать содержимое файла, используется функция `OPEN`. По умолчанию, `OPEN` возвращает посимвольный поток ввода данных, который можно передать множеству функций, которые читают один или несколько символов текста: `READ-CHAR` читает одиночный символ; `READ-LINE` читает строку текста, возвращая ее как строку без символа конца строки; функция `READ` читает s-выражение, возвращая объект Lisp. Если работа с потоком завершена, то вы можете закрыть его с помощью функции `CLOSE`.

Функция `OPEN` требует имя файла как единственный обязательный аргумент. Как можно увидеть в секции "Имена файлов", Common Lisp предоставляет два пути для представления имени файла, но наиболее простой способ - использовать строку, содержащую имя в формате, используемом в файловой системе. Так, предполагая, что `/some/file/name.txt` это файл, возможно открыть его следующим образом:

```
(open "/some/file/name.txt")
```

Вы можете использовать объект, возвращаемый функцией как первый аргумент любой функции, осуществляющей чтение. Например, для того, чтобы напечатать первую строку файла, вы можете комбинировать `OPEN`, `READ-LINE`, `CLOSE` следующим образом:

```
(let ((in (open "/some/file/name.txt")))
  (format t "~a~%" (read-line in))
  (close in))
```

Конечно, при открытии и чтении данных может произойти ряд ошибок. Файл может не существовать, или вы можете непредвиденно достигнуть конца файла в процессе его чтения. По умолчанию, `OPEN` и `READ-*` будут сигнализировать об ошибках в данных ситуациях. В главе 19, я рассмотрю как обработать эти ошибки. Сейчас же, однако, будем использовать более легковесное решение: каждая из этих функций принимает аргументы, которые изменяют ее реакцию на исключительные ситуации.

Если вы хотите открыть файл, который возможно не существует без генерирования ошибки функцией `OPEN`, вы можете использовать аргумент `:if-does-not-exists` для того, чтобы указать другое поведение. Три различных значения допустимы для данного аргумента - `:error`, по умолчанию; `:create`, что указывает на необходимость создания файла и осуществить повторное его открытие второй раз как существующего и `NIL`, что означает возврат `NIL` при неуспешном открытии вместо потока. Итак, возможно изменить предыдущий пример таким образом, чтобы обработать несуществующий файл.

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (format t "~a~%" (read-line in))
    (close in)))
```

Все функции чтения - READ-CHAR, READ-LINE, READ - принимают вспомогательный аргумент, который имеет значение истины по умолчанию, указывающий, должны ли они сигнализировать об ошибке, если они достигли конца файла. Если этот аргумент установлен в NIL, то они возвращают значение их 3го аргумента, который по умолчанию NIL, вместо ошибки. Таким образом, вывести на печать все строки файла можно следующим способом:

```
(let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
          while line do (format t "~a~%" line))
    (close in)))
```

Среди трех вышерассмотренных функций READ – уникальна. Это – та самая функция, которая представляет букву "R" в "REPL", и которая используется для того, чтобы читать исходный код Lisp. Во время вызова она читает целое s-выражение, пропуская пробельные символы и комментарии, и возвращает объект Lisp, представляемый s-выражением. Например, предположим, что файл /some/file/name.txt имеет следующее содержимое:

```
(1 2 3)
456
"строка" ; это комментарий
((a b)
 (c d))
```

Другими словами, он содержит 4 s-выражения: список чисел, число, строку, и список списков. Вы можете читать эти выражения следующим образом:

```
CL-USER> (defparameter *s* (open "/some/file/name.txt"))
*S*
CL-USER> (read *s*)
(1 2 3)
CL-USER> (read *s*)
456
CL-USER> (read *s*)
"строка"
CL-USER> (read *s*)
((A B) (C D))
CL-USER> (close *s*)
T
```

Как было показано в Главе 3, возможно использовать PRINT для того, чтобы выводить объекты Lisp на печать в удобной для прочтения форме. Итак, когда необходимо хранить данные в файлах, PRINT и READ предоставляют простой способ делать это без создания специального формата данных и парсера для их прочтения. Вы даже можете использовать комментарии без ограничений. И, поскольку s-выражения создавались для того, чтобы быть редактируемыми людьми, то они так же хорошо подходят для использования в качестве формата конфигурационных файлов .

Чтение двоичных данных

По умолчанию OPEN возвращает символьные потоки, которые преобразуют байты в символы в соответствии с конкретной схемой кодирования символов .

Для чтения потока байтов необходимо передать функции OPEN ключевой параметр :element-type со значением '(unsigned-byte 8) Полученный поток можно передать функции READ-BYTE, которая будет возвращать целое число от 0 до 255 во время каждого вызова. READ-BYTE, так же, как и функции, работающие с потоками символов, принимает опциональные аргументы, которые указывают, должна ли она сигнализировать об ошибке, если достигнут конец файла, и какое значение возвращать в противном случае. В главе 24 мы построим библиотеку, которая позволит удобно читать структурированные бинарные данные, используя READ-BYTE.

Блочное чтение

Последняя функция для чтения, READ-SEQUENCE, работает с бинарными и символьными потоками. Ей передается последовательность (обычно вектор) и поток, и она пытается заполнить последовательность данными из потока. Функция возвращает индекс первого элемента последовательности, который не был заполнен, либо ее длину, если она была заполнена полностью. Так же возможно передать ключевые аргументы :start и :end, которые указывают на подпоследовательность, которая должна быть заполнена вместо последовательности. Аргумент, определяющий последовательность должен быть типом, который может хранить элементы, определенного для потока типа. Поскольку большинство операционных систем поддерживают какую-либо форму блочных операций ввода/вывода, READ-SEQUENCE скорее всего более эффективна чем чтение последовательных данных несколькими вызовами READ-BYTE или READ-CHAR.

Файловый вывод

Для записи данных в файл необходим поток вывода, который можно получить вызовом функции OPEN с ключевым аргументом :direction :output. Когда файл открывается для записи OPEN предполагает что файл не должен существовать и будет сообщать об ошибке в противном случае. Однако, возможно изменить это поведение с помощью ключевого аргумента :if-exists. Передавая значение :supersede можно вызвать замену существующего файла. Значение :append позволяет осуществлять запись таким образом, что новые данные будут помещены в конец файла, а значение :overwrite возвращает поток, который будет переписывать существующие данные с начала файла. Если же передать NIL, то OPEN вернет NIL вместо потока, если файл уже существует. Характерное использование OPEN для вывода данных выглядит следующим образом:

```
(open "/some/file/name.txt" :direction :output :if-exists :supersede)
```

Common Lisp также предоставляет некоторые функции для записи данных: WRITE-CHAR пишет одиночный символ в поток. WRITE-LINE пишет строку, за которой следует символ конца строки, с учетом реализации для конкретной платформы. Другая функция, WRITE-STRING пишет строку, не добавляя символ конца строки. Две разные функции могут использоваться для того чтобы вывести символ конца строки: TERP - сокращение для "TERminate PRint" (закончить печать) безусловно печатает символ конца строки, а FRESH-LINE печатает символ конца строки только в том случае, если текущая позиция печати не совпадает с началом строки. FRESH-LINE удобна в том случае, когда желательно избежать паразитных пустых строк в текстовом выводе, генерируемом другими последовательно вызываемыми функциями. Например, допустим, что есть одна функция, которая генерирует вывод и после которой обязательно должен идти перенос строки и другая, которая должна начинаться с новой строки. Но, предположим, что если функции вызываются последовательно, то необходимо обеспечить

отсутствие лишних пустых строк в их выводе. Если в начале второй функции используется `FRESH-LINE`, ее вывод будет постоянно начинать с новой строки, но если она вызывается непосредственно после первой функции, то не будет образовываться дополнительной пустой строки.

Некоторые функции позволяют вывести данные Lisp в форме s-выражений: `PRINT` печатает s-выражение, предваряя его символом начала строки, и пробельным символом после. `PRIN1` печатает только s-выражение. И функция `PPRINT` печатает s-выражения как `PRINT` и `PRIN1`, но используя "красивую печать", которая пытается печатать s-выражения в удобном для восприятия виде.

Но не все объекты могут быть напечатаны в том формате, который понимает `READ`. Переменная `*PRINT-READABLY*` контролирует поведение при попытке напечатать какой-либо подобный объект с помощью `PRINT`, `PRIN1` или `PPRINT`. Когда она равна `NIL`, эти функции напечатают объект в таком формате, что `READ` сообщит об ошибке при попытке чтения; в ином случае они просигнализируют об ошибке вместо того, чтобы напечатать объект.

Еще одна функция, `PRINC`, также печатает объекты Лиспа, но в виде, удобном для человеческого восприятия. Например, `PRINC` печатает строки без кавычек. Текстовый вывод может быть еще более гибким и искусным, если задействовать — отчасти загадочную — функцию `FORMAT`. Я расскажу о некоторых важных деталях этой функции, которая, в общем-то, определяет мини-язык для форматированного вывода, в 18 главе.

Для того, чтобы записать двоичные данные в файл, следует открыть файл функцией `OPEN` с тем же самым аргументом `:element-type`, который использовался при чтении данных: `'(unsigned-byte 8)`. После этого можно записывать в поток отдельные байты функцией `WRITE-BYTE`.

Функция блочного вывода `WRITE-SEQUENCE` принимает как двоичный, так и символьный потоки до тех пор, пока все элемента последовательности подходящего типа: либо символы, либо байты. Так же как и `READ-SEQUENCE`, эта функция наверняка более эффективна, чем запись одного элемента последовательности за один вызов функции.

Заккрытие файлов

Любой, кто писал программы, взаимодействующие с файлами, знает, что важно закрывать файлы, когда работа с ними закончена, так как дескрипторы норовят быть дефицитным ресурсом. Если открывают файлы и забывают их закрывать, вскоре обнаруживают, что больше нельзя открыть ни одного файла. На первый взгляд может показаться, что достаточно каждый вызов `OPEN` сопоставить с вызовом `CLOSE`. Например, можно всегда обрамлять код, использующий файл, как показано ниже:

```
(let ((stream (open "/some/file/name.txt")))  
  ;; работа с потоком  
  (close stream))
```

Однако этот метод имеет две проблемы. Первая — он предрасположен к ошибкам: если забыть написать `CLOSE`, то будет происходить утечка дескрипторов при каждом вызове этого кода. Вторая — наиболее значительная — нет гарантии, что `CLOSE` будет достигнут. Например, если в коде, расположенном до `CLOSE`, есть `RETURN` или `RETURN-FROM`, возвращение из `LET` произойдет без закрытия потока. Или, как вы увидите в 19 главе, если какая-либо часть кода до `CLOSE` сигнализирует об ошибке, управление может перейти за пределы `LET` обработчику ошибки и никогда не вернется, чтобы закрыть поток.

Common Lisp предоставляет общее решение того, как удостовериться, что определенный код всегда выполняется: специальный оператор `UNWIND-PROTECT`, о котором я расскажу в 20 главе. Так как открытие файла, работа с ним и последующее закрытие очень часто употребляются, Common Lisp предлагает макрос, `WITH-OPEN-FILE`, основанный на `UNWIND-PROTECT`, для скрывания этих действий. Ниже — основная форма:

```
(with-open-file (stream-var open-argument*)  
  body-form*)
```

Выражения в `body-forms*` вычисляются с `stream-var`, связанной с файловым потоком, открытым вызовом `OPEN` с аргументами `open-argument*`. `WITH-OPEN-FILE` удостоверяется, что поток `stream-var` закрывается до того, как из `WITH-OPEN-FILE` вернется управление. Поэтому читать файл можно следующим образом:

```
(with-open-file (stream "/some/file/name.txt")  
  (format t "~a~%" (read-line stream)))
```

Создать файл можно и так:

```
(with-open-file (stream "/some/file/name.txt" :direction :output)  
  (format stream "Какой-то текст."))
```

Как правило, `WITH-OPEN-FILE` используется в 90-99 процентах файлового ввода/вывода. Вызовы `OPEN` и `CLOSE` понадобятся, если файл нужно открыть в какой-либо функции и оставить поток открытым при возврате из нее. В любом случае, нужно позаботиться о закрытии потока впоследствии, или произойдет утечка файловых дескрипторов и, в конце концов, вы больше не сможете открыть ни одного файла.

Имена файлов

До сих пор мы использовали строки для представления имен файлов. Однако, использование строк как имен файлов привязывает код к конкретной операционной и файловой системам. Точно так же, если конструировать имена в соответствии правилам конкретной схемы именования (скажем, разделение директорий знаком `"/`), то вы также привязаны к одной определенной файловой системе.

Для того, чтобы избежать подобной непереносимости программ, Common Lisp предоставляет другое представление файловых имен: объекты файловых путей. Файловые пути представляют собой файловые имена в структурированном виде, что делает их использование легким без привязки к определенному синтаксису файловых имен. А бремя по переводу между строками в локальном представлении — строками имен — и файловыми путями ложится на плечи реализаций Lisp.

К несчастью, как и со многими абстракциями, спроектированных для скрывания деталей различных базовых систем, абстракция файловых путей приносит свои трудности. В то время, когда разрабатывались файловые пути, множество файловых систем было значительно больше, чем сегодня. Но это мало проясняет ситуацию, если все, о чем вы заботитесь, — представление имен файлов в Unix или Windows. Однако однажды поняв какие части этой абстракции можно забыть, как артефакты истории развития файловых путей, вы сможете ловко управлять именами файлов.

Как правило, когда возникает необходимость в файловом имени, вы можете использовать как строку имени (`namestring`), так и файловый путь. Выбор зависит от того, откуда произошло имя.

Файловые имена, предоставленные пользователем — например, как аргументы или строки конфигурационного файла — как правило, будут строками имен, так как пользователь знает какая операционная система у него запущена, поэтому не следует ожидать, что он будет беспокоиться о представлении файловых имен в Lisp. Но следуя общепринятой практике, файловые имена будут представлены файловыми путями, так как они переносимы. Поток, который возвращает `OPEN`, также представляет файловое имя, а именно, файловое имя, которое было изначально использовано для открытия этого потока. Вместе эти три типа упоминаются как указатели файловых путей. Все встроенные функции, ожидающие файловое имя как аргумент, принимают все три типа указателя файловых путей. Например, во всех предыдущих случаях, когда вы использовали строку для представления файлового имени, вы также могли передавать в функцию объект файлового пути или поток.

Как мы до этого докатились

Историческое разнообразие файловых систем, существующих в период 70-80 годов, можно легко забыть. Кент Питман, один из ведущих технических редакторов стандарта Common Lisp, описал однажды ситуацию в `comp.lang.lisp` (Message-ID: `sfwzo74np6w.fsf@world.std.com`) так:

В момент завершения проектирования Common Lisp господствующими файловыми системами были TOPS-10, TENEX, TOPS-20, VAX VMS, AT&T Unix, MIT Multics, MIT ITS, и это не упоминаю группу систем для мэйнфреймов. В некоторых системах имена файлов были только в верхнем регистре, в других — смешанные, в третьих — чувствительны к регистру, но с возможностью преобразования (как в CL). Какие-то имели групповые символы (wildcards), какие-то — нет. Одни имели `:vверх (:up)` в относительных файловых путях, другие — нет. Также существовали файловые системы без каталогов, файловые системы без иерархической структуры каталогов, файловые системы без типов файлов, файловые системы без версий, файловые системы без устройств и т.д.

Если сейчас посмотреть на абстракцию файловых путей с точки зрения какой-нибудь определенной файловой системы, она выглядит нелепо. Но если взять в рассмотрении даже такие две похожие файловые системы, как в Windows и Unix, то вы можете заметить отличия, от которых можно отвлечься с помощью системы файловых путей. Файловые имена в Windows содержат букву диска в то время, как в Unix нет. Другое преимущество владения абстракцией файловых путей, которая спроектирована, чтобы оперировать большим разнообразием файловых систем, которые существовали в прошлом, — ее вероятная способность управлять файловыми системами, которые будут существовать в будущем. Если, скажем, файловые системы с сохранением всех старых данных и истории операций войдут снова в моду, Common Lisp будет к этому готов.

Как имена путей представляют имена файлов

Файловый путь — это структурированный объект, который представляет файловое имя, используя шесть компонентов: хост, устройство, каталог, имя, тип и версия. Большинство из них принимают атомарные значения, как правило, строки; только директория — структурный компонент, содержащий список имен каталогов (как строки) с предшествующим ключевым словом: `:absolute` (абсолютный) или `:relative` (относительный). Но не все компоненты необходимы на все платформах — это одна из тех вещей, которая вселяет страх в начинающих лисперов, потому что необоснованно сложна. С другой стороны, вам не надо заботиться о том, какие компоненты могут или нет использоваться для представления имен на определенной файловой системе, если только вам не надо создать объект файлового пути с нуля, а это почти никогда и не надо. Взамен, вы обычно получите объект файлового пути, либо позволив реализации преобразовать строку имени, специфичной для какой-то файловой системы, в объект файлового пути, либо создав файловый путь, который перенимает большинство компонент от

какого-либо существующего.

Например, для того, чтобы преобразовать строку имени в файловый путь, используйте функцию `PATHNAME`. Она принимает на вход указатель файлового пути и возвращает эквивалентный объект файлового пути. Если указатель уже является файловым путем, то он просто возвращается. Если это поток, извлекается первоначальное файловое имя и возвращается. Если это строка имени, она анализируется согласно локальному синтаксису файловых имен. Стандарт языка как независимый от платформы документ не определяет какое-либо конкретное отображение строки имени в файловый путь, но большинство реализаций следуют одним и тем же соглашениям на данной операционной системе.

На файловых системах Unix, обычно, используются компоненты директория, имя и тип. В Windows на один компонент больше – обычно устройство или хост – содержит букву диска. На этих платформах строку имени делят на части, а разделителем служит косая черта в Unix и косая или обратная косая черты в Windows. Букву диска в Windows размещают либо в компонент устройства или компонент хост. Все, кроме последнего из оставшихся элементов имени, размещаются в списке, начинающемся с `:absolute` или `:relative` в зависимости от того, начинается ли имя с разделителя или нет (игнорирую букву диска, если таковая присутствует). Список становится компонентом каталог файлового пути. Последний элемент делится по самой крайней точке, если она есть, и полученные две части есть компоненты имя и тип, соответственно.

Можно проверить каждый компонент файлового пути с функциями `PATHNAME-DIRECTORY`, `PATHNAME-NAME` и `PATHNAME-TYPE`.

```
(pathname-directory (pathname "/foo/bar/baz.txt")) → (:ABSOLUTE "foo" "bar")  
(pathname-name (pathname "/foo/bar/baz.txt")) → "baz"  
(pathname-type (pathname "/foo/bar/baz.txt")) → "txt"
```

Другие три функции – `PATHNAME-HOST`, `PATHNAME-DEVICE` и `PATHNAME-VERSION` – позволяют получить остальные три составляющие файлового пути, хотя они и не представляют интереса в Unix. В Windows либо `PATHNAME-HOST`, либо `PATHNAME-DEVICE` возвратит букву диска.

Подобно другим встроенным объектам, файловые пути обладают своим синтаксисом для чтения: `#p`, за которым следует строка, заключенная в двойные кавычки. Это позволяет печатать и считывать `s`-выражения, содержащие объекты файлового пути, но так как синтаксис зависит от алгоритма анализа строки, эти данные могут быть непереносимыми между разными операционными системами.

```
(pathname "/foo/bar/baz.txt") → #p"/foo/bar/baz.txt"
```

Для того, чтобы файловый путь преобразовать обратно в строку имени – например, чтобы представить его пользователю – следует воспользоваться функцией `NAMESTRING`, которая принимает указатель файлового пути и возвращает строку имени. Две других функции – `DIRECTORY-NAMESTRING` и `FILE-NAMESTRING` – возвращают часть строки имени. `DIRECTORY-NAMESTRING` соединяет элементы компонента каталог в локальное имя каталога. `FILE-NAMESTRING` – компоненты имя и тип.

```
(namestring #p"/foo/bar/baz.txt") → "/foo/bar/baz.txt"  
(directory-namestring #p"/foo/bar/baz.txt") → "/foo/bar/"  
(file-namestring #p"/foo/bar/baz.txt") → "baz.txt"
```

Конструирование имен путей

Вы можете создать файловый путь, используя функцию `MAKE-PATHNAME`. Она принимает по одному аргументу-ключу на каждую компоненту файлового пути и возвращает файловый путь, заполненный всеми предоставленными компонентами.

```
(make-pathname
 :directory '(:absolute "foo" "bar")
 :name "baz"
 :type "txt") → #p"/foo/bar/baz.txt"
```

Однако, если вы желаете, чтобы ваши программы были переносимыми, то врядли вы пожелаете создавать файловые пути с нуля, даже если абстракция файловых путей предохраняет вас от непереносимого синтаксиса файловых имен, ведь файловые имена могут быть непереносимыми еще множеством способов. Например, файловое имя `"/home/peter/foo.txt"` не очень-то подходит для OS X, в которой `/home/` представлено `/Users/`.

Другой причиной, по которой не следует создавать файловые пути с нуля, является тот факт, что различные реализации используют компоненты файлового пути с небольшими различиями. Например, как было упомянуто выше, некоторые Windows-реализации LISP хранят букву диска в компоненте устройство в то время, как другие – в компоненте хост. Если вы напишите:

```
(make-pathname :device "c" :directory '(:absolute "foo" "bar") :name "baz")
```

то это может быть правильным при использовании одной реализации, но не другой.

Вместо того, чтобы создавать пути с нуля, проще создать новый файловый путь, используя существующий файловый путь, при помощи аргумента-ключа `:defaults` функции `MAKE-PATHNAME`. С этим параметром можно предоставить указатель файлового пути, из которого будут взяты компоненты, не указанные другими аргументами. Для примера, следующее выражение создает файловый путь с расширением `.html` и компонентами из файлового пути `input-file`:

```
(make-pathname :type "html" :defaults input-file)
```

Предполагая, что значение `input-file` было предоставлено пользователем, этот код – надежен вопреки различиям операционных систем и реализаций, как-то: сожерждит ли файловый путь букву диска и где она хранится.

Используя эту же технику, можно создать файловый путь с другой компонентой директория.

```
(make-pathname :directory '(:relative "backups") :defaults input-file)
```

Однако этот код создаст файловый путь с компонентой директория, равной относительному пути `"backups/"`, безотносительно к любым другим компонентам файла `input-file`. Например:

```
(make-pathname :directory '(:relative "backups")
 :defaults #p"/foo/bar/baz.txt") → #p"backups/baz.txt"
```

Возможно, когда-нибудь вы захотите объединить двое файловых путей, один из которых имеет относительный компонент директория), путем комбинирования их компонент директория. Например, предположим, что имеется относительный файловый путь `#p"foo/bar.html"`, который вы хотите объединить с абсолютным файловым путем `#p"/www/html/"`, чтобы получить `#p"/www/html/foo/bar.html"`. В этом случае `MAKE-PATHNAME` не подойдет; то,

что вам надо, — MERGE-PATHNAMES.

MERGE-PATHNAMES принимает два файловых пути и соединяет их, заполняя при этом компоненты, которые в первом файловом пути равны NIL, соответствующими значениями из второго файлового пути. Это очень похоже на MAKE-PATHNAME, которая заполняет все неопределенные компоненты компонентами, предоставленными аргументом :defaults. Однако, MERGE-PATHNAMES особенно относится к компоненте директория: если директория первого файлового пути — относительна, то директорией окончательного файлового пути будет директория первого пути относительно директории второго. Так:

```
(merge-pathnames #p"foo/bar.html" #p"/www/html/") → #p"/www/html/foo/bar.html"
```

Второй файловый путь также может быть относительным. В этом случае окончательный путь также будет относительным.

```
(merge-pathnames #p"foo/bar.html" #p"html/") → #p"html/foo/bar.html"
```

Для того, чтобы обратить этот процесс, то есть получить файловый путь, который относителен определенной корневой директории, используйте полезную функцию ENOUGH-NAMESTRING.

```
(enough-namestring #p"/www/html/foo/bar.html" #p"/www/") → "html/foo/bar.html"
```

Вы можете соединить ENOUGH-NAMESTRING и MERGE-PATHNAMES для того, чтобы создать файловый путь, относительный другой корневой директории.

```
(merge-pathnames  
  (enough-namestring #p"/www/html/foo/bar/baz.html" #p"/www/")  
  #p"/www-backups/") → #p"/www-backups/html/foo/bar/baz.html"
```

MERGE-PATHNAMES используется стандартными функциями для доступа к файлам, чтобы дополнять незавершенные файловые пути. Например, пусть есть файловый путь, имеющий только компоненты имя и тип.

```
(make-pathname :name "foo" :type "txt") → #p"foo.txt"
```

Если вы попытаетесь передать этот файловый путь как аргумент функции OPEN, недостающие компоненты, как, например, директория, должны быть заполнены, чтобы Lisp смог преобразовать файловый путь в действительное файловое имя. Common Lisp добудет эти значения для недостающих компонент, объединяя данный файловый путь со значением переменной *DEFAULT-PATHNAME-DEFAULTS*. Начальное значение этой переменной определено реализацией, но, как правило, это файловый путь, компонент директория которого представляет ту директорию, в которой Lisp был запущен. Компоненты хост и устройство заполнены подходящими значениями, если в этом есть необходимость. Если MERGE-PATHNAMES вызвана только с одним аргументом, то она объединит аргумент со значением *DEFAULT-PATHNAME-DEFAULTS*. Например, если *DEFAULT-PATHNAME-DEFAULTS* — #p"/home/peter/", то в результате:

```
(merge-pathnames #p"foo.txt") → #p"/home/peter/foo.txt"
```

Два представления для имен директорий

Существует один неприятный момент при работе с файловым путем, который представляет

директорию. Файловые объекты разделяют компоненты директория и имя файла, но Unix и Windows рассматривают директории как еще один тип файла. Поэтому, в этих системах, каждая директория может иметь два различных представления.

Одно из них, которое я назову представлением файла, рассматривает директорию, как любой другой файл и размещает последний элемент строки имени в компоненты имя и тип. Другое представление — представление директории — помещает все элементы имени в компонент директория, оставляя компоненты имя и тип равными NIL. Если `/foo/bar/` — директория, тогда любой из следующих двух файловых путей представляет ее.

```
(make-pathname :directory '(:absolute "foo") :name "bar") ; file form
(make-pathname :directory '(:absolute "foo" "bar")) ; directory form
```

Когда вы создаете файловые пути с помощью `MAKE-PATHNAME`, вы можете получить любое из двух представлений, но нужно быть осторожным, когда имеете дело со строками имен. Все современные реализации Lisp создают представление файла, если только строка имени не заканчивается разделителем пути. Но вы не можете полагаться на то, что строки имени, предоставленные пользователем, будут в том либо ином представлении. Например, предположим, что вы запросили у пользователя имя директории, в которой сохраните файл. Пользователь ввел `"/home/peter"`. Если передать функции `MAKE-PATHNAME` эту строку как аргумент `:defaults`:

```
(make-pathname :name "foo" :type "txt" :defaults user-supplied-name)
```

то в конце концов вы сохраните файл как `/home/foo.txt`, а не `/home/peter/foo.txt`, как предполагалось, так как `"peter"` из строки имени будет помещен в компонент имя, когда `user-supplied-name` будет преобразовано в файловый путь. В переносимой библиотеке файловых путей, которую я обсужу в следующей главе, вы напишите функцию `pathname-as-directory`, которая преобразует файловый объект в представление директории. С этой функцией вы сможете сохранять наверняка файл в директории, указанной пользователем.

```
(make-pathname
 :name "foo" :type "txt" :defaults (pathname-as-directory user-supplied-name))
```

Взаимодействие с файловой системой

Хоть открытие файлов для чтения и записи — наиболее частый вид взаимодействия с файловой системой, порой вам захочется проверить существует ли файл, прочитать содержимое директории, удалить или переименовать файлы, создать директории или получить такую информацию о файле, как: кто его владелец, когда он последний раз был изменен, его длину. Тут-то общность абстракции файловых путей немного нехватает. Стандарт языка не определяет, как функции, взаимодействующие с файловой системой, отображаются на какую-то конкретную файловую систему, поэтому у создателей конкретных реализаций есть некоторая свобода действий.

Несмотря на это большинство функций для взаимодействия с файловой системой просты для понимания. Я расскажу о стандартных функциях и укажу на те, с которыми могут возникнуть проблемы при переносимости кода между реализациями. В следующей главе вы разработаете переносимую библиотеку файловых путей для того, чтобы сгладить эти проблемы с переносимостью.

Для того, чтобы проверить существует ли файл, соответствующий указателю файлового пути — будь-то файловый путь, строка имени или файловый поток, — можно использовать функцию

PROBE-FILE. Если файл, соответствующий указателю, существует, PROBE-FILE вернет "настоящее" имя файла – файловый путь с любыми преобразованиями уровня файловой системой, как, например, следование по символическим ссылкам. В ином случае, она вернет NIL. Однако, не все реализации позволяют использовать ее для того, что проверить существует ли директория. Также, Common Lisp не предоставляет переносимого способа определить, чем является существующий файл – обычным файлом или директорией. В следующей главе вы сделаете функцию-обертку для PROBE-FILE – file-exists-p, которая может проверить существует ли файл и ответить: данное имя есть имя файла или директории.

Так же стандартная функция для чтения списка файлов – DIRECTORY – работает прекрасно в незамысловатых случаях, но из-за различий реализаций ее использование становится мудреным делом. В следующей главе вы определите функцию чтения содержимого директорий, которая сгладит некоторые из различий.

DELETE-FILE и RENAME-FILE делают то, что следует из их названий. DELETE-FILE принимает указатель файлового пути и удаляет указанный файл, возвращая истину в случае успеха. В ином случае она сигнализирует FILE-ERROR.

RENAME-FILE принимает два указателя файлового пути и изменяет имя файла, указанного первым параметром, на имя, указанное вторым параметром.

Вы можете создать директории с помощью функции ENSURE-DIRECTORIES-EXIST. Она принимает указатель файлового пути и удостоверяется, что все элементы компонента директория существуют и являются директориями, создавая их при необходимости. Так как она возвращает файловый путь, который ей был передан, то удобно ее вызывать на месте аргумента другой функции.

```
(with-open-file (out (ensure-directories-exist name) :direction :output)
  ...
)
```

Обратите внимание, что если вы передаете ENSURE-DIRECTORIES-EXIST имя директории, то оно должно быть в представлении директории, или последняя директория не будет создана. Обе функции FILE-WRITE-DATE и FILE-AUTHOR принимают указатель файлового пути. FILE-WRITE-DATE возвращает количество секунд, которое прошло с полуночи 1-го января 1900 года, среднее время по Гринвичу (GMT), до времени последней записи в файл. FILE-AUTHOR возвращает – в Unix и Windows – владельца файла.

Для того, чтобы получить размер файла, используйте функцию FILE-LENGTH. По историческим причинам FILE-LENGTH принимает поток как аргумент, а не файловый путь. В теории это позволяет FILE-LENGTH вернуть размер в терминах типа элементов потока. Однако, так как на большинстве современных операционных системах единственная доступная информация о размере файла – исключая чтение всего файла, чтобы узнать размер – это размер в байтах, что возвращают большинство реализаций, даже если FILE-LENGTH передан символьный поток. Однако, стандарт не требует такого поведения, поэтому ради предсказуемых результатов лучший способ получить размер файла – использовать бинарный поток.

```
(with-open-file (in filename :element-type '(unsigned-byte 8))
  (file-length in))
```

Похожая функция, которая также принимает открытый файловый поток в качестве аргумента, – FILE-POSITION. Когда ей передают только поток, она возвращает текущую позицию в файле – количество элементов, прочитанных из потока или записанного в него. Когда ее вызывают с двумя аргументами, поток и указателем позиции, она устанавливает текущей позицией

указанную. Указатель позиции должен быть ключевым словом `:start`, `:end` или неотрицательное целое число. Два ключевых слова устанавливают позицию потока в начало или конец. Если же передать функции целое число, то позиция переместится в указанную позицию файла. В случае бинарного потока позиция – это просто смещение в байтах от начала файла. Однако, символьные потоки немного сложнее из-за проблем с кодировками. Лучшее, что вы можете сделать, если нужно переместиться на другую позицию в текстовом файле, – всегда передавать `FILE-POSITION` в качестве второго аргумента только значение, которое вернула функция `FILE-POSITION`, вызванная с тем же потоком в качестве единственного аргумента.

Другие операции ввода/вывода

Вдобавок к файловым потокам Common Lisp поддерживает другие типы потоков, которые также можно использовать с разнообразными функциями ввода/вывода для чтения, записи и печати. Например, можно считывать данные из строки или записывать их в строку, используя `STRING-STREAM`, которое вы можете создать функциями `MAKE-STRING-INPUT-STREAM` и `MAKE-STRING-OUTPUT-STREAM`.

`MAKE-STRING-INPUT-STREAM` принимает строку и необязательные начальный и конечный индексы, указывающие часть строки, которую следует связать с потоком, и возвращает символьный поток, который можно передать как аргумент любой функции символьного ввода, как, например, `READ-CHAR`, `READ-LINE` или `READ`. Например, если у вас есть строка, содержащая число с плавающей точкой с синтаксисом Common Lisp, вы можете преобразовать ее в число с плавающей точкой:

```
(let ((s (make-string-input-stream "1.23")))
  (unwind-protect (read s)
    (close s)))
```

`MAKE-STRING-OUTPUT-STREAM` создает поток, который можно использовать с `FORMAT`, `PRINT`, `WRITE-CHAR`, `WRITE-LINE` и т.д. Она не принимает аргументов. Что бы вы не записывали, строковый поток вывода будет накапливать в строке, которую потом можно получить с помощью функции `GET-OUTPUT-STREAM-STRING`. Каждый раз при вызове `GET-OUTPUT-STREAM-STRING` внутренняя строка потока очищается, поэтому существующий строковый поток вывода можно снова использовать.

Однако, использовать эти функции напрямую вы будете редко, так как макросы `WITH-INPUT-FROM-STRING` и `WITH-OUTPUT-TO-STRING` предоставляют более удобный интерфейс. `WITH-INPUT-FROM-STRING` похожа на `WITH-OPEN-FILE` – она создает строковый поток ввода на основе переданной строки и выполняет код в своем теле с потоком, который присвоен переменной, вами предоставленной. Например, вместо формы `LET` с явным использованием `UNWIND-PROTECT`, вероятно, лучше написать:

```
(with-input-from-string (s "1.23")
  (read s))
```

Макрос `WITH-OUTPUT-TO-STRING` также связывает вновь созданный строковый поток вывода с переменной, вами названной, и затем выполняет код в своем теле. После того, как код был выполнен, `WITH-OUTPUT-TO-STRING` вернет значение, которое было бы возвращено `GET-OUTPUT-STREAM-STRING`.

```
CL-USER> (with-output-to-string (out)
           (format out "hello, world ")
           (format out "~s" (list 1 2 3)))
"hello, world (1 2 3)"
```

Другие типы потоков, определенные в стандарте языка, предоставляют различные способы "соединения" потоков, то есть позволяют подключать потоки друг к другу почти в любой конфигурации. BROADCAST-STREAM – поток вывода, который посылает записанные данные множеству потоков вывода, переданного как аргумент функции-конструктору, MAKE-BROADCAST-STREAM. В противоположность этому, CONCATENATED-STREAM – поток ввода, который принимает ввод от множества потоков ввода, перемещаясь от потока к потоку, когда очередной поток достигает конца. Потоки CONCATENATED-STREAM создаются функцией MAKE-CONCATENATED-STREAM, которая принимает любое количество потоков ввода в качестве аргументов.

Еще существуют два вида двунаправленных потоков, которые могут подключать потоки друг к другу – TWO-WAY-STREAM и ECHO-STREAM. Их функции-конструкторы, MAKE-TWO-WAY-STREAM и MAKE-ECHO-STREAM, обе принимают два аргумента, поток ввода и поток вывода, и возвращают поток соответствующего типа, который можно использовать как с потоками ввода, так и с потоками вывода.

В случае TWO-WAY-STREAM потока, каждое чтение вернет данные из потока ввода, и каждая запись пошлет данные в поток вывода. ECHO-STREAM по существу работает точно так же кроме того, что все данные прочитанные из потока ввода также направляются в поток вывода. То есть поток вывода потока ECHO-STREAM будет содержать стенограмму "беседы" двух потоков.

Используя эти пять типов потоков, можно построить почти любую топологию сети потоков, какую бы вы ни пожелали.

В заключение, хотя стандарт Common Lisp не оговаривает какой-либо сетевой API, большинство реализаций поддерживают программирование сокетов и, как правило, реализуют сокет как еще один тип потока. Следовательно, можно использовать с ними все обычные функции вводы/вывода.

Теперь вы готовы двигаться дальше для написания библиотеки, которая позволит сгладить некоторые различия поведения основных функций для работы с файловыми путями в различных реализациях Common Lisp.

15. Практика: переносимая библиотека файловых путей

Как было сказано в предыдущей главе, Common Lisp предоставляет абстракцию файловых путей, и предполагается, что она изолирует вас от деталей того, как различные операционные и файловые системы именуют файлы. Файловые пути предоставляют удобное API для управления именами самими по себе, но когда дело доходит до функций, которые на самом деле взаимодействуют с файловой системой, всё не так гладко.

Корень проблемы, как я упомянул, в том, что абстракция файлового пути была спроектирована для представления файловых имён среди много большего многообразия файловых систем, чем используется в данный момент. К несчастью, сделав файловые пути достаточно абстрактными, чтобы учитывать большое разнообразие файловых систем, создатели Common Lisp оставили разработчикам реализаций на выбор многочисленные варианты, как точно отображать абстракцию файлового пути в любую конкретную файловую систему. Следовательно, различные разработчики реализаций, каждые из которых реализуют абстракцию файлового пути для одной и той же файловой системы, сделав разный выбор в нескольких ключевых точках, могут закончить с соответствующими стандартными реализациями, которые, тем не менее, будут демонстрировать различное поведение для нескольких основных функций, связанных с файловыми путями.

Однако, так или иначе, все реализации обеспечивают одну и ту же базовую функциональность, так что не так сложно написать библиотеку, которая предоставляет единообразный интерфейс для обычных операций в разных реализациях. Это и будет нашей задачей в данной главе. В добавление к предоставлению вам нескольких полезных функций, которые вы будете использовать в будущих главах, написание этой библиотеки даст вам возможность научиться писать код, имеющий дело с различиями в реализациях.

API

Базовые операции, которые будет поддерживать библиотека — получение списка файлов в директории и проверка существования в данной директории файла с данным именем. Вы также напишете функцию для рекурсивного прохода по иерархии директорий с вызовом заданной функции для каждого файлового пути в дереве.

Теоретически, эти операции просмотра директории и проверки существования файла уже предоставлены стандартными функциями `DIRECTORY` и `PROBE-FILE`. Однако, вы увидите, что есть несколько разных путей для реализации этих функций — все в рамках правильных интерпретаций стандарта языка — и вам захочется написать новые функции, которые предоставят единообразное поведение для разных реализаций.

Переменная ***FEATURES*** и обработка условий при считывании.

Перед тем, как реализовать API в библиотеке, которая будет корректно работать на нескольких реализациях Common Lisp, мне нужно показать вам механизм для написания кода, предназначенного для определённой реализации.

В то время, как большая часть кода, которую вы будете писать будет "переносимой" в том смысле, что она будет выполняться одинаково на любой реализации, соответствующей стандарту Common Lisp, вам может внезапно понадобится положиться на функциональность, специфичную для реализации, или написать немного разные куски кода для различных

реализаций. Чтобы помочь вам сделать это без полного разрушения переносимости вашего кода, Common Lisp предоставляет механизм, называемый *обработка условий при считывании*, который позволит вам включать код при определённых условиях, основанных на таких особенностях, как реализация, в которой идёт выполнение.

Этот механизм состоит из переменной `*FEATURES*` и двух дополнительных частей синтаксиса, понимаемых считывателем Lisp. `*FEATURES*` является списком символов; каждый символ представляет собой «свойство», которое присутствует в реализации или используемой ей платформе. Эти символы затем используются в выражениях на свойства, которые вычисляются как истина или ложь, в зависимости от того, присутствуют ли символы из этих выражений в переменной `*FEATURES*`. Простейшее выражение на свойство — одиночный символ; это выражение истинно, если символ входит в `*FEATURES*`, и ложно в противном случае. Другие выражения на свойства — логические выражения, построенные из операторов NOT, AND или OR. Например, если бы вы захотели установить условие, чтобы некоторый код был включен только если присутствуют свойства `foo` и `bar`, вы могли бы записать выражение на свойства `(and foo bar)`.

Считыватель использует выражения на свойства вместе с двумя элементами синтаксиса, `#+` и `#-`. Когда считыватель видит один из них, он сначала читает выражение на свойство и затем вычисляет его, как я только что описал. Когда выражение на свойство, следующее за `#+`, является истинным, считыватель считывает следующее выражение обычным образом. В противном случае он пропускает следующее выражение, считая его пробелом. `#-` работает также, за исключением того, что форма считывается, если выражение на свойство ложно, и пропускается, если оно истинно.

Начальное значение `*FEATURES*` зависит от реализации, и функциональность, подразумеваемая любым присутствующим в ней символом, тоже определяется реализацией. Однако, все реализации включают по крайней мере один символ, указывающий на неё саму. Например, Allegro Common Lisp включает символ `:allegro`, CLISP включает `:clisp`, SBCL включает `:sbcl` и CMUCL включает `:cmu`. Чтобы избежать зависимостей от пакетов, которые могут или не могут существовать в различных реализациях, символы в `*FEATURES*` — обычно ключевые слова, и считыватель связывает `*PACKAGE*` с пакетом `KEYWORD` во время считывания выражений. Таким образом, имя без указания пакета будем прочитано как ключевой символ. Итак, вы могли бы написать функцию, которая ведёт себя немного по-разному в каждой из только что упомянутых реализаций так:

```
(defun foo ()
  #+allegro (do-one-thing)
  #+sbcl (do-another-thing)
  #+clisp (something-else)
  #+cmu (yet-another-version)
  #- (or allegro sbcl clisp cmu) (error "Not implemented"))
```

В Allegro этот код будет считан, как если бы он был написан так:

```
(defun foo ()
  (do-one-thing))
```

тогда как в SBCL считыватель прочитает это:

```
(defun foo ()
  (do-another-thing))
```

а в реализации, отличной от тех, на которые специально установлены условия, будет считано

следующее:

```
(defun foo ()  
  (error "Not implemented"))
```

Так как обработка условий происходит в считывателе, компилятор даже не увидит пропущенные выражения. Это означает, что вы не тратите время выполнения на наличие различных версий для разных реализаций. Также, когда считыватель пропускает выражения, на которые установлены условия, его не волнуют входящие символы, так что пропущенные выражения могут спокойно содержать символы из пакетов, которые не существуют в других реализациях.

Создание пакета библиотеки

Кстати о пакетах, если вы загрузите полный код этой библиотеки, то увидите, что она определена в новом пакете, `com.gigamonkeys.pathnames`. Я расскажу о деталях определения и использования пакетов в главе 21. Сейчас вы должны отметить, что некоторые реализации предоставляют свои пакеты, которые содержат функции с некоторыми такими же именами, как вы определите в этой главе, и делают эти имена доступными из пакета `CL-USER`. Таким образом, если вы попытаетесь определить функции этой библиотеки, находясь в пакете `CL-USER`, вы можете получить сообщения об ошибках о конфликтах с существующими определениями. Чтобы избежать этой возможности, вы можете создать файл с названием `packages.lisp` и следующим содержанием:

```
(in-package :cl-user)  
(defpackage :com.gigamonkeys.pathnames  
  (:use :common-lisp)  
  (:export  
    :list-directory  
    :file-exists-p  
    :directory-pathname-p  
    :file-pathname-p  
    :pathname-as-directory  
    :pathname-as-file  
    :walk-directory  
    :directory-p  
    :file-p))
```

и сделать `LOAD` на него. Тогда в `REPL` или в начале файла, в который вы печатаете определения из этой главы, напечатайте следующее выражение:

```
(in-package :com.gigamonkeys.pathnames)
```

В дополнение к избежанию конфликтов имён с символами, уже доступными в `CL-USER`, создание пакета для библиотеки таким образом также сделает проще использовать её в другом коде, как вы увидите из нескольких будущих глав.

Получение списка файлов в директории

Вы можете реализовать функцию для получения списка файлов одной директории, `list-directory`, как тонкую обёртку вокруг стандартной функции `DIRECTORY`. `DIRECTORY` принимает особый тип файлового пути, называемого *шаблоном файлового пути*, который имеет одну или более компоненту, содержащую специальное значение `:wild`, и возвращает список файловых путей, представляющих файлы в файловой системе, которые соответствуют шаблону. Алгоритм сопоставления — как большинство вещей, которым приходится иметь дело с

взаимодействием между Lisp и конкретной файловой системой — не определяется стандартом языка, но большинство реализаций на Unix и Windows следуют одной и той же базовой схеме.

Функция DIRECTORY имеет две проблемы, с которыми придётся иметь дело list-directory. Главная проблема состоит в том, что определённые аспекты поведения этой функции различаются достаточно сильно для различных реализаций Common Lisp, даже для одной и той же операционной системы. Другая проблема в том, что, хотя DIRECTORY и предоставляет мощный интерфейс для получения списка файлов, её правильное использование требует понимания некоторых достаточно тонких моментов в абстракции файловых путей. С этими тонкостями и стилистическими особенностями различных реализаций, само написание переносимого кода, использующего DIRECTORY для таких простых вещей, как получение списка всех файлов и поддиректорий для единственной директории, могло бы стать разочаровывающим опытом. Вы можете разобраться со всеми тонкостями и характерными особенностями раз и навсегда, написав list-directory и забыв о них после этого.

Одна тонкость, которая обсуждалась в главе 14 — это два способа представлять имя директории в виде файлового пути: в форме директории и в форме файла.

Чтобы DIRECTORY возвратила вам список файлов в /home/peter/, вам надо передать ей шаблон файлового пути, чья компонента директории — это директория, которую вы хотите прочитать, и чьи компоненты имени и типа являются :wild. Таким образом, может показаться, что для получения списка файлов в /home/peter/ вы можете написать это:

```
(directory (make-pathname :name :wild :type :wild :defaults home-dir))
```

где home-dir является файловым путём, представляющим /home/peter/. Это бы сработало, если бы home-dir была бы в форме директории. Но если бы она была бы в файловой форме — например, если бы она была создана разбором строки "/home/peter" - тогда бы это выражение вернуло список всех файлов в /home, так как компонента имени "peter" была бы заменена на :wild.

Чтобы избежать беспокойства о явном преобразовании между представлениями, вы можете определить list-directory так, чтобы она принимала нешаблонный файловый путь в обеих формах, который затем она будет переводить в подходящий шаблон файлового пути.

Чтобы облегчить это, вам следует определить несколько вспомогательных функций. Одна, component-present-p, будет проверять, «существует» ли данная компонента в файловом пути, имея в виду не NIL и не специальное значение :unspecific.. Другая, directory-pathname-p, проверяет, задан ли файловый путь уже в форме директории, и третья, pathname-as-directory, преобразует любой файловый путь в файловый путь в форме директории.

```
(defun component-present-p (value)
  (and value (not (eql value :unspecific))))
(defun directory-pathname-p (p)
  (and
    (not (component-present-p (pathname-name p)))
    (not (component-present-p (pathname-type p)))
    p))
(defun pathname-as-directory (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (not (directory-pathname-p name))
      (make-pathname
        :directory (append (or (pathname-directory pathname) (list :relative))
                          (list (file-namestring pathname)))
        :name nil
        :type nil
        :defaults pathname)
      pathname)))
```

Теперь кажется, что можно создать шаблон файлового пути для передачи DIRECTORY, вызвав MAKE-PATHNAME с формой директории, возвращённой pathname-as-directory. К несчастью, благодаря одной причуде в реализации DIRECTORY в CLISP, всё не так просто. В CLISP, DIRECTORY вернёт файлы без расширений, только если компонента типа шаблона является NIL, но не :wild. Так что вы можете определить функцию, directory-wildcard, которая принимает файловый путь в форме директории или файла, и возвращает шаблон, подходящий для данной реализации, используя проверку условий при считывании для того, чтобы делать файловый путь с компонентой типа :wild во всех реализациях, за исключением CLISP, и NIL в CLISP.

```
(defun directory-wildcard (dirname)
  (make-pathname
    :name :wild
    :type #-clisp :wild #+clisp nil
    :defaults (pathname-as-directory dirname)))
```

Заметьте, что каждое условие при считывании работает на уровне единственного выражения после #-clisp, выражение :wild будет или считано, или пропущено; ровно как и после #+clisp, NIL будет прочитано или пропущено.

Теперь вы можете первый раз вгрызться в функцию list-directory.

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
    (error "Can only list concrete directory names."))
  (directory (directory-wildcard dirname)))
```

Утверждается, что эта функция будет работать в SBCL, CMUCL и LispWorks. К несчастью, остаётся парочка различий, которые надо сгладить. Одно отличие состоит в том, что не все реализации вернут поддиректории данной директории. Allegro, SBCL, CMUCL и LispWorks сделают это. OpenMCL не делает это по умолчанию, но сделает, если вы передадите DIRECTORY истинное значение по специфичному для этой реализации ключевому аргументу :directories. DIRECTORY в CLISP возвращает поддиректории только когда ей передаётся шаблон файлового пути с :wild в последнем элементе компоненты директории и NIL в компонентах имени и типа. В этом случае, он вернёт только поддиректории, так что вам придётся вызвать DIRECTORY

дважды с разными шаблонами и скомбинировать результаты.

Как только вы заставите все реализации возвращать директории, вы узнаете, что они также различаются в том, возвращают ли они имена директорий в форме директорий или файлов. Вы хотите, чтобы `list-directory` всегда возвращала имена директорий в форме директорий, так, чтобы вы могли отличать поддиректории от обычных файлов, основываясь просто на имени. За исключением Allegro, все реализации этой библиотеки поддерживают это. Allegro, с другой стороны, требует передачи `DIRECTORY` характерного для этой реализации аргумента `:directories-are-files` со значением `NIL`, чтобы заставить её вернуть директории в форме файлов.

Как только вы узнали о том, как сделать так, чтобы каждая реализация делала то, что вы хотите, само написание `list-directory` становится просто делом сочетания различных версий при помощи проверки условий при чтении.

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
    (error "Can only list concrete directory names."))
  (let ((wildcard (directory-wildcard dirname)))
    #+(or sbcl cmu lispworks)
    (directory wildcard)
    #+openmcl
    (directory wildcard :directories t)
    #+allegro
    (directory wildcard :directories-are-files nil)
    #+clisp
    (nconc
     (directory wildcard)
     (directory (clisp-subdirectories-wildcard wildcard))))
    #-(or sbcl cmu lispworks openmcl allegro clisp)
    (error "list-directory not implemented")))
```

Функция `clisp-subdirectories-wildcard` на самом деле не является присущей CLISP, но так как она не нужна никакой другой реализации, вы можете ограничить её условием при чтении. В этом случае, так как выражение, следующее за `#+` является целым `DEFUN`, будет или не будет включено всё определение функции, в зависимости от того, присутствует ли `clisp` в `*FEATURES*`.

```
#+clisp
(defun clisp-subdirectories-wildcard (wildcard)
  (make-pathname
   :directory (append (pathname-directory wildcard) (list :wild))
   :name nil
   :type nil
   :defaults wildcard))
```

Проверка существования файла

Чтобы заменить `PROBE-FILE`, вы можете определить функцию с именем `file-exists-p`. Она должна принимать имя файла и, если файл существует, возвращать то же самое имя, и `NIL`, если не существует. Она должна быть способна принимать имя директории и в виде директории, и в виде файла, но должна всегда возвращать файловый путь в форме директории, если файл существует и является директорией. Это позволит вам использовать `file-exists-p` вместе с `directory-pathname-p`, чтобы проверить, является ли данное имя именем файла или директории.

Теоретически, `file-exists-p` достаточно похожа на стандартную функцию `PROBE-FILE`, и на самом деле, в нескольких реализациях — `SBCL`, `LispWorks`, `OpenMCL` — `PROBE-FILE` уже даёт вам то поведение, которого вы хотите от `file-exists-p`. Но не все реализации `PROBE-FILE` ведут себя так.

Функции `PROBE-FILE` в `Allegro` и `CMUCL` близки к тому, чего вы хотите — они принимают имя директории в обеих формах, но, вместо возвращения имени в форме директории, просто возвращают его в той же самой форме, в которой им был передан аргумент. К счастью, если им передаётся имя недиректории в форме директории, они возвращают `NIL`. Так что, в этих реализациях вы можете получить желаемое поведение, сначала передав `PROBE-FILE` имя в форме директории — если файл существует и является директорией, она возвратит имя в форме директории. Если этот вызов вернёт `NIL`, вы попытаетесь снова с именем в форме файла.

`CLISP`, с другой стороны, снова имеет свой собственный способ сделать это. Его `PROBE-FILE` немедленно сигнализирует ошибку, если передано имя в форме директории, вне зависимости от того, существует ли файл или директория с таким именем. Она также сигнализирует ошибку, если в файловой форме передано имя, которое на самом деле является именем директории. Для определения, существует ли директория, `CLISP` предоставляет собственную функцию: `probe-directory` (в пакете `ext`). Она практически является зеркальным отражением `PROBE-FILE`: выдаёт ошибку, если ей передаётся имя в файловой форме или если передано имя в форме директории, которое оказалось именем файла. Единственное различие в том, что она возвращает `T`, а не файловый путь, когда существует названная директория.

Но даже в `CLISP` вы можете реализовать желаемую семантику, обернув вызовы `PROBE-FILE` и `probe-directory` в `IGNORE-ERRORS`.

```
(defun file-exists-p (pathname)
  #+(or sbcl lispworks openmcl)
  (probe-file pathname)
  #+(or allegro cmu)
  (or (probe-file (pathname-as-directory pathname))
      (probe-file pathname))
  #+clisp
  (or (ignore-errors
      (probe-file (pathname-as-file pathname)))
      (ignore-errors
      (let ((directory-form (pathname-as-directory pathname)))
        (when (ext:probe-directory directory-form)
          directory-form))))
  #- (or sbcl cmu lispworks openmcl allegro clisp)
  (error "file-exists-p not implemented"))
```

Функция `pathname-as-file`, которая нужна вам для реализации `file-exists-p` в `CLISP` является обратной для определённой ранее `pathname-as-directory`, возвращающей файловый путь, являющийся эквивалентом аргумента в файловой форме. Несмотря на то, что эта функция нужна здесь только для `CLISP`, она полезна в общем случае, так что определим её для всех реализаций и сделаем частью библиотеки.

```
(defun pathname-as-file (name)
  (let ((pathname (pathname name)))
    (when (wild-pathname-p pathname)
      (error "Can't reliably convert wild pathnames."))
    (if (directory-pathname-p name)
        (let* ((directory (pathname-directory pathname))
               (name-and-type (pathname (first (last directory)))))
          (make-pathname
           :directory (butlast directory)
           :name (pathname-name name-and-type)
           :type (pathname-type name-and-type)
           :defaults pathname))
        pathname)))
```

Проход по дереву директорий

Наконец, чтобы завершить библиотеку, вы можете реализовать функцию, называемую `walk-directory`. В отличие от ранее определённых функций, эта функция не нужна для сглаживания различий между реализациями; она просто использует функции, которые вы уже определили. Однако, она довольно удобна, и вы будете её несколько раз использовать в последующих частях. Она будет принимать имя директории и функцию, и вызывать функцию на всех файлах входящих в директорию рекурсивно. Она также принимает два ключевых аргумента: `:directories` и `:test`. Когда `:directories` истинно, она будет вызывать функцию на именах директорий, как на обычных файлах. Аргумент `:test`, если предоставлен, определяет другую функцию, которая вызывается на каждом файловом пути до того, как будет вызвана главная функция, которая будет вызвана только если тестовая функция возвратит истинное значение.

```
(defun walk-directory (dirname fn &key directories (test (constantly t)))
  (labels
    ((walk (name)
      (cond
        ((directory-pathname-p name)
         (when (and directories (funcall test name))
           (funcall fn name))
         (dolist (x (list-directory name)) (walk x)))
        ((funcall test name) (funcall fn name)))))
    (walk (pathname-as-directory dirname))))
```

Теперь у вас есть полезная библиотека функций для работы с файловыми путями. Как я упомянул, эти функции окажутся полезны в следующих частях, особенно в частях 23 и 27, где вы будете использовать `walk-directory`, чтобы продраться через дерево директорий, содержащих спамерские сообщения и MP3 файлы. Но до того как мы доберёмся до этого, мне, тем не менее, нужно поговорить о объектной ориентации, теме следующих двух глав.

16. Переходим к объектам: Обобщенные функции

Поскольку создание Lisp произошло за пару десятилетий до того момента, когда объектно-ориентированное программирование (ООП) стало популярным, новые программисты на Lisp иногда удивляются, открывая для себя насколько полноценным объектно-ориентированным языком является Common Lisp. Непосредственные предшественники Common Lisp разрабатывались в то время, когда FIXME/*объектной ориентация была*/ООП был волнующей новой идеей и проводилось много экспериментов на тему включения в Lisp идей FIXME/*объектной ориентации*/ООП, особенно реализованных в Smalltalk. Как часть процесса стандартизации Common Lisp объединение идей нескольких этих экспериментов было представлено под названием Common Lisp Object System (Объектная система Common Lisp) или CLOS. Стандарт ANSI включил CLOS в язык, так что сейчас нет смысла говорить о CLOS как об отдельной сущности.

Возможности CLOS внесенные в Common Lisp варьируются от тех, без которых крайне тяжело обойтись, до относительно эзотеричных проявлений философии Lisp как язык-инструмента для построения языков. Полный обзор всех возможностей лежит за пределами данной книги, но в этой и следующей главе я опишу базовые возможности и дам обзор подхода Common Lisp к объектам.

В начале вы должны были заметить, что объектная система Common Lisp предлагает достаточно отличающуюся от других языков реализацию принципов FIXME/*объектной ориентации*/ООП. FIXME/*Если у вас есть глубокое*/Если вы имеете глубокое понимание фундаментальных идей, заложенных в основу FIXME/*объектной ориентации*/ООП, то вы вероятно оцените способ, который использовался в Common Lisp для реализации этих идей. С другой стороны, если у вас есть опыт использования FIXME/*объектной ориентации в основном на одном*/ООП только на одном языке, то подход Common Lisp может показаться вам несколько чуждым; вы должны избегать предположений, что для языка существует только один способ реализации принципов FIXME/*объектной ориентации*/ООП. Если у вас имеется небольшой опыт объектно-ориентированного программирования, то вы не должны испытывать проблем с пониманием изложенных здесь объяснений, хотя может быть лучше игнорировать сравнения с подходами в других языках.

Обобщенные функции и классы

Мощной и фундаментальной особенностью ООП FIXME*Фундаментальной особенностью объектной ориентации* является способ организации программ путем определения типов данных и связи операций с этими типами данных. В частности, вам может понадобиться выполнять операцию и получать определенное поведение, определенное типом объекта (или объектов) для которых эта операция выполняется. Практически все введения в ООП используют классический пример операции рисования, которая может быть применена к объектам, представляющим различные геометрические фигуры. Для рисования окружностей, треугольников или квадратов могут быть реализованы разные операции FIXME*draw (здесь имеется ввиду конкретный метод draw)*, которые будут отображать окружность, треугольник или квадрат, в зависимости от объекта, к которому применяется операция рисования. Различные реализации операции рисования FIXME*draw (здесь имеется ввиду конкретный метод draw)* определяются отдельно, и могут создаваться новые версии для рисования других фигур, не затрагивая ни кода пользователя, ни кода других операция рисования FIXME*Ени кода других реализаций операции рисования draw*. Эта возможность ООП имеет красивое греческое имя "полиморфизм (polymorphism)", переводимое как "много форм", поскольку одна концептуальная операция, такая как

рисование объекта, может иметь множество различных конкретных форм.

Common Lisp, подобно другим современным объектно-ориентированным языкам, основан на классах; все объекты являются экземплярами определенного класса. Класс объекта определяет его представление – встроенные классы, такие как NUMBER и STRING имеют скрытое представление, доступное только через стандартные функции для работы с этими типами, в то время как экземпляры классов, определенных пользователем, состоят из именованных частей, называемых слотами (вы увидите это в следующей главе).

Классы образуют иерархию/классификацию всех объектов. Класс может быть определен как подкласс других классов, называемых суперклассами (или предком) *DELETЕМЕ* *нету там предков*. Класс наследует от суперклассов часть своего определения, а экземпляры класса также считаются и экземплярами суперклассов. В Common Lisp иерархия классов имеет один корень – класс Т, который является прямым (или косвенным) суперклассом для всех остальных классов. Таким образом, в Common Lisp все данные являются экземплярами класса Т. Common Lisp также поддерживает множественное наследование – один класс может иметь несколько прямых предков *ФИХМЕ* *прямых суперклассов*.

Вне семейства языков Lisp, *DELETЕМЕ* (запятая) почти все объектно-ориентированные языки следуют базовому дизайну, заданному языком Simula, когда поведение, связанное с классом, реализуется в виде методов или функций-членов, которые относятся к определенному классу. В этих языках метод, вызываемый для определенного объекта, и класс, к которому этот объект относится, определяют какой код будет запущен. Эта модель запуска *ФИХМЕ* *вызова* методов называется (по терминологии Smalltalk) передачей сообщений (message passing). Концептуально запуск *ФИХМЕ* *вызовов* методов в системах с передачей сообщений начинается путем отправки сообщения, содержащего имя запускаемого метода и необходимые аргументы, объекту для которого метод будет выполняться *ФИХМЕ* *и необходимые аргументы для объекта, метод которого вызывается*. Объект затем использует свой класс для поиска метода, связанного с именем, указанным в сообщении, и запускает его. Поскольку каждый класс может иметь собственный метод для заданного имени, то одно и то же сообщение, посланное разным объектам, может запускать *ФИХМЕ* *вызывать* разные методы.

Ранние реализации объектной системы Lisp работали аналогичным образом, используя *ФИХМЕ* *предоставляя* специальную функцию SEND, которая могла быть использована для отправки сообщения определенному объекту. Однако, это было не совсем удобно, поскольку это делало вызов метода отличным от обычного вызова функции. Синтаксически вызов метода записывался как:

```
(send object 'foo)
```

ВМЕСТО:

```
(foo object)
```

Более важным являлось то, что поскольку методы не являлись функциями, то они не могли быть переданы как аргументы для функций высшего порядка, таких как MAPCAR; если кто-то хотел вызвать метод для всех элементов списка, используя MAPCAR, то он должен был писать вот так:

```
(mapcar #'(lambda (object) (send object 'foo)) objects)
```

ВМЕСТО:

```
(mapcar #'foo objects)
```

В конечном счете, люди работающие над объектной системой *FIXME* *объектными системами* Lisp унифицировали методы и функции путем введения нового вида функций, называемых обобщенными (generic). В дополнение к тому, что это решало проблемы описанные выше, *FIXME* *в дополнение к решению описанных выше проблем* обобщенные функции открыли новые возможности объектной системы, включая много *FIXME* *множество* возможностей, которые просто не имели значения *FIXME* *смысла* в объектных системах с передачей сообщений.

Обобщенные функции являются основой объектной системой Common Lisp и будут описываться на протяжении всей главы. *FIXME* *Обобщенные функции являются сердцем объектной системы Common Lisp и будут темой данной главы до ее окончания* Поскольку я не могу говорить об обобщенных функциях без упоминания классов, то сначала я остановлюсь на том, как определить и использовать обобщенные функции. А в следующей главе я покажу вам как определять ваши собственные классы.

Обобщенные функции и методы

Обобщенные функции определяют *FIXME* *Обобщенная функция определяет* абстрактную операцию, указывая имя операции и список параметров, но не определяют реализацию. Вот, например, как вы можете определить обобщенную функцию draw, которая будет использоваться для отрисовки различных фигур на экране:

```
(defgeneric draw (shape)
  (:documentation "Draw the given shape on the screen."))
```

Синтаксис DEFGENERIC будет описан в следующем разделе; сейчас достаточно упомянуть, что это определение совсем не содержит кода. *FIXME* *Я опишу синтаксис DEFGENERIC в следующем разделе; сейчас лишь замечу, что это определение совсем не содержит кода.*

Обобщенные функции являются "общими" в том смысле, что они могут *FIXME* *в единственное число* (по крайней мере в теории) принимать в качестве аргументов любые объекты. Однако, сама обобщенная функция не делает ничего; если вы просто определили обобщенную функцию, то при ее вызове с любыми аргументами она будет выдавать ошибку. Актуальная *FIXME* *Действительная?* реализация обобщенной функции обеспечивается методами. Каждый метод реализует обобщенную функцию для конкретного класса аргументов. *FIXME* *Каждый метод предоставляет реализацию для конкретных классов аргументов.* Возможно, самое большое отличие между системами с обобщенными функциями и системами с передачей сообщений является то, что методы не относятся к конкретным классам *FIXME* *не принадлежат к классам*; они относятся к обобщенной функции, которая отвечает за определение того, какой метод (или методы) будут в действительности вызваны при вызове обобщенной функции *FIXME* *(или методы) будет запущен в ответ на вызов обобщенной функции.*

Методы указывают какой вид аргументов они могут обрабатывать, путем специализации требуемых параметров, определенных обобщенной функцией. Например, для обобщенной функции draw, вы можете определить один метод, который определяет специализацию параметра shape для объектов, которые относятся к классу *FIXME* *которые являются экземплярами класса circle*, а другой метод, указывает специализацию shape для объектов *FIXME* *экземпляров* класса triangle. Они могут выглядеть следующим образом (не вдаваясь в подробности рисования конкретных фигур):

```
(defmethod draw ((shape circle))
  ...)
(defmethod draw ((shape triangle))
  ...)
```

Когда запускается `FIXME` вызывается обобщенная функция, она сравнивает переданные аргументы с параметрами специализированных методов `FIXME` *сравнивает переданные аргументы со специализацией каждого из ее методов* для того, чтобы найти метод, специализации которого совместимы с переданными аргументами. Если вы вызываете `draw`, передавая экземпляр `circle`, то применяется метод, который специализирует `shape` для класса `circle`, а если вы вызываете передавая `triangle`, то будет вызван метод, который специализирует `shape` для `triangle`. В простых случаях, будет применяться `FIXME` *будет подходить* только один метод, и он будет полностью `DELETEME` *убрать полностью* обрабатывать вызов. В более сложных случаях, могут быть применимы несколько методов; они будут объединены, как это будет описано `FIXME` *как я опишу* в разделе "Комбинация методов", в один `FIXME` *эффективный?* метод, который обработает данный вызов.

Вы можете специализировать параметры двумя способами – обычно вы указываете класс, экземпляром которого должен быть аргумент. Поскольку экземпляры класса также рассматриваются как экземпляры его суперклассов, то метод специализированный для конкретного класса, также применим для аргументов, являющихся экземплярами его подклассов `FIXME` *аргументов, которые могут быть как экземплярами специализирующего класса, так его подклассов*. Другой вид специализации – так называемый `EQL`-специализатор, который определяет конкретный объект к которому применим данный метод.

Когда обобщенная функция имеет только методы специализированные для одного параметра, и все специализации являются специализациями классов, то результат вызова обобщенной функции сильно похож на вызов метода в системе с передачей сообщений – комбинация имени операции и класса объекта, для которого производится запуск операции, определяют `FIXME` *определяющая (комбинация)* какой конкретно метод будет выполнен.

Однако изменение порядка поиска открывает новые возможности, не имеющиеся `FIXME` *которых нет?* в системах с передачей сообщений. Обобщенные функции поддерживают методы, которые специализируются для множества параметров, предоставляя каркас, который делает множественное наследование более управляемым, и который позволяет вам использовать декларативные конструкции для контроля того, как методы комбинируются в эффективный метод, поддерживая несколько общих методов использования без наличия кучи дополнительного кода. Я буду обсуждать эти вопросы в соответствующих разделах `FIXME` *Я опишу эти вопросы очень скоро*. Но сначала вам необходимо взглянуть на основы использования двух макросов, которые используются для определения обобщенных функций: `DEFGENERIC` и `DEFMETHOD`.

DEFGENERIC

Чтобы дать вам почувствовать эти макросы, и ознакомиться с возможностями, которые они предоставляют, я покажу вам некоторый код, который вы могли бы написать как часть банковского приложения, или иначе, игрушечное банковское приложение; главная задача – ознакомиться с возможностями языка, а не научиться писать банковское программное обеспечение. Например, данный код даже не пытается претендовать на работу с разными валютами, а также оставляет в стороне вопросы аудита и целостности транзакций.

Поскольку я буду обсуждать вопросы создания новых классов только в следующей главе, для понимания вы можете просто представить `FIXME` *главе, пока что вы можете считать*, что определенные классы уже существуют: `FIXME` *для начала* предположим, что существует класс

bank-account и он имеет два подкласса – checking-account и savings-account. Иерархия классов выглядит следующим образом:

FIXME нарисовать картинку

Первой обобщенной функцией будет функция withdraw, которая уменьшает баланс на указанное числоFIXMEуказанную сумму. Если баланс меньше указанного числаFIXMEуказанной суммы, то она должна выдать ошибку и оставить баланс в неизменном виде. Вы можете начать с определения обобщенной функции с помощью DEFGENERIC.

Основная форма DEFGENERIC похожа DEFUN за тем исключением, что нет тела функции. Список параметров DEFGENERIC определяет параметры, которые должны приниматься всеми методами, определенными для данной обобщенной функции. Вместо тела DEFGENERIC может содержать различные опции. Одной из опций, которую вы должны всегда указывать, является :documentation, которая используется для указания строки с описанием FIXMENазначения обобщенной функции. Поскольку обобщенная функция является абстрактной, то очень важно,FIXMEявляется полностью абстрактной, то важно, чтобы и пользователь и программист имели четкое представление о том, что она делает. Таким образом, вы можете определить withdraw следующим образом:

```
(defgeneric withdraw (account amount)
  (:documentation "Withdraw the specified amount from the account.
Signal an error if the current balance is less than amount."))
```

DEFMETHOD

Сейчас вы готовы к использованию DEFMETHOD для определения методов, которые реализуют withdraw.

Список параметров метода должен соответствоватьFIXMEсовпадать? спискам параметров, объявленному обобщенной функцией. В данном случае это означает, что все методы определенные для withdraw должны иметь два обязательных параметра. Точнее говоряFIXMEВ более общих чертах, методы должны иметь то же самое количество обязательных и необязательных параметров, и кроме этого, должны уметь принимать любые аргументы, относящиеся к остаточным (&rest) или именованным (&key) аргументамFIXMEпараметрам, определенным в обобщенной функции.

Поскольку базовые действия по списанию денег со счета являются одинаковыми для всех счетов, то вы можете определить метод, который специализирует параметр account для класса bank-account. Вы можете предположить, что функция balance возвращает текущее значение суммы на счете и может быть использована вместе с функцией SETF (и таким образом, вместе с DECF) для установки значения баланса. Функция ERROR является стандартной функцией для сообщения об ошибках и я ее подробно опишу в главе 19. Используя эти две функции, вы можете определить базовыйFIXMEпростой метод withdraw примерно так:

```
(defmethod withdraw ((account bank-account) amount)
  (when (< (balance account) amount)
    (error "Account overdrawn."))
  (decf (balance account) amount))
```

Как видно из этого кода, форма DEFMETHOD более похожа на DEFUN по сравнению с DEFGENERIC. Основным отличием является то, что требуемые параметры могут быть специализированы путем замены имени параметра на список из двух элементов. Первым элементом является имя параметра, а вторым – специализатор, который может быть либо

именем класса, либо EQL-специализатором, который будет обсуждаться немного позже *FIXME*который я опишу чуть позже. Имя параметра может быть любым – оно не обязательно должно совпадать с именем, указанным в объявлении обобщенной функции, хотя это часто используется *FIXME*хотя часто будет.

Этот метод будет использоваться тогда, когда первый аргумент `withdraw` является экземпляром класса `bank-account`. Второй параметр, `amount`, неявно специализируется для класса `T`, а поскольку все объекты являются экземплярами `T`, это никак не затрагивает применимость метода.

Теперь предположим, что все чековые счета имеют защиту от перерасхода. Так что, каждый чековый счет связан с другим счетом в банке, с которого будет производиться списание, когда на чековом счету недостаточно денег для списания. Вы можете предположить, что функция `overdraft-account` получает объект класса `checking-account` и возвращает объект класса `bank-account`, представляющего собой связанный счет.

Таким образом, списание с объекта класса `checking-account` требует выполнения дополнительных шагов по сравнению со списанием с обычного счета *FIXME*списанием со стандартного объекта `bank-account`. Вы сначала должны проверить, является ли списываемая сумма большей, чем имеющаяся на счету, и если это так, то перенести недостающую сумму со связанного счета. Затем, *DELETEME*(запятая) вы можете продолжать вычисления также, как и для обычно счета *FIXME*как и со стандартным объектом `bank-account`.

Так что вы можете захотеть определить метод `withdraw`, который специализирован для `checking-account` для обработки перевода денег с другого счета, а затем передать управление методу, специализированному для обычных счетов (класса `bank-account`) *FIXME*счета и последующей передачи управление методу, специализированному на `bank-account`. Такой метод может выглядеть вот так:

```
(defmethod withdraw ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft)))
    (call-next-method))
```

Функция `CALL-NEXT-METHOD` является частью системы обобщенных функций и используется для комбинации *FIXME*подходящих методов. Она сообщает, что контроль должен быть передан от текущего метода, к методу, специализированному для `bank-account`. Когда он вызывается без аргументов, как это было сделано в нашем примере, следующий в цепочке метод будет вызван с теми же аргументами, которые были переданы обобщенной функции. Он также может быть вызван с явным указанием аргументов, которые будут переданы следующему методу.

Вам не обязательно вызывать `CALL-NEXT-METHOD` в каждом методе. Однако, если вы не будете вызывать эту функцию, то новый метод будет полностью отвечать за реализацию требуемого поведения обобщенной функции. Например, если вы хотите создать подкласс `bank-account`, названный `proxy-account`, который не будет отслеживать свой баланс, а вместо этого будет делегировать списание средств другому счету, то вы можете записать этот метод следующим образом (предполагая, что функция `proxied-account` возвращает соответствующий счет):

```
(defmethod withdraw ((proxy proxy-account) amount)
  (withdraw (proxied-account proxy) amount))
```

В заключение, DEFMETHOD также позволяет вам создавать методы, которые специализированы для конкретного объекта используя EQL-специализатор. Например, предположим, что банковское приложение FIXMEбудет развернуто в каком-то коррумпированном банке. Предположим, что переменная *account-of-bank-president* хранит ссылку на конкретный банковский счет, который относится (как это видно из имени) к президенту банка. ТакжеFIXMEДалее предположим, что переменная *bank* представляет весь банк, а функция embezzle крадет деньги у банка. Президент банка может попросить вас "исправить" функцию withdraw таким образом, чтобы она обрабатывала его счет другим способом.

```
(defmethod withdraw ((account (eql *account-of-bank-president*)) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (incf (balance account) (embezzle *bank* overdraft)))
    (call-next-method)))
```

Однако заметьте, что форма, указанная в EQL-специализаторе, который используется для указания объекта (в нашем случае это *account-of-bank-president*) вычисляется один раз, когда вычисляется DEFMETHOD. Этот метод будет специализирован для значения *account-of-bank-president* в тот момент, когда этот метод был определен; последующие изменения переменной не изменяют метод.

Комбинирование методов

Вне кода метода использование CALL-NEXT-METHOD не имеет смысла. Внутри метода использование этой функции обретает смысл за счет того, что механизмы реализации обобщенных функций при каждом запуске создают эффективный метод из методов, для которых применимы текущие параметры. Идея построения эффективного метода путем комбинации методов, для которых применимы текущие аргументы, является основой концепции обобщенных функций и это позволяет обобщенным функциям реализовывать возможности, которые недоступны в системах с передачей сообщений. Так что следует поближе познакомиться с тем, что в действительности происходит. Люди, которые давно знакомы с концепциями систем с передачей сообщений должны обратить на данный материал особое внимание, поскольку обобщенные функции кардинально меняют диспатчеризацию методов по сравнению с передачей сообщений, делая обобщенную функцию (а не класс) ответственной за передачу управления.

В соответствии с концепцией, эффективный метод строится в три шага: на первом шаге обобщенная функция строит список методов, для которых применимы переданные аргументы. На втором шаге полученный список методов сортируется в соответствии со специализированными параметрами. И в заключение, методы по порядку берутся из списка и их код комбинируется образуя эффективный метод.

Для нахождения методов, для которых применимы данные аргументы, обобщенная функция сравнивает аргументы с соответствующими специализаторами параметров всех определенных методов. Метод считается допустимым, только если все специализаторы совместимы с соответствующими параметрами.

Когда специализатор является именем класса, он считается совместимым, если указанное имя совпадает с именем класса аргумента (или именем одного из суперклассов аргумента). (Заметьте, что параметры без явных специализаторов, неявно специализируются классом Т, так что они будут совместимы с любым аргументом). EQL-специализатор считается совместимым, если аргумент является тем же объектом, что указан в специализаторе.

Поскольку все аргументы проверяются относительно соответствующих специализаторов, все они влияют на результаты выбора подходящих методов. Методы, которые явно специализируют

более одного параметра, называются мультиметодами; я опишу их в разделе "Мультиметоды".

После того, как все соответствующие методы найдены, необходимо отсортировать их, чтобы затем скомбинировать в эффективный метод. Для упорядочения двух методов, обобщенная функция сравнивает их специализаторы параметров слева направо, и первый специализатор, который отличается в списке параметров методов будет определять их порядок, где первым ставится метод с более специфичным специализатором.

Поскольку сортируются только подходящие методы, вы знаете все классы специализаторов, для которых соответствующий аргумент является экземпляром. В типичном случае, если два специализатора класса отличаются, то один будет подклассом другого. В этом случае специализатор, именующий подкласс, считается более специфичным. Поэтому метод, который специализирован для счета с классом `checking-account` будет рассматриваться как более специфичный, чем метод, специализированный для класса `bank-account`.

Множественное наследование, немного усложняет идею специфичности, поскольку аргумент может быть экземпляром двух классов, ни один из которых не является подклассом другого. Если такие классы используются как специализаторы параметров, то обобщенная функция не может упорядочить их используя только правило, что подклассы являются более специфичными, чем их суперклассы. В следующей главе я опишу как понятие специфичности было расширено для обработки множественного наследования. Сейчас достаточно сказать, что существует алгоритм для упорядочения специализаторов классов.

В заключение надо отметить, что EQL-специализатор всегда более специфичен, чем любой специализатор класса, и поскольку рассматриваются только подходящие методы, то если EQL-специализатор конкретного параметра имеют более одного метода, то все они должны иметь одинаковые EQL-специализаторы. Сравнение данных методов происходит на основе других параметров.

Стандартный комбинатор методов

Теперь, когда вы понимаете как подходящие методы находятся и сортируются, вы готовы поближе взглянуть на последний шаг – как отсортированный список методов комбинируется в один эффективный метод. По умолчанию обобщенные функции используют так называемый "стандартный комбинатор методов". Стандартный комбинатор объединяет методы таким образом, что `CALL-NEXT-METHOD` работает как это вы уже увидели – сначала запускается наиболее специфичный метод, и каждый метод может передать управление следующему методу используя `CALL-NEXT-METHOD`.

Однако тут есть больше возможностей. Методы, которые я обсуждал, называются основными методами. Основные методы (как и предполагает их имя) отвечают за реализацию основной функциональности обобщенных функций. Стандартный комбинатор методов также поддерживает три вида вспомогательных методов: `:before`, `:after` и `:around`. Определение вспомогательных методов записывается с помощью `DEFMETHOD` также как и для основных методов, но кроме этого, между именем метода и списком параметров указывается квалификатор метода, который именуется типом метода. Например, метод `:before` для функции `withdraw`, которые специализирует параметр `account` для класса `bank-account` будет начинаться со следующей строки:

```
(defmethod withdraw :before ((account bank-account) amount) ...)
```

Каждый вид вспомогательных методов комбинируется в эффективный метод разными способами. Все применимые методы `:before` (не только наиболее специфические) запускаются

как часть эффективного метода. Они запускаются (как и предполагается их именем) до наиболее специфического основного метода и запускаются, начиная с самого специфического метода. Таким образом, методы `:before` могут быть использованы для выполнения любой подготовительной работы, которая может понадобиться, чтобы убедиться что основной метод может быть выполнен. Например, вы можете использовать метод `:before` специализированный для `checking-account` для реализации защиты от перерасхода для чековых счетов:

```
(defmethod withdraw :before ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft)))))
```

Этот метод `:before` имеет три преимущества по сравнению с использованием основного метода. Во первых, он явно показывает как метод изменяет общее поведение функции `withdraw` – он не пересекается с основным поведением и не изменяет возвращаемое значение.

Следующим преимуществом является то, что главный метод, специализированный для класса, более специфичного, чем `checking-account` не будет противоречить этому методу `:before`, что позволяет автору подкласса `checking-account` более легко расширить поведение `withdraw`, при этом сохраняя старую функциональность.

И наконец, поскольку метод `:before` не должен вызывать `CALL-NEXT-METHOD` для передачи управления оставшимся методам, нет возможности сделать ошибку, забыв указать эту функцию.

Другие вспомогательные методы также организуются в эффективные методы способами, которые отражены в их именах. Все методы `:after` запускаются после основного метода в порядке, когда наиболее специфичный метод вызывается последним (порядок противоположен вызову методов `:before`). Таким образом, методы `:before` и `:after` комбинируются для создания вложенных обвязок вокруг основной функциональности, обеспечиваемой основными методами – каждый более специфичный метод `:before` получит шанс сделать что-то для менее специфичного метода `:before`, основные методы могут успешно выполняться, и затем каждый более специфичный метод `:after` получает шанс убрать что-то за основным методом и менее специфичными методами `:after`.

И наконец, методы `:around` комбинируются практически также как и основные методы, за исключением того, что они выполняются "вокруг" остальных методов. Так что код наиболее специфического метода `:around` запускается до любого кода. Внутри кода метода `:around`, вызов `CALL-NEXT-METHOD` приведет к тому, что будет выполняться код следующего метода `:around`, или, при вызове из наименее специфического метода `:around`, приведет к выполнению цепочки методов `:before`, основного метода и затем методов `:after`. Почти все методы `:around` будут иметь в своем коде вызов `CALL-NEXT-METHOD`, поскольку метод `:around` не будет полностью реализовывать (перехватывать) действия обобщенной функции и всех ее методов, за исключением более специфичных методов `:around`.

Иногда требуется полный перехват действий, но обычно, методы `:around` используются для установки некоторого динамического контекста в котором будут выполняться остальные методы – например, для связывания динамической переменной, или для установки обработчика ошибок (это я буду обсуждать в главе 19). Метод `:around` может не вызывать `CALL-NEXT-METHOD` в тех случаях, если он, например, возвращает кэшированное значение, которое было получено при предыдущих вызовах `CALL-NEXT-METHOD`. В любом случае, метод `:around`, не вызывающий `CALL-NEXT-METHOD`, ответствен за корректную реализацию семантики обобщенной функции для всех классов аргументов, для которых этот метод может применяться, включая и будущие подклассы.

Вспомогательные методы являются лишь удобным и лаконичным способом для выражения некоторых общих подходов (паттернов). Они не позволяют вам сделать то, что вы не смогли бы сделать, объединения основные методы с аккуратным соблюдением нескольких соглашений по кодированию и небольшого количества дополнительного кода. Вероятно, наибольшей выгодой является то, что они обеспечивают унифицированный каркас для расширения функциональности обобщенных функций. Библиотеки часто определяют обобщенные функции и основные методы для них, позволяя пользователям изменять их поведение с помощью вспомогательных методов.

Другие комбинаторы методов

В добавление к стандартному комбинатору методов, язык реализует девять других встроенных комбинаторов методов, известных как "простые встроенные комбинаторы методов". Вы также можете определить собственный комбинатор методов, хотя это довольно редко используемая возможность, описание которой не является предметом этой книги. Я кратко опишу как использовать простые комбинаторы методов, чтобы дать общее представление об имеющихся возможностях.

Все простые комбинаторы используют одинаковую стратегию: вместо запуска наиболее специфического метода, и разрешения ему запускать менее специфичные методы через `CALL-NEXT-METHOD`, простые комбинаторы методов создают эффективный метод, содержащий код всех основных методов, расположенных по порядку, и обернутых вызовом к функции, макросу или специальному оператору, который и дал комбинатору методов соответствующее имя. Девять комбинаторов получили имена от операторов: `+`, `AND`, `OR`, `LIST`, `APPEND`, `NCONC`, `MIN`, `MAX` и `PROGN`. Простые комбинаторы поддерживают только два типа методов – основные методы, которые объединяются так, как было описано выше, и методы `:around`, которые работают также, как и методы `:around` в стандартном комбинаторе.

Например, обобщенная функция, которая использует комбинатор методов `+`, вернет сумму всех результатов, возвращенных вызванными основными методами. Отметьте, что комбинаторы `AND` и `OR` не обязательно будут выполнять все основные методы, поскольку эти макросы могут использовать сокращенную схему работы – обобщенная функция, использующая комбинатор `AND` вернет значение `NIL` сразу же, как один из методов вернет его, или в противном вернет значение, возвращенное последним вызванным методом. Аналогичным образом, комбинатор `OR` вернет первое значение не равное `NIL`, возвращенное любым из его методов.

Для определения обобщенной функции, которая использует конкретный комбинатор методов, вы должны указать опцию `:method-combination` при объявлении `DEFGENERIC`. Значение, указанное данной опцией определяет имя комбинатора методов, который вы хотите использовать. Например, для определения обобщенной функции `priority`, которая возвращает сумму значений, возвращаемых отдельными методами, используя комбинатор методов `+`, вы можете написать следующий код:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination +))
```

По умолчанию, все эти комбинаторы методов комбинируют методы в порядке начиная с наиболее специфического. Однако вы можете изменить порядок, путем указания ключевого слова `:most-specific-last` после имени комбинатора в объявлении функции с помощью `DEFGENERIC`. Порядок скорее всего не имеет значения если вы используете комбинатор `+` и методы не имеют побочных эффектов, но в целях демонстрации, я изменю код `priority` чтобы он использовал порядок `most-specific-last`:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination + :most-specific-last))
```

Основные методы обобщенных функций, которые используют один из этих комбинаторов, должны быть квалифицированы именем комбинатора методов. Таким образом, основной метод, определенный для функции `priority` может выглядеть следующим образом:

```
(defmethod priority + ((job express-job)) 10)
```

Это делает ясным, когда вы смотрите на определение метода, который является частью конкретной обобщенной функции.

Все простые встроенные комбинаторы методов поддерживают методы `:around`, которые работают также как и методы `:around` в стандартном комбинаторе: наиболее специфичный метод `:around` выполняется до любого метода, и он может использовать `CALL-NEXT-METHOD` для передачи контроля менее специфичному методу `:around` до тех пор, пока не будет достигнут основной метод. Опция `:most-specific-last` не влияет на порядок вызова методов `:around`. И, как я отметил ранее, встроенные комбинаторы методов не поддерживают методы `:before` и `:after`.

Подобно стандартному комбинатору методов, эти комбинаторы не позволяют вам сделать ничего из того, что вы не можете сделать "вручную". Вместо этого, они позволяют вам выразить то, что вы хотите, а язык возьмет на себя заботу о связывании всего вместе, делая ваш код более кратким и выразительным.

Честно говоря, примерно в 99 процентах случаев вам будет достаточно стандартного комбинатора методов. Оставшийся один процент случаев скорее всего будет обработан простыми встроенными комбинаторами методов. Но если вы попадете в ситуацию (вероятность - примерно одна сотая процента), когда вам не будет подходить ни один из встроенных комбинаторов, то вы можете посмотреть на описание `DEFINE-METHOD-COMBINATION` в вашем любимом справочнике по Common Lisp.

Мультиметоды

Методы, которые явно специализируют более одного параметра обобщенной функции называются мультиметодами. Мультиметоды - это то, в чем реально расходятся обобщенные функции и передача сообщений. Мультиметодов не существует в системах с передачей сообщений `FIXME` *Мультиметоды не подходят языкам с передачей сообщений*, поскольку они не относятся к конкретному классу; вместо этого, каждый мультиметод определяет часть реализации конкретной обобщенной функции, которая применяется тогда, когда эта функция вызывается с аргументами, которые соответствуют всем специализированным параметрам.

`FIXME` (начало выделенного блока)

Мультиметоды против перегрузки методов

Программисты, использовавшие статически типизованные языки с системами передачи сообщений, такие как Java и C++, могут подумать, что мультиметоды похожи на такую возможность этих языков, как перегрузка методов. Однако эти две возможности языка в достаточной мере отличаются, поскольку перегруженные методы выбираются во время компиляции, основываясь на информации об аргументах, полученной во время компиляции, а не во время исполнения. Чтобы посмотреть как это работает, рассмотрим два следующих класса на

языке Java:

```
public class A {
    public void foo(A a) { System.out.println("A/A"); }
    public void foo(B b) { System.out.println("A/B"); }
}
public class B extends A {
    public void foo(A a) { System.out.println("B/A"); }
    public void foo(B b) { System.out.println("B/B"); }
}
```

Теперь посмотрим, что случится, когда вы запустите метод `main` из этого класса.

```
public class Main {
    public static void main(String[] argv) {
        A obj = argv[0].equals("A") ? new A() : new B();
        obj.foo(obj);
    }
}
```

Когда вы заставляете `main` создать экземпляр `A`, она выдаст `A/A`, как вы и ожидали.

```
bash$ java com.gigamonkeys.Main A
A/A
```

Однако, если вы заставите `main` создать экземпляр `B`, то настоящий тип объекта `obj` будет принят во внимание не полностью.

```
bash$ java com.gigamonkeys.Main B
B/A
```

Если бы перегруженные методы работали также как и мультиметоды в Common Lisp, то программа бы выдала ожидаемый результат: `B/B`. Существует возможность реализации множественной диспатчеризации для систем с передачей сообщений вручную, но это будет противоречить модели системы с передачей сообщений, поскольку метод с множественной диспатчеризацией не должен относиться ни к одному из классов.

FIXME конец блока

Мультиметоды полезны в ситуациях, когда вы не знаете к какому классу определенное поведение должно относиться (в языках с передачей сообщений). Звук, который издает барабан, когда вы стучите по нему палочкой, является функцией барабана, или функцией палочки? Конечно, он принадлежит обоим предметам. Для моделирования такой ситуации в Common Lisp, вы просто определяете функцию `beat`, которая принимает два аргумента.

```
(defgeneric beat (drum stick)
  (:documentation
    "Produce a sound by hitting the given drum with the given stick."))
```

Затем, вы можете определять разные мультиметоды для реализации `beat` для комбинаций, которые вам нужны. Например:

```
(defmethod beat ((drum snare-drum) (stick wooden-drumstick)) ...)
(defmethod beat ((drum snare-drum) (stick brush)) ...)
(defmethod beat ((drum snare-drum) (stick soft-mallet)) ...)
(defmethod beat ((drum tom-tom) (stick wooden-drumstick)) ...)
(defmethod beat ((drum tom-tom) (stick brush)) ...)
(defmethod beat ((drum tom-tom) (stick soft-mallet)) ...)
```

Мультиметоды не помогают от комбинаторного взрыва – если вам необходимо смоделировать пять видов барабанов и шесть видов палочек, и каждая из комбинаций дает отдельный звук, то не способа упростить это; вам необходимо определить тридцать различных методов для реализации всех комбинаций, используя мультиметоды или нет. Мультиметоды помогут вам не писать собственный код для диспатчинга, позволяя использовать встроенные средства диспатчеризации, которые были очень удобны при работе с методами, специализированными для одного параметра.

Мультиметоды также спасают вас от требования чтобы один набор классов был тесно связан с другим набором. В нашем примере барабанов/палочек, не требуется чтобы класс, представляющий барабаны знал о различных классах, представляющих барабанные палочки, и наоборот. Мультиметоды связывают независимые классы для описания их совместного поведения, при этом не требуя никакого взаимодействия между самими классами.

Продолжение следует ...

Я описал основы (и немного больше) обобщенных функций – "глаголов" объектной системы Common Lisp. В следующей главе я покажу вам как определять ваши собственные классы.

17. Переходим к объектам: Классы

Если обобщенные функции являются глаголами объектной системы, то классы являются существительными. Как я упоминал в предыдущей главе, все значения в программах на Common Lisp являются экземплярами какого-то из классов. Более того, все классы образуют иерархию на вершине которой находится класс Т.

Иерархия классов состоит из двух основных семейств классов: встроенных и определенных пользователем. Классы, которые представляют типы данных, которые мы изучали до сих пор – такие как INTEGER, STRING и LIST, являются встроенными. Они находятся в отдельном разделе иерархии классов, организованные соответствующими связями дочерних и родительских классов, и для работы с ними используются функции, которые мы обсуждали *FIXME* которые я описывал на протяжении всей книги. Вы не можете унаследовать от этих классов, но как вы увидели в предыдущем разделе, вы можете определить специализированные методы для них, эффективно расширяя поведения этих классов.

Но когда вы создаете новые существительные – например, классы, которые использовались в предыдущей главе для представления банковских счетов, то вам нужно определить ваши собственные классы. Это и будет темой данной главы.

DEFCLASS

Вы можете создать собственный класс с помощью макроса DEFCLASS. Поскольку поведение класса определяется обобщенными функциями, и методами специализированными для класса, то DEFCLASS отвечает только за определение класса как типа данных.

Класс, как тип данных, состоит из трех частей: имени, отношения к другим классам и имен слотов. Базовая форма DEFCLASS выглядит достаточно просто.

```
(defclass name (direct-superclass-name*)  
  (slot-specifier*))
```

Что такое классы, определенные пользователем?

*"Определенные пользователем классы" – термин не из стандарта языка. Определёнными пользователем классами я называю подклассы класса STANDARD-OBJECT, а также классы, у которых метакласс – STANDARD-CLASS. Но поскольку я не собираюсь говорить о способах определения классов, которые не наследуют STANDARD-OBJECT и чей метакласс – это не STANDARD-CLASS, вам можно не обращать на это внимания. Определённые пользователем – не идеальный термин, поскольку так же реализация может называть некоторые классы. *FIXME* Определённые пользователем классы не идеальный термин, потому что реализация может определять некоторые классы одним способом. Но ещё большей путаницей будет называть эти классы стандартными, поскольку встроенные классы (например, INTEGER и STRING) тоже стандартные, если не сказать больше, потому что они определены стандартом языка, но они не расширяют (не наследуют) STANDARD-OBJECT. Чтобы ещё больше запутать дело, пользователь может также определять классы, не наследующие STANDARD-OBJECT. В частности, макрос DEFSTRUCT тоже определяет новые классы. Но это во многом для обратной совместимости – DEFSTRUCT появился раньше, чем CLOS и был изменен, чтоб определять классы, когда CLOS добавлялся в язык. Но создаваемые им классы достаточно ограничены по сравнению с классами, созданными с помощью DEFCLASS. Итак, я буду обсуждать только классы, создаваемые с помощью DEFCLASS, которые используют заданный по умолчанию метакласс STANDARD-CLASS и, за неимением лучшего термина, назову их "определёнными пользователем классами".*

Также как с функциями и переменными, вы можете использовать в качестве имени класса любой символ. Имена классов находятся в собственном пространстве имен, отдельно от имен функций и переменных, так что вы можете задать для класса то же самое имя, что и существующие функция и переменная. Вы будете использовать имя класса в качестве аргумента функции `MAKE-INSTANCE`, которая создает новые экземпляры классов, определенных пользователем.

Опция `direct-superclass-names` используется для указания имен классов, от которых будет проводиться наследование данного класса. Если ни одного класса не указано, то он будет унаследован от `STANDARD-OBJECT`. Все классы указанные в данной опции должны быть классами, определенными пользователем, чтобы быть уверенным, что каждый новый класс происходит от `STANDARD-OBJECT`. `STANDARD-OBJECT` является подклассом `T`, так что все классы, определенные пользователем, являются частью одной иерархии классов, которая также содержит все встроенные классы.

На время отвлекаясь от упоминания спецификаторов слотов, запись `DEFCLASS` для некоторых из классов, которые мы использовали в предыдущей главе, может выглядеть следующим образом:

```
(defclass bank-account () ...)
(defclass checking-account (bank-account) ...)
(defclass savings-account (bank-account) ...)
```

В разделе "Множественное наследование" я *ФИКМЕ* опишу, что означает указание более чем одного суперкласса в списке опции `direct-superclass-names`.

Спецификаторы слотов

Большая часть `DEFCLASS` состоит из списка спецификаторов слотов. Каждый спецификатор определяет слот, который будет частью экземпляра класса. Каждый слот в экземпляре является местом, который может хранить значение, к которому можно получить доступ через функцию `SLOT-VALUE`. `SLOT-VALUE` в качестве аргументов принимает объект и имя слота и возвращает значение нужного слота в данном объекте. Эта функция может использоваться вместе с `SETF` для установки значений слота в объекте.

Класс также наследует спецификаторы слотов от своих суперклассов, так что набор слотов, присутствующих в любом объекте, является объединением всех слотов, указанных в форме `DEFCLASS` для класса, а также указанных для всех его суперклассов.

По минимуму, спецификатор слота указывает его имя, так что спецификатор может быть простым именем. Например, вы можете определить класс `bank-account` с двумя слотами — `customer-name` и `balance`, например, вот так:

```
(defclass bank-account ()
  (customer-name
   balance))
```

Каждый экземпляр этого класса содержит два слота: один для хранения имени клиента, а второй — для хранения текущего баланса счета. Используя данное определение вы можете создать новые объекты `bank-account` с помощью `MAKE-INSTANCE`.

```
(make-instance 'bank-account) ==> #<BANK-ACCOUNT @ #x724b93ba>
```

Аргументом `MAKE-INSTANCE` является имя класса, а возвращаемым значением — новый объект. Печатное представление объекта определяется обобщенной функцией `PRINT-OBJECT`. В этом случае, подходящим методом будет тот, который предоставляется реализацией и

специализированный для STANDARD-OBJECT. Поскольку, не каждый объект может быть выведен таким образом, чтобы потом быть считанным назад, то метод печати для STANDARD-OBJECT использует синтаксис #<>, который заставит процедуру чтения выдать ошибку, если он попытается прочесть его. Оставшаяся часть представления зависит от реализации, но обычно оно похоже на результат, приведенный выше, включая имя класса и некоторое значение, например, адрес объекта в памяти. В главе 23 вы увидите пример того, как определить метод для PRINT-OBJECT чтобы некоторые классы могли выдавать больше информации *FIXME* при печати.

Используя данное определение bank-account, новые объекты будут создаваться со слотами, которые не связаны со значениями. Любая попытка получить значение для несвязанного значения приведет к выдаче ошибки, так что вы должны задать значение до того, как вы будете считывать значения.

```
(defparameter *account* (make-instance 'bank-account)) ==> *ACCOUNT*
(setf (slot-value *account* 'customer-name) "John Doe") ==> "John Doe"
(setf (slot-value *account* 'balance) 1000) ==> 1000
```

Теперь вы можете получать значения слотов.

```
(slot-value *account* 'customer-name) ==> "John Doe"
(slot-value *account* 'balance) ==> 1000
```

Инициализация объекта

Поскольку мы не можем сделать ничего полезного с объектом, который имеет несвязанные слоты, то было бы хорошо иметь возможность создавать объекты с *FIXME* уже инициализированными слотами. Common Lisp имеет *FIXME* предоставляет три способа управления начальными значениями слотов. Первые два требуют добавления опций в спецификаторы слотов в DEFCLASS: с помощью опции :initarg вы можете указать имя, которое потом будет использоваться как именованный параметр при вызове MAKE-INSTANCE и переданное значение будет сохранено в слоте. Вторая опция – :initform, позволяет вам указать выражение на Lisp, которое будет использоваться для вычисления значения, если при вызове MAKE-INSTANCE не был передан аргумент :initarg. В заключение, для полного контроля за инициализацией объекта, вы можете определить метод для обобщенной функции INITIALIZE-INSTANCE, который вызывается MAKE-INSTANCE.

Спецификатор слота, который включает опции, такие как :initarg или :initform, записывается как список, начинающийся с имени слота, за которым следуют опции. Например, если вы измените определение bank-account таким образом, чтобы позволить передавать имя клиента и начальный баланс при вызове MAKE-INSTANCE, а также чтобы установить для баланса начальное значение равное нулю, вы должны написать:

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name)
   (balance
    :initarg :balance
    :initform 0)))
```

Теперь вы можете одновременно создавать счет и указывать значения слотов.

```
(defparameter *account*  
  (make-instance 'bank-account :customer-name "John Doe" :balance 1000))
```

```
(slot-value *account* 'customer-name) ==> "John Doe"  
(slot-value *account* 'balance) ==> 1000
```

Если вы не передадите аргумент `:balance` при вызове `MAKE-INSTANCE`, то вызов `SLOT-VALUE` для слота `balance` будет получен вычислением формы, указанной опцией `:initform`. Но, если вы не передадите аргумент `:customer-name`, то слот `customer-name` будет не связан, и попытка считывания значения из него, приведет к выдаче ошибки.

```
(slot-value (make-instance 'bank-account) 'balance) ==> 0  
(slot-value (make-instance 'bank-account) 'customer-name) ==> error
```

Если вы хотите убедиться, что имя клиента было задано при создании счета, то вы можете выдать ошибку в начальном выражении (`initform`), поскольку оно будет вычислено только если начальное значение (`initarg`) не было задано. Вы также можете использовать начальные формы, которые создают разные значения при каждом запуске – начальное выражение вычисляется заново для каждого объекта. Для экспериментирования с этими возможностями, вы можете изменить спецификатор слота `customer-name` и добавить новый слот, `account-number`, который инициализируется значением увеличивающегося счетчика.

```
(defvar *account-numbers* 0)  
(defclass bank-account ()  
  ((customer-name  
    :initarg :customer-name  
    :initform (error "Must supply a customer name."))  
   (balance  
    :initarg :balance  
    :initform 0)  
   (account-number  
    :initform (incf *account-numbers*))))
```

В большинстве случаев, комбинации опций `:initarg` и `:initform` будет достаточно для нормальной инициализации объекта. Однако, хотя начальное выражение может быть любым выражением Lisp, оно не имеет доступа к инициализируемому объекту, так что оно не может инициализировать один слот, основываясь на значении другого. Для выполнения такой задачи вам необходимо определить метод для обобщенной функции `INITIALIZE-INSTANCE`.

Основной метод `INITIALIZE-INSTANCE`, специализированный для `STANDARD-OBJECT` берет на себя заботу об инициализации слотов, основываясь на данных, заданных опциями `:initarg` и `:initform`. Поскольку вы не захотите вмешиваться в этот процесс, то наиболее широко применяемым способом является определение метода `:after`, специализированного для вашего класса. Например, предположим, что вы хотите добавить слот `account-type`, который должен быть установлен в значение `:gold`, `:silver` или `:bronze`, основываясь на начальном балансе счета. Вы можете изменить определение класса на следующее, добавляя слот `account-type` без каких либо опций:


```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))
   account-type))
```

После этого вы можете определить метод `:after` для `INITIALIZE-INSTANCE`, который установит значение слота `account-type`, основываясь на значении, которое было сохранено в слоте `balance`.

```
(defmethod initialize-instance :after ((account bank-account) &key)
  (let ((balance (slot-value account 'balance)))
    (setf (slot-value account 'account-type)
          (cond
            ((>= balance 100000) :gold)
            ((>= balance 50000) :silver)
            (t :bronze)))))
```

Указание `&key` в списке параметров требуется чтобы сохранить список параметров подобным **FIXME** *соответствующим* списку параметров обобщенной функции— список параметров, указанный для функции `INITIALIZE-INSTANCE` включает `&key` чтобы позволить отдельным методам передавать собственные именованные параметры, но при этом, он не требует указания конкретных названий. Таким образом, каждый метод должен указывать `&key`, даже если он не указывает ни одного именованного параметра.

С другой стороны, если метод `INITIALIZE-INSTANCE`, специализированный для конкретного класса, указывает именованный параметр, то этот параметр становится допустимым параметром для функции `MAKE-INSTANCE` при создании экземпляра данного класса. Например, если банк иногда платит процент начального баланса в качестве премии при открытии счета, то вы можете реализовать эту функцию, используя метод `INITIALIZE-INSTANCE`, который получает именованный аргумент, указывающий процент премии, например вот так:

```
(defmethod initialize-instance :after ((account bank-account)
                                       &key opening-bonus-percentage)
  (when opening-bonus-percentage
    (incf (slot-value account 'balance)
          (* (slot-value account 'balance) (/ opening-bonus-percentage 100)))))
```

Путем определения метода `INITIALIZE-INSTANCE`, вы делаете `:opening-bonus-percentage` допустимым аргументом функции `MAKE-INSTANCE` при создании объекта `bank-account`.

```
CL-USER> (defparameter *acct* (make-instance
                                'bank-account
                                :customer-name "Sally Sue"
                                :balance 1000
                                :opening-bonus-percentage 5))
*ACCT*
CL-USER> (slot-value *acct* 'balance)
1050
```

Функции доступа

MAKE-INSTANCE и SLOT-VALUE дают вам возможности для создания и работы с экземплярами ваших классов. Все остальные операции могут быть реализованы в терминах этих двух функций. Однако, как знает всякий, знакомый с принципами правильного объектно-ориентированного программирования, прямой доступ к слотам (полям или переменным-членам) объекта может привести к получению уязвимого кода. Проблема заключается в том, что прямой доступ к слотам делает ваш код слишком связанным с конкретной структурой классов. Например, предположим, что вы решили изменить определение bank-account таким образом, что вместо хранения текущего баланса в виде числа, вы храните его в виде списка списаний и помещений денег на счет, вместе с датами этих операций. Код, который имеет прямой доступ к слоту balance скорее всего будет сломан, если вы измените определение класса, удалив слот или храня список в данном слоте. С другой стороны, если вы определите функцию balance, которая осуществляет доступ к слоту, то вы можете позже переопределить ее, чтобы сохранить ее поведение, даже если изменится внутреннее представление данных. И код, который использует такую функцию будет продолжать нормально работать не требуя внесения изменений.

Другим преимуществом использования функций доступа вместо прямого доступа к слотам через SLOT-VALUE является то, что их использование позволяет вам ограничить возможность модификации слота *FIXME* из внешнего кода. Для пользователей класса bank-account может быть удобным использование функций доступа для получения текущего баланса, но вы можете захотеть, чтобы все изменения баланса производились через другие предоставляемые вами функции, такие как deposit и withdraw. Если клиент знает, что он сможет работать с объектами только через определенный набор функций, то вы можете предоставить ему функцию balance, но сделать так, чтобы для нее нельзя было выполнить SETF, чтобы баланс был доступен только для чтения.

В заключение, использование функций доступа делает ваш код более аккуратным, поскольку вы избегаете использования множества менее понятных функций SLOT-VALUE.

Определение функции, которая читает содержимое слота balance является тривиальным.

```
(defun balance (account)
  (slot-value account 'balance))
```

Однако, если вы знаете, что вы будете определять подклассы для bank-account, то может быть хорошей идеей определение balance в качестве обобщенной функции. Таким образом вы можете определить разные методы для balance для некоторых подклассов, или расширить ее возможности с помощью вспомогательных методов. Так что вместо предыдущего примера можете написать следующее:

```
(defgeneric balance (account))
(defmethod balance ((account bank-account))
  (slot-value account 'balance))
```

Как я только что обсуждал, вы не хотите, чтобы пользователь имел возможность устанавливать баланс напрямую, но для других слотов, таких как customer-name, вы также можете захотеть предоставить функцию для установки их значений. Наиболее понятным способом будет определение такой функции как SETF-функции.

SETF-функция является способом расширения функциональности SETF, определяя новый вид места (place), для которого известно как устанавливать его значение. Имя SETF-функции является списком из двух элементов, где первый элемент является символом setf, а второй –

другим символом, обычно именем функции, которая используется для доступа к месту, значение которого будет устанавливать функция SETF. SETF-функция может получать любое количество аргументов, но первым аргументом всегда является значение, присваиваемое выбранному месту. Например, вы можете определить SETF-функцию для установки значения слота `customer-name` в классе `bank-account` следующим образом:

```
(defun (setf customer-name) (name account)
  (setf (slot-value account 'customer-name) name))
```

После вычисления этого определения, выражения, подобные этому:

```
(setf (customer-name my-account) "Sally Sue")
```

будут компилироваться как вызов SETF-функции, которую вы только что определили с значением "Sally Sue" в качестве первого аргумента, и значением `my-account` в качестве второго аргумента.

Конечно, также как с функциями чтения, вы вероятно захотите, чтобы ваша SETF-функция была обобщенной, так что вы должны ее определить примерно так:

```
(defgeneric (setf customer-name) (value account))
(defmethod (setf customer-name) (value (account bank-account))
  (setf (slot-value account 'customer-name) value))
```

И конечно, вы также можете определить функцию чтения для `customer-name`.

```
(defgeneric customer-name (account))
(defmethod customer-name ((account bank-account))
  (slot-value account 'customer-name))
```

Это позволит вам писать следующим образом:

```
(setf (customer-name *account*) "Sally Sue") ==> "Sally Sue"
(customer-name *account*) ==> "Sally Sue"
```

Нет ничего сложного в написании этих функций доступа, но написание этих функций вручную просто не соответствует The Lisp Way. Так что DEFCLASS поддерживает три опции для слотов, которые позволяют вам автоматически создавать функции чтения и записи значений отдельных слотов.

Опция `:reader` указывает имя, которое будет использоваться как имя обобщенной функции, которая принимает объект в качестве своего единственного аргумента. Когда вычисляется DEFCLASS, то создается соответствующая обобщенная функция (если она еще не определена) `DELEТЕМЕскобки`. После этого для данной обобщенной функции создается метод, специализированный для нового класса и возвращающий значение слота. Имя функции может быть любым, но обычно используют то же самое имя, что и имя самого слота. Так что вместо явного задания обобщенной функции `balance` и метода для нее, как это было показано раньше, вы можете просто изменить спецификатор слота `balance` в определении класса `bank-account` на следующее:

```
(balance
  :initarg :balance
  :initform 0
  :reader balance)
```

Опция `:writer` используется для создания обобщенной функции и метода для установки значения слота. Создаваемая функция и метод следуют требованиям для SETF-функции, получая новое значение как первый аргумент, и возвращая его в качестве результата, так что вы можете определить SETF-функцию задавая имя, такое как `(setf customer-name)`. Например, вы можете определить методы чтения и записи для слота `customer-name`, просто изменяя спецификатор слота на следующее определение:

```
(customer-name
  :initarg :customer-name
  :initform (error "Must supply a customer name.")
  :reader customer-name
  :writer (setf customer-name))
```

Поскольку достаточно часто требуется определение обеих функций доступа, то DEFCLASS также имеет опцию `:accessor`, которая создает и функцию чтения, и соответствующую SETF-функцию. Так что вместо предыдущего примера можно написать следующим образом:

```
(customer-name
  :initarg :customer-name
  :initform (error "Must supply a customer name.")
  :accessor customer-name)
```

В заключение, опишу еще одну опцию, о которой вы должны знать: опция `:documentation` позволяет вам задать строку, которая описывает данный слот. Собирая все в кучу и добавляя методы чтения для слотов `account-number` и `account-type`, определение DEFCLASS для класса `bank-account` будет выглядеть примерно так:

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name.")
    :accessor customer-name
    :documentation "Customer's name")
   (balance
    :initarg :balance
    :initform 0
    :reader balance
    :documentation "Current account balance")
   (account-number
    :initform (incf *account-numbers*)
    :reader account-number
    :documentation "Account number, unique within a bank.")
   (account-type
    :reader account-type
    :documentation "Type of account, one of :gold, :silver, or :bronze.)))
```

"WITH-SLOTS" и "WITH-ACCESSORS"

В то время как функции доступа делают ваш код более легким для сопровождения, они все еще достаточно многословны. И конечно будут моменты, когда вы будете писать методы, которые

реализуют низкоуровневое поведение класса, так что вы можете осознанно осуществлять доступ к слотам для установки значений слотов, для которых нет функций записи, или для получения значений из слотов, без использования функций чтения.

Это как раз тот случай, для которого и предназначен макрос `SLOT-VALUE`; однако, он также достаточно многословен. Если функция или метод осуществляют доступ к одному и тому же слоту несколько раз, то исходный код будет засорен вызовами функций доступа и `SLOT-VALUE`. Например, даже достаточно простой метод, такой как следующий пример, который вычисляет пеню для `bank-account` если баланс снижается ниже некоторого минимума, будет засорен вызовами `balance` и `SLOT-VALUE`:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (balance account) *minimum-balance*)
    (decf (slot-value account 'balance) (* (balance account) .01)))))
```

И если вы решите, что вы хотите осуществлять прямой доступ к слоту для того, чтобы избежать вызова вспомогательных методов, то ваш код будет еще больше замусоренным.

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (slot-value account 'balance) *minimum-balance*)
    (decf (slot-value account 'balance) (* (slot-value account 'balance) .01)))))
```

Два стандартных макроса – `WITH-SLOTS` и `WITH-ACCESSORS`, могут помочь избавиться от этого мусора. Оба макроса создают блок кода, в которых могут использоваться простые имена переменных для обращения к слотам определенного объекта. `WITH-SLOTS` предоставляет прямой доступ к слоту, также как при использовании `SLOT-VALUE`, в то время как `WITH-ACCESSORS` предоставляет сокращенный способ вызова функций доступа.

Базовая форма `WITH-SLOTS` выглядит следующим образом:

```
(with-slots (slot*) instance-form
  body-form*)
```

Каждый элемент списка `slot` может быть либо именем слота, *FIXME* которое также будет именем переменной, либо списком из двух элементов, где первый аргумент является именем, которое будет использоваться как переменная, а второй – именем соответствующего слота. Выражение `instance-form` вычисляется один раз для получения объекта, к слотам которого будет производиться доступ. Внутри тела макроса, каждое вхождение имени переменной преобразуется в вызов `SLOT-VALUE` с использованием объекта и имени слота в качестве аргументов. Таким образом, вы можете переписать `assess-low-balance-penalty` вот так:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots (balance) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

или используя списочную запись, вот так:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots ((bal balance)) account
    (when (< bal *minimum-balance*)
      (decf bal (* bal .01)))))
```

Если вы определили `balance` с использованием опции `:accessor`, а не `:reader`, то вы также

можете использовать макрос `WITH-ACCESSORS`. Форма `WITH-ACCESSORS` совпадает с `FIXME` такая же как `WITH-SLOTS` за тем исключением, что каждый элемент списка слотов является списком из двух элементов, содержащих имя переменной и имя функции доступа. Внутри тела `WITH-ACCESSORS`, ссылка на одну из переменных, аналогична вызову соответствующей функции доступа. Если функция доступа разрешает выполнение `SETF`, то тоже самое возможно и для переменной.

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-accessors ((balance balance)) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

Первое вхождение `balance` является именем переменной, а второе – именем функции доступа; они не обязательно должны быть одинаковыми. Например, вы можете написать метод для слияния двух счетов, используя два вызова `WITH-ACCESSORS`, для каждого из счетов.

```
(defmethod merge-accounts ((account1 bank-account) (account2 bank-account))
  (with-accessors ((balance1 balance)) account1
    (with-accessors ((balance2 balance)) account2
      (incf balance1 balance2)
      (setf balance2 0))))
```

Выбор между использованием `WITH-SLOTS` и `WITH-ACCESSORS` примерно таков, как и выбор между использованием `SLOT-VALUE` и функций доступа: низкоуровневый код, которые обеспечивает основную функциональность класса, может использовать `SLOT-VALUE` или `WITH-SLOTS` для прямой работы со слотами `FIXME`, если функции доступа не поддерживают нужный стиль работы, или `FIXME` если хочется явно избежать использования вспомогательных методов, которые могут быть определены для функций доступа. Но в общем вы должны использовать функции доступа или `WITH-ACCESSORS`, если только у вас не имеются конкретные причины не делать этого.

Слоты, выделяемые для классов

Заключительной опцией, которую вам необходимо знать, является опция `:allocation`. Значением опции `:allocation` может быть либо `:instance`, либо `:class`, и по умолчанию оно равно `:instance`, если оно не было явно указано. Когда слот имеет значение опции равное `:class`, то слот имеет только одно значение, которое сохраняется внутри класса и используется всеми экземплярами.

Однако доступ к слотам со значением `:class` производится также как и для слотов со значением `:instance` – доступ производится с помощью `SLOT-VALUE` или функции доступа, что значит, что вы можете получить доступ только через экземпляр класса, хотя это значение не хранится в этом экземпляре. Опции `:initform` и `:initarg` имеют точно такой же эффект, за тем исключением, что начальное выражение вычисляется один раз, при определении класса, а не при создании экземпляра. С другой стороны, передача начальных аргументов `MAKE-INSTANCE` установит значение, затрагивая все экземпляры данного класса.

Поскольку вы не можете получить слот, выделенный для класса, не имея экземпляра класса, то такие слоты не являются полным аналогом статическим членам в таких языках как Java, C++ и Python. В значительной степени, слоты выделенные для класса в основном используются для уменьшения потребляемой памяти; если вы создаете много экземпляров класса и они все имеют ссылку на один и тот же объект (например, список разделяемых ресурсов), то вы можете сократить использование памяти путем объявления такого слота, выделяемым для класса, а не для экземпляра.

Слоты и наследование

Как обсуждалось в предыдущей главе, классы наследуют поведение от своих суперклассов благодаря механизмам обобщенных функции – метод специализированный для класса `A` также применим не только к экземплярам класса `A`, но также и к экземплярам классов, унаследованных от `A`. Классы также наследуют от своих суперклассов слоты, но этот механизм немного отличается.

В Common Lisp конкретный объект может иметь только один слот с определенным именем. Однако возможно, что в иерархии наследования класса несколько классов будут иметь слоты с одним и тем же именем. Это может случиться либо потому, что подкласс включает спецификатор слота с тем же именем, что и слот указанный в суперклассе, либо потому, что несколько суперклассов имеют слоты с одним и тем же именем.

Common Lisp решает эту проблему путем слияния всех спецификаторов с одним и тем же именем из нового класса и всех его суперклассов для создания отдельных спецификаторов для каждого уникального имени слота. При слиянии спецификатор, разные опции спецификаторов слотов рассматриваются по разному. Например, поскольку слот может иметь только одно значение по умолчанию, то если несколько классов указывают опцию `:initform`, то новый класс будет использовать эту опцию из наиболее специализированного класса. Это позволяет подклассам указывать собственное значение по умолчанию, а не те, которые были унаследованы.

С другой стороны, опции `:initargs` не должны быть взаимоисключающими – каждая опция `:initarg` создает именованный параметр, который может быть использован для инициализации слота; множественные параметры не приводят к конфликту, так что новый спецификатор слота будет содержать все опции `:initargs`. Пользователи, вызывающие `FIXME` *Вызывающие* `MAKE-INSTANCE` могут использовать любое из имен, указанных в `:initargs` для инициализации слота. Если пользователь `FIXME` *Если вызывающий* указывает несколько именованных аргументов, которые инициализируют один и тот же слот, то используется то, которое стоит левее всех остальных в списке аргументов `MAKE-INSTANCE`.

Унаследованные опции `:reader`, `:writer` и `:accessor` не включаются в новый спецификатор слота, поскольку методы, созданные при объявлении суперкласса будут автоматически применяться к новому классу. Однако новый класс может создать свои собственные функции доступа, путем объявления собственных опций `:reader`, `:writer` или `:accessor`.

И в заключение, опция `:allocation`, подобно `:initform`, определяется наиболее специализированным классом, определяющим данный слот. Таким образом, возможно сделать так, что экземпляры одного класса будут использовать слот с опцией `:class`, а экземпляры его подклассов могут иметь свои собственные значения опции `:instance` для слота с тем же именем. А их подклассы, в свою очередь, могут переопределить этот слот с опцией `:class`, так что все экземпляры данного класса снова будут делить между собой единственный экземпляр слота. В последнем случае, слот, разделяемый экземплярами под-подклассов отличается от слота, разделяемого оригинальным суперклассом.

Например, у вас имеются следующие классы:

```
(defclass foo ()
  ((a :initarg :a :initform "A" :accessor a)
   (b :initarg :b :initform "B" :accessor b)))
(defclass bar (foo)
  ((a :initform (error "Must supply a value for a"))
   (b :initarg :the-b :accessor the-b :allocation :class)))
```

При создании экземпляра класса `bar`, вы можете использовать унаследованный начальный аргумент `:a` для указания значения для слота `a` и, в действительности, должны сделать это для того, чтобы избежать ошибок, поскольку опция `:initform` определенная `bar` замещает опцию, унаследованную от `foo`. Для инициализации слота `b`, вы можете использовать либо унаследованный аргумент `:b`, либо новый аргумент `:the-b`. Однако, поскольку для слота `b` в определении `bar` указана опция `:allocation`, то указанное значение будет храниться в слоте, используемом всеми экземплярами `bar`. Доступ к этому слоту может быть осуществлен либо с помощью метода обобщенной функции `b`, специализированного для `foo`, либо с помощью нового метода обобщенной функции `the-b`, который специализирован для `bar`. Для доступа к слоту `a` классов `foo` или `bar`, вы продолжите использовать обобщенную функцию `a`.

Обычно, слияние определений слотов происходит достаточно гладко. Однако важно помнить, что при использовании множественного наследования два не относящихся друг к другу слота, имеющих одно и то же имя, в новом классе будут слиты в один слот. Так что методы, специализированные для разных классов могут работать с одним и тем же слотом, когда они будут применяться к классу, унаследованному от этих классов. На практике это не доставляет особых проблем, поскольку, как вы увидите в главе 21, вы можете использовать пакетную систему для того, чтобы избежать коллизий между именами в коде.

Множественное наследование

Все классы, которые вы до сих пор видели имели только один суперкласс. Common Lisp также поддерживает множественное наследование – класс может иметь несколько прямых суперклассов, наследуя соответствующие методы и спецификаторы слотов из всех этих классов.

Множественное наследование не вносит кардинальных изменений в механизмы наследования, которые я уже обсуждал – каждый класс определенный пользователем уже имеет несколько суперклассов, поскольку они все наследуются от `STANDARD-OBJECT`, который унаследован от `T`, так что по крайней мере имеется два суперкласса. Затруднение, которое вносит множественное наследование заключается в том, что класс может иметь более одного непосредственного суперкласса. Это усложняет понятие специфичности класса, которое используется при построении эффективных методов для обобщенных функции и при слиянии спецификаторов слотов.

Так что, если классы должны *FIXME*Если бы классы могли иметь только один непосредственный суперкласс, то упорядочение классов по специфичности будет тривиальным – класс и все его суперклассы могут быть выстроены в линию начиная с самого класса, за которым следует один прямой суперкласс, за которым следует его суперкласс, и так далее, до класса `T`. Но когда класс имеет несколько непосредственных суперклассов, то эти классы обычно не относятся друг к другу – конечно, если один класс был подклассом другого, вам не нужно образовывать подкласс от обоих. В этом случае, правила по которому подклассы более специфичны чем суперклассы недостаточно для упорядочения всех суперклассов. Так что Common Lisp использует второе правило, которое сортирует не относящиеся друг к другу суперклассы по порядку в котором они перечислены в определении непосредственных суперклассов в `DEFCLASS` – классы, указанные в списке первыми, считаются более специфичными, чем классы, указанные в списке последними. Это правило считается достаточно произвольным, но оно позволяет каждому классу иметь линейный список приоритетов *FIXME*список следования классов, который может использоваться для определения того, какой из суперклассов будет считаться более специфичным чем другой. Однако заметьте, что нет глобального упорядочения классов – каждый класс имеет собственный список приоритетов *FIXME*список следования классов, и одни и те же классы могут стоять на разных позициях в списках приоритетов *FIXME*списках следования классов разных классов.

Для того, чтобы увидеть как это работает, давайте добавим новый класс к нашему банковскому приложению: `money-market-account`. Этот счет объединяет в себе характеристики чекового (`checking-account`) и сберегательного (`savings-account`) счетов: клиент может выписывать чеки, но кроме того он получает проценты. Вы можете определить его следующим образом:

```
(defclass money-market-account (checking-account savings-account) ())
```

Список приоритетов класса `money-market-account` будет следующим:

```
(money-market-account
 checking-account
 savings-account
 bank-account
 standard-object
 t)
```

Заметьте, как список удовлетворяет обоим правилам: каждый класс появляется раньше своих суперклассов, а `checking-account` и `savings-account` располагаются в порядке, указанном в `DEFCCLASS`.

Этот класс не определяет своих собственных слотов, но унаследует слоты от обоих суперклассов, включая слоты, которые те унаследовали от своих суперклассов. Аналогичным образом, все методы, которые применимы к любому из классов в списке приоритетов, также будут применимы к объекту `money-market-account`. Поскольку все спецификаторы одинаковых слотов объединяются, то не имеет значения, что `money-market-account` дважды наследует одни и те же слоты из `bank-account`.

Множественное наследование наиболее просто понять когда суперклассы предоставляют совершенно независимые наборы слотов и методов. Например, `money-market-account` унаследует слоты и поведение по работе с чеками от `checking-account`, а слоты и поведение по вычислению процентов – от `savings-account`. Вам не нужно беспокоиться о списке приоритетов класса `money-market-account` в списке следования классов для методов и слотов, унаследованных только от одного или другого суперкласса.

Однако, также возможно унаследовать методы для одних и тех же обобщенных функций от различных суперклассов. В этом случае, в игру включается список приоритетов классов. *Список следования классов*. Например, предположим, что банковское приложение определяет обобщенную функцию `print-statement`, которая используется для генерации месячных отчетов. Вероятно, что уже будут определены методы `print-statement` специализированные для `checking-account` и `savings-account`. Оба этих метода будут применимы для экземпляров класса `money-market-account`, но тот, который специализирован для `checking-account` будет считаться более специфичным, чем специализированный для `savings-account`, поскольку `checking-account` имеет приоритет перед `savings-account` в списке приоритетов классов `money-market-account`.

Предполагается, что унаследованные методы являются основными методами, и вы не определяли других методов, специализированных для `checking-account`, которые будут использоваться, если вы выполните `print-statement` для `money-market-account`. Однако, это не обязательно даст вам то поведение, которое вы хотите, поскольку вы хотите чтобы отчет для нового счета содержал элементы из отчетов по чековому и сберегательному счетам.

Вы можете изменить поведение `print-statement` для `money-market-account` несколькими способами. Непосредственным способом является определение основного метода,

специализированного для `money-market-account`. Это даст вам полный контроль за поведением, но вероятно потребует написания кода для опций, которые я буду вскоре обсуждать. Проблема заключается в том, что хотя вы можете использовать `CALL-NEXT-METHOD` для передачи управления "вверх", следующему методу, а именно, специализированному для `checking-account`, но не существует способа вызвать конкретный менее специфичный метод, например, специализированный для `savings-account`. Так что если вы хотите иметь возможность использования кода, который создает часть отчета, специфичную для `savings-account`, то вам нужно разбить этот код на отдельные функции, которые вы сможете вызвать напрямую из методов `print-statement` классов `money-market-account` и `savings-account`.

Другой возможностью является написание основных методов всех трех классов так, чтобы они вызывали `CALL-NEXT-METHOD`. Тогда метод, специализированный для `money-market-account` будет использовать `CALL-NEXT-METHOD` для вызова метода, специализированного для `checking-account`. Затем, этот метод вызовет `CALL-NEXT-METHOD`, что приведет к запуску метода для `savings-account`, поскольку он будет следующим наиболее специфичным методом в списке приоритетов классов для `money-market-account`.

Конечно, если вы не хотите полагаться на соглашения о стиле кодирования (что каждый метод будет вызывать `CALL-NEXT-METHOD`) чтобы убедиться, что все применимые методы будут вызваны в некоторый момент времени, вы должны подумать об использовании вспомогательных методов. В этом случае, вместо определения основного метода `print-statement` для `checking-account` и `savings-account`, вы можете определить их как методы `:after`, оставляя один основной метод для `bank-account`. Так что `print-statement`, вызванный для `money-market-account`, выдаст базовую информацию о счете, которая будет выведена основным методом, специализированным для `bank-account`, за которым следуют дополнительные детали, выведенные методами `:after` специализированными для `savings-account` и `checking-account`. И если вы хотите добавить детали, специфичные для `money-market-accounts`, вы можете определить метод `:after`, специализированный для `money-market-account`, который будет выполнен последним.

Преимуществом использования вспомогательных методов является то, что становится понятным какие из методов является ответственным за реализацию обобщенной функции, и какие из них вносят дополнительные детали в работу функции. Недостатком этого подхода является то, что вы не получаете точного контроля за порядком, в котором будут выполняться вспомогательные методы – если вы хотите, чтобы часть отчета, приготовленного для `checking-account` печаталась перед частью `savings-account`, то вы должны изменить порядок в котором `money-market-account` наследуются от этих классов. Но это достаточно трагическое изменение, которое затрагивает другие методы и унаследованные слоты. В общем, если вы обнаружите, что рассматриваете изменение списка непосредственных суперклассов как способ тонкой настройки поведения специфических методов, то вы скорее всего должны сделать шаг назад и заново обдумать ваш подход.

С другой стороны, если вы не заботитесь о порядке наследования, но хотите, чтобы он был последовательным для разных обобщенных функций, то использование вспомогательных методов может быть одним из методов. Например, если в добавление к `print-statement` вы имеете функцию `print-detailed-statement`, то вы можете реализовать обе функции используя методы `:after` для разных подклассов `bank-account`, и порядок частей и для основного и детального отчета будет одинаков.

Правильный объектно-ориентированный дизайн

Это все о главных возможностях объектной системы Common Lisp. Если у вас имеется большой

опыт объектно-ориентированного программирования, вы вероятно увидите как возможности Common Lisp могут быть использованы для реализации правильного объектно-ориентированного дизайна. Однако, если у вас небольшой опыт объектно-ориентированного программирования, то вам понадобится провести некоторое время чтобы освоиться с объектно-ориентированным мышлением. К сожалению это достаточно большой раздел, находящийся за пределами данной книги. Или, как указано в справочной странице по объектной системе Perl, "Теперь, вам нужно лишь выйти и купить книгу о методологии объектно-ориентированного дизайна, и провести с нею следующие шесть месяцев". Или вы можете продолжить чтение до практических глав, далее в этой книге, где вы увидите несколько примеров того, как эти возможности используются на практике. Однако сейчас, вы готовы к тому, чтобы взять перерыв и перейти от теории объектно-ориентированного программирования к другой теме – как можно полезно использовать мощную, но немного загадочную функцию Common Lisp – FORMAT.

18. Несколько рецептов для функции FORMAT

Функция FORMAT вместе с расширенным макросом LOOP - одна из двух возможностей Common Lisp, которые вызывают сильную эмоциональную реакцию у многих пользователей Common Lisp. Некоторые их любят, другие ненавидят

Поклонники FORMAT любят ее за мощь и краткость, в то время как противники ее ненавидят за потенциал для возникновения ошибок и непрозрачность. Сложные управляющие строки FORMAT имеют иногда подозрительное сходство с помехами на экране, но FORMAT остается популярным среди программистов на Common Lisp, которые хотят формировать небольшие куски удобочитаемого текста без необходимости нагромождать кучи формирующего вывод кода. Хотя управляющие строки FORMAT могут быть весьма замысловаты, но во всяком случае единственное FORMAT-выражение не сильно замусорит ваш код. Предположим например, что вы хотите напечатать значения в списке, разделенные запятыми. Вы можете написать так:

```
(loop for cons on list
      do (format t "~a" (car cons))
      when (cdr cons) do (format t ", ")))
```

Это не очень плохо, но любой, кто будет читать этот код, должен будет мысленно проанализировать его, прежде чем понять, что все, что он делает - это печать содержимого списка list на стандартный вывод. С другой стороны, вы можете с одного взгляда определить, что следующее выражение печатает список в некоторой форме на стандартный вывод:

```
(format t "~{~a~^, ~}" list)
```

Если вам важна форма, которую примет вывод, тогда вы можете изучить управляющую строку, но если все, что вы хотите - это приблизительно знать, что делает эта строка кода, то это можно увидеть сразу.

Во всяком случае, вам следует по меньшей мере зрительно воспринимать FORMAT, а еще лучше разобраться, на что она способна, прежде, чем вы примкнете к про- или анти-FORMAT'овскому лагерю. Также важно понимать по меньшей мере основы FORMAT, поскольку другие стандартные функции, такие, как функции выбрасывания условий, обсуждаемые в следующей главе, используют управляющие строки в стиле FORMAT для формирования вывода.

Чтобы сильнее запутать дело, FORMAT поддерживает три совершенно разных вида форматирования: печать таблиц с данными, структурная печать (*pretty printing*) s-выражений, и формирование удобочитаемых сообщений со вставленными FIXME (interpolated? вложенными?) значениями. Печать таблиц с текстовыми данными на сегодня несколько устарела; это одно из напоминаний, что Lisp стар, как FORTRAN. В действительности, некоторые директивы, которые вы можете использовать для печати значений с плавающей точкой внутри полей с фиксированной длиной были основаны прямо на edit descriptors FIXME (дескрипторах редактирования?) FORTRAN, которые использовались в FORTRAN для чтения и печати столбцов с данными, расположенными внутри полей с фиксированной длиной. Тем не менее, использование Common Lisp как замены FORTRAN выходят за рамки этой книги, так что я не буду обсуждать эти аспекты FORMAT.

Структурная печать также находится за рамками этой книги - не потому, что она устарела, а потому, что это слишком большая тема. Вкратце, механизм структурной печати Common Lisp - это настраиваемая система для печати блочно-структурированных данных, включая, но не ограничиваясь, s-выражениями, применяющаяся, когда необходимы переменные отступы и динамически увеличивающиеся переводы строк. Это отличный инструмент при необходимости,

но не часто требуемый в повседневном программировании.

Вместо этого я сосредоточусь на частях `FORMAT`, которые вы можете использовать, чтобы формировать удобочитаемые строки со вставленными в них значениями. Даже ограничивая обзор таким образом, остается изрядное количество материала. Не чувствуйте себя так, как будто вы обязаны помнить каждую деталь, описанную в этой главе. Вы можете достичь многого с лишь несколькими идиомами `FORMAT`. Сначала я опишу наиболее важные возможности `FORMAT`; а вам решать, насколько вы хотите стать волшебником `FORMAT`.

Функция `FORMAT`

Как вы видели в предыдущих главах, функция `FORMAT` принимает два аргумента: получатель для своего вывода, и управляющую строку, которая содержит буквенный текст и вложенные директивы. Любые дополнительные аргументы предоставляют значения, используемые директивами управляющей строки, которые вставляют эти значения в печатаемый текст. Я буду ссылаться на эти аргументы как на *аргументы формата*.

Первым аргументом `FORMAT`, получателем для печатаемого текста, может быть `T`, `NIL`, поток, или строка с указателем заполнения. `T` обозначает поток `*STANDARD-OUTPUT*`, в то время как `NIL` заставляет `FORMAT` сформировать свой вывод в виде строки, которую функция затем возвращает. Если получатель - поток, то вывод пишется в поток. А если получатель - строка с указателем заполнения, то форматированный вывод добавляется к концу строки и указатель заполнения соответственно выравнивается. За исключением случая, когда получатель - `NIL` и функция возвращает строку, `FORMAT` возвращает `NIL`.

Второй аргумент - управляющая строка, является, в сущности, программой на языке `FORMAT`. Язык `FORMAT` не целиком "лисповый" - его основной синтаксис основан на символах, а не на s-выражениях, и оптимизирован для краткости, а не для легкого понимания. Вот почему сложные управляющие строки `FORMAT` могут быть приняты за помехи.

Большинство из директив `FORMAT` просто вставляют аргумент внутрь выводимого текста в той или иной форме. Некоторые директивы, такие как `~%`, которая заставляет `FORMAT` выполнить перевод строки, не используют никаких аргументов. Другие, как вы увидите, могут использовать более одного аргумента. Одна из директив даже позволяет вам прыгать по списку аргументов, с целью обработки одного и того же аргумента несколько раз, или в некоторых ситуациях пропустить определенные аргументы. Но прежде, чем я буду обсуждать конкретные директивы, давайте взглянем на общий синтаксис директив.

Директивы `FORMAT`

Все директивы начинаются с тильды (`~`) и кончаются отдельным знаком, который идентифицирует директиву. Вы можете писать этот символ как в верхнем, так и в нижнем регистре. Некоторые директивы принимают *префиксные параметры*, которые пишутся непосредственно после тильды, разделяются запятыми, и используются для управления такими деталями, как - сколько разрядов печатать после десятичной точки при печати числа с плавающей точкой. Например, директива `~$`, одна из директив, использующихся для печати значений с плавающей точкой, по умолчанию печатает два разряда, следующие за десятичной точкой.

```
CL-USER> (format t "~$" pi)
3.14
NIL
```

Тем не менее с префиксным параметром вы можете указать чтобы функция печатала свой

аргумент, к примеру, с пятью десятичными знаками, как здесь:

```
CL-USER> (format t "~5$" pi)
3.14159
NIL
```

Значениями префиксного параметра являются либо числа, записанные как десятичные, или знаки, записанные в виде одинарной кавычки, за которой следует нужный символ. Значение префиксного параметра может быть также получено из аргумента формата двумя способами: префиксный параметр *v* заставляет FORMAT использовать один аргумент формата и назначить его значение префиксному параметру. Префиксный параметр *#* будет вычислен как количество оставшихся аргументов формата. Например:

```
CL-USER> (format t "~v$" 3 pi)
3.142
NIL
CL-USER> (format t "~#$" pi)
3.1
NIL
```

Я дам более правдоподобные примеры использования аргумента *#* в разделе "Условное форматирование".

Вы можете также опустить оба префиксных параметра. Впрочем, если вы хотите указать один параметр, но не желаете указывать параметры, стоящие перед ним, то вы должны включить запятую для каждого пропущенного параметра. Например, директива *~F*, другая директива для печати значений с плавающей точкой, также принимает параметр для управления количеством десятичных разрядов при печати, но это второй по счету параметр, а не первый. Если вы хотите использовать *~F* для печати числа с пятью десятичными разрядами, вы можете написать так:

```
CL-USER> (format t "~~,5f" pi)
3.14159
NIL
```

Также вы можете изменить поведение некоторых директив при помощи *модификаторов двоеточие* и знака *@*, которые ставятся после любого префиксного параметра и до идентифицирующего директиву знака. Эти модификаторы незначительно меняют поведение директивы. Например, с модификатором *двоеточие*, директива *~D*, используемая для вывода целых чисел в десятичном виде, создает число с запятыми, разделяющими каждые три разряда, в то время как знак *@* заставляет *~D* включить знак плюс в случае положительного числа.

```
CL-USER> (format t "~d" 1000000)
1000000
NIL
CL-USER> (format t "~:d" 1000000)
1,000,000
NIL
CL-USER> (format t "~@d" 1000000)
+1000000
NIL
```

В случае необходимости вы можете объединить модификаторы *двоеточие* и *@*, для того чтобы получить оба варианта:

```
CL-USER> (format t "~:@d" 1000000)
+1,000,000
NIL
```

В директивах, где оба модифицированных варианта поведения не могут быть осмысленно объединены, использование обоих модификаторов либо не определено или приобретает третье значение.

Основы Форматирования

Сейчас вы готовы познакомиться с конкретными директивами. Я начну с нескольких наиболее распространенных директив, включая несколько тех, которые вы уже встречали в предыдущих главах.

Наиболее универсальная директива - `~A`, которая использует один аргумент формата любого типа и печатает его в *эстетичной* (удобочитаемой) форме. Например, строки печатаются без кавычек и экранирующих символов (escape characters), а числа печатаются в форме, принятой для соответствующего числового типа. Если вы хотите просто получить значение, предназначенное для прочтения человеком, то эта директива - ваш выбор.

```
(format nil "The value is: ~a" 10) ==> "The value is: 10"
(format nil "The value is: ~a" "foo") ==> "The value is: foo"
(format nil "The value is: ~a" (list 1 2 3)) ==> "The value is: (1 2 3)"
```

Родственная директива, `~S`, также требует один аргумент формата любого типа, и печатает его. Однако `~S` пытается сформировать такой вывод, который мог бы быть прочитан обратно с помощью `READ`. Поэтому, строки должны быть заключены в кавычки, при необходимости знаки должны быть пакетно-специализированы `FIXME` (package-qualified?), и так далее. Объекты, которые не имеют подходящего для `READ` представления печатаются в нечитаемом синтаксисе объектов `FIXME`, `#<>`. С модификатором *двоеточие*, обе директивы `~A` и `~S` порождают `NIL` в виде `()`, а не как `NIL`. Обе директивы `~A` и `~S` также принимают до четырех префиксных параметров, которые могут быть использованы для выравнивания пробелами `FIXME` (padding?), добавляемыми после (или до, при модификаторе `@`) значения, впрочем эти функции действительно полезны лишь при формировании табличных данных.

Две другие часто используемые директивы - это `~%`, которая создает новую строку, и `~&`, которая выполняет *перевод строки*. Разница между ними в том, что `~%` всегда создает новую строку, тогда как `~&` срабатывает только если она уже не находится в начале строки. Это удобно при создании слабо связанных функций, каждая из которых формирует кусок текста, и их нужно объединить различными способами. Например, если одна из функции выводит текст, который кончается новой строкой (`~%`), а другая функция выводит некоторый текст, который начинается с перевода строки (`~&`), то вам не стоит беспокоиться о получении дополнительной пустой строки, если вы вызываете их одну за другой. Обе эти директивы могут принимать единственный префиксный параметр, который обозначает количество выводимых новых строк. Директива `~%` просто выведет заданное количество знаков новой строки, в то время как директива `~&` создаст либо `n - 1` либо `n` новых строк, в зависимости от того, начинается ли она с начала строки.

Реже используется родственная им директива `~~`, которая заставляет `FORMAT` вывести знак тильды. Подобно директивам `~%` и `~&`, она может быть параметризована числом, которое задает количество выводимых тильд.

Знаковые и Целочисленные директивы

Вдобавок к директивам общего назначения, ~A и ~S, FORMAT поддерживает несколько директив, которые могут использоваться для получения значений определенных типов особыми способами. Простейшая из них является директива ~C, которая используется для вывода знаков. Ей не требуются префиксные аргументы, но ее работу можно корректировать с помощью модификаторов *двоеточие* и @. Без модификаций ее поведение не отличается от поведения ~A, за исключением того, что она работает только со знаками. Модифицированные версии более полезны. С модификатором *двоеточие*, ~:C выводит заданные по имени *непечатаемые* знаки, такие как пробел, символ табуляции и перевод строки. Это полезно, если вы хотите создать сообщение к пользователю, в котором упоминается некоторый символ. Например, следующий код:

```
(format t "Syntax error. Unexpected character: ~:c" char)
```

может напечатать такое сообщения:

```
Syntax error. Unexpected character: a
```

а еще вот такое:

```
Syntax error. Unexpected character: Space
```

Вместе с модификатором @, ~@C выведет знак в синтаксисе знаков Lisp:

```
CL-USER> (format t "~@c~%" #\a)
#\a
NIL
```

Одновременно с модификаторами *двоеточие* и @, директива ~C может напечатать дополнительную информацию о том, как ввести символ с клавиатуры, если для этого требуется специальная клавиатурная комбинация. Например, на Macintosh, в некоторых приложениях вы можете ввести нулевой символ (код символа 0 в ASCII или в любом надмножестве ASCII, наподобие ISO-8859-1 или Unicode) удерживая клавиши Control и нажав '@'. В OpenMCL, если вы напечатаете нулевой символ с помощью директивы ~:C, то она сообщит вам следующее:

```
(format nil "~:@c" (code-char 0)) ==> "^@ (Control @)"
```

Однако не все версии Lisp реализуют этот аспект директивы ~C. Даже если они это делают, то реализация может и не быть аккуратной - например, если вы запустите OpenMCL в SLIME, комбинация клавиш C-@ перехватывается Emacs, вызывая команду set-mark-command

Другой важной категорией являются директивы формата, предназначенные для вывода чисел. Хотя для этого вы можете использовать директивы ~A и ~S, но если вы хотите получить над печатью чисел больший контроль, вам следует использовать одну из специально предназначенных для чисел директив. Ориентированные на числа директивы могут быть подразделены на две категории: директивы для форматирования целых чисел и директивы для форматирования чисел с плавающей точкой.

Вот пять родственных директив для форматированного вывода целых чисел: ~D, ~X, ~O, ~B, и ~R. Чаще всего применяется директива ~D, которая выводит целые числа по основанию 10.


```
(format nil "~d" 1000000) ==> "1000000"
```

Как я упоминал ранее, с модификатором *двоеточие* она добавляет запятые.

```
(format nil "~:d" 1000000) ==> "1,000,000"
```

А с модификатором *@*, она всегда будет печатать знак.

```
(format nil "~@d" 1000000) ==> "+1000000"
```

И оба модификатора могут быть объединены.

```
(format nil "~:@d" 1000000) ==> "+1,000,000"
```

С помощью первого префиксного параметра можно задать минимальный размер вывода, а с помощью второго - используемый символ заполнения. По умолчанию символ заполнения - это пробел, и заполнение всегда помещается до самого числа.

```
(format nil "~12d" 1000000) ==> " 1000000"  
(format nil "~12,'0d" 1000000) ==> "000001000000"
```

Эти параметры удобны для форматирования объектов наподобие календарных дат в формате с фиксированной длиной.

```
(format nil "~4,'0d-~2,'0d-~2,'0d" 2005 6 10) ==> "2005-06-10"
```

Третий и четвертый параметры используются в связке с модификатором *двоеточие*: третий параметр определяет знак, используемый в качестве разделителя между группами и разрядами, а четвертый параметр определяет число разрядов в группе. Их значения по умолчанию - запятая и число 3 соответственно. Таким образом, вы можете использовать директиву *~:D* без параметров для вывода больших чисел в стандартном для Соединенных Штатов формате, но можете заменить запятую на точку и группировку с 3 на 4 с помощью *~,,'.',4D*.

```
(format nil "~:d" 100000000) ==> "100,000,000"  
(format nil "~,,'.',4:d" 100000000) ==> "1.0000.0000"
```

Заметим, что вы должны использовать запятые для замещения позиций неуказанных параметров длины и заполняющего символа, позволяя им сохранить свои значения по умолчанию.

Директивы *~X*, *~O*, и *~B* работают подобно директиве *~D*, за исключением того, что они выводят числа в шестнадцатеричном, восьмеричном, и двоичном виде.

```
(format nil "~x" 1000000) ==> "f4240"  
(format nil "~o" 1000000) ==> "3641100"  
(format nil "~b" 1000000) ==> "11110100001001000000"
```

Наконец, директива *~R* - универсальная директива для задания *системы счисления*. Ее первый параметр - число между 2 и 36 (включительно), которое обозначает, какое основание системы счисления использовать. Оставшиеся параметры такие же, что и четыре параметра, принимаемые директивами *~D*, *~X*, *~O*, и *~B*, а модификаторы *двоеточие* и *@* меняют ее поведение схожим образом. Кроме того, директива *~R* ведет себя особым образом при

использовании без префиксных параметров, который я буду обсуждать в разделе "Директивы для Английского языка".

Директивы для Чисел с Плавающей Точкой

Вот четыре директивы для форматирования значений с плавающей точкой: ~F, ~E, ~G и ~\$. Первые три из них - это директивы, основанные на дескрипторах редактирования FIXME (edit descriptor?) FORTRAN. Я пропущу большинство деталей этих директив, поскольку они в основном имеют дело с форматированием чисел с плавающей точкой для использования в табличной форме. Тем не менее вы можете использовать директивы ~F, ~E, и ~\$ для вставки значений с плавающей точкой в текст. С другой стороны директива ~G, или *генерал*, FIXME (генерал?) сочетает аспекты директив ~F и ~E единственным осмысленным способом для печати таблиц.

Директива ~F печатает свой аргумент, который должен быть числом, в десятичном формате, по возможности контролируя количество разрядов после десятичной точки. Директиве ~F, тем не менее, разрешается использовать компьютеризированную научную нотацию FIXME (компьютеризированное экспоненциальное представление?), если число достаточно велико либо мало. Директива ~E, с другой стороны, всегда выводит числа в компьютеризированной научной нотации. Обе эти директивы принимают несколько префиксных параметров, но вам нужно беспокоиться только о втором, который управляет количеством разрядов, печатаемых после десятичной точки.

```
(format nil "~f" pi) ==> "3.141592653589793d0"  
(format nil "~,4f" pi) ==> "3.1416"  
(format nil "~e" pi) ==> "3.141592653589793d+0"  
(format nil "~,4e" pi) ==> "3.1416d+0"
```

Директива ~\$ (значок доллара) похожа на ~F, но несколько проще. Как подсказывает ее имя, она предназначена для вывода денежных единиц. Без параметров она эквивалентна ~,2F. Чтобы изменить количество разрядов, печатаемых после десятичной точки, используйте *первый* параметр, в то время как второй параметр регулирует минимальное количество разрядов, печатающихся до десятичной точки.

```
(format nil "~$" pi) ==> "3.14"  
(format nil "~2,$" pi) ==> "0003.14"
```

С модификатором @ все три директивы, ~F, ~E и ~\$ можно заставить всегда печатать знак, плюс или минус.

Директивы для Английского языка

Некоторые из удобнейших директив FORMAT для формирования удобочитаемых сообщений - те, которые выводят английский текст. Эти директивы позволяют вам выводить числа как английский текст, выводить слова в множественном числе в зависимости от значения аргумента формата, и менять регистр FIXME (case conversion? выполнять преобразование между строчными и прописными буквами?) в секциях вывода FORMAT.

Директива ~R, которую я обсуждал в разделе "Знаковые и Целочисленные директивы", при использовании без указания системы счисления, печатает числа как английские слова или римские цифры. При использовании без префиксного параметра и модификаторов, она выводит число словами как количественное числительное.

```
(format nil "~r" 1234) ==> "one thousand two hundred thirty-four"
```

С модификатором *двоеточие* она выводит число как порядковое числительное.

```
(format nil "~:r" 1234) ==> "one thousand two hundred thirty-fourth"
```

И вместе с модификатором @, она выводит число в виде римских цифр; вместе с @ и *двоеточием* она выводит римские цифры, в которых четверки и девятки записаны как IIII и VIIII вместо IV и IX.

```
(format nil "~@r" 1234) ==> "MCCXXXIV"  
(format nil "~:@r" 1234) ==> "MCCXXXIIII"
```

Для чисел, слишком больших, чтобы быть представленными в заданной форме, ~R ведет себя как ~D.

Чтобы помочь вам формировать сообщений со словами в нужном числе, FORMAT предоставляет директиву ~P, которая просто выводит 's' если соответствующий аргумент не 1.

```
(format nil "file~p" 1) ==> "file"  
(format nil "file~p" 10) ==> "files"  
(format nil "file~p" 0) ==> "files"
```

Тем не менее обычно вы будете использовать ~P вместе с модификатором *двоеточие*, который заставляет ее повторно обработать предыдущий аргумент формата.

```
(format nil "~r file~:p" 1) ==> "one file"  
(format nil "~r file~:p" 10) ==> "ten files"  
(format nil "~r file~:p" 0) ==> "zero files"
```

С модификатором @, который может быть объединен с модификатором *двоеточие*, ~P выводит у или *ies*.

```
(format nil "~r famil~:@p" 1) ==> "one family"  
(format nil "~r famil~:@p" 10) ==> "ten families"  
(format nil "~r famil~:@p" 0) ==> "zero families"
```

Очевидно, что ~P не может разрешить все проблемы образования множественного числа и не может помочь при формировании сообщений на других языках (отличных от английского), она удобна в тех ситуациях, для которых предназначена. А директива ~[, о которой я расскажу очень скоро, предоставит вам более гибкий способ параметризации вывода FORMAT.

Последняя директива, посвященная выводу английского текста - это ~ (, которая позволяет вам контролировать регистр выводимого текста. Каждая ~ (составляет пару с ~), и весь вывод, сгенерированный частью управляющей строки между двумя маркерами будет преобразован в нижний регистр.

```
(format nil "~(~a~)" "FOO") ==> "foo"  
(format nil "~(~@r~)" 124) ==> "cxxiv"
```

Вы можете изменить поведение ~ (модификатором @, чтобы заставить ее начать с прописной буквы первое слово на участке текста, с *двоеточием* заставить ее печатать все слова с прописной

буквы, а с обоими модификаторами - преобразовать весь текст в верхний регистр. (Подходящие для этой директивы слова представляют собой буквенно-цифровые знаки, ограниченные небуквенно-цифровыми знаками или концом текста.)

```
(format nil "~(a~)" "tHe Quick BROWN foX") ==> "the quick brown fox"
(format nil "~@(a~)" "tHe Quick BROWN foX") ==> "The quick brown fox"
(format nil "~:(a~)"p "tHe Quick BROWN foX") ==> "The Quick Brown Fox"
(format nil "~:@(a~)" "tHe Quick BROWN foX") ==> "THE QUICK BROWN FOX"
```

Условное Форматирование

Вдобавок к директивам, вставляющим в выводимый текст свои аргументы и видоизменяющими прочий вывод, FORMAT предоставляет несколько директив, которые реализуют простые управляющие структуры внутри управляющих строк. Одна из них, которую вы использовали в главе 9, это *условная* директива ~[. Эта директива замыкается соответствующей директивой ~], а между ними находятся выражения, разделенные ~;. Работа директивы ~[- выбрать одно из выражений, которое затем обрабатывается в FORMAT. Без модификаторов или параметров, выражение выбирается по числовому индексу; директива ~[использует аргумент формата, который должен быть числом, и выбирает *N*-ное (считая от нуля) выражение, где *N* - значение аргумента.

```
(format nil "~[cero~;uno~;dos~]" 0) ==> "cero"
(format nil "~[cero~;uno~;dos~]" 1) ==> "uno"
(format nil "~[cero~;uno~;dos~]" 2) ==> "dos"
```

Если значение аргумента больше, чем число выражений, то ничего не печатается.

```
(format nil "~[cero~;uno~;dos~]" 3) ==> ""
```

Однако если последний разделитель выражений это ~:; вместо ~;, тогда последнее выражение служит выражением по умолчанию.

```
(format nil "~[cero~;uno~;dos~:;mucho~]" 3) ==> "mucho"
(format nil "~[cero~;uno~;dos~:;mucho~]" 100) ==> "mucho"
```

Также возможно определить выбранное выражение, используя префиксный параметр. Хотя было бы глупо использовать константу, записанную прямо в управляющей строке, вспомним, что знак '#', используемый в качестве префиксного параметра означает число оставшихся для обработки аргументов. Поэтому вы можете определить строку формата следующим образом:

```
(defparameter *list-etc*
  "~#[NONE~;~a~;~a and ~a~:~a, ~a~]~#[~; and ~a~:~, ~a, etc~].")
```

и использовать ее так:

```
(format nil *list-etc*) ==> "NONE."
(format nil *list-etc* 'a) ==> "A."
(format nil *list-etc* 'a 'b) ==> "A and B."
(format nil *list-etc* 'a 'b 'c) ==> "A, B and C."
(format nil *list-etc* 'a 'b 'c 'd) ==> "A, B, C, etc."
(format nil *list-etc* 'a 'b 'c 'd 'e) ==> "A, B, C, etc."
```

Заметим, что управляющая строка в действительности содержит две ~[~] директивы - обе из

которых применяют `#` для выбора используемого выражения. Первая директива использует от нуля до двух аргументов, тогда как вторая использует еще один, если он есть. `FORMAT` молча проигнорирует любые аргументы, сверх использованных во время обработки управляющей строки.

С модификатором двоеточие `~[` может содержать только два выражения; директива использует единственный аргумент и обрабатывает первое выражение, если аргумент `NIL`, и второе выражение в противном случае. Вы уже использовали этот вариант `~[` в главе 9 для формирования сообщений типа принять/отклонить `FIXME` (pass/fail message? сработало или не сработало?), таких как это:

```
(format t "~:[FAIL~;pass~]" test-result)
```

Заметим, что оба выражения могут быть пустыми, но директива должна содержать `~;`.

Наконец, с модификатором `@`, директива `~[` может иметь только одно выражение. Директива использует первый аргумент и если он отличен от `NIL`, обрабатывает выражение, при этом список аргументов восстанавливается заново, чтобы первый аргумент был доступен для использования заново.

```
(format nil "~@[x = ~a ~]~@[y = ~a~]" 10 20) ==> "x = 10 y = 20"
(format nil "~@[x = ~a ~]~@[y = ~a~]" 10 nil) ==> "x = 10 "
(format nil "~@[x = ~a ~]~@[y = ~a~]" nil 20) ==> "y = 20"
(format nil "~@[x = ~a ~]~@[y = ~a~]" nil nil) ==> ""
```

Итерация

Другая директива `FORMAT`, мимоходом виденная вами, это директива итерации `~{`. Эта директива сообщает `FORMAT` перебрать элементы списка или неявного списка аргументов формата.

Без модификаторов, `~{` принимает один аргумент формата, который должен являться списком. Подобно директиве `~[`, которой всегда соответствует директива `~]`, директива `~{` всегда имеет соответствующую замыкающую `~}`. Текст между двумя маркерами обрабатывается как управляющая строка, которая выбирает свой аргумент из списка, поглощенного `FIXME` (consumed?) директивой `~{`. `FORMAT` будет циклически обрабатывать эту управляющую строку до тех пор, пока в перебираемом списке не останется элементов. В следующем примере, `~{` принимает один аргумент формата, список `(1 2 3)`, и затем обрабатывает управляющую строку `"~a, "`, повторяя, пока все элементы списка не будут использованы.

```
(format nil "~{~a, ~}" (list 1 2 3)) ==> "1, 2, 3, "
```

При этом раздражает, что при печати за последним элементом списка следует запятая и пробел. Вы можете исправить это директивой `~^`; внутри тела `~{` директива `~^` заставляет итерацию немедленно остановиться, и если в списке не остается больше элементов, прервать обработку оставшейся части управляющей строки. Таким образом, для предотвращения печати запятой и пробела после последнего элемента в списке, вы можете предварить его `~^`.

```
(format nil "~{~a~^, ~}" (list 1 2 3)) ==> "1, 2, 3"
```

После первых двух итераций, при обработке `~^`, в списке остаются необработанные элементы. При этом на третий раз, после того как директива `~a` обработает `3`, `~^` заставит `FORMAT` прервать итерацию без печати запятой и пробела.

С модификатором @, ~{ обработает оставшийся аргумент формата как список.

```
(format nil "~@{~a~^, ~}" 1 2 3) ==> "1, 2, 3"
```

Внутри тела ~{...~} специальный префиксный параметр # ссылается на число необработанных элементов списка, а не на число оставшихся элементов формата. Вы можете использовать его вместе с директивой ~[для печати разделенного запятыми списка с "and" перед последним элементом, вот так:

```
(format nil "~{~a~#[~;;, and ~:;;, ~]~}" (list 1 2 3)) ==> "1, 2, and 3"
```

Тем не менее этот подход совершенно не работает, когда список имеет длину в два элемента, поскольку тогда добавляется лишняя запятая.

```
(format nil "~{~a~#[~;;, and ~:;;, ~]~}" (list 1 2)) ==> "1, and 2"
```

Вы можете поправить это кучей способов. Следующий пользуется эффектом, который дает директива ~@{, когда она заключена внутри другой директивы ~{ или ~@{ - в этом случае она перебирает все элементы, оставшиеся в обрабатываемом внешней директивой ~{ списке. Вы можете объединить ее с директивой ~#[чтобы следующая управляющая строка форматировала список в соответствии с английской грамматикой:

```
(defparameter *english-list*  
  "~{~#[~;;~a~;~a and ~a~:;;~@{~a~#[~;;, and ~:;;, ~]~}~}~}")  
(format nil *english-list* '()) ==> ""  
(format nil *english-list* '(1)) ==> "1"  
(format nil *english-list* '(1 2)) ==> "1 and 2"  
(format nil *english-list* '(1 2 3)) ==> "1, 2, and 3"  
(format nil *english-list* '(1 2 3 4)) ==> "1, 2, 3, and 4"
```

В то время, как эта управляющая строка приближается к такому классу кода, который трудно понять, после того как он написан FIXME (write-only code?), все-таки это возможно, когда у вас есть немного времени. Внешние ~{...~} принимают список и затем перебирают его. Затем все тело цикла состоит из ~#[...~]; печать производится на каждом шаге цикла, и таким образом зависит от количества обрабатываемых элементов, оставшихся в списке. Разделяя директиву ~#[...~] разделителями выражений ~;; вы можете увидеть, что она состоит из четырех выражений, последнее из которых является выражением по умолчанию, поскольку его предваряет ~:;; в отличие от простой ~;. Первое выражение, выполняющееся при нулевом числе элементов, пусто, оно нужно только в том случае, если обрабатываемых элементов больше не осталось, тогда мы должны остановиться. Второе выражение с помощью простой директивы ~a обрабатывает случай, когда элемент единственный. С двумя элементами справляется "~a and ~a". И выражение по умолчанию, которое имеет дело с тремя и более элементами, состоит из другой директивы итерации, в этот раз используется ~@{ для перебора оставшихся элементов списка, обрабатываемых внешней ~{. В итоге тело цикла представляет собой управляющую строку, которая может корректно обработать список трех или более элементов, что в данной ситуации более чем достаточно. Поскольку цикл ~@{ использует все оставшиеся элементы списка, внешний цикл выполняется только один раз.

Если вы хотите напечатать что-нибудь особенное, например "<empty>", когда список пуст, то у вас есть пара способов это сделать. Возможно проще всего будет вставить нужный вам текст в первое (нулевое) выражение внешней ~#[и затем добавить модификатор *двоеточие* к замыкающей ~} внешнего цикла - двоеточие заставит цикл выполняться по меньшей мере один

раз, даже если список пуст, и в этот момент FORMAT обработает нулевое выражение условной директивы.

```
(defparameter *english-list*
  "~{~#[<empty>~;~a~;~a and ~a~;~@{~a~#[~; and ~;~, ~]~}~::~}")
(format nil *english-list* '()) ==> "<empty>"
```

Удивительно, что директива ~{ предоставляет даже больше вариантов с различными комбинациями префиксных параметров и модификаторов. Я не буду обсуждать их, только скажу, что вы можете использовать целочисленный префиксный параметр для ограничения максимального числа выполнений итерации, и что с модификатором *двоеточие* каждый элемент списка (как настоящего, так и созданного директивой ~@{) сам должен быть списком, чьи элементы затем будут использованы как аргументы управляющей строки в директиве ~: {...~}.

Тройной прыжок

Более простой директивой является ~*, которая позволяет вам перемещаться по списку аргументов формата. В своей основной форме, без модификаторов, она просто пропускает следующий аргумент, при этом ничего не печатается. Однако чаще она используется с модификатором *двоеточие*, который заставляет ее отступить назад, позволяя одному и тому же аргументу быть обработанным дважды. Например, вы можете использовать ~:* для печати числового аргумента один раз словом, а другой раз цифрами, вот так:

```
(format nil "~r ~:* (~d)" 1) ==> "one (1)"
```

Еще вы можете реализовать директиву, похожую на ~:Р для неправильной формы множественного числа (в английском языке) объединяя ~:* с ~[.

```
(format nil "I saw ~r el~:*~[ves~;f~::ves~]." 0) ==> "I saw zero elves."
(format nil "I saw ~r el~:*~[ves~;f~::ves~]." 1) ==> "I saw one elf."
(format nil "I saw ~r el~:*~[ves~;f~::ves~]." 2) ==> "I saw two elves."
```

В этой управляющей строке ~R печатает аргумент формата в виде количественного числительного. Затем директива ~:* возвращается назад, так что число также используется как аргумент директивы ~[, выбирая между выражениями, когда число равно нулю, единице, или чему-нибудь еще

Внутри директивы ~{, ~* пропускает или перескакивает назад через элементы списка. Например, вы можете напечатать только ключи в списке свойств FIXME (использовать термин "список свойств" или plist?), таким образом:

```
(format nil "~{~s~*~^ ~}" '(:a 10 :b 20)) ==> ":A :B"
```

Директиве ~* также может быть задан префиксный параметр. Без модификаторов или с модификатором *двоеточие*, этот параметр определяет число аргументов, на которое нужно передвинуться вперед или назад, и по умолчанию равняется единице. С модификатором @, префиксный параметр определяет абсолютный, отсчитываемый от нуля индекс аргумента, на который нужно переместиться, по умолчанию это нуль. Вариант ~* с @ может быть полезен, если вы хотите использовать различные управляющие строки для формирования различных сообщений для одних и тех же аргументов, и если этим сообщениям нужно использовать свои аргументы в различном порядке

И многое другое...

Осталось еще многое - я не упоминал о директиве `~?`, которая может брать фрагменты управляющих строк из аргументов формата, или директиву `~/`, которая позволяет вам вызвать произвольную функцию для обработки следующего аргумента формата. И еще остались все директивы для формирования табличной и удобочитаемой печати. Но уже затронутых в этой главе директив пока будет вполне достаточно.

В следующей главе вы перейдете к системе условий `FIXME Common Lisp`, аналогичной системам исключений и обработки ошибок, присутствующих в других языках.

19. Обработка исключений изнутри: Условия и Перезапуск

Одной из замечательных особенностей Лиспа является его система *условий*. Она служит тем же целям, что и системы обработки исключений в Java, Python и C++, но является более гибкой. На самом деле её способности выходят за пределы обработки ошибок – условия являются более всеохватывающими, чем исключения, в том смысле, что условия могут представить любое событие во время выполнения программы, которое может представлять интерес в программировании различных уровней стека вызовов. Например в секции "Другие применения условий" вы увидите, что условия могут быть использованы для выдачи предупреждения без нарушения выполнения кода, который выдаёт предупреждение, в то же время позволяя коду выше на стеке вызовов контролировать, напечатано ли предупреждающее сообщение. Пока, однако, я сосредоточусь на обработке ошибок.

Система условий более гибка, чем система исключений, потому что вместо разделения на две части между кодом, который сигнализирует об ошибке и кодом, который обрабатывает её, система условий разделяет ответственность на три части – *сигнализация условия*, его *обработка* и *перезапуск*. В этой главе я опишу, как можно использовать условия в гипотетическом приложении для анализа журнальных файлов. Вы увидите, как можно использовать систему условий, чтобы дать возможность низкоуровневой функции обнаружить проблему в процессе разбора журнального файла и сигнализировать об ошибке, коду на промежуточном уровне предоставить несколько возможных путей исправления такой ошибки и коду на высшем уровне приложения определить политику для выбора стратегии исправления.

Для начала я введу некоторую терминологию: *ошибки*, как я буду использовать этот термин, есть последствие закона Мёрфи. Если может случиться что-то плохое, оно случится: файл, который ваша программа должна прочесть, будет потерян, диск, на который вам надо записать, будет полон, сервер, с которым вы общаетесь, упадёт или исчезнет сеть. Если что-то из этих вещей случилось, это может помешать участку кода выполнить то, что вы хотите. Но это не программная ошибка: нет места в коде, которое вы можете исправить, чтобы сделать несуществующий файл существующим или освободить место на диске. Однако если оставшаяся часть программы зависит от действий, которые должны были быть сделаны, то вам лучше как-то разобраться с происшествием, иначе вы ошибку *закладываете*. Итак, ошибки не всегда бывают программными, но их игнорирование – это всегда ошибка в программе.

Что же означает обработать ошибку? В хорошо написанной программе каждая функция – это чёрный ящик, скрывающий внутреннее содержание своей работы. Программы таким образом строятся из функций разных уровней: функции на высшем уровне построены на функциях более низкого уровня и так далее. Эта иерархия функционирования выражается во время выполнения в форме стека вызовов: когда высший вызывает среднее, а он в свою очередь вызывает нижнее, то поток выполнения находится внизу, но он также проходит через средний и высший уровни, поэтому они все находятся в одном стеке вызовов.

Так как каждая функция является чёрным ящиком, связи между функциями – прекрасное место для работы с ошибками. Каждая функция – нижняя для примера – должна делать какую-то работу. Та, которая её непосредственно вызывает – средняя в нашем случае – рассчитывает на неё в своей работе. Однако ошибка, которая мешает ей сделать свою работу, подвергает всех вызывающих риску: средняя вызывала нижнюю, потому что ей нужна работа, которую нижняя делает; если эта работа не сделана, у средней проблемы. Но это означает, что у вызвавшей среднюю, высшей, тоже проблемы – и так далее вверх по стеку вызовов вплоть до верха программы. С другой стороны, так как каждая функция чёрный ящик, если какая-то функция в стеке вызовов может как-то сделать свою работу не смотря на проблемы у тех, кто пониже, то

никакой функции повыше не надо знать, что проблемы вообще были – все эти функции заботятся только чтобы функции, которые они вызывают, как-то сделали работу, которую от них ждут.

В большинстве языков ошибки обрабатываются путём возврата из упавшей функции и передачи шанса вызывающей функции или всё исправить или упасть самой. Некоторые языки используют обычный механизм возврата из функций, в то время как языки с исключениями возвращают контроль через *выброс* или *возбуждение* исключения. Исключения – это огромное усовершенствование над использованием обычных возвратов функций, но обе схемы страдают от общего порока: пока ищется функция, которая может восстановиться, стек пустеет. Это приводит к тому, что код, который может всё восстановить, должен сделать это без контекста того, что код нижнего уровня пытался сделать, когда случилась ошибка.

Рассмотрим гипотетическую цепочку вызовов из высшей, средней, нижней функций. Если нижняя падает, и средняя не может всё поправить, дело передаётся в суд высшей инстанции. Для высшей, чтобы обработать ошибку, надо или сделать свою работу без помощи средней, или как-то всё изменить, чтобы вызов средней работал, и вызвать её снова. Первый вариант теоретически прост, но предполагает кучу лишнего кода – полное воплощение того, что, предполагалось, сделает средняя. И чем дальше стек освобождается, тем больше работы, которую следует переделать. Второй вариант – исправление ситуации, и ещё одна попытка – запутанный: для высшей, чтобы иметь возможность изменить ситуацию в мире так, чтобы второй вызов средней не закончился ошибкой в нижней, необходимо неподобающее ей знание внутреннего устройства обеих функций: средней и нижней, что противоречит идее, что каждая функция является чёрным ящиком.

Путь языка Лисп

Система обработки ошибок в языке Common Lisp даёт вам выход из этого тупика, позволяя вам отделить код, который исправляет ошибки, от кода, который решает, как их исправлять. Таким образом, вы можете поместить код восстановления в функции низкого уровня, на самом деле не обязывая использовать какую-то конкретную стратегию восстановления, предоставляя решать это коду функций высшего уровня.

Чтобы почувствовать, как это работает, предположим, что вы пишете приложение, которое читает какой-то текстовый файл наподобие журнала Web-сервера. Где-то в вашем приложении будет ваша функция разбора отдельных журнальных записей. Давайте предположим, что вы напишете функцию `parse-log-entry`, которой будет передаваться строка, содержащая одну журнальную запись, и которая предположительно вернёт объект `log-entry`, представляющий эту запись. Эта функция будет вызываться из функции `parse-log-file`, которая читает журнальный файл полностью и возвращает список объектов, представляющий все записи в журнале.

Чтобы всё упростить, от функции `parse-log-entry` не будет требоваться разбирать неправильно сформированные записи. Однако она будет способна распознать, когда её ввод неправилен. Но что она должна делать, когда она определяет неправильный ввод? В языке C вы бы вернули специальное значение, чтобы обозначить, что была проблема. В языках Java или Python вы бы выбросили или возбудили исключение. В Common Lisp вы сигнализируете условие.

Условия

Условие – это объект, чей класс обозначает общую природу условия, а данные экземпляров класса несут информацию о деталях конкретных обстоятельств, которые приводят к сигнализации об условии. В нашей гипотетической программе анализа журнальных файлов вы

можете определить класс условия `malformed-log-entry-error`, с помощью которого `parse-log-entry` будет сигнализировать, если предоставленные ему данные он не сможет разобрать.

Классы условий определяются макросом `DEFINE-CONDITION`, который работает точно так же, как `DEFCLASS`, исключая то, что суперклассом по умолчанию для классов, определённых с `DEFINE-CONDITION`, является `CONDITION`, а не `STANDARD-OBJECT`. Слоты определяются так же, и классы условий могут иметь единственное или множественное наследование от других классов, которые происходят от `CONDITION`. Однако, по историческим причинам от классов условий не требуется быть экземплярами `STANDARD-OBJECT`, так что некоторые из функций, которые вы используете, и классы которых созданы через `DEFCLASS`, не обязательно должны работать с условиями. В частности, слоты условий недоступны для `SLOT-VALUE`: вы должны задать или опцию `:reader`, или опцию `:accessor` для любого слота, чьё значение вы собираетесь использовать. Точно так же новые объекты для условий создаются путём вызова `MAKE-CONDITION`, а не `MAKE-INSTANCE`. `MAKE-CONDITION` инициализирует слоты нового условия, основываясь на полученных `:initarg`, но способа последующей настройки инициализации условия, аналогичного `INITIALIZE-INSTANCE`, не существует.

При использовании системы условий для обработки ошибок вы должны определять ваши условия как подклассы `ERROR`, который является в свою очередь подклассом `CONDITION`. Таким образом, вы можете определить условие `malformed-log-entry-error` со слотом для хранения аргумента, который был передан в `parse-log-entry`, вот так:

```
(define-condition malformed-log-entry-error (error)
  ((text :initarg :text :reader text)))
```

Обработчики Условий

В `parse-log-entry` вы будете сигнализировать `malformed-log-entry-error`, если вы не сможете разобрать журнальную запись. Вы сигнализируете об ошибках функцией `ERROR`, которая вызывает низкоуровневую функцию `SIGNAL` и попадает в отладчик, если условие не обрабатывается. Можно вызвать `ERROR` двумя путями: послать ей уже сформированный объект условия или послать ей имя класса условия и любые аргументы инициализации, необходимые для построения нового условия, и она создаст его для вас. Первое иногда полезно для повторной сигнализации уже существующего объекта условия, а второе более лаконично. Итак, вы можете записать `parse-log-entry` в таком виде, опуская детали собственно разбора журнальной записи:

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
      (make-instance 'log-entry ...)
      (error 'malformed-log-entry-error :text text)))
```

То, что происходит, когда сигнализируется ошибка, зависит от кода в стеке вызовов, стоящего выше `parse-log-entry`. Чтобы избежать попадания в отладчик, вы должны установить *обработчик условия* в одну из функций, предшествующих вызову `parse-log-entry`. Когда условие сигнализируется, механизм сигнализирования просматривает список активных обработчиков условий, ища по классу условия обработчик, способный обработать сигнализируемое условие. Каждый обработчик условия состоит из спецификатора типа, обозначающего, какие типы условий он может обработать, и функции, которая получает единственный аргумент – условие. В каждый момент времени может быть несколько активных обработчиков этого условия на различных уровнях стека вызовов. Когда условие сигнализируется, механизм сигнализирования находит последний установленный обработчик, чей спецификатор типа совместим с сигнализируемым условием и вызывает его функцию, передавая

ей объект условия.

Функция обработки может выбрать, обрабатывать ли ей условие. Функция может отказаться обрабатывать условие, просто нормально завершившись, в этом случае управление возвращается функции `SIGNAL`, которая будет искать следующий установленный обработчик с подходящим спецификатором типа. Для обработки условия функция обязана передать управление минуя функцию `SIGNAL` посредством *нелокального выхода*. В следующей секции вы увидите, как обработчик может выбрать, куда передать контроль. Однако многие обработчики условий просто хотят опустошить стек до того места, где они были установлены, и затем запустить некоторый код. Макрос `HANDLER-CASE` как раз устанавливает такой тип обработчика. Базовая форма `HANDLER-CASE` такова:

```
(handler-case expression
  error-clause*)
```

где каждая конструкция *error-clause* представляет собой следующую форму:

```
(condition-type ([var]) code)
```

Если выражение *expression* завершается нормально, тогда его значение возвращается из конструкции `HANDLER-CASE`. Тело `HANDLER-CASE` должно быть одним выражением, но вы можете использовать `PROGN` для объединения нескольких выражений в одну форму. Если всё же выражение сигнализирует условие, которое является одним из типов *condition-type*, указанных в одной из секций *error-clause*, то код из соответствующей секции будет выполнен, и его значение будет возвращено из `HANDLER-CASE`. Если указана переменная *var*, она будет именем переменной, содержащей объект условия при выполнении кода обработчика. Если код не нуждается в доступе к объекту условия, вы можете опустить имя переменной.

Например, одним из способов обработки условия `malformed-log-entry-error`, сигнализируемого из функции `parse-log-entry` в вызывающую её функцию `parse-log-file`, был бы пропуск неправильной записи. В следующей функции выражение `HANDLER-CASE` вернёт либо имя, возвращаемое `parse-log-entry`, либо `NIL`, если `malformed-log-entry-error` сигнализируется. Слово `it` во фразе `collect it` цикла `LOOP` является ещё одним ключевым словом `LOOP`, которое ссылается на значение последнего выполненного условия, в данном случае – на переменную `entry`).

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
      for entry = (handler-case (parse-log-entry text)
                              (malformed-log-entry-error () nil))
      when entry collect it)))
```

Когда функция `parse-log-entry` завершается нормально, её значение присваивается переменной `entry` и затем накапливается циклом `LOOP`. Но если функция `parse-log-entry` сигнализирует ошибку `malformed-log-entry-error`, то обработчик ошибки вернёт `NIL`, который не будет сохранён.

Обработка исключений в стиле языка Ява

`HANDLER-CASE` в языке Common Lisp является ближайшим аналогом стиля обработки исключений в языках Ява и Питон. То, что вы можете записать на языке Ява как

```
try {
    doStuff();
    doMoreStuff();
} catch (SomeException se) {
    recover(se);
}
```

или на языке Питон как

```
try:
    doStuff()
    doMoreStuff()
except SomeException, se:
    recover(se)
```

в языке Common Lisp вы пишете так:

```
(handler-case
  (progn
    (do-stuff)
    (do-more-stuff))
  (some-exception (se) (recover se)))
```

Эта версия `parse-log-file` имеет один серьёзный недостаток: она делает слишком много всего. Как предполагает её имя, работа `parse-log-file` заключается в разборе файла и выдаче списка объектов `log-entry`; если это невозможно, не её дело решать что сделать взамен. Что если вы захотите использовать `parse-log-file` в приложении, которое захочет сказать пользователю, что журнальный файл повреждён или в таком, которое захочет восстановить неправильную запись путём исправления её и разбора снова? Или приложению достаточно пропускать их до тех пор, пока не накопится определённое количество повреждённых записей.

Вы можете попытаться решить проблему, переместив `HANDLER-CASE` в функцию высшего уровня. Однако тогда у вас не будет способа воплотить текущую политику пропуска отдельных записей – когда ошибка будет просигнализована, стек будет опустошён вплоть до функции высшего уровня, разбор журнального файла будет вообще покинут. Что вам надо, так это способ предоставить текущую стратегию восстановления без требования всегда её использовать.

Перезапуск

Система условий позволяет вам это делать через разделение кода обработки ошибок на две части. Вы помещаете код, который собственно исправляет ошибки, в перезапуски (`restarts`), и обработчик условия может затем обработать условие, запустив подходящий вариант. Код перезапуска можно положить в средне- или низко-уровневую функцию, такую `parse-log-file` или `parse-log-entry`, переместив обработчик условия на высший уровень в приложении.

Для изменения `parse-log-file`, чтобы она устанавливала перезапуск вместо обработчика условия, вы можете сменить `HANDLER-CASE` на `RESTART-CASE`. Форма `RESTART-CASE` довольно похожа на `HANDLER-CASE` за исключением того, что имена перезапусков, это просто имена, не обязательно имена типов условия. В общем, имя перезапуска должно описывать действие, которое перезапуск совершает. В `parse-log-file` вы можете вызвать перезапуск `skip-log-entry` так как он делает именно "пропустить-журнальную-запись". Новая версия

будет выглядеть так:

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
      for entry = (restart-case (parse-log-entry text)
                              (skip-log-entry () nil))
      when entry collect it)))
```

Если вы вызовете эту версию `parse-log-file` на журнальном файле, содержащем повреждённую запись, она не обработает ошибку напрямую; вы попадёте в отладчик. Однако там, среди различных представленных перезапусков от отладчика будет один, называемый `skip-log-entry`, который, если вызовете его, продолжит выполнение `parse-log-file` дальше в прежнем режиме. Для избежания попадания в отладчик, вы можете установить обработчик условия, который вызовет перезапуск `skip-log-entry` автоматически.

Преимущество установки перезапуска вместо обработки ошибки напрямую в `parse-log-file` в том, что это даёт возможность использовать `parse-log-file` в большем количестве ситуаций. Код высшего уровня, который вызывает `parse-log-file`, не обязан вызывать перезапуск `skip-log-entry`. Он может выбрать обработку ошибки на высшем уровне. Или, как я покажу в следующей секции, вы можете добавить перезапуск в `parse-log-entry`, чтобы предоставить другую стратегию исправления и затем обработчик условия может выбрать какую стратегию он хочет использовать.

Но перед этим вам надо увидеть как установить обработчик условия, который будет вызывать `skip-log-entry` перезапуск. Вы можете установить обработчик где угодно в цепочке вызовов ведущей к `parse-log-file`. Это может быть довольно высокий уровень в вашем приложении, не обязательно в функции, непосредственно вызывающей `parse-log-file`. Например, предположим, что главная точка входа в ваше приложение, это функция `log-analyzer`, которая ищет стопку журналов и анализирует их функцией `analyze-log`, которая в итоге приводит к вызову `parse-log-file`. Без какой-либо обработки ошибок, это может выглядеть так:

```
(defun log-analyzer ()
  (dolist (log (find-all-logs))
    (analyze-log log)))
```

Работа `analyze-log` – это вызвать прямо или непрямо `parse-log-file` и затем сделать что-то с возвращённым списком журнальных записей. Сверхпростая версия может выглядеть так:

```
(defun analyze-log (log)
  (dolist (entry (parse-log-file log))
    (analyze-entry entry)))
```

где функция `analyze-entry` предположительно ответственна за извлечение заботящей вас информации про каждую журнальную запись и припрятывание её куда-нибудь.

Таким образом, путь от функции высшего уровня `log-analyzer` к `parse-log-entry`, которая собственно сигнализирует про ошибку, следующий:

{{ pcl:restart-call-stack.png }}

Предполагая, что вы всегда хотите пропускать неправильно сформированные записи, вы можете изменить эту функцию для установки обработчика условия, который вызывает перезапуск `skip-`

log-entry для вас. Однако вы не можете использовать HANDLER-CASE для установки обработчика условия, потому что тогда стек будет опустошён до функции, где HANDLER-CASE появляется. Вместо этого, вам надо использовать макрос нижнего уровня HANDLER-BIND. Основная форма HANDLER-BIND следующая:

```
(handler-bind (binding*) form*)
```

где каждая привязка (binding) представляет собой список из типа условия и обрабатывающей функции с одним аргументом. После привязок с обработчиками, тело HANDLER-BIND может содержать произвольное число форм. В отличие от кода обработчика в HANDLER-CASE, код обработчика должен быть FIXME функцией-объектом и должен принимать единственный аргумент. Более важным различием между HANDLER-BIND и HANDLER-CASE является то, что функция обработки привязанная через HANDLER-BIND будет запущена без опустошения стека – поток контроля будет всё ещё в вызове parse-log-entry если эта функция вызвана. Вызов INVOKE-RESTART найдёт и вызовет последний связанный перезапуск с данным именем. Таким образом вы можете добавить обработчик к log-analyzer который будет вызывать skip-log-entry перезапуск, установленный в parse-log-file таким образом:

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (invoke-restart 'skip-log-entry))))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

В этом HANDLER-BIND обработчик является безымянной функцией, которая вызывает перезапуск skip-log-entry. Вы также могли бы определить функцию с именем, которая делала бы тоже самое и связать всё с ней. Фактически это обычная практика, когда определение перезапуска является определением функции с тем же именем и получающую единственный аргумент, условие, которая вызывает одноимённый перезапуск. Такие функции называются функциями перезапуска. Можно определить функцию перезапуска skip-log-entry так:

```
(defun skip-log-entry (c)
  (invoke-restart 'skip-log-entry))
```

Затем вы могли бы изменить определение log-analyzer на такое:

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error #'skip-log-entry))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

Как уже было сказано, функция перезапуска skip-log-entry полагает, что перезапуск skip-log-entry уже был установлен. Если malformed-log-entry-error просигнализирован кодом, вызванным из log-analyzer без уже установленного skip-log-entry, вызов INVOKE-RESTART будет сигнализировать CONTROL-ERROR, когда не сможет обнаружить перезапуск skip-log-entry. Если вы хотите допустить возможность того, чтобы malformed-log-entry-error могло быть просигнализировано из кода в котором перезапуск skip-log-entry не установлен, то вы можете изменить функцию skip-log-entry таким образом:

```
(defun skip-log-entry (c)
  (let ((restart (find-restart 'skip-log-entry)))
    (when restart (invoke-restart restart))))
```

FIND-RESTART ищет перезапуск с данным именем и возвращает объект, представляющий перезапуск, если перезапуск найден или NIL если нет. Вы можете вызвать перезапуск путём посылки перезапуск-объекта к INVOKE-RESTART. Таким образом, когда skip-log-entry привязывается внутри HANDLER-BIND, она будет обрабатывать условие путём вызова перезапуска skip-log-entry, если тот доступен или, в противном случае, нормально завершится, давая другим обработчикам условия, привязанным выше по стеку, шанс так обработать условие.

Предоставление множественных перезапусков

Так как перезапуски должны быть прямо вызваны, чтобы был какой-то эффект, вы можете определить несколько перезапусков, предоставляющих различную стратегию исправления. Как я упоминал ранее, не все приложения для разбора журналов будут с необходимостью хотеть пропускать неправильные записи. Некоторые приложения могут захотеть, чтобы parse-log-file включала специальный тип объекта для представления неправильной записи в списке log-entry объектов; другие приложения могут иметь несколько путей для восстановления неправильной записи и могут хотеть иметь способ отправки исправленной записи назад в parse-log-entry.

Чтобы позволить существовать более сложным протоколам исправления, перезапуски могут получать произвольные аргументы, которые передаются в вызове INVOKE-RESTART. Вы можете предоставить поддержку для обеих стратегий исправления, которые я упомянул, путём добавления двух перезапусков к parse-log-entry, каждый из которых получает единственный аргумент. Первый просто возвращает значение, которое получил, как значение всего parse-log-entry, в то время как другой пытается разобрать свой аргумент на месте оригинальной журнальной записи.

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
      (make-instance 'log-entry ...)
      (restart-case (error 'malformed-log-entry-error :text text)
        (use-value (value) value)
        (reparse-entry (fixed-text) (parse-log-entry fixed-text))))))
```

Имя USE-VALUE является стандартным именем для такого типа перезапуска. Common Lisp определяет функцию для USE-VALUE аналогично skip-log-entry функции, только что вами определённой. Таким образом, если вы хотели изменить политику по отношению к неправильным записям на ту, которая создана экземпляром malformed-log-entry, вы могли бы сменить log-analyzer на такую функцию (предполагая существование malformed-log-entry класса с иницилирующим аргументом :text):

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (use-value
                        (make-instance 'malformed-log-entry :text (text c))))))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

Вы могли бы точно также поместить эти новые перезапуски в parse-log-file вместо parse-log-entry. Хотя вы, вообще-то, хотите поместить перезапуски в код по-возможности самого нижнего уровня. Не было бы, однако, правильным перемещать перезапуск skip-log-entry в parse-log-entry, так как это может привести к возвращению иногда 'NIL от parse-log-

`entry` в качестве результата нормального завершения, а вы начинали всё с тем, чтобы таких вещей избежать. И это была бы одинаково плохая идея убрать перезапуск `skip-log-entry` исходя из теории, что обработчик условия смог бы достичь того же эффекта через вызов перезапуска `use-value` с `NIL` в качестве аргумента; это потребовало бы от обработчика условия знания внутренней работы `parse-log-file`. Таким образом `skip-log-entry` – правильно абстрагированная часть API журнального разбора.

Другие применения условий

В то время, как условия в основном используются для обработки ошибок, они также могут быть использованы для других целей – вы можете применить условия, обработчики условий и перезапуски для конструирования различных протоколов между низко и высокоуровневым кодом. Ключом для понимания возможностей условий является понимание того, что просто сигнализация условия не влияет на поток контроля.

Примитивная сигнальная функция `SIGNAL` воплощает механизм поиска применимого обработчика условия и вызывает его обрабатывающую функцию. Причина, почему обработчик может отказаться обрабатывать условие и нормально завершиться, это потому что вызов функции обработчика, это обычный вызов функции – когда обработчик завершается, контроль передаётся обратно к `SIGNAL`, которая затем ищет другой, ранее привязанный обработчик, который может обработать условие. Если `SIGNAL` исчерпает обработчики условия до того, как условие будет обработано, она также завершается нормально.

Ранее использованная вами функция `ERROR` вызывает `SIGNAL`. Если ошибка обрабатывается обработчиком условия, который передаёт управление через `HANDLER-CASE` или через вызов перезапуска, тогда вызов `SIGNAL` никогда не завершится. Но если `SIGNAL` завершается, `ERROR` вызывает отладчик путём вызова функции, сохранённой в `*DEBUGGER-HOOK*`. Таким образом вызов `ERROR` никогда не сможет завершиться нормально; условие должно быть обработано или обработчиком условия или в отладчике.

Другая сигнализирующая условие функция `WARN`, представляет пример протокола ещё одного типа, построенный на системе условий. Подобно `ERROR`, `WARN` вызывает `SIGNAL` для сигнализации условия. Но, если `SIGNAL` завершается, `WARN` не вызывает отладчик – она печатает условие в `*ERROR-OUTPUT*` и возвращает `NIL`, позволяя своему вызывающему продолжать работу. `WARN` также устанавливает перезапуск `MUFFLE-WARNING`, `FIXME` обёртку вызова `SIGNAL`, которая может быть использована обработчиком условия, чтобы сделать возврат из `WARN` без печатания чего либо. Функция `MUFFLE-WARNING` перезапуска, ищет и вызывает одноимённый перезапуск, сигнализируя `CONTROL-ERROR`, если такого перезапуска нет. Конечно условие, сигнализируемое `WARN`, также может быть обработано другим путём – обработчик условия может

"поспособствовать"

, чтобы предупреждение об ошибке обрабатывалось, как если бы ошибка действительно была.

Например в приложении для разбора журналов, если журнальная запись немного неправильна, но всё же поддаётся разбору, вы могли бы указать `parse-log-entry` разобрать немного повреждённую запись, но сигнализировать `WARN`, когда она будет делать это. Затем большее приложение может выбрать либо позволить предупреждению быть напечатанным, либо скрыть предупреждение, либо рассматривать предупреждение как ошибку, исправляя ситуацию как это делалось при `malformed-log-entry-error`.

Третья сигнализирующая ошибки функция, `CERROR`, представляет ещё один протокол. Подобно `ERROR`, `CERROR` сбросит вас в отладчик, если условие, которая она сигнализирует, не будет обработано. Однако как и `WARN`, она устанавливает перезапуск перед сигнализацией условия.

Перезапуск, `CONTINUE`, приведёт к нормальному завершению `CERROR` – если перезапуск был вызван обработчиком условия, он вообще сохранит вас от попадания в отладчик. `FIXME` В противном случае, вы можете использовать перезапуск, как только окажетесь в отладчике для продолжения вычислений сразу после вызова `CERROR`. Функция `CONTINUE` находит и вызывает `CONTINUE` перезапуск, если он доступен, и возвращает `NIL` в противном случае.

Вы также можете конструировать ваши собственные протоколы поверх `SIGNAL` – везде, где низкоуровневый код нуждается в передаче информации назад по стеку вызовов в высокоуровневый код, механизм условий является подходящим механизмом для использования. Но для большинства целей один из стандартных протоколов ошибок или предупреждений должен подойти.

Вы будете использовать систему условий в будущих практических главах, как в виде обычной обработки ошибок, так и, в 25-й главе, для помощи в обработке мудрёного особого случая при разборе ID3 файлов. К сожалению, это судьба обработки ошибок, всегда идти мелким шрифтом в программных текстах – надлежащая обработка ошибок или отсутствие таковой, является часто наибольшим отличием иллюстративного кода и утяжелённого кода промышленного качества. Хитрость написания последнего относится больше к принятию особого строгого образа мышления о программном обеспечении, чем к деталям конструкции языка. Так что если вашей целью является написание такого типа программ, то вы найдёте систему условий в Common Lisp отличным инструментом для написания надёжного кода и прекрасно подходящей для стиля последовательных улучшений (*incremental development style*).

Написание надёжных программ

Для информации по написанию надёжных программ нет ничего лучше, чем начать с поиска книги Гленфорда Меерса "Надёжность программного обеспечения" *Software Reliability* Glenford J. Meyers (John Wiley & Sons, 1976). Написанное Bertrand Meyer про Design By Contract также показывает полезный путь в размышлениях про правильность программ. Например, смотрите главы 11 и 12 его *Object-Oriented Software Construction* (Prentice Hall, 1997). Запомните, однако, что Bertrand Meyer является изобретателем Eiffel, статически типизированного крепостнического и дисциплинарного языка из школы Algol/Ada. Хотя у него есть много умных мыслей про объектную ориентированность и программную надёжность, всё же существует довольно большой разрыв между его видением программирования и путём Лиспа. Наконец, для прекрасного обзора большинства проблем, окружающих построение отказоустойчивых систем, смотрите главу 3 из классической *Transaction Processing: Concepts and Techniques* (Morgan Kaufmann, 1993) от Jim Gray и Andreas Reuter.

В следующей главе я дам быстрый обзор некоторых из 25 специальных операторов, которых у вас пока не было шанса использовать, по крайней мере прямо.

20. Специальные операторы

Между прочим, наиболее впечатляющим аспектом условной системы, описанной в предыдущей главе, является то, что если бы она не была частью языка, то она могла бы быть полностью написана в виде отдельной библиотеки. Это возможно, поскольку специальные операторы Common Lisp (когда никто не осуществляет прямой доступ к обработке или выдаче условий) обеспечивают достаточный уровень доступа к низкоуровневым частям языка, что делает возможным контроль раскрутки стека (unwinding of the stack).

В предыдущих главах я обсуждал наиболее часто используемые специальные операторы, но было бы неправильным ознакомиться только с их частью. Для этого есть две причины - во-первых, некоторые из редко используемых специальных операторов используются не часто просто потому, что то, что они обрабатывают, обычно не используется в большинстве программ. Ознакомиться с ними будет полезно хотя бы для того, чтобы вы знали, что они существуют. А во-вторых, поскольку 25 специальных операторов (вместе с правилами вычисления вызовов функций и основными типами данных) составляют основу для остальной части языка, так что знакомство с ними поможет вам понять как язык устроен.

В этой главе я буду обсуждать специальные операторы (некоторые вкратце, а некоторые – более подробно), так что вы сможете увидеть как они связаны между собой. Я буду указывать на те, которые вы сможете напрямую использовать в вашем коде, на те, которые могут служить основой для конструкций, которые вы будете использовать все время, а также на те, которые вы будете редко использовать напрямую, но которые вы сможете использовать в коде, генерируемом макросами.

Контроль вычисления

К первой категории специальных операторов относятся три оператора, которые обеспечивают базовый контроль вычисления выражений. Это QUOTE, IF и PROGN, про которые я уже рассказывал. Однако было бы неправильным не отметить то, как каждый из этих специальных операторов предоставляет контроль за вычислением одной или нескольких форм. QUOTE предотвращает вычисление выражения и позволяет вам получить s-выражение в виде данных. IF реализует базовый оператор логического выбора, на основе которого могут быть построены все остальные условные выражения. А PROGN обеспечивает возможность вычисления последовательности выражений.

Манипуляции с лексическим окружением

Наибольший класс специальных операторов содержит операторы, которые манипулируют и производят доступ к лексическому окружению. LET и LET*, которые мы уже обсуждали, являются примерами специальных операторов, которые манипулируют лексическим окружением, поскольку они вводят новые лексические связи для переменных. Любая конструкция, такая как DO или DOTIMES, которая связывает лексические переменные будет развернута в LET или LET*. Специальный оператор SETQ является одним из операторов для доступа к лексическому окружению, поскольку он может быть использован для установки значений переменных, чьи связи были созданы с помощью LET и LET*.

Однако, не только переменные могут быть поименованы внутри лексического окружения. Хотя большинство функций и определены глобально с использованием DEFUN, но все равно возможно создание локальных функций с помощью специальных операторов FLET и LABELS, локальных макросов с помощью MACROLET, а также специальных видов макросов (называемых символьными макросами) с помощью SYMBOL-MACROLET.

Точно также как и `LET` позволяет вам ввести переменную, чьей областью видимости будет тело `LET`, `FLET` и `LABELS` позволяют вам определить функцию, которая будет видна только внутри области видимости `FLET` или `LABELS`. Эти специальные операторы являются очень удобными, если вам нужна локальная функция, которая является слишком сложной для ее определения как `LAMBDA`, или если вам нужно вызвать ее несколько раз. Оба этих оператора имеют одинаковую форму, которая выглядит так:

```
(flet (function-definition*)  
      body-form*)
```

или так:

```
(labels (function-definition*)  
        body-form*)
```

где каждая из `function-definition` имеет следующую форму:

```
(name (parameter*) form*)
```

Разница между `FLET` и `LABELS` заключается в том, что имена функций, которые определены с помощью `FLET`, могут использоваться только в теле `FLET`, в то время как имена, определенные с помощью `LABELS` могут использоваться сразу, включая тела функций, определенных с помощью `LABELS`. Таким образом, `LABELS` может определять рекурсивные функции, а `FLET` — не может. Может показаться ограничением то, что `FLET` не может быть использован для определения рекурсивных функций, но Common Lisp предоставляет и `FLET` и `LABELS` по той причине, что иногда бывает полезным иметь возможным написать локальную функцию, которая может вызвать другую функцию с тем же именем, либо глобальную, либо определенную в охватывающей области видимости.

Внутри тела `FLET` или `LABELS`, вы можете использовать имена определенных функций, точно также как и имена любых других функций, включая использование со специальным оператором `FUNCTION`. Поскольку вы можете использовать `FUNCTION` для получения объекта-функции, представляющего функцию, определенную с помощью `FLET` или `LABELS`, и поскольку `FLET` и `LABELS` могут быть в области видимости других связывающих форм, таких как `LET`, то эти функции могут использоваться как замыкания (closures).

Поскольку локальные функции могут ссылаться на переменные из охватывающего окружения, то они могут часто записываться таким образом, чтобы принимать меньше параметров, чем эквивалентные вспомогательные функции. Это очень удобно, когда вам необходимо в качестве параметра-функции передать функцию, которая принимает единственный аргумент. Например, в следующей функции, которую вы увидите снова в главе 25, функция `count-version`, определенная с помощью `FLET`, принимает единственный аргумент, как этого требует функция `walk-directory`, но она также может использовать переменную `versions`, заданную охватывающим `LET`:

```
(defun count-versions (dir)  
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))  
    (flet ((count-version (file)  
              (incf (cdr (assoc (major-version (read-id3 file)) versions)))))  
      (walk-directory dir #'count-version :test #'mp3-p)  
      versions)))
```

Эта функция также может быть записана с использованием анонимной функции вместо

использования FLET, но задание имени делает исходный текст более понятным.

И когда вспомогательная функция должна быть рекурсивной, она не может быть анонимной. Когда вам не нужно определять рекурсивную вспомогательную функцию как глобальную функцию, то вы можете использовать LABELS. Например, следующая функция, collect-leaves, использует рекурсивную вспомогательную функцию walk для прохода по дереву и сбора всех объектов дерева в список, который затем возвращается collect-leaves (после его реверсирования):

```
(defun collect-leaves (tree)
  (let ((leaves ()))
    (labels ((walk (tree)
              (cond
                ((null tree))
                ((atom tree) (push tree leaves))
                (t (walk (car tree))
                   (walk (cdr tree))))))
      (walk tree))
    (nreverse leaves)))
```

Снова отметьте, как внутри функции walk вы можете сослаться на переменную leaves, объявленную окружающим LET.

FLET и LABELS также являются полезными при раскрытии макросов – макрос может раскрываться в код, который содержит FLET или LABELS для создания функций, которые могут быть использованы внутри тела макроса. Этот прием может быть использован либо для введения функций, которые будет вызывать пользователь макроса, либо для организации кода, генерируемого макросом. Это может служить примером того, как может быть определена функция, такая как CALL-NEXT-METHOD, которая может быть использована только внутри определения метода.

К той же группе, что FLET и LABELS, можно отнести специальный оператор MACROLET, который вы можете использовать для определения локальных макросов. Локальные макросы работают также, как и глобальные макросы, определенные с помощью DEFMACRO, за тем исключением, что они не затрагивают глобальное пространство имен. Когда вычисляется MACROLET, то выражения в теле вычисляются с использованием локального определения макроса, которое возможно скрывает глобальное определение, а также используя локальные определения из окружающих выражений. Подобно FLET и LABELS, MACROLET может использоваться напрямую, но оно также очень удобно для использования кодом, сгенерированным макросом – путем обертки в MACROLET некоторого кода, написанного пользователем, макрос может предоставлять конструкции, которые могут быть использованы только внутри этого кода, или скрывать глобально определенный макрос. Вы увидите примеры использования MACROLET в главе 31.

В заключение, еще одним специальным оператором для определения макросов является SYMBOL-MACROLET, который определяет специальный вид макросов, называемых символьными макросами (symbol macro). Символьные макросы аналогичны обычным, за тем исключением, что они не могут принимать аргументы, и их используют как обычный символ, а не в листовой записи. Другими словами, после того, как вы определили символьный макрос с некоторым именем, любое использование этого символа как значения будет раскрыто, и вместо него будет вычислена соответствующая форма. Это как раз относится к тому, как макросы, такие как WITH-SLOTS и WITH-ACCESSORS получают возможность определения "переменных", которые осуществляют доступ к состоянию определенного объекта. Например, следующее выражение WITH-SLOTS:

```
(with-slots (x y z) foo (list x y z)))
```

может быть раскрыто в код, который использует SYMBOL-MACROLET:

```
(let ((#:g149 foo))
  (symbol-macrolet
    ((x (slot-value #:g149 'x))
     (y (slot-value #:g149 'y))
     (z (slot-value #:g149 'z)))
    (list x y z)))
```

Когда вычисляется выражение `(list x y z)`, то символы `x`, `y` и `z` будут заменены на соответствующие раскрытия, такие как `(slot-value #:g149 'x)`.

Символьные макросы, наиболее часто используются локально и определяются с помощью SYMBOL-MACROLET, но Common Lisp также предоставляет макрос DEFINE-SYMBOL-MACRO, который определяет глобальный символьный макрос. Символьный макрос, определенный с помощью SYMBOL-MACROLET скрывает другие макросы с тем же именем, определенные с помощью DEFINE-SYMBOL-MACRO или охватывающих выражений SYMBOL-MACROLET.

Локальный поток управления

Следующие четыре специальных оператора, о которых я буду говорить, также создают и используют имена в лексическом окружении, но для целей изменения контроля потока, а не для определения новых функций и макросов. Я упоминал ранее все четыре из этих специальных операторов, потому что они предоставляют низкоуровневые механизмы, используемые для других особенностей языка. Вот они: BLOCK, RETURN-FROM, TAGBODY, and GO. Первые два, BLOCK and RETURN-FROM, используются вместе для написания кода, который совершает выход немедленно из секции кода – я говорил про RETURN-FROM в Главе 5, как о способе немедленного выхода из функции, но он работает и в более общих случаях, чем тот. Два другие, TAGBODY and GO, предоставляют вполне низкоуровневую goto конструкцию, которая составляет основу для всех высокоуровневых конструкций цикла, которые вы уже видели.

Общий скелет формы BLOCK таков:

```
(block name
  form*)
```

name является символом и *form**, это формы языка. Формы выполняются по порядку и значение последней формы возвращается как значение всего BLOCK, если не использован RETURN-FROM для возврата из блока ранее. RETURN-FROM форма, как вы видели в Главе 5, состоит из имени блока, из которого выходят, и, по желанию, формы, которая предоставляет возвращаемое значение. Когда RETURN-FROM выполняется, это является причиной немедленного выхода из упомянутого в нём BLOCK. Если RETURN-FROM вызван с формой, возвращающей значение, BLOCK вернёт это значение; в ином случае BLOCK вернёт NIL.

Именем для BLOCK может быть любой символ, включая NIL. Многие из стандартных макросов конструкций контроля, таких как DO, DOTIMES и DOLIST, генерируют расширение, состоящее из BLOCK, названного NIL. Это позволяет вам использовать макрос RETURN, который скорее является синтаксическим сахаром для `(return-from nil ...)`, чтобы прерывать такие циклы. Так, следующий цикл напечатает не более десяти случайных чисел, остановившись сразу же, как только достигнет числа большего чем 50:

```
(dotimes (i 10)
  (let ((answer (random 100)))
    (print answer)
    (if (> answer 50) (return)))))
```

Задающие функции макросы, такие как DEFUN, FLET и LABELS, с другой стороны, оборачивают свои тела в BLOCK с тем же именем, что и функция. Вот почему вы можете пользоваться RETURN-FROM для возврата из функции.

TAGBODY и GO имеют такое же отношение друг к другу, как BLOCK и RETURN-FROM: TAGBODY форма задаёт контекст в котором имена определены, что может быть использовано GO. Скелет TAGBODY следующий:

```
(tagbody
  tag-or-compound-form*)
```

где каждая *tag-or-compound-form*, это или символ, называемый *тег*, или не пустая форма в виде списка. Формы списка выполняется по порядку и теги игнорируются за исключением, про которое я скоро скажу. После выполнения последней формы в TAGBODY, TAGBODY возвращает NIL. В любом месте внутри лексической области видимости TAGBODY вы можете использовать специальный оператор GO для немедленного перехода на любой тег и выполнение продолжится с формы, следующей за тегом. Например, вы можете написать простейший бесконечный цикл с TAGBODY и GO в виде этого:

```
(tagbody
  top
  (print 'hello)
  (go top))
```

Заметьте, что в то время, как имена тегов должны появляться на верхнем уровне в TAGBODY, не заключёнными внутри форм, специальный оператор GO может появляться где угодно внутри области видимости TAGBODY. Это означает, что вы можете написать цикл, который выполняется случайное число раз, например так:

```
(tagbody
  top
  (print 'hello)
  (when (plussp (random 10)) (go top)))
```

Ещё более глупый пример TAGBODY, который показывает вам, что можно иметь множество тегов в одном TAGBODY, выглядит так:

```
(tagbody
  a (print 'a) (if (zerop (random 2)) (go c))
  b (print 'b) (if (zerop (random 2)) (go a))
  c (print 'c) (if (zerop (random 2)) (go b)))
```

Эта форма будет прыгать вокруг случайно печатаемых *a*, *b* и *c* до тех пор, пока последнее RANDOM выражение вернёт 1 и управление наконец достигнет конца TAGBODY.

TAGBODY редко используется прямо, так как почти всегда удобней писать итеративные конструкции в терминах существующих циклических макросов. Однако он становится удобным для перевода алгоритмов, написанных на других языках в Коммон Лисп либо автоматически,

либо в ручную. Примером инструмента автоматического перевода является транслятор FORTRAN-to-Common Lisp, f2cl, который переводит исходный код на Фортране, в исходный код на Коммон Лисп для того, чтобы сделать различные библиотеки из Фортрана доступными для программистов на Коммон Лисп. Так как многие библиотеки Фортрана были написаны до революции структурного программирования, они полны операторов *goto*. f2cl транслятор может просто переводить такие *goto* в GO внутри соответствующих TAGBODY.

Аналогично, TAGBODY и GO могут быть полезны, когда переводятся алгоритмы, написанные или прозой, или диаграммами переходов – например в классической серии Дональда Кнута "Искусство программирования", он описывает алгоритмы, используя формат "рецептов": step 1, do this; step 2, do that; step 3, go back to step 2; и так далее. Для примера на странице 142, "Искусства программирования", Том 2: Получисленные алгоритмы, 3-е издание (Addison-Wesley, 1998), он описывает Алгоритм S, который вы увидите в Главе 27, в такой форме:

Алгоритм S (Метод выбора последовательности). Для выбора n случайных записей из множества N , где $0 < n \leq N$.

```
* S1. [Инициализировать.] Установить  $t \leftarrow 0$ ,  $m \leftarrow 0$ . (В этом алгоритме  $m$ 
представляет количество записей уже выбранных, а  $t$  общее количество
записей которые мы просмотрели.)
* S2. [Сгенерировать U.] Сгенерировать случайное число  $U$ , равномерно
распределённое между нулём и единицей.
* S3. [Проверить.] Если  $(N - t)U \geq n - m$ , то перейти к шагу S5.
* S4. [Выбрать.] Выбрать следующую запись в последовательность и увеличить
 $m$  и  $t$  на 1. Если  $m < n$ , то перейти к шагу S2; иначе
последовательность закончена и алгоритм завершается.
* S5. [Пропустить.] Пропустить следующую запись (не включать её в
последовательность), увеличить  $t$  на 1, и вернуться к шагу S2.
```

Это описание может быть легко переведено в Коммон Лисп функцию, после переименования нескольких переменных таким образом:

```
(defun algorithm-s (n max) ; max это N в алгоритме Кнута
  (let (seen ; t в в алгоритме Кнута
        selected ; m в алгоритме Кнута
        u ; U в алгоритме Кнута
        (records ())) ; список, где мы сохраняем выбранные записи
    (tagbody
      s1
        (setf seen 0)
        (setf selected 0)
      s2
        (setf u (random 1.0))
      s3
        (when (>= (* (- max seen) u) (- n selected)) (go s5))
      s4
        (push seen records)
        (incf selected)
        (incf seen)
        (if (< selected n)
            (go s2)
            (return-from algorithm-s (nreverse records)))
      s5
        (incf seen)
        (go s2))))
```

Это не самый красивый код, но легко проверить, что это канонический перевод алгоритма Кнута.

Однако этот код, в отличие от прозаического описания у Кнута, может быть запущен и проверен. Затем вы можете начать рефакторинг, проверяя после каждого изменения, что функция всё ещё работает.

Сложив все кусочки, вы возможно получите что-то наподобие этого:

```
(defun algorithm-s (n max)
  (loop for seen from 0
    when (< (* (- max seen) (random 1.0)) n)
    collect seen and do (decf n)
    until (zerop n)))
```

Хотя это может быть не очевидно, что этот код правильно представляет Алгоритм S, но если вы пришли к нему посредством серии функций, которые вели себя идентично оригинальному дословному переводу рецепта Кнута, у вас есть хорошее основание верить, что он правильный.

Раскрытие стека

Другим интересным аспектом языка является то, что специальные операторы дают вам контроль за поведением стека вызовов функций. Например, хотя вы обычно используете BLOCK и TAGBODY для управления потоком выполнения команд внутри отдельной функции, вы также можете использовать их вместе с замыканиями, для выполнения немедленного нелокального выхода из функции в точку, находящуюся ниже на стеке. Это происходит потому, что имена BLOCK и таги TAGBODY могут быть FIXME closed over by any code внутри лексического окружения BLOCK или TAGBODY. Например, рассмотрим следующую функцию:

```
(defun foo ()
  (format t "Entering foo~%")
  (block a
    (format t " Entering BLOCK~%")
    (bar #'(lambda () (return-from a)))
    (format t " Leaving BLOCK~%"))
  (format t "Leaving foo~%"))
```

Анонимная функция, переданная bar использует RETURN-FROM для выхода из BLOCK. Но это выражение RETURN-FROM не будет вычислено до тех пор, пока анонимная функция не будет выполнена с помощью FUNCALL или APPLY. Теперь, предположим что bar выглядит следующим образом:

```
(defun bar (fn)
  (format t " Entering bar~%")
  (baz fn)
  (format t " Leaving bar~%"))
```

Все равно, анонимная функция не будет вызвана. Теперь посмотрим на baz.

```
(defun baz (fn)
  (format t " Entering baz~%")
  (funcall fn)
  (format t " Leaving baz~%"))
```

И в заключение, функция выполняется. Но к чему приведет вызов RETURN-FROM из блока, который находится на несколько уровней выше по стеку? На проверку все работает отлично – стек отматывается к той точке, где BLOCK был создан и контроль возвращается из соответствующего выражения BLOCK. Вызовы FORMAT в функциях foo, bar и baz

демонстрируют это:

```
CL-USER> (foo)
Entering foo
  Entering BLOCK
    Entering bar
      Entering baz
Leaving foo
NIL
```

Заметьте, что единственным выдаваемым сообщением `Leaving` . . . является то, которое было вставлено после выражения `BLOCK` в функции `foo`.

Поскольку имена блоков имеют лексическую область видимости, `RETURN-FROM` всегда возвращается из наименьшего охватывающего блока `BLOCK` в лексическом окружении, в котором задана форма `RETURN-FROM`, даже если `RETURN-FROM` выполняется в отличающемся динамическом контексте. Например, `bar` также может содержать форму `BLOCK` с именем `a`, например, вот так:

```
(defun bar (fn)
  (format t " Entering bar~%")
  (block a (baz fn))
  (format t " Leaving bar~%"))
```

Это дополнительное выражение `BLOCK` никак не изменит поведение `foo` – поиск имен, таких как `a`, производится лексически, во время компиляции, а не динамически, так что дополнительный блок не влияет на работу `RETURN-FROM`. И наоборот, имя `BLOCK` может быть использовано только тем выражением `RETURN-FROM`, которое задано внутри лексической области видимости выражения `BLOCK`; нет никакой возможности для кода, который определен вне блока, выполнить выход из него, кроме как выполнения замыкания, которая выполнит `RETURN-FROM` и лексического окружения `BLOCK`.

`TAGBODY` и `GO` работают точно также как и `BLOCK` и `RETURN-FROM`. Когда вы выполняете замыкание, которое содержит выражение `GO`, и `GO` вычисляется, то стек отматывается назад к соответствующей форме `TAGBODY`, и затем уже переходит к соответствующему тегу.

Однако, имена блоков `BLOCK` и таги `TAGBODY`, достаточно сильно отличаются от связывания лексических переменных. Как обсуждалось в главе 6, лексические связывания имеют неопределенный экстенд (`FIXME extent`), что означает, что связывания не исчезают даже после того, как связывающая форма будет возвращена. С другой стороны, выражения `BLOCK` и `TAGBODY` имеют динамический экстенд – вы можете выполнить `RETURN-FROM` из `BLOCK` или `GO` из тага `TAGBODY` только когда `BLOCK` или `TAGBODY` находятся на стеке вызова функций. Другими словами, замыкание, которое захватывает имя блока, или таг `TAGBODY`, может быть передано вниз по стеку, для последующего выполнения, но оно не может быть возвращено вверх по стеку. Если вы выполните замыкание, которое будет пытаться выполнить `RETURN-FROM` из `BLOCK`, после того, как `BLOCK` будет завершен, то вы получите ошибку. Аналогично, попытка выполнить `GO` для `TAGBODY` которое больше не существует, также вызовет ошибку.

Маловероятно, что вам понадобится использовать `BLOCK` и `TAGBODY` для такого способа раскрутки стека. Но вы скорее всего будете использовать этот подход как часть системы условий и рестартов, так что понимание того, как оно работает, поможет вам понять что в точности делается при запуске рестарта.

`CATCH` и `THROW` являются еще одной парой специальных операторов, которые приводят к раскрутке стека. Вы будете использовать эти операторы еще реже, чем описанные выше – они

являются наследием ранних версий Lisp, которые не имели в своем составе систему условий Common Lisp. Они не должны путаться с конструкциями `try/catch` и `try/except` из таких языков, как Java и Python.

CATCH и THROW являются динамическими аналогами конструкций BLOCK и RETURN-FROM. Так что вы используете CATCH для какого-то кода и затем используете THROW для выхода из блока CATCH с возвратом указанного значения. Разница заключается в том, что связь между CATCH и THROW устанавливается динамически – вместо лексически ограниченного имени, метка CATCH является объектом, называемым тагом catch, и любое выражение THROW вычисляется внутри динамического экстенда CATCH, так что "выбрасывание" (FIXME throws) этот объекта будет приводить к раскрутке стека к блоку CATCH и приводить к немедленному возврату. Так что вы можете написать новые версии функций `foo`, `bar` и `baz` используя CATCH и THROW вместо BLOCK и RETURN-FROM:

```
(defparameter *obj* (cons nil nil)) ; некоторый произвольный объект
(defun foo ()
  (format t "Entering foo~%")
  (catch *obj*
    (format t " Entering ''CATCH''~%")
    (bar)
    (format t " Leaving ''CATCH''~%")))
  (format t "Leaving foo~%"))
(defun bar ()
  (format t " Entering bar~%")
  (baz)
  (format t " Leaving bar~%"))
(defun baz ()
  (format t " Entering baz~%")
  (throw *obj* nil)
  (format t " Leaving baz~%"))
```

Заметьте, что нет необходимости передавать замыкание вниз по стеку – `baz` может напрямую вызвать THROW. Результат будет таким же как и раньше.

```
CL-USER> (foo)
Entering foo
  Entering ''CATCH''
    Entering bar
      Entering baz
Leaving foo
NIL
```

Однако, CATCH и THROW слишком динамичные. И в CATCH, и в THROW, форма, представляющая таг, вычисляется, что означает, что ее значение в обоих случаях определяется во время выполнения. Так что, если некоторый код в `bar` присвоит новое значение `*obj*`, то THROW в `baz` не будет пойман в том же блоке CATCH. Это делает использование CATCH и THROW более тяжелым чем BLOCK и RETURN-FROM. Единственным преимуществом версии `foo`, `bar` и `baz`, которая использует CATCH и THROW, является то, что нет необходимости передавать замыкание вниз по стеку, для возврата из CATCH – любой код, который выполняется внутри динамического экстенда CATCH может заставить вернуться к нему, путем "бросания" (FIXME throwing) нужного объекта.

В старых диалектах Lisp в которых не было ничего подобного системе условий Common Lisp, CATCH и THROW использовались для обработки ошибок. Однако, для того, чтобы сделать ее сопровождаемой, таги catch обычно были FIXME quoted symbols, так что вы могли понять, глядя на CATCH и THROW, где они будут перехвачены во время выполнения. В Common Lisp вы будете

редко иметь нужду в использовании CATCH и THROW, поскольку система условий намного более гибкая.

Последним специальным оператором, относящимся к управлению стеком вызовов, является UNWIND-PROTECT, который я вскользь упоминал раньше. UNWIND-PROTECT позволяет вам контролировать что происходит при раскрутке стека и быть уверенным в том, что определенный код всегда выполняется, независимо от того, как поток выполнения покидает область видимости UNWIND-PROTECT – обычным способом, путем запуска рестарта, или любым другим способом, описанным в этом разделе. Базовая форма UNWIND-PROTECT выглядит примерно так:

```
(unwind-protect protected-form
  cleanup-form*)
```

Сначала вычисляется одиночное выражение protected-form, и затем, вне зависимости от способа его завершения, будут вычислены выражения заданные cleanup-forms. Если protected-form завершается нормальным образом, то его результат будет возвращен UNWIND-PROTECT после вычисления возвратит cleanup-forms. Выражения cleanup-forms вычисляются в том же самом динамическом окружении, что и UNWIND-PROTECT, так что все динамические переменные, связи, перезапуски и обработчики условий, будут доступны коду cleanup-forms, так как они были видны перед выполнением UNWIND-PROTECT.

Вы редко будете использовать UNWIND-PROTECT напрямую. Наиболее часто, вы его будете использовать как основу для макросов в стиле WITH-, таких как WITH-OPEN-FILE, который вычисляет произвольное количество выражений в контексте, где они имеют доступ к некоторому ресурсу, который должен быть освобожден после того, как все выполнено, вне зависимости от того, как выражения были завершены – нормальным образом, или через рестарт или любой другой нелокальный выход. Например, если вы пишете библиотеку для работы с базами данных, которая определяет функции open-connection и close-connection, то вы можете написать вот такой вот макрос:

```
(defmacro with-database-connection ((var &rest open-args) &body body)
  `(let ((,var (open-connection ,@open-args)))
    (unwind-protect (progn ,@body)
      (close-connection ,var))))
```

что позволяет вам писать в следующем стиле:

```
(with-database-connection (conn :host "foo" :user "scott" :password "tiger")
  (do-stuff conn)
  (do-more-stuff conn))
```

и не беспокоиться о закрытии соединения к базе данных, поскольку UNWIND-PROTECT позаботится о его закрытии, вне зависимости от того, что случится в теле формы with-database-connection.

Множественные значения

Еще одним свойством Common Lisp, которое я упоминал вскользь в главе 11, когда я обсуждал GETHASH, является возможность возвращения множества значений из одного выражения. Сейчас мы обсудим эту функциональность более подробно. Правда не совсем правильно обсуждать эту функциональность в главе про специальные операторы, поскольку этот функционал не реализуется отдельными операторами, а глубоко интегрирован в язык. Операторами, которые вы наиболее часто будете использовать с множественными значениями,

являются макросы и функции, а не специальные операторы. Но базовая возможность получения множественных значений обеспечивается специальным оператором `MULTIPLE-VALUE-CALL`, на основе которого построен более часто используемый макрос `MULTIPLE-VALUE-BIND`.

Ключом к пониманию множественных значений является тот факт, что возврат множества значений совершенно отличается от возврата списка — если форма возвращает множество значений, то до тех пор пока вы не сделаете что-то специальное для их получения, все значения, кроме первого (основного) будут игнорироваться. Для того, чтобы увидеть это отличие, рассмотрим функцию `GETHASH`, которая возвращает два значения: найденное значение и логическое значение, которое равно `NIL` если значение не было найдено. Если бы эти значения возвращались в виде списка, то при каждом вызове `GETHASH` вам требовалось бы выделять найденное значение из списка, вне зависимости от того, нужно ли вам второе возвращаемое значение или нет. Предположим, что у вас есть хэш-таблица `*h*`, которая содержит числа. Если бы `GETHASH` возвращал бы список, то вы бы не могли написать что-то подобное:

```
(+ (gethash 'a *h*) (gethash 'b *h*))
```

поскольку `+` ожидает, что его аргументами будут числа, а не списки. Но поскольку механизм работы с множественными значениями просто игнорирует дополнительные возвращаемые значения, которые вам не нужны, то этот код будет работать нормально.

Имеется два аспекта использования множественных значений — возврат множественных значений, и получение не основных значений, возвращаемых формами, которые возвращают множественные значения. Начальной точкой для возврата множественных значений являются функции `VALUES` и `VALUES-LIST`. Это обычные функции, а не специальные операторы, так их параметры передаются как обычно. `VALUES` принимает переменное число аргументов и возвращает их как множественные значения; `VALUES-LIST` принимает единственный аргумент — список значений, и возвращает все его содержимое в виде множественных значений. Иначе говоря:

```
(values-list x) === (apply #'values x)
```

Механизм возврата множественных значений зависит от конкретной реализации, также как и механизм передачи аргументов функциям. Почти все языковые конструкции возвращающие значения своих подвыражений, будут "передавать" множественные значения, возвращаемые подвыражениями. Так что, функция, которая возвращает результат вызова `VALUES` или `VALUES-LIST` сама будет возвращать множественные значения, и то же самое будет делать и функция, вызвавшая эту функцию, и т.д..

Но когда выражение вычисляется в позиции значения, то используется только основное значение — вот почему предыдущий пример с добавлением работает как ожидается. Специальный оператор `MULTIPLE-VALUE-CALL` предоставляет механизм, который позволяет вам работать с множественными значениями, возвращаемыми выражениями. `MULTIPLE-VALUE-CALL` аналогичен `FUNCALL` за тем исключением, что `FUNCALL` является обычной функцией, и как следствие — использует только основное значение из переданных аргументов, в то время как `MULTIPLE-VALUE-CALL` передает функции, указанной в качестве первого аргумента, все значения, возвращенные остальными выражениями, указанными в качестве аргументов.

```
(funcall #' + (values 1 2) (values 3 4)) ==> 4  
(multiple-value-call #' + (values 1 2) (values 3 4)) ==> 10
```

Но скорее всего вы достаточно редко будете просто передавать все значения, возвращенные одной функцией, в другую функцию. Скорее всего вы захотите сохранить множественные

значения в отдельные переменные и затем что-то сделать с ними. Макрос `MULTIPLE-VALUE-BIND`, с которым вы встречались в главе 11, является наиболее часто используемым оператором для работы с множественными значениями. В общем виде он выглядит вот так:

```
(multiple-value-bind (variable*) values-form
  body-form*)
```

Выражение `values-form` вычисляется, и множественные значения, которые возвращаются им, присваиваются указанным переменным. Затем вычисляются выражения `body-forms` используя указанные переменные. Так что:

```
(multiple-value-bind (x y) (values 1 2)
  (+ x y)) ==> 3
```

Другой макрос – `MULTIPLE-VALUE-LIST`, еще более простой – он принимает одно выражение, вычисляет его и собирает полученные множественные значения в список. Другими словами, этот макрос выполняет действия обратные действию `VALUES-LIST`.

```
CL-USER> (multiple-value-list (values 1 2))
(1 2)
CL-USER> (values-list (multiple-value-list (values 1 2)))
1
2
```

Однако, если вы обнаружите, что часто используете `MULTIPLE-VALUE-LIST`, то это может быть сигналом того, что некоторая функция должна возвращать список, а не множественные значения.

И в заключение, если вы хотите присвоить множественные значения, возвращенные формой, существующим переменным, то вы можете использовать функцию `VALUES` для выполнения `SETF`. Например:

```
CL-USER> (defparameter *x* nil)
*X*
CL-USER> (defparameter *y* nil)
*Y*
CL-USER> (setf (values *x* *y*) (floor (/ 57 34)))
1
23/34
CL-USER> *x*
1
CL-USER> *y*
23/34
```

EVAL-WHEN

Еще одним специальным оператором, принципы работы которого вам нужно понимать для того, чтобы писать некоторые виды макросов, является `EVAL-WHEN`. По некоторым причинам, книги о Lisp часто считают `EVAL-WHEN` орудием только для экспертов. Но единственным требованием для понимания `EVAL-WHEN` является понимание того, как взаимодействуют две функции – `LOAD` и `COMPILE-FILE`. И понимание принципов работы `EVAL-WHEN` будет важным для вас, когда вы начнете писать сложные макросы, такие как, мы будем писать в главах 24 и 31.

В предыдущих главах я немного касался вопросов совместной работы `LOAD` и `COMPILE-FILE`, но стоит рассмотреть этот вопрос снова. Задачей `LOAD` является загрузка файла и вычисление

всех выражений верхнего уровня, содержащихся в нем. Задачей `COMPILE-FILE` является компиляция исходного текста в файл `FASL`, который может быть затем загружен с помощью `LOAD`, так что `(load "foo.lisp")` и `(load "foo.fasl")` являются практически эквивалентными.

Поскольку `LOAD` вычисляет каждое выражение до чтения следующего, побочный эффект вычисления выражений, находящихся ближе к началу файла, может воздействовать на то, как будут читаться и вычисляться формы, находящиеся дальше в файле. Например, вычисление выражения `IN-PACKAGE` изменяет значение переменной `*PACKAGE*`, что затронет процедуру чтения всех остальных выражений. Аналогичным образом, выражение `DEFMACRO` находящееся раньше в файле, может определить макрос, который будет использоваться кодом, находящимся далее в файле.

С другой стороны, `COMPILE-FILE` обычно не вычисляет выражения в процессе компиляции; это происходит в момент загрузки файла `FASL`. Однако `COMPILE-FILE` должен вычислять некоторые выражения, такие как `IN-PACKAGE` и `DEFMACRO`, чтобы поведение `(load "foo.lisp")` и `(load "foo.fasl")` было консистентным.

Так как же работают макросы, такие как `IN-PACKAGE` и `DEFMACRO`, в тех случаях, когда они обрабатываются `COMPILE-FILE`? В некоторых версиях Lisp, существовавших до разработки Common Lisp, компилятор файлов просто знал, что он должен вычислять некоторые макросы в добавление к их компиляции. Common Lisp избегает необходимости в таких хаков путем введения специального оператора `EVAL-WHEN` взятого из Maclisp. Этот оператор, как и предполагает его имя, позволяет контролировать то, когда определенные части кода должны вычисляться. В общем виде выражение `EVAL-WHEN` выглядит вот так:

```
(eval-when (situation*)  
  body-form*)
```

Есть три возможных условия (`situation`) – `:compile-toplevel`, `:load-toplevel` и `:execute`, и то, которое вы укажете, будет определять то, когда будут вычислены выражения указанные `body-forms`. `EVAL-WHEN` с несколькими условиями аналогичен записи нескольких выражений `EVAL-WHEN` с разными условиями, но с одинаковыми выражениями. Для объяснения того, что означают эти условия, нам необходимо объяснить что делает `COMPILE-FILE` (который также называют компилятором файлов) в процессе компиляции файла.

Для того, чтобы объяснить как `COMPILE-FILE` компилирует выражения `EVAL-WHEN`, я должен описать отличия между компиляцией выражений верхнего уровня (`FIXME top-level form`), и компиляцией остальных выражений. Выражения верхнего уровня, грубо говоря, это те, которые будут скомпилированы в исполняемый код, который будет выполнен при загрузке файла `FASL`. Так что, все выражения, которые находятся верхнем уровне (`FIXME top level of a source`) файла с исходным текстом, будут скомпилированы как выражения верхнего уровня. Аналогичным образом, все выражения указанные в выражении `PROGN` верхнего уровня, будут также скомпилированы как выражения верхнего уровня (`PROGN` сам ничего не делает – он лишь группирует вместе указанные выражения), и они будут выполнены при загрузке `FASL`. Аналогичным образом, выражения указанные в `MACROLET` или `SYMBOL-MACROLET` будут скомпилированы как выражения верхнего уровня, поскольку после того, как компилятор раскроет локальные и символьные макросы, в скомпилированном коде не останется никаких упоминаний `MACROLET` или `SYMBOL-MACROLET`. И в заключение, раскрытие макроса верхнего уровня будет скомпилировано как выражение верхнего уровня.

Таким образом, `DEFUN` указанный на верхнем уровне исходного текста является выражением верхнего уровня – код, который определяет функцию и связывает ее с именем, будет выполнен

при загрузке FASL, но выражения внутри тела функции, которые не будут выполнены до тех пор, пока функция не будет вызвана, не являются выражениями верхнего уровня. Большинство выражений компилируются одинаково вне зависимости от того, на верхнем ли уровне они или нет, но семантика выражения EVAL-WHEN зависит от того, будет ли оно скомпилировано как выражение верхнего уровня, выражение не верхнего уровня, или просто вычислено, и все это в комбинации с условиями, указанными в выражении.

Условия `:compile-toplevel` и `:load-toplevel` контролируют поведение EVAL-WHEN, которое компилируется как выражение верхнего уровня. Когда присутствует условие `:compile-toplevel`, то компилятор файла вычислит заданные выражения во время компиляции. Когда указано условие `:load-toplevel`, то он будет компилировать выражения как выражения верхнего уровня. Если ни одно из этих условий не указано в выражении EVAL-WHEN верхнего уровня, то компилятор просто игнорирует его.

Когда EVAL-WHEN компилируется как выражение не верхнего уровня, то он либо компилируется как PROGN, в том случае, если указано условие `:execute`, либо просто игнорируется. Аналогичным образом, вычисляемое выражение (FIXME evaluated) EVAL-WHEN (что включает в себя выражения EVAL-WHEN верхнего уровня в исходном тексте, обрабатываемом LOAD и EVAL-WHEN, вычисляемый во время компиляции когда он является подвыражением другого EVAL-WHEN с условием `:compile-toplevel`) также рассматривается как PROGN если указано условие `:execute`, и игнорируется в противном случае. (FIXME может быть стоит сделать табличку с условиями и стадиями компиляции?)

Таким образом, макрос, такой как IN-PACKAGE может производить необходимые действия и во время компиляции и при загрузке из исходного кода путем раскрытия в выражения EVAL-WHEN выглядящие примерно так:

```
(eval-when (':compile-toplevel' '':load-toplevel' '':execute'))  
  (setf *package* (find-package "PACKAGE-NAME")))
```

значение `*PACKAGE*` будет выставлено во время компиляции из-за условия `:compile-toplevel`, во время загрузки FASL из-за условия `:load-toplevel` и во время загрузки исходного кода из-за условия `:execute`.

Существует два широко распространенных способа использования EVAL-WHEN. Первый – если вы хотите написать макрос, которому необходимо сохранить некоторую информацию во время компиляции, и которая будет использоваться при генерации раскрытий других макро-выражений в том же файле. Это обычно нужно для определяющих (FIXME definitional) макросов, когда определение, расположенное в начале файла, могут влиять на код, генерируемый для определения, расположенного далее в том же файле. Вы будете писать такие макросы в главе 24.

Вам также может понадобиться использовать EVAL-WHEN если вы захотите поместить определение макроса и вспомогательной функции, которая используется в этом макросе, в том же файле с исходным текстом, что и код, использующий данный макрос. DEFMACRO уже использует EVAL-WHEN в своем раскрытии, так что определение макроса становится доступным для использования сразу. Но обычно DEFUN не делает определение функции доступным во время компиляции, а если вы используете макрос в том же файле, где он определен, то вам необходимо чтобы были определены и все функции, используемые в макросе. Если вы обернете все определения вспомогательных функций в выражение EVAL-WHEN с условием `:compile-toplevel`, то определения будут доступны при раскрытии макросов. Вы наверное захотите включить также условия `:load-toplevel` и `:execute` поскольку макросы будут требовать наличие определения функций после того, как файл скомпилирован и загружен, или если вы

загружает файл с исходным текстом вместо компиляции.

Другие специальные операторы

Все оставшиеся четыре специальных оператора – `LOCALLY`, `THE`, `LOAD-TIME-VALUE` и `PROGV`, позволяют получить доступ к некоторым частям нижележащего языка, к которым доступ не может быть осуществлен другими способами. `LOCALLY` и `THE` являются частями системы объявлений `Common Lisp`, которая используется для "связывания" некоторых вещей с компилятором, что не изменит работу вашего кода, но позволит генерировать лучший код – более быстрый, более понятные сообщения об ошибках, и т.п. Мы коротко обсудим объявления в главе 32.

Еще два оператора – `LOAD-TIME-VALUE` и `PROGV`, используются не часто, и объяснение того, почему это происходит, займет больше времени, чем объяснение того, что они делают. Я расскажу вам то, что они делают, так что просто вы будете иметь эту информацию. Когда-нибудь вы встретите один из них, и тогда вы будете готовы к пониманию их работы.

`LOAD-TIME-VALUE` используется (как видно из его имени) для создания значения во время загрузки. Когда компилятор обрабатывает код, который содержит выражение `LOAD-TIME-VALUE`, он генерирует код, который выполнит подвыражения лишь один раз, во время загрузки `FASL`, и код, содержащий выражение `LOAD-TIME-VALUE` будет ссылаться на вычисленное значение. Другими словами, вместо того, чтобы писать что-то подобное:

```
(defvar *loaded-at* (get-universal-time))  
(defun when-loaded () *loaded-at*)
```

вы можете просто написать вот так:

```
(defun when-loaded () (load-time-value (get-universal-time)))
```

В коде, не обрабатываемом `COMPILE-FILE`, выражение `LOAD-TIME-VALUE` вычисляется однажды, во время компиляции, что может происходить когда вы явно компилируете функцию с помощью `COMPILE`, или даже раньше, из-за неявной компиляции кода во время его вычисления. В некомпилируемом коде, `LOAD-TIME-VALUE` вычисляет свои подвыражения при каждом вычислении.

И в заключение, `PROGV` создает новые динамические привязки для переменных, чьи имена определяются во время выполнения. Это в основном полезно при реализации встраиваемых интерпретаторов для языков, с динамической областью видимости переменных. Базовая структура этого выражения выглядит вот так:

```
(progv symbols-list values-list  
  body-form*)
```

где `symbols-list` является выражением, которое вычисляется в список символов, а `values-list` – выражение, которое вычисляется в список значений. Каждый символ динамически связывается с соответствующим значением, и затем вычисляются выражения, указанные в `body-forms`. Разница между `PROGV` и `LET` заключается в том, что `symbols-list` вычисляется во время выполнения, и имена связываемых переменных вычисляются динамически. Как я уже сказал, этот оператор не понадобится вам очень часто.

И это вся информация о специальных операторах. В следующей главе, я вернусь к практическим темам и покажу вам как использовать пакетную систему `Common Lisp` для получения контроля за

пространствами имен, так что вы можете писать библиотеки и приложения, которые смогут сосуществовать без разрушения друг друга.

21. Программирование по-взрослому: Пакеты и Символы

В 4-й главе я рассказывал, как считыватель Lisp переводит текстовые имена в объекты, которые затем передаются вычислителю в виде так называемых *символов*. Оказывается, что иметь встроенный тип данных, специально для представления имён, очень удобно для многих видов программирования. Это, однако, не тема данной главы. В этой главе я расскажу об одном из наиболее явных и практических аспектов работы с именами: как избегать конфликта между независимо разрабатываемыми кусками кода.

Предположим, для примера, что вы пишете программу и решаете использовать чью-то библиотеку. Вы не хотите знать имена всех функций, переменных, классов или макросов внутри этой библиотеки, чтобы избежать путаницы между её именами и именами, которые вы используете в своей программе. Вы бы хотели, чтобы большинство имён в библиотеке и имён в вашей программе рассматривались отдельно, даже если они окажутся одинаковыми в написании. В то же самое время, вы хотите, чтобы некоторые имена, определённые в библиотеке, были легко доступны — те имена, что составляют её публичный API, который вы как раз хотите использовать.

В Common Lisp эта проблема пространства имён сводится просто к вопросу контроля за тем, как считыватель переводит текстовые имена в символы: если вы хотите чтобы два появления одного и того же имени рассматривались интерпретатором одинаково, вы должны убедиться, что считыватель использует один и тот же символ для представления каждого из них. И наоборот, если нужно, чтобы два имени рассматривались как разные, даже если они совпадают побуквенно, вам надо, чтобы считыватель создал разные символы для представления этих имён.

Как считыватель использует пакеты

В главе 4 я коротко рассказал, как считыватель в LISP переводит имена в символы, но я пропустил множество деталей — теперь настало время поближе взглянуть на то, что же происходит на самом деле.

Начну с описания синтаксиса имён, понимаемых считывателем, и того, как он соотносится с пакетами. На данный момент можете представить себе пакеты, как таблицы, которые отображают строки в символы. Как вы увидите в дальнейшем, в действительности это отображение можно регулировать более тонко, чем в простой таблице соответствий, но не теми способами, которые использует считыватель. Каждый пакет также имеет имя, которое может быть использовано для поиска пакета через функцию `FIND-PACKAGE`.

Две ключевые функции, которые считыватель использует для доступа к отображению имя-в-символ в пакете, это `FIND-SYMBOL` и `INTERN`. Обе они получают строку и, необязательно, пакет. Если пакет не указывается, на его место подставляется значение глобальной переменной `*PACKAGE*`, называемой также *текущим пакетом*.

`FIND-SYMBOL` ищет в пакете символ с именем, совпадающим со строкой, и возвращает его или `NIL`, если символ не найден. `INTERN` также возвратит существующий символ, но если его нет, создаст новый, назначит строку его именем и поместит в пакет.

Большинство имён используемых вами — неспециализированные имена, не содержащие двоеточий. Когда считыватель читает такое имя, он переводит его в символ путём изменения всех не экранированных букв в верхний регистр и передавая полученную строку `INTERN`. Таким образом каждый раз, когда считыватель читает то же имя в том же пакете, он получает тот же объект-символ. Это важно, так как интерпретатор использует объектное совпадение символов,

чтобы определить к какой функции, переменной или другому программному элементу данный символ относится. То есть причина, по которой выражение вида (hello-world) преобразуется в вызов определённой hello-world функции, это возврат считывателем одного и того же символа, и когда он читает вызов функции, и когда он читал форму DEFUN, которая эту функцию определяла. Имя, содержащее двоеточие или двойное двоеточие, является пакетно-специализированным именем. Когда считыватель читает пакетно-специализированное имя, он разбивает его в месте двоеточия(й) и берёт первую часть, как имя пакета, а вторую как имя символа. Затем считыватель просматривает соответствующий пакет и использует его для перевода имени в символ-объект.

Имена, содержащие только одинарное двоеточие, относятся к внешним символам, тем, которые пакет экспортирует для общего пользования. Если названный пакет не содержит символа с данным именем, или содержит, но он не экспортирован, считыватель сигнализирует об ошибке. Имена с двойным двоеточием могут обозначать любой символ в названном пакете, хотя это обычно плохой тон – множество экспортированных символов определяют публичный интерфейс пакета и, если вы не уважаете решение автора пакета о том, какие имена делать внешними, а какие внутренними, вы напрашиваетесь на неприятности в дальнейшем. С другой стороны, иногда авторы пакетов забывают экспортировать символы, явно предназначенные для внешнего пользователя. В таком случае имена с двойным двоеточием позволяют вам работать, не дожидаясь выпуска следующей версии пакета.

Ещё два аспекта в синтаксисе символов, которые понимает считыватель, это ключевые и внепакетные символы. Ключевые символы записываются как имена, начинающиеся с двоеточия. Такие символы добавляются в пакет, названный KEYWORD, и экспортируются автоматически. Кроме того, когда считыватель добавляет символ в KEYWORD, он также определяет константную переменную с символом в качестве имени, так и значения. Вот почему вы можете использовать ключевые слова в списке аргументов без кавычки спереди – когда вычисляется их значение, оно оказывается равным им самим. Таким образом:

```
(eq1 ':foo :foo) ==> T
```

Имена ключевых символов, как и всех остальных, перед интернированием преобразуются считывателем к верхнему регистру. В имя не включается двоеточие.

```
(symbol-name :foo) ==> "FOO"
```

Внепакетные символы записываются с поставленными впереди #:. Эти имена (без #:) преобразуются в верхний регистр, как обычно, и затем переводятся в символы, но эти символы не добавляются ни в один пакет; каждый раз, когда считыватель читает имя с #:, он создаёт новый символ. Таким образом:

```
(eq1 '#:foo '#:foo) ==> NIL
```

Вы очень редко будете пользоваться в записи таким синтаксисом, если вообще будете, но иногда вы будете видеть его при печати s-выражения с символом, возвращённым функцией GENSYM.

```
(gensym) ==> #:G3128
```

Немного про словарь пакетов и символов

Как я упомянул ранее, соответствие между именами и символами, предоставленными пакетом, устроено более гибко, чем простая таблица соответствий. Ядром каждого пакета является

поисковая таблица имя-в-символ, но символ в пакете можно сделать доступным через неспециализированное имя и другим путём. Чтобы поговорить об этих иных механизмах, вам понадобятся некоторые термины.

Для начала, все символы, которые могут быть найдены в данном пакете с помощью функции `FIND-SYMBOL`, называются *доступными* в этом пакете. Иными словами, доступными символами в пакете являются те, на которые можно ссылаться неспециализированными именами, когда пакет является текущим.

Символ может быть доступен в пакете двумя способами. Первый, когда пакетная имя-в-символ таблица содержит запись для этого символа. В этом случае мы говорим, что символ *присутствует* в пакете. Когда считыватель интернирует новый символ в пакет, он добавляет его в таблицу имя-в-символ. Первый пакет, в который интернирован символ, называется *домашним пакетом* этого символа.

Другой способ, которым символ может быть доступен в пакете, это когда пакет *наследует* его. Пакет наследует символы из других пакетов путём *использования* их. Наследуются только внешние символы из используемых пакетов. Символ делается внешним в пакете путём его *экспорта*. В дополнение к тому, что он наследуется использующим пакетом, экспорт символа — как вы видели в предыдущих разделах — делает возможным ссылаться на символ через специализированное имя с двоеточием.

Чтобы сохранить отображение имён в символы однозначным, пакетная система позволяет только одному символу быть доступным в данном пакете для каждого имени. То есть, в пакете не могут присутствовать символ и наследованный символ с одним и тем же именем или два разных наследованных символа из двух разных пакетов с одинаковым именем. Однако вы можете разрешить конфликт, делая один из доступных символов *скрывающим* символом, что делает другой символ с таким же именем недоступным. В дополнение к своей таблице имя-в-символ, каждый пакет содержит список скрывающих символов.

Существующий символ может быть *импортирован* в другой пакет через добавление его в пакетную таблицу имя-в-символ. Таким образом, один и тот же символ может присутствовать во многих пакетах. Иногда вы будете импортировать символы просто потому, что вы хотите их сделать доступными в импортирующем пакете без использования их домашнего пакета. В другой раз вы экспортируете символ потому, что только присутствующий символ может быть экспортирован или быть скрывающим символом. Например, если пакету надо использовать два пакета, которые содержат внешние символы с одинаковым именем, один из символов должен быть импортирован в использующий пакет, чтобы быть добавленным в список скрывающих и сделать другой символ недоступным.

Наконец, присутствующий символ может быть сделан *внепакетным*, что ведёт к его удалению из таблицы имя-в-символ, и, если это был скрывающий символ, из списка скрывающих. Вы можете выкинуть символ из пакета для разрешения конфликта между символом и внешним символом пакета, который вы хотите использовать. Символ, который не присутствует ни в одном из пакетов, названный *внепакетным* символом, не может быть более прочтён считывателем и будет печататься, используя `#:foo` синтаксис.

Три стандартных пакета

В следующем разделе я покажу вам как создавать ваши собственные пакеты, включая создание одного пакета с использованием другого и как экспортировать, скрывать и импортировать символы. Но вначале давайте посмотрим на несколько пакетов, которыми вы уже пользовались. Когда вы запускаете Лисп, значением `*PACKAGE*` обычно является пакет `COMMON-LISP-USER`, так же известный как `CL-USER`. `CL-USER` использует пакет `COMMON-LISP`, который

экспортирует все имена из языкового стандарта. Таким образом, когда вы набираете выражение в REPL, все имена стандартных функций, макросов, переменных и так далее, будут преобразованы в символы, экспортированные из COMMON-LISP, и все другие имена будут интернированы в пакет COMMON-LISP-USER. Например имя *PACKAGE* экспортировано из COMMON-LISP – если вы хотите увидеть значение *PACKAGE*, наберите следующее:

```
CL-USER> *package*  
#<The COMMON-LISP-USER package>
```

потому что COMMON-LISP-USER использует COMMON-LISP. Или вы можете задать пакетно-специализированное имя.

```
CL-USER> common-lisp:*package*  
#<The COMMON-LISP-USER package>
```

Вы даже можете использовать CL, псевдоним COMMON-LISP.

```
CL-USER> cl:*package*  
#<The COMMON-LISP-USER package>
```

Однако *X* не является символом из COMMON-LISP, так что если вы наберёте:

```
CL-USER> (defvar *x* 10)  
*X*
```

считыватель прочтёт DEFVAR как символ из пакета COMMON-LISP и *X*, как символ из COMMON-LISP-USER.

REPL не может запускаться в пакете COMMON-LISP, потому что вам не позволено вводить никакие символы в него; COMMON-LISP-USER работает как "черновой" пакет в котором вы можете создавать собственные имена, в то же время имея лёгкий доступ ко всем символам из COMMON-LISP. Обычно, все созданные вами пакеты будут так же использовать COMMON-LISP, так что вы не должны писать нечто вроде:

```
(cl:defun (x) (cl:+ x 2))
```

Третий стандартный пакет - это пакет KEYWORD, которых считыватель Лиспа использует, чтобы хранить имена, начинающиеся с двоеточия. Таким образом, вы так же можете ссылаться на любой ключевой символ, с явным указанием пакета, как здесь:

```
CL-USER> :a  
:A  
CL-USER> keyword:a  
:A  
CL-USER> (eq1 :a keyword:a)  
T
```

Определение собственных пакетов

Работать в COMMON-LISP-USER замечательно для экспериментов в REPL, но как только вы начнёте писать настоящую программу, вы захотите определить новый пакет, чтобы различные программы, загруженные в одну среду Lisp, не топтались на именах друг друга. И когда вы пишете библиотеку, которую намереваетесь использовать в различных контекстах, вы захотите

определить различные пакеты и затем экспортировать символы, которые составляют публичный API библиотеки.

Однако перед тем, как вы начнёте определять пакет, важно понять одну вещь, про то, чем пакеты *не занимаются*. Пакеты не предоставляют прямой контроль за тем, кто может вызвать какую функцию и к какой переменной кому позволено иметь доступ. Они предоставляют вам базовое управление пространством имён через контроль за тем, как считыватель преобразует текстовые имена в символьные объекты, но не за тем, как потом в интерпретаторе этот символ рассматривается как имя функции, переменной или чего-нибудь ещё. Таким образом, бессмысленно говорить про экспорт функции или переменной из пакета. Вы можете экспортировать символ чтобы сделать определённое имя легко доступным, но пакетная система не позволяет вам ограничивать как оно будет использовано.

Запомнив это, вы можете начать рассмотрение того, как определять пакеты и увязывать их друг с другом. Вы определяете новые пакеты через макрос `DEFPACKAGE`, который даёт возможность не только создать пакет, но и определить, какие пакеты он будет использовать, какие символы экспортирует, какие символы импортирует из других пакетов, и разрешить конфликты посредством скрытия символов.

Я буду описывать различные опции в свете того, как вы можете использовать пакеты в написании программы, организующей почтовые сообщения в поисковую базу данных. Программа целиком гипотетическая, так же, как и библиотеки, на которые я буду ссылаться – смысл в том, чтобы взглянуть, как могут быть структурированы пакеты, используемые в такой программе.

Первый пакет, который вам нужен, это тот, который предоставляет пространство имён для приложений – вы захотите именовать ваши функции, переменные и так далее, без заботы о коллизии имён с не относящимся к делу кодом. Так вы определите новый пакет посредством `DEFPACKAGE`.

Если приложение достаточно просто, чтобы обойтись без библиотек сверх средств, предоставляемых самим языком, вы можете определить простой пакет примерно так:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp))
```

Здесь определяется пакет, названный `COM.GIGAMONKEYS.EMAIL-DB`, который наследует все символы, экспортируемые пакетом `COMMON-LISP`.

У вас, на самом деле, есть выбор, как представлять имена пакетов и, как вы увидите, имена символов в `DEFPACKAGE`. Пакеты и символы называются с помощью строк. Однако, в форме `DEFPACKAGE`, вы можете задать имена пакетов и символов через *строковые обозначения*. Строковыми обозначениями являются строки, которые обозначают сами себя; символы, которые обозначают свои имена; или знак, который означает однобуквенную строку, содержащую только этот знак. Использование ключевых символов, как в вышеприведённом `DEFPACKAGE`, является общепризнанным стилем, который позволяет вам писать имена в нижнем регистре – считыватель преобразует для вас имена в верхний регистр. Так же можно записывать `DEFPACKAGE` с помощью строк, но тогда вы должны писать их все в верхнем регистре, потому, что настоящие имена большинства символов и пакетов фактически в верхнем регистре из-за соглашения о преобразовании, которое выполняет считыватель.

```
(defpackage "COM.GIGAMONKEYS.EMAIL-DB"
  (:use "COMMON-LISP"))
```

Вы могли бы также использовать неключевые символы – имена в `DEFPACKAGE` не

интерпретируются – но тогда, при каждом акте считывания формы DEFPACKAGE, эти символы интернировались бы в текущий пакет, что, по меньшей мере, загрязняло бы его пространство имён и могло бы в дальнейшем привести к проблемам при использовании пакета.

Чтобы прочесть код в этом пакете, вы должны сделать его текущим пакетом с помощью макроса IN-PACKAGE:

```
(in-package :com.gigamonkeys.email-db)
```

Если вы напечатаете это выражение в REPL, оно изменит значение *PACKAGE* и повлияет на то, как REPL будет читать последующие выражения до тех пор, пока вы не измените это другим вызовом IN-PACKAGE. Точно также, если вы включите IN-PACKAGE в файл, который загрузите посредством LOAD или скомпилируете посредством COMPILE-FILE, это изменит пакет, влияя на то, как последующие выражения будут читаться из этого файла.

Установив текущим пакетом COM.GIGAMONKEYS.EMAIL-DB, вы можете, кроме имён, унаследованных от пакета COMMON-LISP, использовать любые имена, какие вы хотите, для любых целей. Таким образом, вы можете определить новую функцию hello-world, которая будет сосуществовать с функцией hello-world, ранее определённой в COMMON-LISP-USER. Вот как ведёт себя существующая функция:

```
CL-USER> (hello-world)
hello, world
NIL
```

Теперь можно переключиться в новый пакет с помощью IN-PACKAGE. Заметьте, как изменилось приглашение – точная форма зависит от реализации окружения разработки, но в SLIME приглашение по умолчанию состоит из аббревиатуры имени пакета.

```
CL-USER> (in-package :com.gigamonkeys.email-db)
#<The COM.GIGAMONKEYS.EMAIL-DB package>
EMAIL-DB>
```

Вы можете определить новую hello-world в этом пакете:

```
EMAIL-DB> (defun hello-world () (format t "hello from EMAIL-DB package~%"))
HELLO-WORLD
```

И протестировать её вот так:

```
EMAIL-DB> (hello-world)
hello from EMAIL-DB package
NIL
```

Переключитесь теперь обратно в CL-USER.

```
EMAIL-DB> (in-package :cl-user)
#<The COMMON-LISP-USER package>
CL-USER>
```

Со старой функцией ничего не случилось.


```
CL-USER> (hello-world)
hello, world
NIL
```

Упаковка библиотек для повторного использования

Во время работы над базой данных почтовых сообщений вы можете написать несколько функций, относящихся к сохранению и извлечению текста, но в которых нет ничего конкретного для работы именно с почтой. Вы можете осознать, что эти функции могут быть полезны в других программах и решите перепаковать их в библиотеку. Вы должны будете определить новый пакет, и, в то же время, экспортировать некоторые имена, чтобы сделать их доступными другим пакетам.

```
(defpackage :com.gigamonkeys.text-db
  (:use :common-lisp)
  (:export :open-db
           :save
           :store))
```

Итак, вы используете пакет `COMMON-LISP`, потому что внутри `COM.GIGAMONKEYS.TEXT-DB` вам понадобится доступ к стандартным функциям. Пункт `:export` определяет имена, которые будут внешними в `COM.GIGAMONKEYS.TEXT-DB`, и, таким образом, доступными в пакетах, которые будут `:use` (использовать) его. Следовательно, после определения этого пакета, вы можете изменить определение главного пакета программы на следующее:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db))
```

Теперь код, записанный в `COM.GIGAMONKEYS.EMAIL-DB`, может использовать неспециализированные имена для экспортированных символов из `COMMON-LISP` и `COM.GIGAMONKEYS.TEXT-DB`. Все прочие имена будут продолжать добавляться в пакет `COM.GIGAMONKEYS.EMAIL-DB`.

Импорт отдельных имён

Предположим теперь, что вы нашли стороннюю библиотеку функций для манипуляций с почтовыми сообщениями. Имена, использованные в API библиотеки, экспортированы в пакете `COM.ACME.EMAIL`, так, что вы могли бы сделать `:use` на этот пакет, чтобы получить доступ к этим именам. Однако, предположим, вам нужна только одна функция из этой библиотеки, а другие экспортированные в ней символы конфликтуют с именами, которые вы уже используете (или собираетесь использовать) в вашем собственном коде. В таком случае, вы можете импортировать этот единственный нужный вам символ с помощью пункта `:import-from` в `DEFPACKAGE`. Например, если имя нужной вам функции `parse-email-address`, вы можете изменить `DEFPACKAGE` на такой:

```
(defpackage :com.gigamonkeys.email-db
  (:use :common-lisp :com.gigamonkeys.text-db)
  (:import-from :com.acme.email :parse-email-address))
```

Теперь, где бы имя `parse-email-address` ни появилось в коде, прочитанном из пакета `COM.GIGAMONKEYS.EMAIL-DB`, оно будет прочитано как символ из `COM.ACME.EMAIL`. Если надо импортировать более чем один символ из пакета, можно включить несколько имён после

имени пакета в один пункт `:import-from`. `DEFPACKAGE` также может включать несколько пунктов `:import-from` для импорта символов из разных пакетов.

По воле случая, вы можете попасть и в противоположную ситуацию – пакет экспортирует кучу имён, которые вам нужны, кроме нескольких. Вместо того, чтобы перечислять все символы, которые вам нужны в пункте `:import-from`, лучше сделать `:use` на этот пакет и затем перечислить имена, которые не нужны для наследования в пункте `:shadow`. Предположим, например, что пакет `COM.ACME.TEXT` экспортирует кучу имён функций и классов нужных в обработке текста. Далее положим, что большая часть этих функций и классов нужны вам в вашем коде, но одно имя, `build-index`, конфликтует с уже вами задействованным именем. Можно сделать `build-index` из `COM.ACME.TEXT` недоступным через его сокрытие.

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index))
```

Пункт `:shadow` приведёт к созданию нового символа с именем `BUILD-INDEX` и добавлению его прямо в таблицу имя-символ в `COM.GIGAMONKEYS.EMAIL-DB`. Теперь, если считыватель прочтёт имя `BUILD-INDEX`, он переведёт его в символ из таблицы `COM.GIGAMONKEYS.EMAIL-DB`, вместо того, чтобы, в ином случае, наследовать его из `COM.ACME.TEXT`. Этот новый символ также добавляется в список скрывающих символов, который является частью пакета `COM.GIGAMONKEYS.EMAIL-DB`, так что, если вы позже задействуете другой пакет, который тоже экспортирует символ `BUILD-INDEX`, пакетная система будет знать, что тут нет конфликта, и вы хотите, чтобы символ из `COM.GIGAMONKEYS.EMAIL-DB` использовался вместо любого другого символа с таким же именем, унаследованного из другого пакета.

Похожая ситуация может возникнуть, если вы захотите задействовать два пакета, которые экспортируют одно и то же имя. В этом случае считыватель не будет знать какое унаследованное имя использовать, когда он прочтёт это имя в тексте. В такой ситуации вы должны исправить неоднозначность путём сокрытия конфликтного имени. Если вам не нужно имя ни из одного пакета, вы можете скрыть его с помощью пункта `:shadow`, создав новый символ с таким же именем в вашем пакете. Но если вы всё же хотите использовать один из наследуемых символов, тогда вам надо устранить неоднозначность с помощью пункта `:shadowing-import-from`. Так же, как и пункт `:import-from`, пункт `:shadowing-import-from` состоит из имени пакета за которым следуют имена, импортируемые из этого пакета. Например, если `COM.ACME.TEXT` экспортирует имя `SAVE`, которое конфликтует с именем, экспортированным `COM.GIGAMONKEYS.TEXT-DB`, можно устранить неоднозначность следующим `DEFPACKAGE`:

```
(defpackage :com.gigamonkeys.email-db
  (:use
   :common-lisp
   :com.gigamonkeys.text-db
   :com.acme.text)
  (:import-from :com.acme.email :parse-email-address)
  (:shadow :build-index)
  (:shadowing-import-from :com.gigamonkeys.text-db :save))
```

Пакетная механика

До этого объяснялись основы того, как использовать пакеты для управления пространством имён в некоторых распространённых ситуациях. Однако ещё один уровень использования пакетов, который стоит обсудить – неустоявшиеся механизмы управления с кодом, который использует различные пакеты. В этом разделе я расскажу о некоторых правилах "правой руки", о том, как организовать код, где поместить ваши формы `DEFPACKAGE`, относящиеся к коду, который использует ваши пакеты через `IN-PACKAGE`.

Так как пакеты используются считывателем, пакет должен быть определён до того, как вы сможете сделать `LOAD` на него или сделать `COMPILE-FILE` над файлом, который содержит выражение `IN-PACKAGE`, переключаящее на тот пакет. Пакет также должен быть определен до того, как другие формы `DEFPACKAGE` смогут сослаться на него. Например, если вы собираетесь указать `:use COM.GIGAMONKEYS.TEXT-DB` в `COM.GIGAMONKEYS.EMAIL-DB`, то `DEFPACKAGE` для `COM.GIGAMONKEYS.TEXT-DB` должен быть выполнен раньше, чем `DEFPACKAGE` для `COM.GIGAMONKEYS.EMAIL-DB`.

Лучшим первым шагом для того, чтобы убедиться, что пакеты будут существовать тогда, когда они понадобятся, будет поместить все ваши `DEFPACKAGE` в файлы, отдельно от кода, который должен быть прочитан в тех пакетах. Некоторые парни предпочитают создавать файлы `foo-package.lisp` для каждого пакета в отдельности, другие делают единый файл `packages.lisp`, который содержит все `DEFPACKAGE` формы для группы родственных пакетов. Любой метод разумен, хотя метод "один файл на пакет" также требует, чтобы вы выстроили загрузку файлов в правильном порядке в соответствии с межпакетными зависимостями.

В любом случае, как только все формы `DEFPACKAGE` отделены от кода, который будет в них прочитан, вы должны выстроить `LOAD` файлов, содержащих `DEFPACKAGE`, перед тем, как вы будете компилировать или загружать любые другие файлы. Для простых программ это можно сделать руками: просто `LOAD` на файл или файлы, содержащие формы `DEFPACKAGE`, возможно, сперва компилируя их с помощью `COMPILE-FILE`. Затем `LOAD` на файлы, которые используют те пакеты, также, если надо, сперва компилируя их через `COMPILE-FILE`. Заметьте, однако, что пакет не существует до тех пор, пока вы не сделали `LOAD` его определения в виде исходного текста или скомпилированной версии, созданной `COMPILE-FILE`. Таким образом, если вы компилируете всё, вы должны по-прежнему делать `LOAD` определениям пакетов, перед тем, как вы сможете сделать `COMPILE-FILE` какому-нибудь файлу, читающемуся в тех пакетах.

Продельвание всех этих операций руками со временем утомляет. Для простых программ можно автоматизировать все шаги с помощью файла `load.lisp`, который будет содержать подходящие вызовы `LOAD` и `COMPILE-FILE` в нужном порядке. Затем можно просто сделать `LOAD` этому файлу. Для более сложных программ вы захотите использовать средство *системных определений* для управления загрузкой и компиляцией файлов в правильном порядке.

Ещё одно ключевое правило "правой руки", это то, что каждый файл должен содержать только одну форму `IN-PACKAGE`, и это должна быть первая форма в файле, отличная от комментариев. Файлы, содержащие формы `DEFPACKAGE`, должны начинаться с `(in-package "COMMON-LISP-USER")`, и все другие файлы должны содержать `IN-PACKAGE` для одного из ваших пакетов.

Если вы нарушите это правило и переключите пакет в середине файла, человек, читающий файл, будет в растерянности, если он не заметит где случился второй `IN-PACKAGE`. Также многие среды разработки Лисп, в частности такая, как `SLIME`, основанная на `Emacs`, ищут `IN-PACKAGE`, чтобы определить пакет, который им надо использовать для общения с `Common Lisp`. Множественные формы `IN-PACKAGE` в одном файле приводят в растерянность такие инструменты.

С другой стороны, всё хорошо, если есть несколько файлов, читающихся в одном и том же

пакете, каждый с одинаковой формой `IN-PACKAGE`. Это просто вопрос того, как вам следует организовывать свой код.

Другая часть пакетной механики имеет дело с тем, как именовать пакеты. Пакетные имена живут в плоском пространстве имён — имена пакетов это просто строки и различные пакеты должны иметь текстуально отличные имена. Таким образом вам надо учитывать возможность конфликта между именами пакетов. Если вы используете пакеты, которые сами же и разрабатываете, то возможно и обойдётесь короткими именами для своих пакетов. Однако, если вы планируете использовать библиотеки третьих лиц или публиковать свой код для использования другими программистами, вам надо следовать соглашениям для имён, которые минимизируют возможность коллизии имён для различных пакетов. Многие Lisp-программисты в наше время взяли на вооружение Java-стиль в именах, наподобие того, что вы видели в этой главе, состоящие из обращённых доменных имён Интернета, с последующей точкой и строкой описания.

Пакетные ловушки

Как только вы освоитесь с созданием пакетов, вы больше не будете тратить время на размышления про них. В них нет ничего такого. Однако пара ловушек, которые часто подстерегают новичков в Lisp, заставляют пакетную систему казаться сложнее и недружественнее, чем она есть на самом деле.

Первая ловушка чаще всего проявляется во время работы с REPL. Вы будете искать какую-нибудь библиотеку, определяющую некоторые нужные функции. Вы попытаетесь вызвать одну из них так:

```
CL-USER> (foo)
```

и вас отбросит в отладчик с ошибкой:

```
attempt to call `FOO' which is an undefined function.
[Condition of type UNDEFINED-FUNCTION]
Restarts:
 0: [TRY-AGAIN] Try calling FOO again.
 1: [RETURN-VALUE] Return a value instead of calling FOO.
 2: [USE-VALUE] Try calling a function other than FOO.
 3: [STORE-VALUE] Setf the symbol-function of FOO and call it again.
 4: [ABORT] Abort handling SLIME request.
 5: [ABORT] Abort entirely from this (lisp) process.
```

```
(попытка вызвать `FOO', которая является неопределённой функцией
[Случай типа UNDEFINED-FUNCTION]
Перезапуск:
 0: [TRY-AGAIN] Попытаться вызвать FOO снова.
 1: [RETURN-VALUE] Возвратить значение вместо вызова FOO.
 2: [USE-VALUE] Попытаться вызвать функцию, другую чем FOO.
 3: [STORE-VALUE] Setf символ-функцию FOO и вызвать снова.
 4: [ABORT] Прервать обработку SLIME запроса.
 5: [ABORT] Прервать полностью этот (Лисп) процесс.
)
```

Ну конечно — вы забыли использовать пакет библиотеки. Итак, вы выходите из отладчика и пытаетесь сделать `USE-PACKAGE` на библиотечный пакет в надежде получить доступ к имени `FOO`, чтобы можно было вызвать эту функцию.

```
CL-USER> (use-package :foolib)
```

Но это снова приводит вас к попаданию в отладчик с сообщением об ошибке:

```
Using package `FOOLIB' results in name conflicts for these symbols: FOO
[Condition of type PACKAGE-ERROR]
Restarts:
  0: [CONTINUE] Unintern the conflicting symbols from the `COMMON-LISP-USER'
package.
  1: [ABORT] Abort handling SLIME request.
  2: [ABORT] Abort entirely from this (lisp) process.
```

```
Использование пакета `FOOLIB' приводит к конфликту имён для этих символов: FOO
[ Условие типа PACKAGE-ERROR]
Перезапуск:
  0: [CONTINUE] Вывести конфликтующие символы из пакета `COMMON-LISP-USER'.
  1: [ABORT] Прервать обработку SLIME запроса.
  2: [ABORT] Прервать полностью этот (Лисп) процесс.
```

Что такое? Проблема в том, что в первую попытку вызвать `foo`, считыватель прочёл имя `foo` интернировал его в `CL-USER` перед тем, как интерпретатор получил управление и обнаружил, что только что введённое имя не является именем функции. Этот новый символ затем и законфликтовал с таким же именем, экспортированным из пакета `FOOLIB`. Если бы вы вспомнили о `USE-PACKAGE FOOLIB` перед тем, как попытались вызвать `foo`, считыватель читал бы `foo` как унаследованный символ и не вводил бы `foo` символ в `CL-USER`.

Однако, ещё не всё потеряно, потому, что первый же первый же перезапуск, предлагаемый отладчиком, исправит ситуацию правильным образом: он выведет символ `foo` из `COMMON-LISP-USER`, возвращая пакет `CL-USER` обратно в состояние, в котором он был до вашего вызова `foo`, позволит `USE-PACKAGE` сделать своё дело и дать возможность унаследованной `foo` стать доступной в `CL-USER`.

Такого рода проблемы могут возникать, когда загружаются и компилируются файлы. Например, если вы определили пакет `MY-APP` для кода, предназначенного для использования функций с именами из пакета `FOOLIB`, но забыли сделать `:use FOOLIB`, когда компилируете файлы с (`in-package :my-app`) внутри, считыватель введёт новые символы в `MY-APP` для имён, которые предполагались быть прочитанными как символы из `FOOLIB`. Когда вы попытаетесь запустить скомпилированный код, вы получите ошибки о неопределённых функциях. Если вы затем попытаетесь переопределить пакет `MY-APP`, добавив `:use FOOLIB`, то получите ошибку конфликта символов. Решение то же самое: выберите перезапуск с выводом конфликтующих символов из `MY-APP`. Затем вам надо будет перекомпилировать код в пакете `MY-APP`, и он будет ссылаться на унаследованные имена.

Очередная ловушка представляет собой предыдущую наоборот. В её случае у вас есть определённый пакет – назовём его, снова, `MY-APP` – который использует другой пакет, скажем, `FOOLIB`. Теперь вы начинаете писать код в пакете `MY-APP`. Хотя вы использовали `FOOLIB`, чтобы иметь возможность ссылаться на функцию `foo`, `FOOLIB` может также экспортировать и другие символы. Если вы используете один из таких символов – скажем, `bar` – как имя функции в вашем собственном коде, Lisp не станет возмущаться. Вместо этого, имя вашей функции будет символом, экспортированным из `FOOLIB`, и функция перекроет предыдущее определение `bar` из `FOOLIB`.

Эта ловушка гораздо более коварна, потому что она не вызывает появление ошибки – с точки зрения интерпретатора это просто запрос на ассоциацию новой функции со старым именем,

нечто вполне законное. Это подозрительно только потому, что код, делающий переопределение, был прочитан со значением `*PACKAGE*`, отличным от пакета данного имени. Но интерпретатору не обязательно знать об этом. Однако, во большинстве Лиспов, вы получите предупреждение про "переопределение `VAR`, сначала определённом в ?". Надо быть внимательным к таким предупреждениям. Если вы перекрыли определение из библиотеки, можете восстановить его, перезагрузив код библиотеки через `LOAD`.

Последняя, относящаяся к пакетам ловушка, относительно тривиальна, но на неё попадают большинство Lisp-программистов как минимум несколько раз: вы определяете пакет, который использует `COMMON-LISP` и, возможно, несколько библиотек. Затем в `REPL` вы переходите в этот пакет чтобы поиграться. После этого вы решили покинуть Lisp и пробуете вызвать (`quit`). Однако `quit` не имя из пакета `COMMON-LISP` – оно определено в зависимости от реализации в некотором определяемом реализацией пакете, который оказывается используется пакетом `COMMON-LISP-USER`. Решение просто – смените пакет обратно на `CL-USER` для выхода. Или используйте `SLIME REPL` сокращение для выхода, что, к тому же, убережёт вас от необходимости помнить, что в некоторых реализациях Common Lisp функцией для выхода является `exit`, а не `quit`.

Вы почти закончили свой тур по Common Lisp. В следующей главе я расскажу о деталях расширенного макроса `LOOP`. После этого, остаток книги посвящён "практикам": спам-фильтру, библиотеке для разбора двоичных файлов, и различным частям потокового MP3 сервера с Веб интерфейсом.

22. LOOP для мастеров с черным поясом.

В главе 7 я кратко описал расширенный макрос **LOOP**. Как я упоминал тогда, **LOOP** по существу предоставляет язык специального назначения для написания конструкций итерирования.

Это может показаться весьма хлопотным: изобретение целого языка лишь для написания циклов. Но, если вы задумаетесь о способах использования циклов в программах, эта идея действительно станет обретать смысл. Любая программа любого размера всегда будет содержать циклы. И хотя все они не будут одинаковыми, они также не будут и совершенно различными; при детальном рассмотрении будут выделены образцы (в частности, если включить в них код непосредственно предшествующий и следующий за циклами): образцы инициализации перед циклом, образцы действий внутри цикла и образцы действий после завершения цикла. Язык **LOOP** фиксирует эти образцы, так что вы можете выражать их явно.

Макрос **LOOP** имеет множество частей: одной из главных претензий противников **LOOP** является то, что он слишком сложен. В этой главе я покажу, что это не так, дав вам систематическое описание различных частей **LOOP** и того, как эти части использовать вместе.

Части LOOP

Вы можете делать в **LOOP** следующее:

- * Итерировать переменную численно или по различным структурам данных.
- * Сбирать, подсчитывать, суммировать, искать максимальное и минимальное значения по данным, просматриваемым во время цикла.
- * Выполнять произвольные выражения Lisp.
- * Решать, когда остановить цикл.
- * Осуществлять определенные действия при заданных условиях.

Вдобавок, **LOOP** предоставляет синтакс для следующего:

- * Создание локальных переменных для использования внутри цикла.
- * Задание произвольных выражений Lisp для выполнения перед и после цикла.

Базовой структурой **LOOP** является набор предложений (*clauses*), каждое из которых начинается с ключевого слова *loop*. То, как каждое предложение анализируется макросом **LOOP**, зависит от такого ключевого слова. Некоторые из главных ключевых слов, которые вы видели в главе 7, следующие: *for*, *collecting*, *summing*, *counting*, *do* и *finally*.

Управление итерированием

Большинство из так называемых предложений управления итерированием начинаются с ключевого слова *loop for* или его синонима *as*, за которыми следует имя переменной. Что следует за именем переменной, зависит от типа предложения *for*.

Подвыражения (*subclauses*) предложений *for* могут итерировать по следующему:

- * Численные интервалы, вверх или вниз.
- * Отдельные элементы списка.
- * *cons*-ячейки, составляющие список.
- * Элементы вектора, включая подтипы, такие как строки и битовые векторы.
- * Пары хэш-таблицы.
- * Символы пакета.
- * Результаты повторных вычислений заданной формы.

Одиночный цикл может содержать несколько предложений `for`, каждое из которых именует собственную переменную. Если цикл содержит несколько предложений `for`, он завершается как только любое из них достигает своего условия завершения. Например, следующий цикл:

```
(loop
  for item in list
  for i from 1 to 10
  do (something))
```

выполнится максимум 10 раз, но может завершиться и раньше, если список содержит менее десяти элементов.

Подсчитывающие циклы (Counting Loops)

Предложения арифметического итерирования управляют числом раз, которое будет выполнено тело цикла, путем изменения переменной в пределах интервала чисел, выполняя тело на каждом шаге. Такие предложения состоят из от одного до трех следующих *предложных оборотов* (*prepositional phrases*), идущих после `for` (или `as`): оборот *откуда* (*from where*), оборот *докуда* (*to where*) и оборот *по сколько* (*by how much*).

Оборот *откуда* задает начальное значение для переменной предложения. Он состоит из одного из предлогов (prepositions) `from`, `downfrom` или `upfrom`, за которыми следует форма, предоставляющая начальное значение (число).

Оборот *докуда* задает точку останова цикла и состоит из одного из предлогов `to`, `upto`, `below`, `downto` или `above`, за которыми следует форма, предоставляющая точку останова. С `upto` и `downto` цикл завершится (без выполнения тела) когда переменная перейдет точку останова; с `below` и `above` он завершится на итерацию ранее.

Оборот *по сколько* состоит из предлога `by` и формы, которая должна вычисляться в положительное число. Переменная будет изменяться (вверх или вниз, что определяется другими оборотами) на эту величину на каждой итерации, или на единицу, если оборот опущен.

Вы должны задать по меньшей мере один из этих предложных оборотов. По умолчанию цикл начинается с нуля, переменная на каждой итерации увеличивается на единицу, и цикл продолжается вечно, или, более точно, пока другое предложение не остановит цикл. Вы можете изменить любое из этих умолчаний путем добавления соответствующего предложного оборота. Единственным неудобством является то, что если вы хотите декрементный цикл, не существует значения *откуда* по умолчанию, поэтому вы должны явно указать его с помощью `from` или `downfrom`. Таким образом, следующее:

```
(loop for i upto 10 collect i)
```

накапливает первые одиннадцать целых чисел (с нуля до десяти), но поведение этого:

```
(loop for i downto -10 collect i) ; неверно
```

не определено. Вместо этого вам нужно написать так:

```
(loop for i from 0 downto -10 collect i)
```

Также заметьте что, так как **LOOP** является макросом, который запускается во время компиляции, то он может определить направление изменения переменной только по предлогам,

но не по значениям форм, которые не могут быть известны до времени выполнения. Поэтому следующее:

```
(loop for i from 10 to 20 ...)
```

работает хорошо, используя значение приращения по умолчанию. Но это:

```
(loop for i from 20 to 10 ...)
```

не знает, что нужно считать от двадцати до десяти. Хуже того, это выражение не выдаст вам никакой ошибки: оно просто не выполнит цикл, так как *i* уже больше десяти. Вместо этого вы должны написать так:

```
(loop for i from 20 downto 10 ...)
```

или так:

```
(loop for i downfrom 20 to 10 ...)
```

Наконец, если вам просто нужен цикл, повторяющийся определенное число раз, вы можете заменить предложение следующей формы:

```
for i from 1 to number-form
```

на предложение `repeat` следующего вида:

```
repeat number-form
```

Эти предложения идентичны по своему действию, за исключением того, что предложение `repeat` не создает явной переменной цикла.

Организация циклов по коллекциям и пакетам

Предложения `for` для итерирования по спискам гораздо проще, чем арифметические предложения. Они поддерживают только два предложных оборота: `in` и `on`.

Оборот такой формы:

```
for var in list-form
```

итерирует переменную по всем элементам списка, являющегося результатом вычисления *list-form*.

```
(loop for i in (list 10 20 30 40) collect i) ==> (10 20 30 40)
```

Иногда это предложение дополняется оборотом `by`, который задает функцию для продвижения по списку. Значением по умолчанию является **CDR**, но можно использовать любую функцию, принимающую список и возвращающую подсписок. Например, вы можете накапливать каждый второй элемент списка с помощью `loop` следующим образом:

```
(loop for i in (list 10 20 30 40) by #'cddr collect i) ==> (10 30)
```

Предложный оборот `on` используется для итерирования по cons-ячейкам, составляющим список.

```
(loop for x on (list 10 20 30) collect x) ==> ((10 20 30) (20 30) (30))
```

Этот оборот также принимает предлог `by`:

```
(loop for x on (list 10 20 30 40) by #'cddr collect x) ==> ((10 20 30 40) (30 40))
```

Итерирование по элементам вектора (что включает строки и битовые векторы) подобно итерированию по элементам списка, за исключением использования предлога `across` вместо `in`. Например:

```
(loop for x across "abcd" collect x) ==> (#\a #\b #\c #\d)
```

Итерирование по хэш-таблице или пакету немного более сложно, так как хэш-таблицы и пакеты содержат различные множества значений, по которым вы можете захотеть итерировать: ключи или значения в хэш-таблице или различные виды символов в пакете. Оба вида итерирования следуют сходному образцу, который в базовом виде выглядит так:

```
(loop for var being the things in hash-or-package ...)
```

Для хэш-таблиц возможными значениями для *things* являются `hash-keys` и `hash-values`, означающие, что `var` будет связываться с последовательными значениями ключей или самими значениями хэш-таблицы, соответственно. Форма *hash-or-package* вычисляется лишь один раз для получения значения, которое должно быть хэш-таблицей.

Для итерирования по пакету *things* может быть `symbols`, `present-symbols` и `external-symbols`, и `var` будет связываться с каждым символом, доступным в пакете, каждым символом, присутствующим в пакете (другими словами, интернированным или импортированным в этот пакет), или с каждым символом, экспортированным из пакета, соответственно. Форма *hash-or-package* вычисляется для предоставления имени пакета, который будет искаться как с помощью **FIND-PACKAGE**, или объекта пакета. Для частей предложения `for` также доступны синонимы. На месте *the* вы можете использовать `each`, вместо `in – of`, а также *things* можно записывать в единственном числе (например, `hash-key` или `symbol`).

И наконец, так как часто при итерировании по хэш-таблицам нужны и ключи, и сами значения, предложения для хэш-таблиц поддерживают использование специального `using`-подпредложения.

```
(loop for k being the hash-keys in h using (hash-value v) ...)
(loop for v being the hash-values in h using (hash-key k) ...)
```

Оба этих цикла будут связывать `k` с каждым ключем в хэш-таблице, а `v` – с соответствующим значением. Обратите внимание, что первый элемент `using`-подпредложения должен быть записан в единственном числе.

Equals-Then итерирование

Если ни одно из остальных предложений `for` не предоставляет именно ту форму итерирования переменной, которая вам нужна, вы можете получить полный контроль над итерированием, используя предложение *equals-then*. Это предложение подобно связывающим предложениям (binding clauses) в циклах **DO**, преведенных к более Algol-подобному синтаксису. Образец

использования следующий:

```
(loop for var = initial-value-form [ then step-form ] ...)
```

Как обычно, *var* – имя итерируемой переменной. Ее начальное значение получается путем однократного вычисления *initial-value-form* перед первой итерацией. На каждой последующей итерации вычисляется *step-form* и ее значение становится новым значением *var*. В отсутствие *then*-части предложения *initial-value-form* перевычисляется на каждой итерации для предоставления нового значения. Заметьте, что это отличается от связывающего предложения **DO** без *step*-формы.

step-form может ссылаться на другие переменные *loop*, включая переменные, созданные другими предложениями *for* цикла *loop*. Например:

```
(loop repeat 5
  for x = 0 then y
  for y = 1 then (+ x y)
  collect y) ==> (1 2 4 8 16)
```

Заметьте, однако, что каждое предложение *for* вычисляется отдельно в порядке своего появления. Поэтому в предыдущем цикле на второй итерации *x* устанавливается в значение *y* до того, как *y* изменится (другими словами, в 1). Но *y* затем устанавливается в значение суммы своего старого значения (все еще 1) и нового значения *x*. Если порядок предложений *for* изменить, результат изменится.

```
(loop repeat 5
  for y = 1 then (+ x y)
  for x = 0 then y
  collect y) ==> (1 1 2 4 8)
```

Часто, однако, вам нужно, чтобы *step*-формы для нескольких переменных были вычислены перед тем, как любая из этих переменных получит свое новое значение (подобно тому как это происходит в **DO**). В этом случае вы можете объединить несколько предложений *for*, заменив все кроме первого *for* на *and*. Вы уже видели такую запись в **LOOP**-версии вычисления чисел Фибоначчи в главе 7. Вот другой вариант, основанный на двух предыдущих примерах:

```
(loop repeat 5
  for x = 0 then y
  and y = 1 then (+ x y)
  collect y) ==> (1 1 2 3 5)
```

Локальные переменные

В то время как главные переменные, необходимые внутри цикла, обычно явно объявляются в предложениях *for*, иногда вам понадобятся вспомогательные переменные, которые вы можете объявить с помощью предложений *with*.

```
with var [ = value-form ]
```

Имя *var* станет именем локальной переменной, которая перестанет существовать после завершения цикла. Если предложение *with* содержит часть *= value-form*, то перед первой итерацией цикла переменная будет проинициализирована значением *value-form*.

В `loop` может быть несколько предложений `with`; каждое предложение вычисляется независимо в порядке их появления, и значение присваивается перед началом обработки следующего предложения, что позволяет последующим переменным зависеть от значения уже объявленных переменных. Взаимно независимые переменные могут быть объявлены в одном предложении `with` с использованием `and` между такими декларациями.

Деструктурирование переменных

Очень удобной возможностью **LOOP**, о которой я ранее не упоминал, является возможность деструктурирования списковых значений, присваиваемых переменным цикла. Это позволяет разбирать на части значение списков, которые иначе присваивались бы переменной цикла, подобно тому, как работает **DESTRUCTURING-BIND**, но немного более простым способом. В общем, вы можете заменить любую переменную цикла в предложениях `for` или `with` деревом символов, и списковое значение, которое было бы присвоено простой переменной, будет деструктурировано на переменные, именованные символами дерева. Простой пример выглядит следующим образом:

```
CL-USER> (loop for (a b) in '((1 2) (3 4) (5 6))
           do (format t "a: ~a; b: ~a~%" a b))
a: 1; b: 2
a: 3; b: 4
a: 5; b: 6
NIL
```

Такое дерево также может включать в себя точечные пары. В этом случае имя после точки работает как **&rest** параметр: с ним будет связан список, содержащий все оставшиеся элементы списка. Это особенно полезно с `for/on` циклом, так как значением всегда является список. Например, этот **LOOP** (который я использовал в главе 18 для вывода элементов списка, разделенных запятыми):

```
(loop for cons on list
      do (format t "~a" (car cons))
      when (cdr cons) do (format t ", "))
```

может также быть записан следующим образом:

```
(loop for (item . rest) on list
      do (format t "~a" item)
      when rest do (format t ", "))
```

Если вы хотите игнорировать значение деструктурированного списка, вы можете использовать **NIL** на месте имени переменной.

```
(loop for (a nil) in '((1 2) (3 4) (5 6)) collect a) ==> (1 3 5)
```

Если список деструктурирования содержит больше переменных, чем значений в списке, лишние переменные получают значение **NIL**, что делает переменные по существу похожими на **&optional** параметры. Не существует, однако, эквивалента **&key** параметрам.

Накопление значения

Предложения накопления значения вероятно являются наиболее мощной частью **LOOP**. Хотя предложения управления итерированием предоставляют лаконичный синтаксис для выражения

базовых механизмов итерирования, они не отличаются разительно от подобных механизмов, предоставляемых **DO**, **DOLIST** и **DOTIMES**.

С другой стороны, операторы накопления значения предоставляют возможность лаконичной записи общих идиом накопления во время итерирования. Каждое предложение накопления начинается с глагола и следует следующему образцу:

```
verb form [ into var ]
```

Каждый раз, при прохождении цикла, предложение накопления вычисляет *form* и сохраняет значение способом, определяемым глаголом *verb*. С подпредложением *into* значение сохраняется в переменную под именем *var*. Переменная является локальной в цикле, как если бы она была объявлена в предложении *with*. Без подпредложения *into* предложение накопления накапливает значения в переменную по умолчанию для всего выражения цикла.

Возможными глаголами являются *collect*, *append*, *nconc*, *count*, *sum*, *maximize* и *minimize*. Также доступны синонимы в форме причастий настоящего времени: *collecting*, *appending*, *nconcing*, *counting*, *summing*, *maximizing* и *minimizing*.

Предложение *collect* строит список, содержащий все значения *form* в порядке их просмотра. Эта конструкция особенно полезна, так как код, который вы бы написали для накопления списка, равный по эффективности сгенерированному **LOOP** коду, будет гораздо более сложным, чем вы обычно пишете вручную. Родственными *collect* являются глаголы *append* и *nconc*. Эти глаголы также накапливают значения в список, но они объединяют значения, которые должны быть списками, в единый список как с помощью функций **APPEND** и **NCONC**.

Остальные предложения накопления значения используются для накопления численных значений. Глагол *count* подсчитывает число раз, которое форма *form* была истинна, *sum* подсчитывает сумму значений, которые принимала форма *form*, *maximize* подсчитывает максимальное из этих значения, а *minimize* — минимальное. Представим, например, что вы определили переменную **random**, содержащую список случайных чисел.

```
(defparameter *random* (loop repeat 100 collect (random 10000)))
```

Следующий цикл вернет список, содержащий различную сводную информацию о числах из **random**:

```
(loop for i in *random*
      counting (evenp i) into evens
      counting (oddp i) into odds
      summing i into total
      maximizing i into max
      minimizing i into min
      finally (return (list min max total evens odds)))
```

Безусловное выполнение

Хоть и удобная для конструкций накопления значения, **LOOP** не была бы очень хорошим средством итерации общего назначения, если бы не предоставляла способа выполнения произвольного кода в теле цикла.

Самым простым способом выполнения произвольного кода внутри тела цикла является использование предложения *do*. По сравнению с вышеописанными предложениями со всеми их предлогами и подвыражениями, *do* следует модели простоты по Йоде. Предложение *do* состоит

из слова `do` (или `doing`), за которым следует одна или более форм Lisp, которые вычисляются при вычислении предложения `do`. Предложение `do` заканчивается закрывающей скобкой цикла `loop` или следующим ключевым словом `loop`.

Например, для печати чисел от одного до десяти, вы можете записать следующее:

```
(loop for i from 1 to 10 do (print i))
```

Еще одной формой непосредственного выполнения является предложение `return`. Это предложение состоит из слова `return`, за которым следует одна форма Lisp, которая вычисляется, а результат немедленно возвращается как значение цикла `loop`.

Вы также можете прервать цикл из предложения `do` путем использования любого обычного оператора управления потоком вычислений Lisp, таких как **RETURN** и **RETURN-FROM**. Обратите внимание, что предложение `return` всегда возвращает управление из непосредственно охватывающего выражения **LOOP**, в то время как с помощью **RETURN** и **RETURN-FROM** в предложении `do` можно вернуть управление из любого охватывающего выражения. Например, сравните следующее:

```
(block outer
  (loop for i from 0 return 100) ; 100 возвращается из LOOP
  (print "This will print")
  200) ==> 200
```

С ЭТИМ:

```
(block outer
  (loop for i from 0 do (return-from outer 100)) ; 100 возвращается из BLOCK
  (print "This won't print")
  200) ==> 100
```

Предложения `do` и `return` вместе называются предложениями безусловного выполнения.

Условное выполнение

Так как предложение `do` может содержать произвольные формы Lisp, вы можете использовать любые выражения Lisp, включая конструкции управления, такие как **IF** и **WHEN**. Таким образом, следующее является одним из способов написания цикла, печатающего только четные числа от одного до десяти:

```
(loop for i from 1 to 10 do (when (evenp i) (print i)))
```

Однако иногда вам понадобится условное управление на уровне предложений цикла `loop`. Например, представим, что вам нужно просуммировать только четные числа от одного до десяти путем использования предложения `summing`. Вы не сможете написать такой цикл с помощью предложения `do`, так как не существует способа "вызвать" `sum i` в середине обычной формы Lisp. В случаях, подобных этому, вам нужно использовать одно из собственных условных выражений **LOOP**:

```
(loop for i from 1 to 10 when (evenp i) sum i) ==> 30
```

LOOP предоставляет три условных конструкции, и все они следуют этому базовому образцу:

Условный оператор *conditional* может быть *if*, *when* или *unless*. *test-form* — это любая обычная форма Lisp, а предложение *loop-clause* может быть предложением накопления значения (*count*, *collect* и так далее), предложением безусловного выполнения или другим предложением условного выполнения. Несколько предложений цикла могут быть объединены в одну условную конструкцию путем соединения их с помощью *and*.

Несколько условных предложений могут быть объединены в одно условное, путем соединения их с помощью *and*.

Дополнительным синтаксическим сахаром является возможность использования в первом предложении *loop* после формы условия переменной *it* для ссылки на значение, возвращенное этой формой условия. Например, следующий цикл накапливает не равные **NIL** значения, найденные в *some-hash* по ключам из *some-list*:

```
(loop for key in some-list when (gethash key some-hash) collect it)
```

Условное выражение выполняется при каждой итерации цикла. Предложения *if* и *when* выполняют свои предложения *loop* если форма *test-form* вычисляется в истину. *unless* же выполняет предложения только если *test-form* вычисляется в **NIL**. В отличие от так же названных операторов Common Lisp, *if* и *when* в **LOOP** являются синонимами — в их поведении нет никакой разницы.

Все три условных предложения могут также принимать ветвь *else*, в которой за *else* следует другое предложение *loop* либо несколько предложений, объединенных *and*. Если условные предложения являются вложенными, множество предложений, связанных с внутренним условным предложением, может быть завершено с помощью слова *end*. *end* является необязательным, если оно не нужно для разрешения неоднозначности с вложенными условными предложениями: конец условного предложения будет определен по концу цикла либо по началу другого предложения, не присоединенного с помощью *and*.

Следующий довольно глупый цикл демонстрирует различные формы условных предложений **LOOP**. Функция *update-analysis* будет вызываться на каждой итерации цикла с последними значениями различных переменных, накапливаемых предложениями внутри условных предложений.

```

(loop for i from 1 to 100
  if (evenp i)
    minimize i into min-even and
    maximize i into max-even and
    unless (zerop (mod i 4))
      sum i into even-not-fours-total
    end
    and sum i into even-total
  else
    minimize i into min-odd and
    maximize i into max-odd and
    when (zerop (mod i 5))
      sum i into fives-total
    end
    and sum i into odd-total
  do (update-analysis min-even
      max-even
      min-odd
      max-odd
      even-total
      odd-total
      fives-total
      even-not-fours-total))

```

Начальные установки и подытоживание

Одним из ключевых озарений проектировщиков языка **LOOP** было осознание того, что циклы часто предваряются некоторым кодом, занимающимся начальной установкой каких-то вещей, и завершаются кодом, осуществляющим что-то со значениями, вычисленными в цикле. Простой пример на Perl мог бы выглядеть так:

```

my $evens_sum = 0;
my $odds_sum = 0;
foreach my $i (@list_of_numbers) {
  if ($i % 2) {
    $odds_sum += $i;
  } else {
    $evens_sum += $i;
  }
}
if ($evens_sum > $odds_sum) {
  print "Sum of evens greater\n";
} else {
  print "Sum of odds greater\n";
}

```

Циклической сущностью в этом коде является инструкция `foreach`. Но сам цикл `foreach` не является независимым: код в теле цикла ссылается на переменные, объявленные в двух строках перед циклом. А работа, осуществляемая циклом является абсолютно бесполезной без инструкции `if` после цикла, которая фактически сообщает о результате. В Common Lisp, к тому же, конструкция **LOOP** является выражением, возвращающим значение, и поэтому потребность в осуществлении чего-либо, а именно генерации возвращаемого значения, даже больше.

Поэтому проектировщики **LOOP** предоставили возможность включения такого, на самом деле являющегося частью цикла, кода в сам цикл. Для этого **LOOP** предоставляет два ключевых слова, `initially` и `finally`, которые вводят код для запуска снаружи главного тела цикла.

После слов `initially` или `finally` эти предложения включают все формы Lisp до начала следующего предложения цикла либо до его конца. Все формы `initially` комбинируются в единую *вводную часть* (*prologue*), которая запускается однократно непосредственно после инициализации всех локальных переменных цикла и перед его телом. Формы `finally` схожим образом комбинируются в *заключительную часть* (*epilogue*) и выполняются после последней итерации цикла. И вводная, и заключительная части могут ссылаться на локальные переменные цикла.

Вводная часть всегда запускается, даже если тело цикла не выполняется ни разу. В то же время цикл может вернуть значение без выполнения заключительной части в одном из следующих случаев:

- * Выполнение предложения `return`.
- * **RETURN**, **RETURN-FROM** или другая конструкция передачи управления была вызвана из формы Lisp, находящейся в теле цикла.
- * Цикл завершается по предложению `always`, `never` или `thereis`, которые я обсужу в следующей секции.

Внутри кода заключительной части для явного предоставления возвращаемого циклом значения могут использоваться **RETURN** или **RETURN-FROM**. Это явно возвращаемое значение имеет приоритет над любым значением, которое может иначе предоставляться предложениями накопления или критерия остановки.

Для возможности использования **RETURN-FROM** для возврата из указываемого цикла (полезно при вложенных выражениях **LOOP**) вы можете дать **LOOP** имя с помощью ключевого слова `loop named`. Если предложение `named` используется в цикле, оно должно идти первым. В качестве простого примера предположим что у вас есть список списков и вы хотите найти в одном из вложенных списков элемент, который удовлетворяет некоторому критерию. Вы можете найти его с помощью пары вложенных циклов подобным образом:

```
(loop named outer for list in lists do
  (loop for item in list do
    (if (what-i-am-looking-for-p item)
        (return-from outer item))))
```

Критерии завершения

Хотя предложения `for` и `repeat` предоставляют базовую инфраструктуру для управления числом итераций, иногда вам понадобится прервать цикл до его завершения. Вы уже видели, как с помощью предложения `return` или операторов **RETURN** и **RETURN-FROM** внутри предложения `do` можно немедленно прервать цикл; но как есть общие образцы для накопления значений, так существуют и общие образцы для принятия решений, когда останавливать цикл. Такие образцы поддерживаются в **LOOP** с помощью предложений завершения `while`, `until`, `always`, `never` и `thereis`. Все они следуют одинаковому образцу:

```
loop-keyword test-form
```

Все эти предложения вычисляют форму *test-form* на каждой итерации и на основе возвращаемого ей значения принимают решение, завершить ли выполнение цикла. Они отличаются в том, что происходит при завершении ими цикла (если он завершается), и как они определяют необходимость такого завершения.

Ключевые слова `loop while` и `until` предоставляют "мягкие" предложения завершения. Если они решают завершить цикл, управление передается в заключительную часть, пропуская

оставшуюся часть тела цикла. Затем заключительная часть может вернуть значение или сделать еще что-либо для завершения цикла. Предложение `while` останавливает цикл как только контрольная форма *test-form* вычисляется в ложное значение, а `until`, наоборот, - как только в истинное.

Другая форма мягкого завершения предоставляется макросом **LOOP-FINISH**. Это обычная форма Lisp, не предложение `loop`, поэтому она может использоваться в любом месте внутри форм Lisp предложения `do`. **LOOP-FINISH** также приводит к немедленному переходу к заключительной части, и может быть полезен, когда решение о прерывании цикла не может быть легко уместено в единственную форму, могущую использоваться в предложениях `while` или `until`.

Остальные три предложения, `always`, `never` и `thereis`, останавливают цикл гораздо более жестко: они приводят к немедленному возврату из цикла, пропуская не только все последующие предложения `loop`, но и заключительную часть. Они также предоставляют значение по умолчанию даже если не приводят к завершению цикла. Однако, если цикл не завершается ни по одному из этих критериев, заключительная часть запускается и может вернуть значение, отличное от значения по умолчанию, предоставляемого предложениями завершения.

Так как эти предложения предоставляют свои собственные возвращаемые значения, они не могут комбинироваться с предложениями накопления за исключением содержащих подвыражение `into`. Иначе компилятор (или интерпретатор) должен просигнализировать ошибку во время выполнения. Предложения `always` и `never` возвращают только булевы значения, поэтому они наиболее полезны в случае, если вам нужно использовать выражение цикла как часть предиката. Вы можете использовать `always` для проверки того, что контрольная форма вычисляется в истинное значение на каждой итерации цикла. И наоборот, `never` проверяет, что контрольная форма на каждой итерации вычисляется в **NIL**. Если контрольная форма "не срабатывает" (возвращает **NIL** в предложении `always` или не **NIL** в предложении `never`), цикл немедленно прерывается, возвращая **NIL**. Если же цикл выполняется до конца, предоставляется значение по умолчанию: **T**.

Например, если вы хотите проверить, что все числа в списке `numbers` являются четными, вы можете написать следующее:

```
(if (loop for n in numbers always (evenp n))
    (print "All numbers even."))
```

Также вы можете записать следующее:

```
(if (loop for n in numbers never (oddp n))
    (print "All numbers even."))
```

Предложение `thereis` используется для проверки, вычисляется ли контрольная форма в истинное значение хотя бы раз. Как только контрольная форма возвращает значение не равное **NULL**, цикл останавливается, возвращая это значение. Если же цикл доходит до конца, предложение `thereis` предоставляет возвращаемое значение по умолчанию: **NIL**.

```
(loop for char across "abc123" thereis (digit-char-p char)) ==> 1
```

```
(loop for char across "abcdef" thereis (digit-char-p char)) ==> NIL
```

Сложим все вместе

Вы увидели все основные возможности **LOOP**. Вы можете комбинировать все выше обсужденные предложения следуя следующим правилам:

- * Предложение `named`, если указывается, должно быть первым предложением.
- * После предложения `named` идут все остальные предложения `initially`, `with`, `for` и `repeat`.
- * Затем идут предложения тела: условного и безусловного выполнения, накопления, критериев завершения.
- * Завершается цикл предложениями `finally`.

Макрос **LOOP** раскрывается в код, который осуществляет следующие действия:

- * Инициализирует все локальные переменные цикла, которые объявлены в предложениях `with` или `for`, а также неявно созданы предложениями накопления. Начальные значения форм вычисляются в порядке появления соответствующих предложений в цикле.
- * Выполняет формы, предоставляемые предложениями `initially` (вводная часть), в порядке их появления в цикле.
- * Итерирует, выполняя тело цикла как описано в следующем абзаце.
- * Выполняет формы, предоставляемые предложениями `finally` (заключительная часть), в порядке их появления в цикле.

Во время работы цикла сначала соответствующим образом изменяются все переменные управления итерацией, а затем выполняются все предложения условного и безусловного выполнения, накопления, критериев завершения в том порядке, в каком они появляются в коде цикла. Если любое из предложений тела цикла завершает цикл, оставшаяся часть тела пропускается и происходит возврат из цикла, возможно после выполнения завершающей части.

И это описывает почти все, связанное с **LOOP**. Вы будете использовать **LOOP** далее в этой книге довольно часто, поэтому стоило получить некоторое представление о нем. Ну а после вам самим решать, насколько интенсивно использовать **LOOP**.

И теперь вы готовы к погружению в практические главы, составляющие оставшуюся часть этой книги. Для начала мы напишем антиспамовый фильтр.

23. Практика: спам-фильтр

В 2002-м году Paul Graham, имея некоторое количество свободного времени после продажи Viaweb Yahoo, написал статью "A Plan for Spam"), которая привела к небольшой революции в технологии фильтрации спама. До статьи Graham, большинство спам-фильтров были написаны в терминах рукописных правил: если сообщение имеет в заголовке слово XXX, то вероятно оно является спамом; Если в сообщении имеется три или больше слов в строке написанных ЗАГЛАВНЫМИ БУКВАМИ, то вероятно, что это тоже спам. Graham провел несколько месяцев пытаясь написать фильтр, который бы использовал такие правила, до того, как осознал что это фундаментально неправильная задача.

Для того, чтобы узнать индивидуальные признаки спама вы должны попытаться влезть в шкуру спамера, и я публично заявляю, что я хочу провести как можно меньше времени в этом качестве.

Чтобы не пытаться думать как спамер, Graham решил попробовать отделять спам от не-спама, используя статистику, собранную о том, какие слова появляются в обоих типах сообщений. Фильтр может отслеживать то, как часто отдельные слова появляются и в спаме и в не-спаме, и затем использовать частоты вхождения этих слов в сообщения, чтобы вычислить вероятность того, к какой группе относится сообщение. Он назвал этот подход Байесовской фильтрацией (Bayesian filtering) по ассоциации с названием статистического подхода, который он использовал для вычисления частот слов.

Сердце спам-фильтра

В этой главе вы реализуете основную функциональность системы фильтрации спама. Вы не будете писать полноценное приложение; вместо этого, вы сосредоточитесь на функциях для классификации новых сообщений и тренировки фильтра.

Это приложение будет достаточно большим, так что было бы удобным определение нового пакета для того, чтобы избежать конфликта имен. Например, в исходном коде, который вы можете загрузить с сайта данной книги, я использую имя пакета COM.GIGAMONKEYS.SPAM, определяя пакет, который использует и стандартный пакет COMMON-LISP и пакет COM.GIGAMONKEYS.PATHNAMES из главы 15. Определение выглядит следующим образом:

```
(defpackage :com.gigamonkeys.spam
  (:use :common-lisp :com.gigamonkeys.pathnames))
```

Любой файл, содержащий код для данного приложения должен начинаться со строки:

```
(in-package :com.gigamonkeys.spam)
```

Вы можете продолжать использовать это имя пакета, или можете заменить com.gigamonkeys на домен, который находится под вашим контролем.

Вы можете также ввести данное выражение в REPL чтобы переключиться на этот пакет и протестировать функции, которые вы пишете. В SLIME это приведет к смене строки приглашения с CL-USER> на SPAM>, вот так:

```
CL-USER> (in-package :com.gigamonkeys.spam)
#<The COM.GIGAMONKEYS.SPAM package>
SPAM>
```

После того, как вы определили пакет, вы можете начать писать код. Основная функция, которую вы должны реализовать, выполняет простую работу – получает текст сообщения в качестве аргумента, и классифицирует сообщение как спам, не-спам или неопределенное. Вы можете легко реализовать эту функцию путем определения ее в терминах других функций, которые вы напишете позже.

```
(defun classify (text)
  (classification (score (extract-features text))))
```

Читая этот код, начиная с самых вложенных функций, первым шагом в классификации сообщения будет извлечение свойств (features), которые затем будут переданы функции score. В функции score вы вычислите значение, которое может быть преобразовано в одну из классификаций (спам, не-спам или неопределенное) функцией classification. Из этих трех функций, функция classification является самой простой. Вы можете предположить, что score будет возвращать значение около 1 если сообщение является спамом, около 0, если оно не является спамом, и около .5, если система не может корректно классифицировать его.

Так что вы можете реализовать classification следующим образом:

```
(defparameter *max-ham-score* .4)
(defparameter *min-spam-score* .6)
(defun classification (score)
  (cond
    ((<= score *max-ham-score*) 'ham)
    ((>= score *min-spam-score*) 'spam)
    (t 'unsure)))
```

Функция extract-features также достаточно проста, хотя она требует большего количества кода для реализации. В настоящее время, свойствами, которые вы будете извлекать из сообщения, будут слова из текста сообщения. Для каждого слова вам необходимо отслеживать количество вхождений в сообщения, указанные как спам и не-спам. Удобным способом хранения всех этих данных вместе со словом, является определение класса word-feature с тремя слотами.

```
(defclass word-feature ()
  ((word
    :initarg :word
    :accessor word
    :initform (error "Must supply :word")
    :documentation "The word this feature represents.")
   (spam-count
    :initarg :spam-count
    :accessor spam-count
    :initform 0
    :documentation "Number of spams we have seen this feature in.")
   (ham-count
    :initarg :ham-count
    :accessor ham-count
    :initform 0
    :documentation "Number of hams we have seen this feature in.)))
```

Вы будете хранить все свойства в хэш-таблице, так что вы сможете легко находить объект представляющий заданное свойство. Вы можете определить специальную переменную, *feature-database*, для хранения указателя на данную таблицу.

```
(defvar *feature-database* (make-hash-table :test #'equal))
```

Вы должны использовать DEFVAR вместо DEFPARAMETER, поскольку вы не хотите, чтобы *feature-database* была очищена, если в ходе работы вы заново загрузите файл, содержащий определение этой переменной – она может содержать данные, которые вы не хотите потерять. Конечно, это означает, что если вы хотите очистить накопленные данные, то вы не можете просто заново вычислить выражение DEFVAR. Так что вы должны определить функцию clear-database.

```
(defun clear-database ()  
  (setf *feature-database* (make-hash-table :test #'equal)))
```

Для нахождения свойств, присутствующих в заданном сообщении, код должен будет выделить отдельные слова, и затем найти соответствующий объект word-feature в таблице *feature-database*. Если *feature-database* не содержит такого свойства, то вам необходимо создать новый объект word-feature чтобы хранить данные о новом слове. Вы можете поместить эту логику в отдельную функцию, intern-feature, которая получает слово и возвращает соответствующее свойство, создавая его, если это необходимо.

```
(defun intern-feature (word)  
  (or (gethash word *feature-database*)  
      (setf (gethash word *feature-database*)  
            (make-instance 'word-feature :word word))))
```

Вы можете выделить из сообщения отдельные слова с помощью регулярных выражений. Например, используя библиотеку Common Lisp Portable Perl-Compatible Regular Expression (CL-PPCRE), написанную Weitz, вы можете написать extract-words следующим образом:

```
(defun extract-words (text)  
  (delete-duplicates  
    (cl-ppcre:all-matches-as-strings "[a-zA-Z]{3,}" text)  
    :test #'string=))
```

Теперь все что вам остается реализовать в extract-features – это совместить вместе extract-words и intern-feature. Поскольку extract-words возвращает список строк и вы хотите получить список, в котором каждая строка преобразована в соответствующий объект word-feature, то тут самое время применить MAPCAR.

```
(defun extract-features (text)  
  (mapcar #'intern-feature (extract-words text)))
```

Вы можете проверить эти функции в интерпретаторе, например вот так:

```
SPAM> (extract-words "foo bar baz")  
("foo" "bar" "baz")
```

И вы можете убедиться, что DELETE-DUPPLICATES работает правильно:

```
SPAM> (extract-words "foo bar baz foo bar")  
("baz" "foo" "bar")
```

Вы также можете проверить работу extract-features.

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE @ #x71ef28da> #<WORD-FEATURE @ #x71e3809a>
 #<WORD-FEATURE @ #x71ef28aa>)
```

Однако как вы можете видеть, стандартный метод печати произвольных объектов не особо информативен. В процессе работы над этой программой, было бы полезно иметь возможность печатать объекты `word-feature` в более понятном виде. К счастью, как я упоминал в главе 17, печать объектов реализована в терминах обобщенной функции `PRINT-OBJECT`, так что для изменения способа печати объектов `word-feature` вам нужно определить метод для `PRINT-OBJECT`, специализированный для `word-feature`. Для того, чтобы сделать реализацию таких методов более легкой, Common Lisp предоставляет макрос `PRINT-UNREADABLE-OBJECT`.

Использование `PRINT-UNREADABLE-OBJECT` выглядит следующим образом:

```
(print-unreadable-object (object stream-variable &key type identity)
  body-form*)
```

Аргумент `object` является выражением, которое вычисляется в объект, который должен быть напечатан. Внутри тела `PRINT-UNREADABLE-OBJECT`, `stream-variable` связывается с потоком, в который вы можете напечатать все, что вам нужно. Все что вы напечатаете в этот поток, будет выведено в `PRINT-UNREADABLE-OBJECT` и заключено в стандартный синтаксис для не читаемых объектов — `#<>`.

`PRINT-UNREADABLE-OBJECT` также позволяет вам включать в вывод тип объекта и признак подлинности (`FIXME identity`) путем указания именованных параметров `type` и `identity`. Если они имеют не-`NIL` значение, то вывод будет начинаться с имени класса и заканчиваться признаком подлинности (`FIXME identity`) объекта, точно также, как это делается стандартным методом `PRINT-OBJECT` для объектов, унаследованных от `STANDARD-OBJECT`. Для `word-feature`, вы вероятно захотите определить метод `PRINT-OBJECT`, который будет включать в вывод тип, но не включать подлинность (`FIXME identity`), а также значения слотов `word`, `ham-count` и `spam-count`. Такой метод может выглядеть вот так:

```
(defmethod print-object ((object word-feature) stream)
  (print-unreadable-object (object stream :type t)
    (with-slots (word ham-count spam-count) object
      (format stream "~s :hams ~d :spams ~d" word ham-count spam-count))))
```

Теперь вы можете протестировать работу `extract-features` в интерпретаторе и увидите какие свойства были выделены из сообщения.

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE "baz" :hams 0 :spams 0>
 #<WORD-FEATURE "foo" :hams 0 :spams 0>
 #<WORD-FEATURE "bar" :hams 0 :spams 0>)
```

Тренируем фильтр

Теперь, когда у вас имеется способ отслеживания отдельных свойств, вы почти готовы для реализации функции `score`. Но сначала вам нужно написать код, который вы будете использовать для тренировки фильтра, так что `score` будет иметь хоть какие-то данные для использования. Вы можете определить функцию `train`, которая получает некоторый текст и символ, определяющий к какому типу относится это сообщений (спам или не спам), и которая для всех свойств, присутствующих в заданном тесте, увеличивает либо счетчик для спама, либо

счетчик не спама, а также глобальный счетчик обработанных сообщений. Снова, вы можете использовать подход разработки "сверху вниз" (top-down) и реализовать эту функцию в терминах других, еще не существующих функций.

```
(defun train (text type)
  (dolist (feature (extract-features text))
    (increment-count feature type))
  (increment-total-count type))
```

Вы уже написали `extract-features`, так что следующим шагом будет реализация `increment-count`, которая получает объект `word-feature` и тип сообщения, и увеличивает соответствующий слот данного свойства. Поскольку нет причин думать, что логика увеличения этих счетчиков будет применяться для различных видов объектов, то вы можете написать ее как обычную функцию. Поскольку вы определили и `ham-count` и `spam-count` с опциями `:accessor`, то для увеличения соответствующего слота вы можете использовать вместе `INCF` и функции доступа, созданные при вычислении `DEFCLASS`.

```
(defun increment-count (feature type)
  (ecase type
    (ham (incf (ham-count feature)))
    (spam (incf (spam-count feature))))))
```

Конструкция `ECASE` является вариантом конструкции `CASE`, которые обе похожи на конструкцию `case` в языках, произошедших от `Algol` (переименованный в `switch` в `C` и его производных). Обе эти конструкции вычисляют свой первый аргумент и затем находят выражение, чей первый элемент (ключ) имеет то же самое значение в соответствии с логикой сравнения `EQL`. В нашем случае это означает, что вычисляется переменная `type`, возвращая значение, переданное как второй аргумент функции `increment-count`.

Ключи поиска не вычисляются. Другими словами, значение переменной `type` будет сравниваться с непосредственными объектами (literal objects), считанными процедурой чтения `Lisp` как часть выражения `ECASE`. В этой функции, это означает что ключи являются символами `ham` и `spam`, а не значениями переменных с именами `ham` и `spam`. Так что, если `increment-count` будет вызвана вот так:

```
(increment-count some-feature 'ham)
```

то значением `type` будет символ `ham`, и будет вычислено первое выражение `ECASE`, что приведен к увеличению счетчика для не спама. С другой стороны, если мы вызовем эту функцию вот так:

```
(increment-count some-feature 'spam)
```

то будет выполнено второе выражение, увеличивая счетчик для спама. Заметьте, что при вызове `increment-count` символы `ham` и `spam` маскируются, иначе это приведет к тому, что они будут считаться именами переменных. Но они не маскируются, когда они используются в `ECASE`, поскольку `ECASE` не вычисляет ключи сравнения.

Буква `E` в `ECASE` обозначает "исчерпывающий" (`EXHAUSTIVE` "exhaustive") или "ошибка" ("error"), обозначая, что `ECASE` должен выдать ошибку, если сравниваемое значение не совпадает ни с одним, из перечисленных ключей. Обычное выражение `CASE` возвращает `NIL`, если не было найдено совпадений.

Для реализации `increment-total-count`, вам нужно решить, где вы будете хранить счетчики; в настоящий момент, достаточно использовать две глобальные переменные: `*total-spams*` и `*total-hams*`.

```
(defvar *total-spams* 0)
(defvar *total-hams* 0)
(defun increment-total-count (type)
  (ecase type
    (ham (incf *total-hams*))
    (spam (incf *total-spams*)))))
```

Вы должны использовать `DEFVAR` для определения этих двух переменных по той же причине, что и для переменной `*feature-database*` — они будут хранить данные, которые вы не хотите потерять лишь потому, что вы в процессе разработки заново считали исходный код. Но вы можете захотеть, чтобы эти переменные также сбрасывались при очистке `*feature-database*`, так что вы должны добавить несколько строк в функцию `clear-database`, как это показано здесь:

```
(defun clear-database ()
  (setf
    *feature-database* (make-hash-table :test #'equal)
    *total-spams* 0
    *total-hams* 0))
```

Пословная статистика

Сердцем статистического спам-фильтра являются функции, которые вычисляют статистические вероятности. Математические нюансы того, как эти вычисления производятся не являются темой данной книги — заинтересованные читатели могут обратиться к нескольким статьям Gary Robinson. Однако я сосредоточусь на том, как это все реализуется.

Начальной точкой для статистических вычислений является набор измеренных значений — частоты сохраненные в переменных `*feature-database*`, `*total-spams*` и `*total-hams*`. Предполагая, что набор сообщений, на которых происходила тренировка, является статистически репрезентативным, мы можем рассматривать полученные частоты как вероятности появления соответствующих свойств в спаме и не спаме.

Основная идея классификации сообщения заключается в выделении всех свойств, вычисления вероятностей для отдельных свойств, и затем объединения всех вычисленных вероятностей в значение для всего сообщения. Сообщения, с большим количеством "спамовых" свойств и малым количеством "не спамовых" будут иметь значения около 1, а сообщения, с большим количеством "не спамовых" свойств и малым количеством "спамовых", получают значение около 0.

Сначала вам нужно иметь статистическую функцию, которая вычисляет базовую вероятность, что сообщение, содержащее данное свойство, является спамом. С нашей точки зрения, вероятность, что сообщение, содержащее заданное свойство является спамом, равно отношению числа спам-сообщений, содержащих данное свойство, к общему количеству сообщений, содержащих данное свойство. Так что это значение будет вычисляться вот так:

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (/ spam-count (+ spam-count ham-count)))))
```

Проблема с вычисляемым этой функцией значением заключается в том, что оно сильно зависит от полной вероятности, что любое сообщение будет считаться как спам, или как не спам. Например, предположим, что у вас в девять раз больше не спама, чем спама. Тогда полностью нейтральное свойство, в соответствии с данной функцией, будет появляться в одном спамовом сообщении, против девяти не спамовых сообщений, давая вам вероятность спама, равную $1/10$.

Но вы более заинтересованы в вероятности, что данное свойство будет появляться в спамовых сообщениях, независимо от общей вероятности получения спама или не спама. Таким образом, вам нужно разделить число вхождений в спам на количество спамовых сообщений, на которых происходила тренировка, и то же самое сделать для число вхождений в не спамовые сообщения. Для того, чтобы избежать получения ошибок *division-by-zero* (деление на ноль), если либо **total-spams**, либо **total-hams** равно нулю, вам необходимо считать соответствующие частоты равными нулю. (Если общее число спамовых или не спамовых сообщений равно нулю, то соответствующие счетчики в свойствах, также должны быть равны нулю, так что вы можете рассматривать полученную частоту равной нулю).

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (let ((spam-frequency (/ spam-count (max 1 *total-spams*)))
          (ham-frequency (/ ham-count (max 1 *total-hams*))))
      (/ spam-frequency (+ spam-frequency ham-frequency)))))
```

Эта версия страдает от другой проблемы – она не обращает внимания на число проанализированных сообщений. Предположим, что вы производили обучение на 2000 сообщений, половина спама и половина не спама. Теперь рассмотрим два свойства, которые входят только в сообщения со спамом. Одно из них входит во все сообщения, а второе - только в одно из них. В соответствии с текущей реализацией *spam-probability*, появление любого из свойств, сообщает, что сообщение является спамом с вероятностью 1.

Однако, все равно возможно, что свойство, которое входит только в одно сообщение, в действительности является нейтральным свойством – оно достаточно редко появляется в спаме и не спаме. Если вы проведете обучение на следующих двух тысячах сообщений, может быть, что оно появится еще раз, теперь – в не спаме, что сделает его нейтральным, с вероятностью вхождения в спам, равной .5.

Так что, вы можете захотеть вычислять вероятность, которая как-либо зависит от числа данных (*number of data points*), которые входят в вероятность каждого свойства. В своих статьях Robinson предложил функцию, основанную на Байесовском понимании включения наблюдаемых данных в априорные знания или предположения. Проще говоря, вы вычисляете новую вероятность начиная с предполагаемой априорной вероятности и веса, данного этой вероятности, а затем добавляя новую информацию. Функция предложенная Robinson'ом выглядит вот так:

```
(defun bayesian-spam-probability (feature &optional
                                   (assumed-probability 1/2)
                                   (weight 1))
  (let ((basic-probability (spam-probability feature))
        (data-points (+ (spam-count feature) (ham-count feature))))
    (/ (+ (* weight assumed-probability)
          (* data-points basic-probability))
       (+ weight data-points))))
```

Robinson предложил значения $1/2$ для *assumed-probability* и 1 для *weight*. Используя эти значения, свойство, которое один раз встретилось в спаме, и ни разу в не спаме, будет иметь значение *bayesian-spam-probability* равное 0.75, а свойство, которое встречается 10 раз

в спаме, и ни разу в не спаме, будет иметь значение `bayesian-spam-probability` приблизительно равное 0.955, а то, которое входит в 1000 спамовых сообщений, и ни разу в не спам, будет иметь вероятность приблизительно равную 0.9995.

Комбинирование вероятностей

Теперь, когда вы можете вычислить `bayesian-spam-probability` для каждого из свойств в сообщении, последним шагом будет реализация функции `score` для комбинирования отдельных вероятностей в одно значение в диапазоне между 0 и 1.

Если бы вероятности отдельных свойств были бы независимы, то можно было бы перемножить их для получения общей вероятности. Но к сожалению в действительности они зависимы — некоторые свойства появляются вместе с другими, а некоторые никогда не появляются с другими.

Robinson предложил использовать метод комбинации вероятности, предложенный статистиком R. A. Fisher (Фишер). Не вдаваясь в детали того, как этот метод работает, он выглядит следующим образом: сначала вы комбинируете вероятности путем их умножения. Это дает вам число, близкое к нулю если в вашей выборке много свойств с низкими вероятностями. Потом вы берете логарифм данного числа и умножаете его на -2. Фишер в 1950 показал, что если отдельные вероятности были независимыми, и соответствовали равномерному распределению между 0 и 1, то результирующее значение будет соответствовать χ^2 -квадрат распределению. Это значение и удвоенное число вероятностей может быть передано в обратную функцию χ^2 -квадрат, которая вернет вероятность, которая отражает вероятность получения значения, которое больше значения, полученного комбинированием того же числа произвольно выбранных вероятностей. Когда обратная функция χ^2 -квадрат возвращает маленькую вероятность, это означает что использовалось неправильно число малых значений (либо большое число значений с относительно малой вероятностью, либо несколько очень малых значений) в отдельных вероятностях.

Для использования этой вероятности для определения является ли сообщения спамом или нет, вы должны начать с *нулевой гипотезы (null hypothesis)*, предполагаемого предположения, несостоятельность которого вы надеетесь доказать. Нулевая гипотеза заключается в том, что классифицируемое сообщение в действительности является произвольным набором свойств. Если это так, то отдельные вероятности (вероятности того, что каждое свойство может появиться в спаме) также будут произвольными. Так что, произвольная выборка свойств обычно будет содержать некоторые свойства с большой вероятностью появления в спаме, а другие свойства будут иметь низкую вероятность появления в спаме. Если вы скомбинируете эти произвольно выбранные вероятности в соответствии с методом Фишера, то вы должны получить усредненное значение, для которого обратная функция χ^2 -квадрат сообщит вероятность успеха. Но если обратная функция χ^2 -квадрат возвращает низкую вероятность, то это означает, что к сожалению, вероятности, которые вошли в объединенное значение, были выбраны не произвольным образом; использовалось слишком много значений с низкой вероятностью чтобы это было случайным. Так что вы можете избавиться от нулевой гипотезы и вместо этого использовать альтернативную гипотезу, что свойства были взяты из противоположного примера — с несколькими свойствами с высокой вероятностью, и большим числом с низкой вероятностью. Другими словами, это должно быть не спамовое сообщение.

Однако, метод Фишера не является симметричным, поскольку обратная функция χ^2 -квадрат возвращает вероятность, что заданное число вероятностей, выбранных произвольным образом, может быть скомбинировано таким образом, чтобы значение было больше чем полученное путем объединения настоящих вероятностей. Эта асимметрия работает на вас, поскольку когда вы отвергаете нулевую гипотезу, вы знаете что существует более правильная гипотеза. Когда вы

комбинируете отдельные вероятности с помощью метода Фишера, и вам говорят, что существует высокая вероятность, что нулевая гипотеза является неправильно (что сообщение не является произвольным набором слов), то это означает, что сообщение вероятно является не спамом. Возвращенное число не является вероятностью, что сообщение не является спамом, но, по крайней мере, является хорошим признаком этого. И наоборот, комбинация отдельных вероятностей не спамовых свойств по Фишеру, дает вам признак того, что сообщение обладает свойствами спама.

Для получения окончательного результата вам необходимо объединить эти два значения в одно число, которое даст вам рейтинг "спам-не спам" в диапазоне от 0 до 1. Метод, рекомендованный Робинсоном, заключается в добавлении к числу $1/2$ половины разницы между значениями вероятности отнесения к спаму и не спаму, или, другими словами, среднее значение вероятности отнесения к спаму и единицы минус вероятность отнесения к не спаму. Это приводит нужному эффекту, так что если два значения согласованы (высокая вероятность спама и низкая не спама, или наоборот), то вы получите четкий индикатор близкий по значению к 0 или 1. Но когда оба значения высоки или низки, то вы получите окончательное значение приблизительно равное $1/2$, что будет рассматриваться как "неопределенное".

Функция `score`, которая реализует эту схему выглядит следующим образом:

```
(defun score (features)
  (let ((spam-probs ()) (ham-probs ()) (number-of-probs 0))
    (dolist (feature features)
      (unless (untrained-p feature)
        (let ((spam-prob (float (bayesian-spam-probability feature) 0.0d0)))
          (push spam-prob spam-probs)
          (push (- 1.0d0 spam-prob) ham-probs)
          (incf number-of-probs))))
    (let ((h (- 1 (fisher spam-probs number-of-probs)))
          (s (- 1 (fisher ham-probs number-of-probs))))
      (/ (+ (- 1 h) s) 2.0d0))))
```

Вы берете список свойств и выполняете цикл, строя два списка вероятностей – один список вероятностей, что сообщение, содержащее каждое из свойств, является спамом, и другой список, что сообщение не является спамом. Для оптимизации, вы также можете подсчитать количество вероятностей и передать это число функции `fisher`, чтобы избежать подсчета в самой функции `fisher`. Число, возвращенное функцией `fisher` будет маленьким, если отдельные вероятности содержат слишком много малых вероятностей чтобы рассматриваться как произвольный текст. Так что, малое число Фишера для "спамовых" вероятностей означает, что там содержится много не-спамовых свойств; вычитая число Фишера из 1, вы получаете вероятность того, что сообщение не является спамом. Соответственно, вычитая число Фишера для "не-спамовых" вероятностей из 1, дает вам вероятность того, что сообщение является спамом. Комбинируя эти два числа вы получаете общую вероятность принадлежности к спаму в диапазоне между 0 и 1.

Внутри цикла вы можете использовать функцию `untrained-p` для пропуска тех свойств, которые не встречались в процессе обучения. Эти свойства будут иметь счетчики спама и не спама равные нулю. Функция `untrained-p` является тривиальной.

```
(defun untrained-p (feature)
  (with-slots (spam-count ham-count) feature
    (and (zerop spam-count) (zerop ham-count))))
```

Новой функцией является `fisher`. Предполагая, что вы уже имеете функцию `inverse-chi-`

square, to fisher является достаточно простой.

```
(defun fisher (probs number-of-probs)
  "The Fisher computation described by Robinson."
  (inverse-chi-square
    (* -2 (log (reduce #'* probs)))
    (* 2 number-of-probs)))
```

К сожалению, существует небольшая проблема с этой прямолинейной реализацией. Хотя использование REDUCE является кратким и идиоматичным способом умножения списка чисел, но в этом конкретном приложении существует опасность, что произведение будет слишком маленьким чтобы быть представленным как число с плавающей запятой. В этом случае, результат будет (FIXME underflow to zero) преобразован в ноль. И если произведение вероятностей будет равен нулю, то все будет напрасно, поскольку вызов LOG для нуля либо выдаст ошибку, либо, в некоторых реализациях, приведет к получению специального значения "отрицательная бесконечность", которое приведет к тому, что все последующие вычисления станут бессмысленными. Это очень нежелательно, поскольку метод Фишера очень чувствителен к малым значениям (близким к нулю), и при умножении часто возникает вероятность (FIXME underflow) переполнения снизу.

К счастью, для того, чтобы избежать данной проблемы вы можете использовать немного знаний из школьной математики. Вспомните, что логарифм произведения является суммой логарифмов соответствующих членов. Так что вместо умножения вероятностей, и затем вычисления логарифма, вы можете использовать сумму логарифмов вероятностей. А поскольку REDUCE может принимать именованный параметр :key, то вы можете использовать ее для проведения всех вычислений. Так что вместо этого кода:

```
(log (reduce #'* probs))
```

напишите вот этот:

```
(reduce #'+ probs :key #'log)
```

Обратная функция Chi-квадрат

Реализации функции inverse-chi-square приведенная в данном разделе, является практически прямым переводом на Lisp версии функции написанной на Робинсоном на Python. Точное математическое значение этой функции не рассматривается в этой книге, но вы можете получить примерное знание о том, что она делает, путем размышления о том, как значения, которые вы передаете функции fisher будут влиять на результат: большее количество малых значений, переданных fisher, приведет к меньшему значению произведения вероятностей. Логарифм от малого числа приведет к получению отрицательного числа, с большим абсолютным значением, которое после умножения на -2, станет еще большим положительным числом. Таким образом, чем больше будет вероятностей с малым значением переданно fisher, тем большее значение будет передано inverse-chi-square. Конечно, число используемых вероятностей также влияет на значение переданное inverse-chi-square. Поскольку вероятности, по определению имеют значение меньше или равное 1, то большее количество вероятностей входящих в произведение будет приводить к меньшему значению вероятности, и соответственно, к большему числу, переданному функции inverse-chi-square. Так что функция inverse-chi-square должна возвращать низкую вероятность в тех случаях, когда число Фишера является ненормально большим для числа вероятностей, которые входят в него. Следующая функция делает следующее:

```
(defun inverse-chi-square (value degrees-of-freedom)
  (assert (evenp degrees-of-freedom))
  (min
    (loop with m = (/ value 2)
      for i below (/ degrees-of-freedom 2)
      for prob = (exp (- m)) then (* prob (/ m i))
      summing prob)
    1.0))
```

Возвращаясь к главе 10, вспоминаем, что функция EXP возводит число e (основание натурального алгоритма) в заданную степень. Таким образом, чем больше используемое значение, тем меньше будет начальное значение `prob`. Но это начальное значение затем будет выравнено вверх, для каждой из степеней свободы, пока `m` больше чем число степеней свободы. Поскольку значение возвращаемое функцией `inverse-chi-square` рассматривается как другая вероятность, то важно ограничить значение возвращаемое MIN, поскольку ошибки округления при умножении и возведении в степень могут привести к тому, что LOOP вернет сумму которая больше 1.

Тренируем фильтр

Поскольку вы написали `classify` и `train` таким образом, чтобы они принимали аргумент-строку, то вы можете работать с ними интерактивно. Если вы еще это не сделали, то вы должны переключиться на пакет, в рамках которого вы писали код, путем вычисления формы `IN-PACKAGE` в строке ввода, или используя сокращенную форму, реализованную в SLIME – `change-package`. Для использования этой возможности SLIME, наберите запятую, и затем наберите имя в строке ввода. Нажатие Tab при наборе имени пакета приведет к автоматическому дополнению имени, основываясь на именах пакетов, которые знает Lisp. Теперь вы можете выполнить любую функцию, которая является частью спам-фильтра. Сначала вы должны убедиться, что база данных свойств пуста.

```
SPAM> (clear-database)
```

Теперь вы можете тренировать фильтр с помощью конкретного текста.

```
SPAM> (train "Make money fast" 'spam)
```

И посмотреть что думает по этому поводу функция классификации.

```
SPAM> (classify "Make money fast")
SPAM
SPAM> (classify "Want to go to the movies?")
UNSURE
```

Хотя вам нужны лишь результаты классификации, было бы хорошо видеть и вычисленную оценку. Самым простым способом получения обоих значений не затрагивая остальной код, будет изменение функции `classification` таким образом, чтобы она возвращала несколько значений.

```
(defun classification (score)
  (values
    (cond
      ((<= score *max-ham-score*) 'ham)
      ((>= score *min-spam-score*) 'spam)
      (t 'unsure))
    score))
```

Вы можете сделать это изменение и затем перекомпилировать лишь одну функцию. Поскольку функция `classify` возвращает то, что вернула функция `classification`, то она также будет возвращать два значения. Но поскольку, основное возвращаемое значение не затрагивается, то пользователи данной функции, ожидающие лишь одно значение, никак не будут затронуты данными изменениями. Теперь, когда вы будете тестировать `classify`, вы сможете увидеть какое значение было передано функции `classification`.

```
SPAM> (classify "Make money fast")
SPAM
0.863677101854273D0
SPAM> (classify "Want to go to the movies?")
UNSURE
0.5D0
```

И теперь вы сможете увидеть что произойдет, если потренируете фильтр с некоторым количеством не-спамового текста.

```
SPAM> (train "Do you have any money for the movies?" 'ham)
1
SPAM> (classify "Make money fast")
SPAM
0.7685351219857626D0
```

Этот текст все равно считается спамом, но с меньшей оценкой, поскольку слово `money` входил в не-спамовый текст.

```
SPAM> (classify "Want to go to the movies?")
HAM
0.17482223132078922D0
```

А сейчас этот текст правильно распознается как не-спам, из-за наличия в нем слова `movies`, которое сейчас считается не-спамовым свойством.

Однако, вы не должны хотеть, чтобы фильтр тренировался вручную. То что вам действительно нужно – простой способ указать на пачку файлов и провести обучение фильтра на них. И если вы хотите протестировать то, как фильтр работает, то вы можете использовать его для классификации другого набора файлов известного типа, и проанализировать результаты. Так что последним кусочком кода, который вы напишете в данной главе, будет набор тестов, которые будут тестировать фильтр относительно наборов сообщений известных типов, используя части из них для обучения, а затем измеряя точность с которой фильтр классифицирует оставшуюся часть сообщений.

Тестируем фильтр

Для тестирования фильтра вам нужны наборы сообщений известных типов. Вы можете использовать сообщения из вашего почтового ящика, или вы можете взять один из наборов

сообщений, размещенных в Интернете. Например, набор сообщений SpamAssassin содержит несколько тысяч сообщений, классифицированных вручную на спам, явный не-спам и не-спам, который тяжело отличить от спама. Чтобы сделать пользование тестами более простым, вы можете определить вспомогательные функции, которые управляются массивом пар имя файла/тип. Вы можете определить функцию, которая принимает имя файла, и тип, и добавляет их в набор, следующим образом:

```
(defun add-file-to-corpus (filename type corpus)
  (vector-push-extend (list filename type) corpus))
```

Значение `corpus` (набор) должно быть изменяемым вектором с указателем заполнения. Например, вы можете создать новый набор следующим образом:

```
(defparameter *corpus* (make-array 1000 :adjustable t :fill-pointer 0))
```

Если у вас спам и не спам уже находятся в разных каталогах, то вы можете захотеть добавить все файлы в каталоге используя один и тот же тип. Вот функция, которая использует функцию `list-directory` из главы 15, чтобы выполнить эту задачу:

```
(defun add-directory-to-corpus (dir type corpus)
  (dolist (filename (list-directory dir))
    (add-file-to-corpus filename type corpus)))
```

Например, предположим, что у вас есть каталог `mail`, содержащий два подкаталога, `spam` и `ham`, каждый содержащий сообщения соответствующего типа (спам и не-спам); вы можете добавить все файлы из этих двух каталогов к набору, хранящемуся в `*corpus*`, используя следующие команды:

```
SPAM> (add-directory-to-corpus "mail/spam/" 'spam *corpus*)
NIL
SPAM> (add-directory-to-corpus "mail/ham/" 'ham *corpus*)
NIL
```

Теперь вам нужна функция для проверки классификатора. Основная стратегия работы будет заключаться в выборе произвольной части набора сообщений для обучения фильтра, а затем тестирования работы путем классификации оставшейся части набора, сравнивая классификацию, возвращенную нашей функцией с известными результатами. Главной вещью которую вы захотите узнать является то, насколько правильно работает классификатор – сколько процентов сообщений классифицировано правильно. Но вы вероятно также будете заинтересованы в информации о том, какие сообщения были неправильно классифицированы и в чем заключается ошибка – больше неправильных пропусков или фальшивых срабатываний? Для того, чтобы сделать более простым выполнение различных видов анализа поведения классификатора, вы должны определить функции тестирования, для построения списка вычисленных значений, которые вы затем сможете проанализировать как захотите.

Основная функция тестирования будет выглядеть следующим образом:


```
(defun test-classifier (corpus testing-fraction)
  (clear-database)
  (let* ((shuffled (shuffle-vector corpus))
        (size (length corpus))
        (train-on (floor (* size (- 1 testing-fraction)))))
    (train-from-corpus shuffled :start 0 :end train-on)
    (test-from-corpus shuffled :start train-on)))
```

Эта функция начинает работу с очистки базы свойств. Затем она перемешивает набор писем используя функцию, которую мы реализуем далее, и определяет, на основе параметра `testing-fraction`, сколько значений мы будем использовать для обучения и сколько мы оставим для тестирования. Две вспомогательные функции: `train-from-corpus` и `test-from-corpus` будут принимать именованные параметры `:start` и `:end`, что позволит работать над частью заданного набора сообщений.

Функция `train-from-corpus` достаточно проста – это цикл по соответствующей части набора сообщений с использованием `DESTRUCTURING-BIND` для выделения имени файла и типа из списка, находящегося в каждом элементе, и затем передача данных параметров для обучения. Поскольку некоторые почтовые сообщения, особенно такие, которые имеют вложения, имеют достаточно большой размер, то вы должны ограничить количество знаков, которое будет выделяться из сообщения. Функция будет получать текст с помощью функции `start-of-file`, которую мы реализуем далее, которая будет принимать в качестве параметров имя файла и максимальное количество возвращаемых знаков. `train-from-corpus` выглядит примерно так:

```
(defparameter *max-chars* (* 10 1024))
(defun train-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) do
    (destructuring-bind (file type) (aref corpus idx)
      (train (start-of-file file *max-chars*) type))))
```

Функция `test-from-corpus` выглядит аналогично, за тем исключением, что вы захотите возвращать список, содержащий результаты каждой из операции классификации, так что вы в последующем сможете проанализировать эти результаты. Так что вы должны захватывать и определенный тип сообщения, и вычисленное значение, возвращенные функцией `classify`, и собирать эти данные в список состоящий из имени файла, известного типа, типа, возвращенного функцией `classify` и вычисленное значение. Чтобы сделать результаты более понятными для человека, вы можете включить в список именованные параметры, описывающие соответствующие значения.

```
(defun test-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) collect
    (destructuring-bind (file type) (aref corpus idx)
      (multiple-value-bind (classification score)
        (classify (start-of-file file *max-chars*))
        (list
         :file file
         :type type
         :classification classification
         :score score)))))
```

Набор вспомогательных функций

Для окончания реализации функции `test-classifier`, вам необходимо написать две вспомогательных функции, которые напрямую не относятся к фильтрации спама – `shuffle-`

vector и start-of-file.

Простым и эффективным способом реализации shuffle-vector будет использование алгоритма Фишера-Ятеса (Fisher-Yates). Вы можете начать с реализации функции nshuffle-vector, которая перемешивает вектор, используя то же самое хранилище. Имя функции соответствует тому же самому соглашению по именованию деструктивных функций, таких как NCONC и NREVERSE. Она выглядит следующим образом:

```
(defun nshuffle-vector (vector)
  (loop for idx downfrom (1- (length vector)) to 1
        for other = (random (1+ idx))
        do (unless (= idx other)
              (rotatef (aref vector idx) (aref vector other))))
  vector)
```

Недеструктивная версия просто делает копию оригинального вектора, и передает его в деструктивную версию.

```
(defun shuffle-vector (vector)
  (nshuffle-vector (copy-seq vector)))
```

Другая вспомогательная функция – start-of-file, также достаточно проста. Наиболее эффективным способом считывания содержимого файла в память, будет создание массива соответствующего размера и использования функции READ-SEQUENCE для его заполнения. Так что вы можете создать массив знаков с размером, равным размеру файла, или максимальному количеству считываемых знаков, в зависимости от того, какое из значений будет меньше. К сожалению, как я упоминал в главе 14, функция FILE-LENGTH не особенно хорошо работает для текстовых потоков, поскольку количество знаков в файле может зависеть от используемой кодировки знаков и конкретного текста. В наихудшем случае, единственным способом точного определения количества знаков в файле, является считывание всего файла. Таким образом, неясно что вернет FILE-LENGTH для текстового потока; в большинстве реализаций FILE-LENGTH всегда возвращает число байт в файле, которое может быть больше, чем число знаков, которое может быть прочитано из файла.

Однако, READ-SEQUENCE возвращает количество прочитанных знаков. Так что вы можете попробовать считать количество знаков, определенное с помощью FILE-LENGTH, и вернуть подстроку, если количество считанных знаков было меньше.

```
(defun start-of-file (file max-chars)
  (with-open-file (in file)
    (let* ((length (min (file-length in) max-chars))
          (text (make-string length))
          (read (read-sequence text in)))
      (if (< read length)
          (subseq text 0 read)
          text))))
```

Анализ результатов

Теперь вы готовы к написанию кода для анализа результатов, сгенерированных test-classifier. Мы должны вспомнить, что test-classifier возвращает список, возвращенный test-from-corpus, в котором каждый элемент является списком свойств (plist), описывающим результаты классификации одного файла. Этот список содержит имя файла, известный тип, результат классификации и вычисленное значение, возвращенное

функцией `classify`. Первой частью нашего аналитического кода, который вы должны написать, является функция, которая будет возвращать признак того, была ли классификация правильной, или нет (пропущенный спам или не спам, и т.п.). Вы можете использовать `DESTRUCTURING-BIND` для получения элементов `:type` и `:classification` списка результатов (используя опцию `&allow-other-keys` для того, чтобы `DESTRUCTURING-BIND` игнорировал другие пары имя-значение), а затем используя вложенные выражения `ECASE` для преобразования отдельных сочетаний в конкретный символ.

```
(defun result-type (result)
  (destructuring-bind (&key type classification &allow-other-keys) result
    (ecase type
      (ham
        (ecase classification
          (ham 'correct)
          (spam 'false-positive)
          (unsure 'missed-ham)))
      (spam
        (ecase classification
          (ham 'false-negative)
          (spam 'correct)
          (unsure 'missed-spam))))))
```

Вы можете проверить эту функцию в интерпретаторе.

```
SPAM> (result-type '(:FILE #p"foo" :type ham :classification ham :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type spam :classification spam :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type ham :classification spam :score 0))
FALSE-POSITIVE
SPAM> (result-type '(:FILE #p"foo" :type spam :classification ham :score 0))
FALSE-NEGATIVE
SPAM> (result-type '(:FILE #p"foo" :type ham :classification unsure :score 0))
MISSED-HAM
SPAM> (result-type '(:FILE #p"foo" :type spam :classification unsure :score 0))
MISSED-SPAM
```

Наличие этой функции делает анализ результатов `test-classifier` более простым. Например, вы можете начать с определения следующих функций-предикатов для каждого типа результатов.

```
(defun false-positive-p (result)
  (eq1 (result-type result) 'false-positive))
(defun false-negative-p (result)
  (eq1 (result-type result) 'false-negative))
(defun missed-ham-p (result)
  (eq1 (result-type result) 'missed-ham))
(defun missed-spam-p (result)
  (eq1 (result-type result) 'missed-spam))
(defun correct-p (result)
  (eq1 (result-type result) 'correct))
```

С помощью этих функций, вы можете просто использовать функции работы со списками и последовательностями, которые обсуждались в главе 11, для того чтобы выделить и подсчитать разные типы результатов.

```
SPAM> (count-if #'false-positive-p *results*)
6
SPAM> (remove-if-not #'false-positive-p *results*)
((:FILE #p"ham/5349" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9999983107355541d0)
 (:FILE #p"ham/2746" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.6286468956619795d0)
 (:FILE #p"ham/3427" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9833753501352983d0)
 (:FILE #p"ham/7785" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9542788587998488d0)
 (:FILE #p"ham/1728" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.684339162891261d0)
 (:FILE #p"ham/10581" :TYPE HAM :CLASSIFICATION SPAM :SCORE
0.9999924537959615d0))
```

Вы также можете использовать символы, возвращенные `result-type` в качестве ключей хэш-таблицы или ассоциативного списка (`alist`). Например, вы можете написать функцию, которая будет печатать итоговые результаты и процентное соотношение каждого типа в результате, используя ассоциативный список, который отображает каждый из типов в соответствующий счетчик.

```
(defun analyze-results (results)
  (let* ((keys '(total correct false-positive
                  false-negative missed-ham missed-spam))
         (counts (loop for x in keys collect (cons x 0))))
    (dolist (item results)
      (incf (cdr (assoc 'total counts)))
      (incf (cdr (assoc (result-type item) counts))))
    (loop with total = (cdr (assoc 'total counts))
          for (label . count) in counts
          do (format t "~&~@(~a~):~20t~5d~,5t: ~6,2f%~%"
                     label count (* 100 (/ count total))))))
```

Эта функция выдаст следующий результат, если ей передать список результатов, созданный с помощью `test-classifier`:

```
SPAM> (analyze-results *results*)
Total: 3761 : 100.00%
Correct: 3689 : 98.09%
False-positive: 4 : 0.11%
False-negative: 9 : 0.24%
Missed-ham: 19 : 0.51%
Missed-spam: 40 : 1.06%
NIL
```

И в качестве заключительного этапа анализа, вы можете захотеть взглянуть на то, почему отдельное сообщение было классифицировано таким образом. Следующая функция сделает это:

```

(defun explain-classification (file)
  (let* ((text (start-of-file file *max-chars*))
         (features (extract-features text))
         (score (score features))
         (classification (classification score)))
    (show-summary file text classification score)
    (dolist (feature (sorted-interesting features))
      (show-feature feature))))
(defun show-summary (file text classification score)
  (format t "~&~a" file)
  (format t "~2%~a~2%" text)
  (format t "Classified as ~a with score of ~,5f~%" classification score))
(defun show-feature (feature)
  (with-slots (word ham-count spam-count) feature
    (format
      t "~&~2t~a~30thams: ~5d; spams: ~5d;~,10tprob: ~,f~%"
      word ham-count spam-count (bayesian-spam-probability feature))))
(defun sorted-interesting (features)
  (sort (remove-if #'untrained-p features) #'< :key #'bayesian-spam-probability))

```

Что далее?

Конечно, вы могли бы сделать больше реализовав эту задачу. Для превращения написанного кода в полноценное приложение для фильтрации спама, вам понадобилось бы найти способ интеграции его в вашу почтовую систему. Один из способов, который можно было бы применить для того, чтобы можно было использовать любой почтовый клиент, является написание кода, который бы позволял выполнять данное приложение как прокси для POP3 – протокола, который большинство клиентов используют для скачивания почты с почтовых серверов. Такая прокси могла бы забирать почту с настоящего POP3-сервера, и раздавать ее вашим почтовым клиентам после того, как сообщения либо будут помечены как спам с помощью дополнительных заголовков, так что фильтры ваших почтовых клиентов смогут распознать такие сообщения, либо будут отложены в сторону. Конечно, вам необходим способ для общения с фильтром на тему неправильной классификации сообщений – поскольку вы установите приложение как сервер, то вы также должны будете предоставить Web-интерфейс для работы с ним. Я буду обсуждать вопросы построения Web-интерфейсов в главе 26, и мы создадим его, для другого приложения, в главе 29.

Или вы можете захотеть улучшить основы классификации – наиболее вероятным местом для начала работы будет улучшение работы `extract-features`. В частности, вы должны сделать процедуру разбивки на слова более осведомленной о структуре почтового сообщения – вы можете выделить различные виды свойств для слов, появляющихся в теле письма, и для тех, которые появляются в заголовках сообщения. И конечно, вы можете декодировать различные виды кодировок сообщений, таких как `base64` и `quoted printable`, поскольку спаммеры часто пытаются изменить свои сообщения используя эти кодировки.

Но я оставляю эти улучшения на ваше усмотрение. Теперь вы готовы продолжить ваш путь к построению потокового MP3-сервера, начав с написания библиотеки для разбора двоичных файлов.

24. Практика. Разбор двоичных файлов

В этой главе я покажу вам как создать библиотеку, которую мы сможем использовать при написании кода для чтения и записи двоичных файлов. Мы воспользуемся этой библиотекой в главе 25 для написания программы разбора тегов ID3, механизма, используемого для хранения метаданных в файлах MP3, таких как исполнитель и название альбома. Эта библиотека является также примером использования макросов для расширения языка новыми конструкциями, превращая его в язык специального назначения, предназначенный для решения специфических задач, в данном случае чтения и записи двоичных файлов. Так как мы разработаем эту библиотеку за один присест, включая несколько промежуточных версий, вам может показаться, что мы напишем очень много кода. Но после того, как все будет сказано и сделано, вся библиотека целиком займет менее 150 строк кода, а самый большой макрос будет длиной всего в 20 строк.

Двоичные файлы

На достаточно низком уровне абстракции все файлы являются "двоичными" в том смысле, что они просто содержат набор чисел, закодированных в двоичной форме. Однако, обычно различают *текстовые файлы*, в которых все числа могут быть интерпретированы как знаки, представляющие человекочитаемый текст, и *двоичные файлы*, который содержат данные, которые при интерпретации их как знаков, выдают непечатаемые знаки

Двоичные форматы файлов обычно проектируются в целях повышения компактности данных и эффективности их разбора — это и является их главным преимуществом над текстовыми форматами. Для достижения этих критериев двоичные файлы обычно имеют такую структуру на диске (on-disk structures), которая легко отображается на структуры данных, используемые программой для представления в памяти хранящихся в файлах данных.

Разработанная библиотека предоставит нам простой способ описания соответствия между структурами на диске, определенных двоичным форматом файла, и объектами Lisp в памяти. Использование этой библиотеки сделает легким написание программ, осуществляющих чтение двоичных файлов, преобразование их в объекты Lisp для дальнейших манипуляций и запись в другой двоичный файл.

Основы двоичного формата

Начальной точкой в чтении и записи двоичных файлов является открытие файла для чтения и записи отдельных байтов. Как я описывал в главе 14, и **OPEN**, и **WITH-OPEN-FILE**, принимают ключевой аргумент `:element-type`, который устанавливает базовую единицу передачи данных для потока. Для работы с двоичными файлами нужно указать `(unsigned-byte 8)`. Входной поток, открытый с таким параметром `:element-type`, будет возвращать числа от 0 до 255 при каждой его передаче в вызов **READ-BYTE**. И наоборот, мы можем записывать байты в выходной поток с типом элементов `(unsigned-byte 8)` путем передачи чисел от 0 до 255 в **WRITE-BYTE**.

Выше уровня отдельных байтов большинство двоичных форматов используют минимальное количество примитивных типов данных: различные представления чисел, текстовые строки, битовые поля и так далее, которые затем комбинируются в более сложные структуры. Поэтому вашим первым заданием будет определение каркаса для написания кода чтения и записи примитивных типов данных, используемых данным двоичным форматом.

В качестве простого примера представим, что мы имеем дело с двоичным форматом, который использует беззнаковые 16-битные целые числа в качестве примитивного типа данных. Для

осуществления чтения таких целых нам нужно прочитать два байта, а затем скомбинировать их в одно число путем умножения одного байта на 256 (то есть 2^8) и добавления к нему второго байта. Предположив, например, что двоичный формат определяет хранение таких 16-битных сущностей в *обратном порядке байтов (big-endian)*, когда наиболее значащий байт идет первым, мы можем прочитать такое число с помощью следующей функции:

```
(defun read-u2 (in)
  (+ (* (read-byte in) 256) (read-byte in)))
```

Однако, Common Lisp предоставляет более удобный способ осуществления такого рода операций с битами. Функция **LDB**, чье имя происходит от load byte, может быть использовано для извлечения и присваивания (с помощью **SETF**) любого/любому числу идущих подряд бит из целого числа. Число бит и их местоположение в целом числе задается спецификатором байта, создаваемом функцией **BYTE**. **BYTE** получает два аргумента, число бит для извлечения (или присваивания) и позицию самого правого бита, где наименее значимый бит имеет нулевую позицию. **LDB** принимает спецификатор байта и целое, из которого нужно извлечь биты, и возвращает положительное целое, представляющее извлеченные биты. Таким образом мы можем извлечь наименее значащие восемь бит из целого числа подобным образом:

```
(ldb (byte 8 0) #xabcd) ==> 205 ; 205 is #xcd
```

Для получения следующих восьми бит нам нужно использовать спецификатор байта (byte 8 8) следующим образом:

```
(ldb (byte 8 8) #xabcd) ==> 171 ; 171 is #xab
```

Мы можем использовать **LDB** с **SETF** для присваивания заданным битам целого числа, сохраненного в SETFable месте.

```
CL-USER> (defvar *num* 0)
*NUM*
CL-USER> (setf (ldb (byte 8 0) *num*) 128)
128
CL-USER> *num*
128
CL-USER> (setf (ldb (byte 8 8) *num*) 255)
255
CL-USER> *num*
65408
```

Итак, мы можем написать read-u2 также следующим образом:

```
(defun read-u2 (in)
  (let ((u2 0))
    (setf (ldb (byte 8 8) u2) (read-byte in))
    (setf (ldb (byte 8 0) u2) (read-byte in))
    u2))
```

Для записи числа как 16-битового целого, нам нужно извлечь отдельные байты и записать их один за одним. Для извлечения отдельных байтов нам просто нужно воспользоваться **LDB** с такими же спецификаторами байтов.

```
(defun write-u2 (out value)
  (write-byte (ldb (byte 8 8) value) out)
  (write-byte (ldb (byte 8 0) value) out))
```

Конечно мы также можем закодировать целые множеством других способов: с различным числом байтов, в различном порядке, а также путем использования знакового или беззнакового форматов.

Строки в двоичных файлах

Текстовые строки являются еще одним видом примитивных типов данных, который мы можем найти во многих двоичных форматах. Мы не можем считывать и записывать строки напрямую, читая файлы побайтно — нам нужно побайтно декодировать и кодировать их, как мы делали это для двоично-кодируемых чисел. И, как кодировать целые числа мы можем не одним способом, кодировать строки мы также можем множеством способов. Поэтому для начала формат двоичных чисел должен определять то, как кодируются отдельные знаки.

Для преобразования байт в знаки нам необходимо знать используемые знаковый код (*character code*) и кодировку знаков (*character encoding*). Знаковый код определяет отображение множества положительных целых чисел на множество знаков. Каждое число отображения называется *единицей кодирования* (*code point*). Например ASCII является знаковым кодом, который отображает числа интервала 0-127 на знаки, использующиеся в латинском алфавите. Кодировка знаков, с другой стороны, определяет как кодовые единицы представляются в виде последовательности байт в байт-ориентированной среде, такой как файл. Для кодов, которые используют восемь или менее бит, таких как ASCII и ISO-8859-1, кодировка тривиальна: каждое численное значение кодируется единственным байтом.

Почти так же просты чистые двухбайтовые кодировки, такие как UCS-2, которые осуществляют отображение между 16-битными значениями и знаками. Единственной причиной, по которой двухбайтовые кодировки могут оказаться более сложными чем однобайтовые, является то, что нам может понадобиться также знать, подразумевается ли кодирование 16-битных значений в обратном порядке байт, либо же в прямом.

Кодировки с переменной длиной используют различное число октетов для различных численных значений, делая их более сложным, но позволяя им быть более лаконичными в большинстве случаев. Например UTF-8, кодировка, спроектированная для использования с кодом знаков Unicode, использует лишь один октет для кодирования значений из интервала 0-127 и в то же время до четырех октетов для кодирования значений до 1,114,111.

Так как единицы кодирования из интервала 0-127 отображаются в Unicode на те же знаки, что и в кодировке ASCII, то закодированный кодировкой UTF-8 текст, состоящий только из знаков ASCII, будет эквивалентен этому же тексту, но закодированному кодировкой ASCII. С другой стороны, текст, состоящий преимущественно из знаков, требующих четырех байт в UTF-8, может быть более компактно закодировано простой двухбайтовой кодировкой.

Common Lisp предоставляет две функции для преобразования между численными кодами знаков и объектами знаков: **CODE-CHAR**, которая получает численный код и возвращает знак, и **CHAR-CODE**, которая получает знак и возвращает его численный код. Стандарт языка не определяет, какую кодировку знаков должны использовать реализации языка, поэтому нет гарантии того, что мы сможем представить любой знак, который может быть закодирован в данном формате файла, как знак Lisp. Однако почти все современные реализации Common Lisp используют ASCII, ISO-8859-1 или Unicode в качестве своего внутреннего знакового кода. Так как Unicode является надмножеством ISO-8859-1, который в свою очередь является

надмножеством ASCII, то, если ваша реализация Lisp использует Unicode, **CODE-CHAR** и **CHAR-CODE** могут быть использованы напрямую для преобразования любого из этих трех знаковых кодов.

Вдобавок к определению кодирования знаков кодирование строк должно также определять то, как кодируется длина строк. В двоичных форматах файлов обычно используются три техники.

Простейшая заключается в том, чтобы никак не кодировать длину, которая неявно определяется по местоположению строки в некоторой большей структуре: некоторый элемент файла может всегда быть строкой определенной длины, либо строка может быть последним элементом структуры данных переменной длины, общий размер которой определяет как много байт осталось прочитать как данные строки. Оба этих подхода используются в тегах ID3, как мы увидим в следующей главе.

Другие две техники могут использоваться для кодирования строк переменной длины без необходимости полагаться на контекст. Одной из них является кодирование длины строки, за которой следуют данные этой строки — анализатор считывает численное значение (в каком-то заданном целочисленном формате), а затем считывает это число знаков. Другой техникой является запись данных строки, за которыми следует разделитель, который не может появиться внутри строки, такой как нулевой знак (null character).

Различные представления имеют различные преимущества и недостатки, но когда мы имеем дело с уже заданными двоичными форматами, мы не имеем никакого контроля над тем, какая кодировка используется. Однако, никакая из кодировок не является более сложной для чтения/записи, чем любая другая. Вот, например, функция, осуществляющая чтение завершающейся нулевым знаком строки ASCII, подразумевающая, что ваша реализация Lisp использует ASCII либо одно из ее надмножеств, такое как ISO-8859-1 или Unicode, в качестве своей внутренней кодировки:

```
(defconstant +null+ (code-char 0))
(defun read-null-terminated-ascii (in)
  (with-output-to-string (s)
    (loop for char = (code-char (read-byte in))
          until (char= char +null+) do (write-char char s))))
```

Макрос **WITH-OUTPUT-TO-STRING**, который упоминался в главе 14, является простым способом построения строки в случае, когда мы не знаем, какой длины она окажется. Этот макрос создает **STRING-STREAM** и связывает его с указанным именем переменной, в данном случае *s*. Все знаки, записанные в этот поток, будут собраны в строку, которая затем будет возвращена в качестве значения формы **WITH-OUTPUT-TO-STRING**.

Для записи строки нам просто нужно преобразовать знаки обратно в численные значения, которые могут быть записаны с помощью **WRITE-BYTE**, а затем записать признак конца строки после ее содержимого.

```
(defun write-null-terminated-ascii (string out)
  (loop for char across string
        do (write-byte (char-code char) out))
  (write-byte (char-code +null+) out))
```

Как показывает этот пример, главной интеллектуальной задачей (может и не совсем таковой, но все же) чтения и записи базовых элементов двоичных файлов является понимание того, как именно интерпретировать байты файла и отображать их на типы данных Lisp. Если формат двоичного файла хорошо определен, это может оказаться довольно простой задачей.

Фактически написание функций для чтения и записи данных, закодированных определенным образом, является просто вопросом программирования.

Теперь мы можем перейти к задаче чтения и записи более сложных структур на диске (on-disk structures) и отображения их на объекты Lisp.

Составные структуры

Так как двоичные форматы обычно используются для представления данных способом, который делает легким их отображение на структуры данных в памяти, не должно вызывать удивление то, что сложные структуры на диске (on-disk structures) обычно определяются схожим способом с тем, как языки программирования определяют структуры данных в памяти. Обычно сложные структуры на диске состоят из некоторого числа именованных частей, каждая из которых является либо примитивным типом, таким как число или строка, либо другой сложной структурой, либо коллекцией таких значений.

Например тег ID3, определенный версией 2.2 спецификации, состоит из заголовка, в свою очередь состоящего из ISO-8859-1 строки длиной в три знака, которыми всегда являются "ID3"; двух однобайтных беззнаковых целых, которые задают старший номер версии и ревизию спецификации; восьми бит, являющихся булевыми флагами; и четырех байт, которые кодируют размер тега в кодировке, особенной для спецификации ID3. За заголовком идет список *фреймов*, каждый из которых имеет свою собственную внутреннюю структуру. За фреймами идет столько нулевых байт, сколько необходимо для заполнения тега до размера, указанного в заголовке.

Если вы глядите на мир через призму объектной ориентации, сложные структуры выглядят весьма похожими на классы. Например мы можем написать класс для представления тега ID3.

```
(defclass id3-tag ()
  ((identifier :initarg :identifier :accessor identifier)
   (major-version :initarg :major-version :accessor major-version)
   (revision :initarg :revision :accessor revision)
   (flags :initarg :flags :accessor flags)
   (size :initarg :size :accessor size)
   (frames :initarg :frames :accessor frames)))
```

Экземпляр этого класса может быть отличным местом хранения данных, необходимых для представления тега ID3. Затем мы можем написать функции чтения и записи экземпляров этого класса. Например, предположив существование функций чтения соответствующих примитивных типов данных, функция `read-id3-tag` может выглядеть следующим образом:

```
(defun read-id3-tag (in)
  (let ((tag (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) tag
      (setf identifier (read-iso-8859-1-string in :length 3))
      (setf major-version (read-u1 in))
      (setf revision (read-u1 in))
      (setf flags (read-u1 in))
      (setf size (read-id3-encoded-size in))
      (setf frames (read-id3-frames in :tag-size size)))
    tag))
```

Функция `write-id3-tag` будет структурирована схожим образом: мы будем использовать соответствующие функции `write-*` для записи значений, хранящихся в слотах объекта `id3-tag`.

Несложно увидеть, как мы можем написать соответствующие классы для представления всех

сложных структур данных спецификации наряду с функциями `read-foo` и `write-foo` для каждого класса и необходимых примитивных типов. Но также легко заметить, что все функции чтения и записи будут весьма похожими, отличающимися только тем, данные каких типов они читают, и именами слотов, в которые они сохраняют эти данные. Это станет особенно утомительным, когда мы учтем тот факт, что описание структуры тега ID3 заняло почти четыре строки текста, в то время как мы уже написали одиннадцать строк кода все еще не написав `write-id3-tag`.

Что нам действительно нужно, так это способ описания структур наподобие тега ID3 в форме, которая лаконична так же, как и псевдокод спецификации, и чтобы это описание раскрывалось в код, который определяет класс `id3-tag` и функции, осуществляющие преобразование между байтами на диске и экземплярами этого класса. Звучит как работа для системы макросов.

Проектирование макросов

Так как мы уже имеем примерное представление о том, какой код ваш макрос должен генерировать, следующим шагом в соответствии с процессом написания макросов, описанным мною в главе 8, является смена ракурса и размышления о том, как должен выглядеть вызов этого макроса. Так как целью является иметь возможность написания чего-то столь же краткого, как и псевдокод спецификации ID3, мы можем начать с него. Заголовок тега ID3 определяется следующим образом:

```
ID3/file identifier "ID3"
ID3 version $02 00
ID3 flags %xx000000
ID3 size 4 * %0xxxxxxx
```

В нотации спецификации это означает, что слот "file identifier" тега ID3 является строкой "ID3" в кодировке ISO-8859-1. Слот version состоит из двух байт, первый из которых, для данной версии спецификации, имеет значение 2 и второй, опять же для данной версии, — 0. Слот flags имеет размер в восемь бит, все из которых, кроме первых двух, имеют нулевое значение, а size состоит из четырех байт, каждый из которых содержит 0 в своем старшем разряде.

Некоторая часть информации не охватывается этим псевдокодом. Например то, как именно интерпретируются четыре байта, кодирующие размер, описывается несколькими строками текста. Схожим образом спецификация описывает текстом то, как после заголовка сохраняются фрейм и последующие байты заполнения. Но, все же, большая часть того, что нам нужно знать, для того, чтобы написать код чтения и записи тега ID3, задается этим псевдокодом. Таким образом, мы должны иметь возможность написания варианта этого псевдокода s-выражением, которое раскроется в класс и определения функций, которые нам иначе бы пришлось писать вручную: что-то, возможно, вроде этого:

```
(define-binary-class id3-tag
  ((file-identifier (iso-8859-1-string :length 3))
   (major-version u1)
   (revision u1)
   (flags u1)
   (size id3-tag-size)
   (frames (id3-frames :tag-size size))))
```

Основной идеей является то, что эта форма определяет класс `id3-tag` подобно тому, как мы можем сделать сами с помощью **DEFCLASS**, но вместо определения таких вещей как `:initarg` и `:accessor`, каждое определение слота состоит из имени слота — `file-identifier`, `major-version`, и т.д. — и информации о том, как этот слот представляется на диске. Так как мы

всего лишь немного пофантазировали, нам не нужно беспокоиться о том, как именно макрос `define-binary-class` будет знать, что делать с такими выражениями как `(iso-8859-1-string :length 3)`, `u1`, `id3-tag-size` и `(id3-frames :tag-size size)`; пока каждое выражение содержит информацию, необходимую для знания того, как читать и записывать определенные данные, все должно быть хорошо.

Делаем мечту реальностью

Хорошо, достаточно фантазий о хорошо выглядящем коде; теперь нужно приступить к работе по написанию `define-binary-class`: написанию кода, который будет преобразовывать краткое выражение, описывающее как выглядит тег ID3, в код, который может представлять этот тег в памяти, считывать с диска и записывать его обратно.

Для начала нам стоит определить пакет для нашей библиотеки. Вот файл пакета, который поставляется с версией, которую вы можете скачать с web-сайта книги:

```
(in-package :cl-user)
(defpackage :com.gigamonkeys.binary-data
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :define-binary-class
           :define-tagged-binary-class
           :define-binary-type
           :read-value
           :write-value
           :*in-progress-objects*
           :parent-of-type
           :current-binary-object
           :+null+))
```

Пакет `COM.GIGAMONKEYS.MACRO-UTILITIES` содержит макросы `with-gensyms` и `once-only` из главы 8.

Так как мы уже имеем написанную вручную версию кода, который хотим сгенерировать, не должно быть очень сложно написать такой макрос. Просто разберем его на небольшие части, начав с версии `define-binary-class`, которая просто генерирует форму **DEFCLASS**.

Если мы вновь взглянем на форму `define-binary-class`, то увидим, что она принимает два аргумента: имя `id3-tag` и список спецификаторов слотов, каждый из которых сам является двух-элементным списком. По этим частям нам нужно построить соответствующую форму **DEFCLASS**. Очевидно, что наибольшее различие между формой `define-binary-class` и правильной формой **DEFCLASS** заключается в спецификаторах слотов. Одиночный спецификатор слота из `define-binary-class` выглядит подобным образом:

```
(major-version u1)
```

Но это не является верным спецификатором слота для **DEFCLASS**. Вместо этого нам нужно что-то вот такое:

```
(major-version :initarg :major-version :accessor major-version)
```

Достаточно просто. Для начала определим простую функцию преобразования символа в соответствующий ключевой символ.

```
(defun as-keyword (sym) (intern (string sym) :keyword))
```

Теперь определим функцию, которая получает спецификатор слота `define-binary-class` и возвращает спецификатор слота **DEFCLASS**.

```
(defun slot->defclass-slot (spec)
  (let ((name (first spec)))
    `(,name :initarg , (as-keyword name) :accessor ,name)))
```

Мы можем протестировать эту функцию в REPL после переключения в наш новый пакет путем вызова **IN-PACKAGE**.

```
BINARY-DATA> (slot->defclass-slot '(major-version u1))
(MAJOR-VERSION :INITARG :MAJOR-VERSION :ACCESSOR MAJOR-VERSION)
```

Выглядит хорошо *FIXME* (так по-русски не пишут. выглядит хорошо, но вот-вот случится жопа). Теперь написание первой версии `define-binary-class` тривиально.

```
(defmacro define-binary-class (name slots)
  `(defclass ,name ()
    , (mapcar #'slot->defclass-slot slots)))
```

Это простой макрос, написанный в `template`-стиле: `define-binary-class` генерирует форму **DEFCLASS** путем подстановки (interpolating) имени класса и списка спецификаторов слотов, сконструированного путем применения `slot->defclass-slot` к каждому элементу списка спецификаторов слотов формы `define-binary-class`.

Для просмотра кода, который генерирует этот макрос, мы можем вычислить в REPL следующее выражение:

```
(macroexpand-1 '(define-binary-class id3-tag
  ((identifier (iso-8859-1-string :length 3))
   (major-version u1)
   (revision u1)
   (flags u1)
   (size id3-tag-size)
   (frames (id3-frames :tag-size size)))))
```

Результат, слегка переформатированный в целях улучшения читаемости, должен казаться вам знакомым, так как это в точности то определение класса, которое мы написали вручную ранее:

```
(defclass id3-tag ()
  ((identifier :initarg :identifier :accessor identifier)
   (major-version :initarg :major-version :accessor major-version)
   (revision :initarg :revision :accessor revision)
   (flags :initarg :flags :accessor flags)
   (size :initarg :size :accessor size)
   (frames :initarg :frames :accessor frames)))
```

Чтение двоичных объектов

Следующим шагом нам нужно заставить `define-binary-class` также генерировать функцию, которая может прочитать экземпляр нового класса. Взгляните вновь на функцию `read-id3-tag`, написанную нами ранее. Она выглядит слегка более мудрено, так как не является столь же однородной: для чтения значений каждого слота, нам приходится вызывать различные функции, не говоря уже о том, что имя функции, `read-id3-tag`, хоть и получается из имени

определяемого нами класса, не является одним из аргументов `define-binary-class`, а следовательно не может быть просто подставлено в шаблон способом, сходным с тем, как мы делали это для имени класса.

Мы можем решить обе эти проблемы путем придумывания и следования такому соглашению по именованию, при котором макрос сможет вычислять имя функции, основываясь на имени типа в спецификаторе слота. Однако это потребует от `define-binary-class` генерирования имени `read-id3-tag`, что возможно, но является плохой идеей. Макросам, создающим глобальные определения, следует в общем случае использовать только имена, переданные им; макросы, сами генерирующие имена, могут привести к сложнопредсказуемым, а также сложным для отладки конфликтам имен, когда сгенерированные имена оказываются теми же, что уже используются где-то.

Мы можем избежать оба этих неудобства заметив, что все функции, считывающие значения определенного типа, имеют в своей сути одинаковую цель: считывание значения определенного типа из потока. Говоря просто, мы можем увидеть, что все они являются экземплярами одной обобщенной операции. И простое использование слова "обобщенный" должно подтолкнуть вас прямо к решению проблемы: вместо определения множества независимых функций, имеющих различные имена, мы можем определить одну обобщенную функцию `read-value` с методами, специализированными для чтения значений различных типов.

Таким образом, вместо определения функций `read-iso-8859-1-string` и `read-u1`, мы можем определить `read-value` как обобщенную функцию, принимающую два обязательных аргумента: тип и поток, а также возможно некоторые ключевые аргументы.

```
(defgeneric read-value (type stream &key)
  (:documentation "Read a value of the given type from the stream."))
```

Путем указания **&key** без самих ключевых параметров, мы позволяем различным методам определять свои собственные **&key** параметры, но не требуя этого от них. Но это требует того, что каждый метод, специализирующий `read-value`, должен будет включить либо **&key**, либо **&rest** в свой список параметров, чтобы быть совместимым с обобщенной функцией.

Затем мы определяем методы, использующие специализаторы EQL для специализации аргумента типа по имени типа значений, которые хотим считывать.

```
(defmethod read-value ((type (eql 'iso-8859-1-string)) in &key length) ...)
(defmethod read-value ((type (eql 'u1)) in &key) ...)
```

Затем мы можем заставить `define-binary-class` сгенерировать метод `read-value`, специализированный по имени типа `id3-tag`, и этот метод может быть реализован в терминах вызовов `read-value` с соответствующими типами слотов в качестве первого аргумента. Код, который мы хотим сгенерировать, выглядит следующим образом:

```
(defmethod read-value ((type (eql 'id3-tag)) in &key)
  (let ((object (make-instance 'id3-tag)))
    (with-slots (identifier major-version revision flags size frames) object
      (setf identifier (read-value 'iso-8859-1-string in :length 3))
      (setf major-version (read-value 'u1 in))
      (setf revision (read-value 'u1 in))
      (setf flags (read-value 'u1 in))
      (setf size (read-value 'id3-encoded-size in))
      (setf frames (read-value 'id3-frames in :tag-size size)))
    object))
```

Теперь, так же, как для генерации формы **DEFCLASS** нам нужна была функция, транслирующая спецификатор слота `define-binary-class` в спецификатор слота **DEFCLASS**, теперь нам нужна функция, получающая спецификатор слота `define-binary-class` и генерирующая соответствующую форму **SETF**, то есть что-то, получающее вот такое:

```
(identifier (iso-8859-1-string :length 3))
```

и возвращающее это:

```
(setf identifier (read-value 'iso-8859-1-string in :length 3))
```

Однако, существует различие между этим кодом и спецификатором слота **DEFCLASS**: этот код включает в себя ссылку на переменную `in`, параметр метода `read-value`, который не был получен из спецификатора слота. Он не обязательно должен называться `in`, но какое бы имя мы не использовали, оно должно быть тем же, что используется в списке параметров метода, а также в других вызовах `read-value`. Сейчас мы можем уклониться от проблемы того, откуда получается это имя, определив `slot->read-value` таким образом, чтобы она принимала второй аргумент, содержащий имя переменной потока.

```
(defun slot->read-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(setf ,name (read-value ',type ,stream ,@args))))
```

Функция `normalize-slot-spec` нормализует второй элемент спецификатора слота, преобразуя символ, такой как `u1`, в список `(u1)`, так что **DESTRUCTURING-BIND** может осуществить его разбор. Она выглядит так:

```
(defun normalize-slot-spec (spec)
  (list (first spec) (mklist (second spec))))
(defun mklist (x) (if (listp x) x (list x)))
```

Мы можем протестировать `slot->read-value` с каждым типом спецификаторов слотов.

```
BINARY-DATA> (slot->read-value '(major-version u1) 'stream)
(SETF MAJOR-VERSION (READ-VALUE 'U1 STREAM))
BINARY-DATA> (slot->read-value '(identifier (iso-8859-1-string :length 3))
'stream)
(SETF IDENTIFIER (READ-VALUE 'ISO-8859-1-STRING STREAM :LENGTH 3))
```

Со всеми этими функциями мы уже готовы добавить `read-value` в `define-binary-class`. Если мы возьмем вручную написанный метод `read-value` и удалим из него все то, что касается определенного класса, у нас останется следующий каркас:

```
(defmethod read-value ((type (eql ...)) stream &key)
  (let ((object (make-instance ...)))
    (with-slots (...) object
      ...
      object)))
```

Все, что нам нужно сделать, это добавить этот каркас в шаблон `define-binary-class`, заменив многоточия кодом, который заполнит этот каркас подходящими именами и кодом. Мы также захотим заменить переменные `type`, `stream` и `object` сгенерированными **GENSYM** именами для избежания потенциальных конфликтов с именами слотов, что мы можем сделать с

помощью макроса `with-gensyms`, рассмотренного в главе 8.

Также, так как макрос должен раскрываться в одиночную форму, мы должны "обернуть" какую-то вокруг **DEFCLASS** и **DEFMETHOD**. Обычно для макросов, которые раскрываются в несколько определений, используется **PROGN** из-за специальной трактовки, которую она получает от компилятора, когда находится на верхнем уровне файла, что было обсуждено в главе 20.

Таким образом, мы можем изменить `define-binary-class` следующим образом:

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
      (defclass ,name ()
        , (mapcar #'slot->defclass-slot slots))
      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let ((,objectvar (make-instance ',name)))
          (with-slots , (mapcar #'first slots) ,objectvar
            ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))
            ,objectvar))))))
```

Запись двоичных объектов

Генерация кода для записи экземпляра двоичного класса происходит схожим образом. Для начала мы можем определить обобщенную функцию `write-value`.

```
(defgeneric write-value (type stream value &key)
  (:documentation "Write a value as the given type to the stream."))
```

Затем мы определяем вспомогательную функцию, которая транслирует спецификатор слота `define-binary-class` в код, который записывает этот слот с помощью `write-value`. Как и для функции `slot->read-value`, эта вспомогательная функция принимает имя переменной потока в качестве параметра.

```
(defun slot->write-value (spec stream)
  (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
    `(write-value ',type ,stream ,name ,@args)))
```

После этого мы можем добавить шаблон `write-value` в макрос `define-binary-class`.

```
(defmacro define-binary-class (name slots)
  (with-gensyms (typevar objectvar streamvar)
    `(progn
      (defclass ,name ()
        , (mapcar #'slot->defclass-slot slots))
      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let ((,objectvar (make-instance ',name)))
          (with-slots , (mapcar #'first slots) ,objectvar
            ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))
            ,objectvar))
      (defmethod write-value ((,typevar (eql ',name)) ,streamvar ,objectvar
        &key)
        (with-slots , (mapcar #'first slots) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```


Добавление наследования и помеченных (tagged) структур

Хотя эта версия `define-binary-class` будет обрабатывать автономные (stand-alone) структуры, двоичные форматы файлов часто определяют такие структуры на диске (on-disk structures), которые было бы естественно моделировать отношениями подклассов и суперклассов. Поэтому мы можем захотеть расширить `define-binary-class` для поддержки наследования.

Родственной техникой, используемой во многих двоичных форматах, является такая, при которой имеется множество структур данных на диске, точный тип которых может быть определен только путем считывания некоторых данных, которые указывают, как осуществлять разбор последующих байтов. Например, фреймы, которые составляют большую часть тега ID3, все разделяют общую структуру заголовка, состоящую из строкового идентификатора и длины. Для чтения фрейма нам нужно прочитать идентификатор и использовать его значение для определения вида просматриваемого фрейма, а следовательно того, как осуществлять разбор его тела.

Текущая версия макроса `define-binary-class` не предоставляет способа осуществления такого рода считывания: мы можем использовать `define-binary-class` для определения класса, представляющего любой вид фрейма, но мы не имеем возможности узнать, какой тип фрейма считывать, без считывания по меньшей мере идентификатора. И если другой код считывает идентификатор, чтобы определить какой тип передавать функции `read-value`, то это нарушит работу `read-value`, поскольку она ожидает возможности считать все данные, составляющие экземпляр класса, создаваемый ею.

Мы можем решить эту проблему добавив возможность наследования в `define-binary-class`, а затем написав другой макрос `define-tagged-binary-class`, предназначенный для определения "абстрактных" классов, экземпляры которых не создаются напрямую, но по которым могут быть специализированы методы `read-value`, которые знают, как считывать достаточно данных для определения конкретного класса, экземпляра которого нужно создать.

Первым шагом добавления возможности наследования в `define-binary-class` является добавление в макрос параметра, принимающего список суперклассов.

```
(defmacro define-binary-class (name (&rest superclasses) slots) ...
```

Затем, в шаблоне **DEFCLASS** подставим это значение вместо пустого списка.

```
(defclass ,name ,superclasses  
  ...)
```

Однако нужно сделать немного больше. Нам также нужно изменить методы `read-value` и `write-value` так, чтобы методы, сгенерированные при определении суперкласса, могли бы быть использованы методами, сгенерированными как часть подкласса, для чтения и записи наследуемых слотов.

Способ, которым работает текущая версия `read-value`, совершенно не подходит, так как он инстанцирует объект перед его заполнением. То есть, у вас есть метод, ответственный за чтение полей суперкласса, инстанцирующий один объект, и метод подкласса, инстанцирующий и заполняющий другой объект.

Мы можем исправить эту проблему путем разделения `read-value` на две части: одну ответственную за инстанцирование правильного вида объекта, а другую — за заполнение слотов

уже существующего объекта. На стороне записи все несколько проще, но и там мы можем использовать схожую технику.

Поэтому мы определим две новые обобщенные функции: `read-object` и `write-object`, обе получающие существующий объект и поток. Методы этих обобщенных функций будут ответственны за чтение и запись слотов, специфичных для классов, для которых они специализированы.

```
(defgeneric read-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Fill in the slots of object from stream."))
(defgeneric write-object (object stream)
  (:method-combination progn :most-specific-last)
  (:documentation "Write out the slots of object to the stream."))
```

Определение этих обобщенных функций с использованием комбинатора методов **PROGN** с опцией `:most-specific-last` позволяет нам определять методы, специализированные для каждого двоичного класса и работающие только со слотами, действительно определенными в таком классе; комбинатор методов **PROGN** скомбинирует все применимые методы так, что метод, специализированный для наименее специфичного класса в иерархии, выполнится первым, считывая или записывая слоты, определенные в этом классе, затем выполнится метод, специализированный для следующего наименее специфичного класса, и т.д. И, так как теперь вся тяжелая, специфичная для класса работа осуществляется методами `read-object` и `write-object`, нам даже не нужно определять специализированные методы `read-value` и `write-value`: мы можем определить методы по умолчанию, которые считают аргумент типа именем двоичного класса.

```
(defmethod read-value ((type symbol) stream &key)
  (let ((object (make-instance type)))
    (read-object object stream)
    object))
(defmethod write-value ((type symbol) stream value &key)
  (assert (typep value type))
  (write-object value stream))
```

Обратите внимание на то, как мы можем использовать **MAKE-INSTANCE** в качестве обобщенной фабрики объектов (generic object factory): хотя обычно мы вызываем **MAKE-INSTANCE** с закавыченным (quoted) символом в качестве первого аргумента, так как обычно знаем, экземпляр какого именно класса хотим создать, мы можем использовать любое выражение, которое вычисляется в имя класса, как в данном случае используем параметр `type` метода `read-value`.

Действительные изменения, внесенные в `define-binary-class` для определения методов `read-object` и `write-object` вместо `read-value` и `write-value`, довольно незначительны.

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(progn
      (defclass ,name ,superclasses
        , (mapcar #'slot->defclass-slot slots))
      (defmethod read-object progn ((,objectvar ,name) ,streamvar)
        (with-slots , (mapcar #'first slots) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))
      (defmethod write-object progn ((,objectvar ,name) ,streamvar)
        (with-slots , (mapcar #'first slots) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Отслеживание унаследованных слотов

Это определение будет работать для многих случаев. Однако, оно не обрабатывает одну достаточно распространенную ситуацию, а именно когда нам нужен подкласс, которому необходимо ссылаться на унаследованные слоты в своих собственных определениях слотов. Например, имея текущее определение `define-binary-class`, мы можем определить следующий одиночный класс:

```
(define-binary-class generic-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)
   (data (raw-bytes :bytes size))))
```

Ссылка на `size` в определении `data` работает ожидаемым образом, так как выражения, считывающие и записывающие слот `data`, обернуты формой **WITH-SLOTS**, которая перечисляет все слоты объекта. Однако, если мы попытаемся разделить этот класс на два класса следующим образом:

```
(define-binary-class frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3)))
(define-binary-class generic-frame (frame)
  ((data (raw-bytes :bytes size))))
```

мы получим предупреждение времени компиляции при компиляции определения `generic-frame` и ошибку времени выполнения при попытке его использования, так как в методах `read-object` и `write-object`, специализированных для `generic-frame`, не будет лексически видимой переменной `size`.

Что нам нужно, так это отслеживание слотов, определенных каждым двоичным классом, а затем включение всех наследуемых слотов в формы **WITH-SLOTS** методов `read-object` и `write-object`.

Наиболее простым способом отслеживания подобной информации является связывание ее с символом, именуемым класс. Как мы обсуждали в главе 21, каждый символьный объект имеет ассоциированный с ним список свойств, доступ к которому можно получить с помощью функций **SYMBOL-PLIST** и **GET**. Мы можем связать произвольную пару ключ/значение с символом, добавив их в список свойств этого символа с помощью вызова **SETF** для результата **GET**. Например, если двоичный класс `foo` определяет три слота `x`, `y` и `z`, мы можем отследить этот факт, добавив в список свойств символа `foo` ключ `slots` со значением `(x y z)` с помощью следующего выражения:

```
(setf (get 'foo 'slots) '(x y z))
```

Мы хотим осуществлять этот учет как часть вычисления `define-binary-class` для `foo`. Однако, не совсем очевидно, куда поместить это выражение. Если мы будем вычислять его при вычислении раскрытий макросов, это выражение вычислится при компиляции формы `define-binary-class`, но не во время последующей загрузки файла, содержащего полученный скомпилированный код. С другой стороны, если мы включим это выражение в раскрытие макроса, то оно не будет вычисляться во время компиляции, а это означает, что при компиляции файла с несколькими формами `define-binary-class` никакой информации о том, какие классы определяют какие слоты, не будет доступно до полной загрузки файла, что слишком поздно.

Это как раз тот случай, для которого предназначен специальный оператор **EVAL-WHEN**, который мы обсуждали в главе 20. Обернув форму в **EVAL-WHEN** мы можем контролировать то, вычисляется ли она во время компиляции, либо во время загрузки скомпилированного кода, либо в обоих случаях. Для таких случаев, как данный, когда мы хотим собрать некоторую информацию во время компиляции формы макроса, к которой мы хотим также иметь доступ после загрузки скомпилированной формы, нам следует обернуть выражения сбора этой информации в **EVAL-WHEN** следующим образом:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get 'foo 'slots) '(x y z)))
```

и включить форму **EVAL-WHEN** в раскрытие, генерируемое макросом. Итак, мы можем сохранить слоты и прямые суперклассы двоичного класса добавив следующую форму в раскрытие, генерируемое `define-binary-class`:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ',name 'slots) ',(mapcar #'first slots))
  (setf (get ',name 'superclasses) ',superclasses))
```

Теперь мы можем определить три вспомогательные функции для осуществления доступа к этой информации. Первая просто возвращает слоты, определенные двоичным классом. Хорошей идеей является возвращение копии списка, так как мы не хотим, чтобы посторонний код модифицировал список слотов после того, как двоичный класс уже был определен.

```
(defun direct-slots (name)
  (copy-list (get name 'slots)))
```

Следующая функция возвращает слоты, унаследованные от других двоичных классов.

```
(defun inherited-slots (name)
  (loop for super in (get name 'superclasses)
        nconc (direct-slots super)
        nconc (inherited-slots super)))
```

И наконец, мы можем определить функцию, которая возвращает список, содержащий имена всех определенных и унаследованных слотов.

```
(defun all-slots (name)
  (nconc (direct-slots name) (inherited-slots name)))
```

Итак, мы хотим, чтобы при вычислении раскрытия формы `define-binary-class`, генерировалась форма **WITH-SLOTS**, содержащая имена всех слотов, определенных в новом классе и во всех его суперклассах. Однако, мы не можем использовать `all-slots` во время генерации раскрытия, так как информация не будет доступна до того момента, когда это раскрытие будет скомпилировано. Вместо этого нам следует воспользоваться следующей функцией, получающей список спецификаторов слотов и список суперклассов, переданных в `define-binary-class`, и использующую их для вычисления списка всех слотов нового класса:

```
(defun new-class-all-slots (slots superclasses)
  (nconc (mapcan #'all-slots superclasses) (mapcar #'first slots)))
```

Имея эти функции мы можем изменить `define-binary-class` таким образом, чтобы он сохранял информацию об определяемом в данный момент классе и использовал уже сохраненную информацию о слотах суперклассов для генерации форм **WITH-SLOTS**.

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(progn
      (eval-when (:compile-toplevel :load-toplevel :execute)
        (setf (get ',name 'slots) ',(mapcar #'first slots))
        (setf (get ',name 'superclasses) ',superclasses))
      (defclass ,name ,superclasses
        , (mapcar #'slot->defclass-slot slots))
      (defmethod read-object progn ((,objectvar ,name) ,streamvar)
        (with-slots , (new-class-all-slots slots superclasses) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots)))
      (defmethod write-object progn ((,objectvar ,name) ,streamvar)
        (with-slots , (new-class-all-slots slots superclasses) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Помеченные структуры

Имея возможность определения двоичных классов, которые расширяют другие двоичные классы, мы готовы определить новый макрос, предназначенный для определения классов, представляющих "помеченные" структуры. В основе нашего способа чтения помеченных структур будет определение специализированного метода `read-value`, который знает как считывать значения, составляющие начало структуры, и как затем использовать эти значения для определения подкласса, экземпляра которого нужно создать. Затем этот метод создает экземпляр класса с помощью **MAKE-INSTANCE**, передавая уже прочитанные значения в качестве аргументов инициализации (`initargs`), и передает объект в `read-object`, позволяя действительному классу объекта определять как именно считывается остальная часть структуры.

Новый макрос `define-tagged-binary-class` будет выглядеть как `define-binary-class` с добавочной опцией `:dispatch`, используемой для указания формы, которая должна вычисляться в имя двоичного класса. Форма `:dispatch` будет вычислена в том контексте, в котором имена слотов, определенных помеченным классом, связываются с переменными, содержащими значения, прочитанные из файла. Класс, чье имя возвращается этой формой, должен принимать аргументы инициализации (`initargs`), соответствующие именам слотов, определенных помеченным классом. Это легко обеспечивается в том случае, если форма `:dispatch` всегда вычисляется в имя класса, являющегося подклассом помеченного класса.

Например, представив, что мы имеем функцию `find-frame-class`, которая отображает строковый идентификатор на двоичный класс, представляющий определенный вид фрейма ID3, мы можем определить помеченный двоичный класс `id3-frame` следующим образом:

```
(define-tagged-binary-class id3-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

Раскрытие `define-tagged-binary-class` будет содержать **DEFCLASS** и метод `write-object` так же, как и раскрытие `define-binary-class`, но вместо метода `read-object` оно будет содержать метод `read-value`, выглядящий следующим образом:

```
(defmethod read-value ((type (eql 'id3-frame)) stream &key)
  (let ((id (read-value 'iso-8859-1-string stream :length 3))
        (size (read-value 'u3 stream)))
    (let ((object (make-instance (find-frame-class id) :id id :size size)))
      (read-object object stream)
      object)))
```

Так как раскрытия `define-tagged-binary-class` и `define-binary-class` будут идентичными за исключением метода чтения, мы можем вынести общие части во вспомогательный макрос `define-generic-binary-class`, который принимает метод чтения в качестве параметра и подставляет его в свое раскрытие.

```
(defmacro define-generic-binary-class (name (&rest superclasses) slots read-
method)
  (with-gensyms (objectvar streamvar)
    `(progn
      (eval-when (:compile-toplevel :load-toplevel :execute)
        (setf (get ',name 'slots) ',(mapcar #'first slots))
        (setf (get ',name 'superclasses) ',superclasses))
      (defclass ,name ,superclasses
        , (mapcar #'slot->defclass-slot slots))
      ,read-method
      (defmethod write-object progn ((,objectvar ,name) ,streamvar)
        (declare (ignorable ,streamvar))
        (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->write-value x streamvar)) slots))))))
```

Теперь мы можем определить `define-binary-class` и `define-tagged-binary-class` так, чтобы они раскрывались в вызов `define-generic-binary-class`. Вот новая версия `define-binary-class`, которая генерирует при полном своем раскрытии тот же код, что и более ранняя:

```
(defmacro define-binary-class (name (&rest superclasses) slots)
  (with-gensyms (objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
      (defmethod read-object progn ((,objectvar ,name) ,streamvar)
        (declare (ignorable ,streamvar))
        (with-slots ,(new-class-all-slots slots superclasses) ,objectvar
          ,@(mapcar #'(lambda (x) (slot->read-value x streamvar)) slots))))))
```

А вот `define-tagged-binary-class` вместе с двумя вспомогательными функциями, которые он использует:

```

(defmacro define-tagged-binary-class (name (&rest superclasses) slots &rest op-
tions)
  (with-gensyms (typevar objectvar streamvar)
    `(define-generic-binary-class ,name ,superclasses ,slots
      (defmethod read-value ((,typevar (eql ',name)) ,streamvar &key)
        (let* ,(mapcar #'(lambda (x) (slot->binding x streamvar)) slots)
          (let ((,objectvar
                  (make-instance
                   ,@(or (cdr (assoc :dispatch options))
                        (error "Must supply :dispatch form."))
                   ,@(mapcan #'slot->keyword-arg slots))))
            (read-object ,objectvar ,streamvar
                          ,objectvar))))))
  (defun slot->binding (spec stream)
    (destructuring-bind (name (type &rest args)) (normalize-slot-spec spec)
      `((,name (read-value ',type ,stream ,@args))))
  (defun slot->keyword-arg (spec)
    (let ((name (first spec)))
      `(, (as-keyword name) ,name)))

```

Примитивные двоичные типы

Хотя `define-binary-class` и `define-tagged-binary-class` делают легким определение сложных структур, мы все еще должны написать методы `read-value` и `write-value` для примитивных типов данных вручную. Мы могли бы смириться с этим, специфицировав, что пользователь библиотеки должен написать соответствующие методы `read-value` и `write-value` для поддержки примитивных типов, используемых его двоичными классами.

Однако, вместо документирования того, как написать подходящую пару методов `read-value/write-value`, мы можем предоставить макрос, делающий это автоматически. Это также даст пользу уменьшения "протечки" абстракции, созданной `define-binary-class`. Сейчас `define-binary-class` зависит от наличия методов `read-value` и `write-value`, определенных неким специфическим образом, который в действительности является деталью реализации. Определив макрос генерации методов `read-value` и `write-value` для примитивных типов мы скроем эти детали за управляемой нами абстракцией. Приняв позднее решение изменить реализацию `define-binary-class`, мы сможем изменить наш макрос определения примитивных типов так, что не потребуется вносить изменения в код, использующий нашу библиотеку двоичных данных.

Итак, мы должны определить последний макрос, `define-binary-type`, который будет генерировать методы `read-value` и `write-value` для чтения и записи значений, представляющих экземпляры уже существующих, а не определенных с помощью `define-binary-class`, классов.

В качестве конкретного примера рассмотрим тип, используемый классом `id3-tag`: строку знаков фиксированной длины, закодированную с помощью ISO-8859-1. Я подразумеваю, как делал это и раньше, что внутренней кодировкой вашей реализации Lisp является ISO-8859-1 или ее надмножество, так что вы можете использовать функции **CODE-CHAR** и **CHAR-CODE** для преобразования байт в знаки и обратно.

Как всегда, нашей целью является написание макроса, который позволит нам ограничиться выражением только самой существенной информации, необходимой для генерации требуемого кода. В данном случае есть четыре части такой существенной информации: имя типа, `iso-8859-1-string`; **&key** параметры, которые должны приниматься методами `read-value` и `write-value`, в нашем случае это `length`; код считывания из потока; код записи в поток. Вот

выражение, содержащее все четыре части информации:

```
(define-binary-type iso-8859-1-string (length)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in))))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out)))))
```

Теперь все, что нам нужно, так это макрос, осуществляющий разбор такой формы и преобразующий ее в две **DEFMETHOD**, обернутых в форму **PROGN**. Если мы определим список параметров `define-binary-type` следующим образом:

```
(defmacro define-binary-type (name (&rest args) &body spec) ...
```

то внутри макроса параметр `spec` будет списком, содержащим определения процедур чтения и записи. Вы можете использовать функцию **ASSOC** для извлечения элементов `spec` с помощью меток `:reader` и `:writer`, а затем **DESTRUCTURING-BIND** для разбора **REST**-части каждого элемента.

Остальная работа является Всего лишь делом подстановки извлеченных значений в шаблоны квазичитирования (backquoted templates) методов `read-value` и `write-value`.

```
(defmacro define-binary-type (name (&rest args) &body spec)
  (with-gensyms (type)
    `(progn
      ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
        `(defmethod read-value ((,type (eql ',name)) ,in &key ,@args)
          ,@body))
      ,(destructuring-bind ((out value) &body body) (rest (assoc :writer spec))
        `(defmethod write-value ((,type (eql ',name)) ,out ,value &key ,@args)
          ,@body)))))
```

Обратите внимание на вложенность шаблонов квазичитирования: самый внешний шаблон начинается с заковыченной формы **PROGN**. Этот шаблон состоит из символа **PROGN** и двух "раскавыченных" (comma-unquoted) выражений **DESTRUCTURING-BIND**. Таким образом, внешний шаблон заполняется путем вычисления выражений **DESTRUCTURING-BIND** и подстановки значений их результатов. Каждое выражение **DESTRUCTURING-BIND**, в свою очередь, также содержит шаблон квазичитирования, каждый из которых используется для генерации определения метода для подстановки его во внешний шаблон.

Теперь данная ранее форма `define-binary-type` раскрывается в такой код:

```
(progn
  (defmethod read-value ((#:g1618 (eql 'iso-8859-1-string)) in &key length)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (code-char (read-byte in))))
      string))
  (defmethod write-value ((#:g1618 (eql 'iso-8859-1-string)) out string &key
length)
    (dotimes (i length)
      (write-byte (char-code (char string i)) out)))))
```


Конечно, теперь, когда у нас есть такой хороший макрос для определения двоичных типов, привлекательной кажется мысль об улучшении этого макроса таким образом, чтобы он проделывал больше работы. Сейчас нам следует сделать лишь одно небольшое улучшение, которое окажется весьма полезным, когда мы начнем использовать нашу библиотеку при работе с реальными форматами файлов, такими как, например, теги ID3.

Теги ID3, подобно многим другим двоичным форматам, используют множество примитивных типов, являющихся небольшими вариациями на одну тему, такими как беззнаковые целые размерами один, два, три и четыре байта. Конечно же мы можем определить каждый такой тип с помощью `define-binary-type`. Но мы также можем вынести общий алгоритм чтения и записи *n*-байтных беззнаковых целых во вспомогательную функцию.

Но представим, что мы уже определили двоичный тип, `unsigned-integer`, который принимает параметр `:bytes` для указания того, как много байт нужно считывать и записывать. Используя этот тип мы можем определить слот, представляющий однобайтное целое, с помощью спецификатора `(unsigned-integer :bytes 1)`. Но, если определенный двоичный формат определяет множество слотов этого типа, было бы неплохо иметь возможность легкого определения нового типа, скажем `u1`, означающего то же, что и используемый тип. На самом деле, несложно изменить `define-binary-type` так, чтобы он поддерживал две формы: длинную форму, состоящую из пары `:reader` и `:writer`, и короткую форму, которая определяет новый двоичный тип в терминах уже существующего типа. Используя короткую форму `define-binary-type` вы можете определить `u1` следующим образом:

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
```

что раскроется в следующее:

```
(progn
  (defmethod read-value ((#:g161887 (eql 'u1)) #:g161888 &key)
    (read-value 'unsigned-integer #:g161888 :bytes 1))
  (defmethod write-value ((#:g161887 (eql 'u1)) #:g161888 #:g161889 &key)
    (write-value 'unsigned-integer #:g161888 #:g161889 :bytes 1)))
```

Для поддержки и длинной, и короткой форм `define-binary-type` нам нужно различать эти два случая основываясь на значении аргумента `спес`. Если `спес` содержит два элемента, он представляет длинную форму, а эти два элемента должны быть определениями `:reader` и `:writer`, извлечение которых было реализовано нами раньше. Если же этот аргумент содержит лишь один элемент, этот элемент должен быть спецификатором типа, разбор которого будет отличаться. Мы можем использовать **ECASE** для осуществления дифференциации по длине аргумента `спес` с целью дальнейшего осуществления разбора этого аргумента и генерации соответствующего форме (длинной или короткой) раскрытия.

```

(defmacro define-binary-type (name (&rest args) &body spec)
  (ecase (length spec)
    (1
     (with-gensyms (type stream value)
       (destructuring-bind (derived-from &rest derived-args) (mklist (first
spec))
         `(progn
            (defmethod read-value ((,type (eq1 ',name)) ,stream &key ,@args)
              (read-value ',derived-from ,stream ,@derived-args))
            (defmethod write-value ((,type (eq1 ',name)) ,stream ,value &key
,@args)
              (write-value ',derived-from ,stream ,value ,@derived-args))))))
    (2
     (with-gensyms (type)
       `(progn
          ,(destructuring-bind ((in) &body body) (rest (assoc :reader spec))
            `(defmethod read-value ((,type (eq1 ',name)) ,in &key ,@args)
              ,@body))
          ,(destructuring-bind ((out value) &body body) (rest (assoc :writer
spec))
            `(defmethod write-value ((,type (eq1 ',name)) ,out ,value &key
,@args)
              ,@body))))))

```

Стек обрабатываемых в данный момент объектов

Последней частью функциональности, которая нам понадобится в следующей главе, является возможность обращения к двоичному объекту, чтение или запись которого производится в данный момент. То есть, во время чтения или записи вложенных сложных объектов было бы удобно иметь возможность получения доступа к объекту, чтение или запись которого производится в данный момент. Благодаря динамическим переменным и методам `:around` мы можем добавить это улучшение, написав всего лишь около дюжины строк кода. Для начала нам нужно определить динамическую переменную, которая будет содержать стек объектов, чтение или запись которых осуществляется в данный момент.

```
(defvar *in-progress-objects* nil)
```

Затем мы можем определить методы `:around` для `read-object` и `write-object`, которые помещают объект, чтение или запись которого будет осуществляться, в определенную ранее переменную перед вызовом **CALL-NEXT-METHOD**.

```

(defmethod read-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))
(defmethod write-object :around (object stream)
  (declare (ignore stream))
  (let ((*in-progress-objects* (cons object *in-progress-objects*)))
    (call-next-method)))

```

Обратите внимание как мы пересвязали `*in-progress-objects*` со списком, содержащим новый элемент в своем начале, вместо присвоения ему нового значения. Поступив так, мы получили тот эффект, что в конце **LET**, после возврата из **CALL-NEXT-METHOD**, старое значение `*in-progress-objects*` будет восстановлено (то есть, последний помещенный в стек элемент будет из него удален).

Имея эти определения методов, мы можем предоставить две удобные функции для получения отдельных объектов из стека обрабатываемых объектов. Функция `current-binary-object` будет возвращать вершину стека, то есть объект, чей метод `read-object` или `write-object` был вызван наиболее недавно. Вторая, `parent-of-type`, получает аргумент, который должен быть именем класса двоичного типа, и возвращает наиболее недавно помещенный в стек объект данного типа, используя функцию **TYPEP**, которая проверяет, является ли переданный ей объект экземпляром определенного типа.

```
(defun current-binary-object () (first *in-progress-objects*))  
(defun parent-of-type (type)  
  (find-if #'(lambda (x) (typep x type)) *in-progress-objects*))
```

Эти две функции могут быть использованы из любого кода, который будет вызван в динамической протяженности вызова `read-object` или `write-object`. Мы увидим один пример использования `current-binary-object` в следующей главе.

Теперь у нас есть все инструменты, нужные для создания библиотеки разбора ID3, поэтому мы готовы перейти к следующей главе, где мы именно этим и займемся.

Практика: разбор ID3

Имея библиотеку для разбора двоичных данных, вы уже готовы к созданию кода для чтения и записи в каком-то реальном двоичном формате, например, формате тэгов ID3. Тэги ID3 используются для хранения дополнительной информации в звуковых файлах MP3. Работа с тэгами ID3 будет хорошей проверкой для библиотеки для работы с двоичными данными, потому что формат ID3 – это настоящий, используемый на практике, формат – смесь инженерных компромиссов и характерных решений, которые, тем не менее, выполняют своё назначение. На тот случай, если вы случайно пропустили революцию свободного обмена данными, вот краткий обзор того, что представляют собой тэги ID3, и как они относятся к файлам MP3.

MP3, или звуковой слой 3 для MPEG, – это формат для хранения сжатых звуковых данных, разработанный исследователями из Фраунгоферовского института интегральных схем и стандартизованный "Группой экспертов кино", объединённым комитетом организаций ISO и IEC. Однако, формат MP3 сам по себе определяет только то, как хранить звуковые данные. Это не страшно до тех пор, пока все ваши звуковые файлы обрабатываются каким-то одним приложением, которое может хранить эти метаданные вне звуковых файлов, сохраняя их связь со звуковыми файлами. Однако, как только люди стали обмениваться файлами MP3 через Интернет через такие файлообменные системы, как Napster, они быстро обнаружили, что нужно как-то вставлять метаданные внутрь самих файлов MP3.

Поскольку стандарт MP3 уже был оформлен, и значительная часть программного обеспечения и оборудования уже была написана и спроектирована, причём так, что они знали, как декодировать существующий формат файлов MP3, то любая схема внедрения информации в файл MP3 была бы такой, что эта информация вынуждено была бы невидима декодерам файлов MP3. Вот тут и появился ID3.

Первоначально формат ID3, изобретённый программистом Эриком Кэмпом, представлял собой 128 байт, прилепленных в конце файла MP3, где их не замечало бы большинство программ, работающих с MP3. Эта информация состояла из четырёх тридцатибуквенных полей, являвшихся названием песни, названием альбома, именем исполнителя и комментарием, одного четырёхбайтового поля года и одного однобайтового поля кода жанра произведения. Кэмп придумал стандартные значения первых 80-ти кодов жанров. Nullsoft, производитель программы Winamp, очень популярного MP3-плеера, позже добавили в этот список ещё что-то около 60 жанров.

Этот формат было легко разбирать, но он был достаточно ограничен. Не было способа сохранить названия более чем в 30 символов, было ограничение в 256 жанров, и значения кодов жанров должны были одинаково восприниматься всеми пользователями, использующими ID3. Не было даже способа сохранить номер дорожки на исходном диске до тех пор, пока другой программист, Микаэль Мутшлер, не предложил вставлять номер дорожки в поле комментария, отделяя его от остального комментария нулевым байтом, так, чтобы существующее ПО, использующее ID3, которое предположительно читало бы до первого нулевого символа в каждом текстовом поле, игнорировало бы его. Версия Кэмп теперь называется "ID3 версия 1" (ID3v1), а версия Мутшлера - "ID3 версия 1.1" (ID3v1.1)

Предложения первой версии, как бы ни ограничены они были, являлись хотя бы частичным решением проблемы хранения метаданных, так что они были применены многими программами копирования музыки, сохранявшими тэги ID3 в файлах MP3, и MP3-плеерами, вытаскивавшими эту информацию из тэгов ID3 и показывавшими их пользователю.

Однако, к 1998 году все эти ограничения стали совсем уже раздражающими, и новая группа разработчиков, возглавляемая Мартином Нильсоном, начала работу над совершенно новой

схемой хранения метаданных, которую ID3v2. Формат ID3v2 крайне гибок, разрешает включать много видов информации практически без ограничения длины. Также он берёт на вооружение некоторые особенности формата MP3 файла для того, чтобы разместить тэги ID3v2 в начале файла MP3.

Однако, разбирать тэги в формате ID3v2 – задача значительно более сложная, чем тэги в формате версии 1. В этой главе мы будем использовать библиотеку разбора бинарных данных из предыдущей главы для того, чтобы разработать код, который сможет читать и писать тэги в формате ID3v2. Ну или по крайней мере сделаем какое-то приемлимое начало, поскольку если ID3v1 достаточно прост, то ID3v2 порой причудлив до невозможности. Реализация всех закоулков и потаённых уголков спецификации была бы порядочно сложной работой, особенно если бы вы хотели поддержать все три версии, которые были документированы. На самом деле вы можете игнорировать многие возможности в этих спецификациях, поскольку они очень редко используются в "дикой природе". В качестве закуски вы можете опустить поддержку всей версии 2.4, поскольку она не была широко воспринята и в основном всего лишь добавляла некую вовсе не нужную гибкость по сравнению с версией 2.3. Я сконцентрируюсь на версии 2.2 и 2.3, потому что обе они широко используются и достаточно сильно отличаются друг от друга, чтобы сделать нашу работу интересной.

Структура тэга ID3v2.

До того, как начать кодировать, вам нужно познакомиться с общей структурой тэгов ID3v2. Каждый тэг начинается с заголовка, содержащего информацию о тэге в общем. Первые три байта заголовка содержат строку "ID3" в кодировке ISO-8859-1. То есть это байты с кодами 73, 68 и 51. Затем идут два байта, которые кодируют "старшую версию" и ревизию спецификации ID3, которой тэг намеревается соответствовать. Далее идёт один байт, биты которого интерпретируются как различные флаги. Значение каждого из флагов зависит от версии спецификации. Некоторые из флагов могут влиять на то, как обрабатывается весь тэг целиком. Байты "старшей версии" на самом деле используются для записи младшей версии спецификации, в то время как ревизия используется для хранения подверсии спецификации. Таким образом поле "старшая версия" тэга, соответствующего спецификации версии 2.3.0, будет 3. Поле ревизии всегда равно нулю, поскольку каждая новая спецификация ID3v2 увеличивала младшую версию, оставляя подверсию нулём. Значение, хранимое в поле старшей версии тэга, как вы увидите, имеет сильное влияние на то, как надо разбирать всю оставшуюся часть тэга.

Последнее поле в заголовке тэга – это число, закодированное в четырех байтах, в каждом из которых используется лишь по семь бит, содержащее размер всего тэга без учета заголовка. В тэгах версии 2.3 в заголовке может быть еще несколько дополнительных полей; все остальное – это данные, разделенные на фреймы. Разные типы фреймов хранят разные виды информации: от простого текста вроде названия песни до встроенного изображения. Каждый фрейм начинается с заголовка, содержащего строковый идентификатор и размер. В версии 2.3 заголовок фрейма также содержит два байта флагов и – при выставленном флаге – дополнительный однобайтовый код, указывающий, как закодирован остаток фрейма.

Фреймы – идеальный пример FIXME tagged структур данных: чтобы FIXME пропарсить тело фрейма, надо прочитать заголовок и использовать идентификатор, чтобы определить, какой вид данных ты читаешь.

Заголовок ID3 не указывает прямо, сколько фреймов в тэге – он говорит, насколько тот большой, но раз фреймы могут быть разной длины, единственным способом узнать количество фреймов будет прочитать их данные. К тому же размер, записанный в заголовке, может быть больше, чем реальное количество байтов в данных фреймов; после фреймов могут идти нули для

выравнивания под указанный размер. Это позволяет программам изменять тэг без переписывания всего MP3-файла.

Итак, наши главные задачи: чтение заголовка ID3; определение версии, 2.2 или 2.3; чтение данных всех фреймов до конца тэга или до блока выравнивания.

Defining a Package

Как и с другими библиотеками, которые мы разработали ранее, тот код, который мы напишем в этой главе, FIXME worth putting в отдельный пакет. Нам надо будет обращаться к функциям из библиотек `binary` и `pathname` из глав 15 и 24, и надо экспортировать имена функций, которые составляют API этого пакета. Определим его так:

```
(defpackage :com.gigamonkeys.id3v2
  (:use :common-lisp
        :com.gigamonkeys.binary-data
        :com.gigamonkeys.pathnames)
  (:export
   :read-id3
   :mp3-p
   :id3-p
   :album
   :composer
   :genre
   :encoding-program
   :artist
   :part-of-set
   :track
   :song
   :year
   :size
   :translated-genre))
```

Как обычно, вы можете, и наверное, вам даже следует заменить `"com.gigamonkeys"` в имени пакета на ваш собственный домен.

Integer Types

Можно начать с определения бинарных типов для чтения и записи некоторых примитивов FIXME(primitive types – слово типов повторяется два раза), использующихся в формате ID3, несколько целочисленных типов разного размера и четыре вида строк.

ID3 использует беззнаковые целые, закодированные в одном, двух, трех или четырех байтах. FIXME If you first write a general unsigned-integer binary type that takes the number of bytes to read as an argument, you can then use the short form of define-binary-type to define the specific types. The general unsigned-integer type looks like this:

```
(define-binary-type unsigned-integer (bytes)
  (:reader (in)
    (loop with value = 0
      for low-bit downfrom (* 8 (1- bytes)) to 0 by 8 do
        (setf (ldb (byte 8 low-bit) value) (read-byte in))
      finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* 8 (1- bytes)) to 0 by 8
      do (write-byte (ldb (byte 8 low-bit) value) out))))
```

Теперь можно пользоваться короткой формой `define-binary-type` для определения типов для каждого размера целого из формата ID3:

```
(define-binary-type u1 () (unsigned-integer :bytes 1))
(define-binary-type u2 () (unsigned-integer :bytes 2))
(define-binary-type u3 () (unsigned-integer :bytes 3))
(define-binary-type u4 () (unsigned-integer :bytes 4))
```

Еще один тип, который надо уметь читать и писать, это 28-ми битное значение из заголовка. Это размер, закодированный не как обычно – количеством бит, кратным 8, таким как 32 – а 28-ю `FIXME PUNCTUATION`, потому что тэг ID3 не может содержать байт `#xff`, за которым идут три включенных бита – такой `FIXME pattern` имеет особое значение для MP3-декодеров. В принципе, ни одно поле в заголовке ID3 не может содержать такую последовательность байтов, но если бы размер тэга был закодирован обычным беззнаковым целым, то были бы проблемы. Чтобы исключить такую возможность, размер кодируется в семи младших битах каждого байта, все старшие всегда нули.

Таким образом, оно может быть считано и записано во многом как беззнаковое целое, только размер байта, который передается в LDB, должен быть 7, а не 8. Это сходство наводит на мысль, что если добавить параметр `bits-per-byte` к существующему бинарному типу `unsigned-integer`, тогда можно определить новый тип `id3-tag-size`, используя короткую форму `define-binary-type`. Новая версия `unsigned-integer` такая же, как старая, только `bits-per-byte` заменяет прописанную везде в старой 8-ку. Выглядит так:

```
(define-binary-type unsigned-integer (bytes bits-per-byte)
  (:reader (in)
    (loop with value = 0
      for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte do
        (setf (ldb (byte bits-per-byte low-bit) value) (read-byte in))
      finally (return value)))
  (:writer (out value)
    (loop for low-bit downfrom (* bits-per-byte (1- bytes)) to 0 by bits-per-byte
      do (write-byte (ldb (byte bits-per-byte low-bit) value) out))))
```

Теперь определение `id3-tag-size` становится тривиальным:

```
(define-binary-type id3-tag-size () (unsigned-integer :bytes 4 :bits-per-byte 7))
```

Также надо изменить определения `u1–u4` для указания, что там 8 бит в байте:

```
(define-binary-type u1 () (unsigned-integer :bytes 1 :bits-per-byte 8))
(define-binary-type u2 () (unsigned-integer :bytes 2 :bits-per-byte 8))
(define-binary-type u3 () (unsigned-integer :bytes 3 :bits-per-byte 8))
(define-binary-type u4 () (unsigned-integer :bytes 4 :bits-per-byte 8))
```

String Types

Еще один из примитивных типов, который повсеместен в тэге ID3, это строка `FIXME PUNCTUATION`. В предыдущей главе мы обсудили некоторые вещи, на которые надо обратить внимание, когда имеешь дело со строками в бинарных файлах, такие как разница между кодом знака и кодировкой.

ID3 использует две разные кодировки: ISO 8859-1 и Unicode. ISO 8859-1, также известный как Latin-1, `FIXME PUNCTUATION` – это 8-ми битная кодировка, которая дополняет ASCII буквами

из языков Восточной Европы. Другими словами, одни и те же коды от 0 до 127 указывают на одни и те же знаки ASCII и ISO 8859-1, но ISO 8859-1 FIXME also provides mappings for code points up to 255. Unicode – это кодировка, сделанная, чтобы обеспечить кодом FIXME virtually каждый знак всех на свете языков. Unicode – надмножество ISO 8859-1 так же, как ISO 8859-1 – надмножество ASCII: коды 0-255 отображаются на одни и те же знаки ISO 8859-1 и Unicode. (Таким образом, Unicode еще и надмножество ASCII.FIXME PUNCTUATION)

Поскольку ISO 8859-1 является 8-ми битной кодировкой, она использует один байт на знак. Для Unicode-строк ID3 использует кодировку UCS-2 с меткой порядка байтов.

Чтение и запись этих двух кодировок не является проблемой – это всего лишь вопрос чтения и записи беззнаковых чисел в разных форматах, и мы только что написали код для этого. Трюк в том, чтобы перевести эти числовые значения в объекты знаков языка Lisp.

Ваша реализация Lisp возможно использует или Unicode, или ISO 8859-1 в качестве внутренней кодировки. И раз все значения от 0 до 255 отображаются на одни и те же знаки в ISO 8859-1 и Unicode, то можно использовать функции CODE-CHAR и CHAR-CODE для их транслирования в обе кодировки. Однако, если ваш Lisp поддерживает только ISO 8859-1, тогда можно будет FIXME represent only the first 255 Unicode characters as Lisp characters. Другими словами, в такой реализации Lisp, если вы попытаетесь обработать тэг ID3, который использует строки Unicode, и любая из этих строк содержит знак с кодом, большим 255, то вы получите ошибку, когда попытаетесь перевести этот код в Lisp FIXME character. Пока будем считать, что мы или используем Lisp, поддерживающий Unicode, или не будем работать с файлами, содержащими знаки вне досягаемости ISO 8859-1.

FIXME The other issue with encoding strings is how to know how many bytes to interpret as character data. ID3 использует две стратегии, рассмотренные в предыдущей главе: некоторые строки заканчиваются нулевым символом, тогда как другие встречаются на позициях, по которым можно определить количество байт для считывания: или когда строка в том расположении всегда одной длины, или когда она в конце составной структуры, чей размер известен. Тем не менее обратите внимание, что количество байт не обязательно совпадает с количеством знаков в строке.

Складывая все эти варианты вместе, получим, что формат ID3 использует четыре способа чтения и записи строк: два вида знаков на два вида разграничения строковых данных.

Очевидно, значительная часть логики чтения и записи строк будет полностью совпадать. Так что, можно начать с определения двух бинарных типов: один для чтения строк заданной длины (в знаках) FIXME(не понял, почему в знаках, а не в байтах) и другой для чтения FIXME terminated строк. Оба пользуются тем, что тип, передаваемый в read-value и write-value, это такие же данные; FIXME you can make the type of character to read a parameter of these types. Этой техникой мы будем пользоваться довольно часто в этой главе.


```
(define-binary-type generic-string (length character-type)
  (:reader (in)
    (let ((string (make-string length)))
      (dotimes (i length)
        (setf (char string i) (read-value character-type in)))
      string))
  (:writer (out string)
    (dotimes (i length)
      (write-value character-type out (char string i)))))
(define-binary-type generic-terminated-string (terminator character-type)
  (:reader (in)
    (with-output-to-string (s)
      (loop for char = (read-value character-type in)
        until (char= char terminator) do (write-char char s))))
  (:writer (out string)
    (loop for char across string
      do (write-value character-type out char)
      finally (write-value character-type out terminator))))
```

С этими типами несложно будет прочитать строки ISO 8859-1. Поскольку character-type, который передается в read-value и write-value должен быть именем бинарного типа, то надо определить iso-8859-1-char. Здесь же неплохо разместить немного FIXME sanity checking on the code points of characters you read and write.

```
(define-binary-type iso-8859-1-char ()
  (:reader (in)
    (let ((code (read-byte in)))
      (or (code-char code)
          (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (if (<= 0 code #xff)
          (write-byte code out)
          (error "Illegal character for iso-8859-1 encoding: character: ~c with
code: ~d" char code)))))
```

Теперь определение строк ISO 8859-1 становится тривиальным:

```
(define-binary-type iso-8859-1-string (length)
  (generic-string :length length :character-type 'iso-8859-1-char))
(define-binary-type iso-8859-1-terminated-string (terminator)
  (generic-terminated-string :terminator terminator :character-type 'iso-8859-1-
char))
```

Чтение строк UCS-2 лишь немногим сложнее. Трудности возникают из-за того, что можно кодировать UCS-2 двумя способами: в порядке байтов от старшего к младшему (big-endian) или от младшего к старшему (little-endian). Поэтому строки UCS-2 начинаются с двух дополнительных байтов, которые называются меткой порядка байтов, состоящих из числового значения #xfeff, закодированных или в порядке big-endian, или в little-endian. При чтении строки UCS-2, надо прочитать метку порядка байтов, а потом, в зависимости от ее значения, читать знаки в порядке big-endian или в little-endian. Так что понадобится два разных типа знаков UCS-2. Но нужна только одна версия FIXME sanity-checking code. Значит можно определить параметризованный бинарный тип:

```
(define-binary-type ucs-2-char (swap)
  (:reader (in)
    (let ((code (read-value 'u2 in)))
      (when swap (setf code (swap-bytes code)))
      (or (code-char code) (error "Character code ~d not supported" code))))
  (:writer (out char)
    (let ((code (char-code char)))
      (unless (<= 0 code #xffff)
        (error "Illegal character for ucs-2 encoding: ~c with char-code: ~d"
char code))
      (when swap (setf code (swap-bytes code)))
      (write-value 'u2 out code))))
```

где функция `swap-bytes` определена ниже, с использованием преимущества функции `LDB`, с которой можно делать `SETF` и, соответственно, `ROTATEF`. `FIXME` второй вариант чуть более неформальный: где функция `swap-bytes` определена ниже, с использованием преимущества `LDB`, которую можно `SETF`ить и, соответственно, `ROTATEF`ить.

where the `swap-bytes` function can be defined as follows, taking advantage of `LDB` being `SETFable` and thus `ROTATEFable`:

```
(defun swap-bytes (code)
  (assert (<= code #xffff))
  (rotatef (ldb (byte 8 0) code) (ldb (byte 8 8) code))
  code)
```

Используя `ucs-2-char`, определим два типа знаков, которые будут использоваться в качестве аргумента `character-type` функций обобщенных строк.

```
(define-binary-type ucs-2-char-big-endian () (ucs-2-char :swap nil))
(define-binary-type ucs-2-char-little-endian () (ucs-2-char :swap t))
```

Затем нужна функция, которая возвращает тип знаков, которые будут использоваться в зависимости от метки порядка байтов.

```
(defun ucs-2-char-type (byte-order-mark)
  (ecase byte-order-mark
    (#xfeff 'ucs-2-char-big-endian)
    (#xfffe 'ucs-2-char-little-endian)))
```

Now you can define `length-` and `terminator-delimited` string types for UCS-2-encoded strings которые читают метку порядка байтов и определяют, какой вариант знаков UCS-2 передавать в качестве аргумента `character-type` в `read-value` и `write-value`. `FIXME` The only other wrinkle is, что надо переводить аргумент `length`, который дан в байтах, в количество знаков, что надо прочесть, учитывая метку порядка байтов.

```

(define-binary-type ucs-2-string (length)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in))
          (characters (1- (/ length 2))))
      (read-value
        'generic-string in
        :length characters
        :character-type (ucs-2-char-type byte-order-mark))))
  (:writer (out string)
    (write-value 'u2 out #xfeff)
    (write-value
      'generic-string out string
      :length (length string)
      :character-type (ucs-2-char-type #xfeff))))
(define-binary-type ucs-2-terminated-string (terminator)
  (:reader (in)
    (let ((byte-order-mark (read-value 'u2 in)))
      (read-value
        'generic-terminated-string in
        :terminator terminator
        :character-type (ucs-2-char-type byte-order-mark))))
  (:writer (out string)
    (write-value 'u2 out #xfeff)
    (write-value
      'generic-terminated-string out string
      :terminator terminator
      :character-type (ucs-2-char-type #xfeff))))

```

ID3 Tag Header

Закончив с основными примитивными типами, мы готовы перейти к более общей картине и начать определять бинарные классы для представления сначала тэга ID3 в целом, а потом и отдельных фреймов.

Если заглянуть в спецификацию ID3v2.2, то мы увидим, что в основе структуры тэга такой заголовок:

```

ID3/file identifier "ID3"
ID3 version $02 00
ID3 flags %xx000000
ID3 size 4 * %0xxxxxxx

```

за которым идут данные фреймов и выравнивание. Поскольку мы уже определили типы для чтения и записи всех полей в этом заголовке, определение класса, который сможет читать заголовок ID3, это всего лишь вопрос их объединения.

```

(define-binary-class id3-tag ()
  ((identifier (iso-8859-1-string :length 3))
   (major-version u1)
   (revision u1)
   (flags u1)
   (size id3-tag-size)))

```

Если у вас под рукой есть какой-нибудь MP3-файл, вы можете проверить всю эту кучу кода и заодно посмотреть, какую версию тэга ID3 он содержит. Для начала напишем функцию, которая считывает только что определенный id3-tag из начала файла. Надо понимать, тем не менее, что тэг ID3 не обязан находиться в начале файла, хотя в наши дни он почти всегда там. Чтобы

найти тэг ID3 где-то еще в файле, последний можно просканировать в поисках последовательности байтов 73, 68, 51 (другими словами, это строка "ID3"). Правда, сейчас уже, наверное, можно считать, что файлы начинаются с тэгов.

```
(defun read-id3 (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (read-value 'id3-tag in)))
```

На основе этой функции можно написать другую, которая получает имя файла и печатает информацию из заголовка тэга вместе с именем файла.

```
(defun show-tag-header (file)
  (with-slots (identifier major-version revision flags size) (read-id3 file)
    (format t "~a ~d.~d ~8,'0b ~d bytes -- ~a~%"
            identifier major-version revision flags size (enough-namestring
file))))
```

Она выдаст примерно следующее:

```
ID3V2> (show-tag-header "/usr2/mp3/Kitka/Wintersongs/02 Byla Cesta.mp3")
ID3 2.0 00000000 2165 bytes -- Kitka/Wintersongs/02 Byla Cesta.mp3
NIL
```

Конечно, чтобы определить, какая версия ID3 встречается чаще всего в вашей библиотеке, лучше бы иметь функцию, которая выдает сводку по всем MP3-файлам в директории. Такую легко реализовать с помощью функции `walk-directory` из главы 15. Для начала определим вспомогательную функцию, которая проверяет, что у файла расширение MP3.

```
(defun mp3-p (file)
  (and
    (not (directory-pathname-p file))
    (string-equal "mp3" (pathname-type file))))
```

Затем соединим `show-tag-header`, `mp3-p` с `walk-directory`, чтобы печатать сводку по заголовкам ID3 в файлах в заданной директории.

```
(defun show-tag-headers (dir)
  (walk-directory dir #'show-tag-header :test #'mp3-p))
```

Однако, если у вас много MP3-файлов, вы можете пожелать просто посчитать, сколько тэгов ID3 каждой версии у вас в MP3 коллекции. Для получения этой информации, можно было бы написать такую функцию:

```
(defun count-versions (dir)
  (let ((versions (mapcar #'(lambda (x) (cons x 0)) '(2 3 4))))
    (flet ((count-version (file)
              (incf (cdr (assoc (major-version (read-id3 file)) versions)))))
      (walk-directory dir #'count-version :test #'mp3-p)
      versions)))
```

Другая функция, которая понадобится в главе 29, для проверки, что файл действительно начинается с тэга ID3, которую можно определить вот так:

```
(defun id3-p (file)
  (with-open-file (in file :element-type '(unsigned-byte 8))
    (string= "ID3" (read-value 'iso-8859-1-string in :length 3))))
```

ID3 Frames

As I discussed earlier, основная часть тэга ID3 разделена на фреймы. Каждый фрейм имеет структуру, похожую на структуру всего тэга. Каждый фрейм начинается с заголовка, указывающего вид фрейма и размер фрейма в байтах. Структура заголовка фрейма немного разная у версий 2.2 и 2.3 формата ID3, и так получилось, что нам придется работать с обеими формами. Для начала сфокусируемся на разборе версии 2.2.

Заголовок в версии 2.2 состоит из трех байт, которые кодируют трехбуквенную ISO 8859-1 строку, за которой идет трехбайтовое беззнаковое число, FIXME which specifies размер фрейма в байтах без шестибайтового заголовка. Строка указывает тип фрейма, что определяет, как мы будем разбирать данные. Это как раз та ситуация, для которой мы определили макрос `define-tagged-binary-class`. Мы можем определить FIXME tagged класс, который читает заголовок фрейма и затем подбирает подходящий конкретный класс, используя функцию, которая отображает ID на имя класса.

```
(define-tagged-binary-class id3-frame ()
  ((id (iso-8859-1-string :length 3))
   (size u3))
  (:dispatch (find-frame-class id)))
```

Now you're ready to start implementing concrete frame classes. Теперь мы готовы начать строить реализацию конкретных классов фреймов. However, спецификация определяет FIXME quite a few – 63 в версии 2.2 и FIXME even more в более поздних версиях. Даже считая типы фреймов, которые имеют общую структуру, эквивалентными, мы все еще получим 24 уникальных типа в версии 2.2. Но только несколько из них используется на практике. Так что, вместо того, чтобы сразу приступить So rather than immediately setting to work defining classes for each of the frame types, you can start by writing a generic frame class that lets you read the frames in a tag without parsing the data within the frames themselves. This will give you a way to find out what frames are actually present in the MP3s you want to process. You'll need this class eventually anyway because the specification allows for experimental frames that you'll need to be able to read without parsing.

Since the size field of the frame header tells you exactly how many bytes long the frame is, you can define a generic-frame class that extends `id3-frame` and adds a single field, `data`, that will hold an array of bytes.

```
(define-binary-class generic-frame (id3-frame)
  ((data (raw-bytes :size size))))
```

The type of the data field, `raw-bytes`, just needs to hold an array of bytes. You can define it like this:

```
(define-binary-type raw-bytes (size)
  (:reader (in)
    (let ((buf (make-array size :element-type '(unsigned-byte 8))))
      (read-sequence buf in)
      buf))
  (:writer (out buf)
    (write-sequence buf out)))
```

For the time being, you'll want all frames to be read as generic-frames, so you can define the find-frame-class function used in id3-frame's :dispatch expression to always return generic-frame, regardless of the frame's id.

```
(defun find-frame-class (id)
  (declare (ignore id))
  'generic-frame)
```

Now you need to modify id3-tag so it'll read frames after the header fields. There's only one tricky bit to reading the frame data: although the tag header tells you how many bytes long the tag is, that number includes the padding that can follow the frame data. Since the tag header doesn't tell you how many frames the tag contains, the only way to tell when you've hit the padding is to look for a null byte where you'd expect a frame identifier.

To handle this, you can define a binary type, id3-frames, that will be responsible for reading the remainder of a tag, creating frame objects to represent all the frames it finds, and then skipping over any padding. This type will take as a parameter the tag size, which it can use to avoid reading past the end of the tag. But the reading code will also need to detect the beginning of the padding that can follow the tag's frame data. Rather than calling read-value directly in id3-frames :reader, you should use a function read-frame, which you'll define to return NIL when it detects padding, otherwise returning an id3-frame object read using read-value. Assuming you define read-frame so it reads only one byte past the end of the last frame in order to detect the start of the padding, you can define the id3-frames binary type like this:

```
(define-binary-type id3-frames (tag-size)
  (:reader (in)
    (loop with to-read = tag-size
      while (plusp to-read)
        for frame = (read-frame in)
        while frame
          do (decf to-read (+ 6 (size frame)))
          collect frame
        finally (loop repeat (1- to-read) do (read-byte in))))
  (:writer (out frames)
    (loop with to-write = tag-size
      for frame in frames
      do (write-value 'id3-frame out frame)
      (decf to-write (+ 6 (size frame)))
      finally (loop repeat to-write do (write-byte 0 out)))))
```

You can use this type to add a frames slot to id3-tag.

```
(define-binary-class id3-tag ()
  ((identifier (iso-8859-1-string :length 3))
   (major-version u1)
   (revision u1)
   (flags u1)
   (size id3-tag-size)
   (frames (id3-frames :tag-size size))))
```

26. Практика. Web-программирование с помощью AllegroServe

В этой главе вы ознакомитесь с одним из способов разработки Web-приложений на Common Lisp: используя AllegroServe – Web-сервер с открытым исходным кодом. Это не означает, что вы найдете здесь исчерпывающее введение в AllegroServe. И я определенно не собираюсь описывать ничего более чем небольшую часть огромной темы Web-программирования. Моей целью является описание достаточного количества базовых вещей по части использования AllegroServe, которые позволят нам в главе 29 разработать приложение для просмотра библиотеки MP3-файлов и проигрывания их на MP3-клиенте. Кроме того, данная глава может служить кратким введением в Web-программирование для новичков.

30-секундное введение в Web-программирование на стороне сервера

Хотя в настоящее время Web-программирование обычно означает использование одного из доступных программных каркасов (frameworks) и различных протоколов, основы Web-программирования не особо изменились с момента их появления в начале 1990-х. Для простых приложений, таких как мы напишем в главе 29, вам необходимо понять только несколько основных концепций, так что в этом разделе я сделаю их быстрый обзор. Опытные Web-программисты могут лишь просмотреть, а то и вовсе пропустить этот раздел.

Для начала вам необходимо понимание ролей Web-браузера и Web-сервера в Web-программировании. Хотя современные браузеры поставляются с кучей свистелок и дуделок (FIXME bells and whistles), основной функциональностью Web-браузера является запрос Web-страниц с Web-сервера и их отображение. Обычно эти страницы пишутся на Hypertext Markup Language (HTML, язык разметки гипертекста), который указывает браузеру как отображать страницу, включая информацию о том, где вставить изображения и ссылки на другие страницы. HTML состоит из текста, *размеченного* с помощью *тегов*, которые структурируют текст, а эту структуру браузер использует при отображении страницы. Например, простой HTML-документ выглядит вот так:

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
    <p>This is a picture: </p>
    <p>This is a <a href="another-page.html">link</a> to another page.</p>
  </body>
</html>
```

Рисунок 26-1 показывает как браузер отображает эту страницу.

Рисунок 26-1. Пример Web-страницы

Браузер и сервер общаются между собой используя протокол, называемый Hypertext Transfer Protocol (HTTP, протокол передачи гипертекста). Хотя вам не нужно беспокоиться относительно деталей протокола, полезным будет понимание того, что он полностью состоит из последовательности запросов, инициированных браузером, и ответов, сгенерированных сервером. Таким образом, браузер подключается к Web-серверу и посылает запрос, который включает в себя, как минимум, адрес желаемого ресурса (URL) и версию протокола HTTP, используемую

браузером. Браузер также может включать в запрос дополнительные данные; таким образом браузер отправляет HTML-формы на сервер.

Для ответа на запрос сервер отправляет ответ, состоящий из наборов заголовков и тела ответа. Заголовки содержат информацию о теле, такую как тип его данных (например, HTML, обычный текст или изображение), а тело ответа содержит сами данные, которые затем отображаются браузером. Сервер также может отправлять ответ-ошибку, который сообщит браузеру о том, что его запрос не может быть обработан по некоторой причине.

И это почти все. После того, как браузер получил завершенный ответ от сервера, между сервером и браузером не происходит никакого общения до тех пор, пока браузер не решит запросить страницу у сервера в следующий раз. Это основное ограничение Web-программирования – для кода, выполняемого на сервере, не существует способа воздействовать на то, что пользователь увидит в браузере, до тех пор, пока браузер не сделает новый запрос к серверу.

Некоторые Web-страницы, называемые *статическими* (*static*) страницами, являются просто файлами с разметкой на языке HTML, хранимые на Web-сервере и считываемые, когда приходит соответствующий запрос. *Динамические* (*dynamic*) страницы, с другой стороны, состоят из HTML, генерируемого при каждом запросе страницы браузером. Например, динамическая страница может быть сгенерирована путем осуществления запроса к базе данных, а затем конструирования HTML для представления результатов этого запроса.

При генерировании ответа на запрос код, выполняемый на сервере, получает четыре основных части информации, на основании которой он работает. Первой является запрошенный адрес (URL). Но обычно URL используется самим Web-сервером для определения того, какой код ответственен за генерирование ответа. Затем, если URL содержит знак вопроса, то все, что следует за ним, рассматривается как *строка запроса* (*query string*), которая обычно игнорируется самим Web-сервером и передается им коду, который будет генерировать ответ. В большинстве случаев строка запроса содержит набор пар имя-значение. Запрос от браузера может также содержать *POST-данные*, которые также обычно состоят из пар имя-значение. POST-данные обычно используются для передачи содержимого форм HTML. Пары имя-значение, переданные либо через строку запроса, либо через дополнительные данные, обычно называют *параметрами запроса* (*query parameters*).

В заключение, для того, чтобы связать между собой последовательность отдельных запросов от одного и того же браузера, код, выполняющийся на сервере, может установить *cookie* путем отправки специального заголовка в своем ответе браузеру; этот заголовок будет содержать некий набор данных. После установки cookie браузер будет слать его при каждом запросе, отправляемом этому серверу. Браузер не заботится о данных, хранимых в cookie, — он просто отправляет их обратно на сервер, и код, работающий там, может обрабатывать их так, как захочет.

Это все базовые элементы, на основании которых основано 99 процентов кода, выполняемого на Web-сервере. Браузер отправляет запрос, сервер находит код, который будет обрабатывать запрос, и запускает его, а код использует параметры запроса и cookies для определения того, что именно нужно сделать.

AllegroServe

Вы можете отдавать Web-страницы с помощью Common Lisp разными способами; существует по крайней мере три реализации Web-серверов с открытым исходным кодом, написанных на Common Lisp, а также подключаемые модули, такие как `mod_lisp` и `Lisplets`, которые позволяют Web-серверу Apache или любому контейнеру Java Servlet делегировать обработку запросов серверу

Lisp, работающему в отдельном процессе.

В этой главе мы будем использовать Web-сервер с открытым исходным кодом AllegroServe, изначально написанный John Foderaro из Franz Inc. AllegroServe включен в версию Allegro, доступную с сайта Franz для использования с этой книгой. Если вы не используете Allegro, то вы можете использовать PortableAllegroServe, ответвление (fork) кода AllegroServe, которое включает в себя уровень (layer) совместимости с Allegro, что позволяет PortableAllegroServe работать почти на всех реализациях Common Lisp. Код, который мы напишем в этой главе и в главе 29, должен работать как на стандартном AllegroServe, так и на PortableAllegroServe.

AllegroServe реализует модель программирования сходную по духу с Java Servlets – каждый раз, когда браузер запрашивает страницу, AllegroServe разбирает запрос и ищет объект, называемый *сущностью* (*entity*), который будет обрабатывать запрос. Некоторые классы сущностей, предоставляемые как часть AllegroServe, умеют обрабатывать статическое содержимое: либо отдельные файлы, либо содержимое каталога. Другие, которые я буду обсуждать большую часть главы, запускают произвольный код на Lisp для генерирования ответа.

Но перед тем как начать, вам необходимо знать, как запускать AllegroServe и как настроить его для обработки файлов. Первым шагом является загрузка кода AllegroServe в ваш образ Lisp. В Allegro вы можете просто набрать (`require :aserve`). В других реализациях Lisp (а также в Allegro), вы можете загрузить PortableAllegroServe путем загрузки файла `INSTALL.lisp`, находящегося в корне каталога `portableaserve`. Загрузка AllegroServe создаст три новых пакета: `NET.ASERVE`, `NET.HTML.GENERATOR` и `NET.ASERVE.CLIENT`.

После загрузки сервера, вы можете запустить его с помощью функции `start` из пакета `NET.ASERVE`. Чтобы иметь простой доступ к символам, экспортированным из пакета `NET.ASERVE`, из пакета `COM.GIGAMONKEYS.HTML` (который мы скоро обсудим), а также остальных частей Common Lisp, нам нужно создать новый пакет:

```
CL-USER> (defpackage :com.gigamonkeys.web
            (:use :cl :net.aserve :com.gigamonkeys.html))
#<The COM.GIGAMONKEYS.WEB package>
```

Теперь переключитесь на этот пакет с помощью следующего выражения `IN-PACKAGE`:

```
CL-USER> (in-package :com.gigamonkeys.web)
#<The COM.GIGAMONKEYS.WEB package>
WEB>
```

Теперь мы можем использовать имена, экспортированные из `NET.ASERVE`, без указания квалификатора. Функция `start` запускает сервер. Она принимает множество именованных параметров, но единственный нужный нам — `:port`, который указывает номер порта, на котором сервер будет принимать запросы. Возможно вам понадобится использовать большой номер порта, такой как 2001, вместо порта по умолчанию для HTTP-серверов, 80, поскольку в Unix-подобных операционных системах только администратор (`root`) может использовать порты с номером меньше 1024. Для запуска AllegroServe на порту 80 под Unix вам необходимо запустить Lisp с правами администратора (`root`), а затем использовать параметры `:setuid` и `:setgid` чтобы заставить `start` переключить пользовательский контекст после открытия этого порта. Вы можете запустить сервер на порту 2001 с помощью следующей команды:

```
WEB> (start :port 2001)
#<WSERVER port 2001 @ #x72511c72>
```

Теперь сервер выполняется в вашей среде Lisp. Возможно, что при попытке запуска сервера вы получите ошибку вида "port already in use". Это означает, что данный порт уже используется каким-то сервером на вашей машине. В таком случае самым простым решением будет использование другого порта, передав другой аргумент функции `start`, а затем использование нового значения во всех адресах, встречаемых на протяжении данной главы.

Вы можете продолжить взаимодействие с Lisp с помощью REPL, поскольку AllegroServe запускает отдельные нити для обработки запросов браузеров. Это означает, среди прочего, что вы можете использовать REPL для того, чтобы заглянуть во "внутренности" сервера во время его работы, что делает тестирование и отладку намного более легкой, по сравнению с тем, когда сервер представляет собой "черный ящик".

Предполагая, что вы запустили Lisp на той же машине, где находится и ваш браузер, вы можете проверить, что сервер запущен путем перехода в браузере по следующему адресу: <http://localhost:2001/>. В данный момент вы получите в браузере сообщение об ошибке `page-not-found` (страница не найдена), поскольку вы пока ничего не опубликовали. Но сообщение об ошибке придет от AllegroServe; это видно по строке внизу страницы. С другой стороны, если браузер отображает ошибку, сообщающую что-то вроде "The connection was refused when attempting to contact localhost:2001", то это означает, что сервер не запущен, или вы запустили его на порту с номером, отличным от 2001.

Теперь мы можем публиковать файлы. Предположим, что у нас есть файл `hello.html` в каталоге `/tmp/html` со следующим содержимым:

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

Вы можете опубликовать его с помощью функции `publish-file`.

```
WEB> (publish-file :path "/hello.html" :file "/tmp/html/hello.html")
#<NET.ASERVE::FILE-ENTITY @ #x725eddea>
```

Аргумент `:path` задает путь, который будет использоваться в URL, запрашиваемом браузером, а аргумент `:file` является именем файла на файловой системе. После вычисления выражения `publish-file` вы можете задать в браузере адрес <http://localhost:2001/hello.html>, и он должен отобразить что-то наподобие того, что изображено на рисунке 26-2.

Рисунок 26-2. <http://localhost:2001/hello.html>

Вы также можете опубликовать целый каталог с помощью функции `publish-directory`. Но сначала давайте избавимся от уже опубликованной сущности с помощью следующего вызова `publish-file`:

```
WEB> (publish-file :path "/hello.html" :remove t)
NIL
```

Теперь вы можете опубликовать каталог `/tmp/html/` целиком (включая его подкаталоги) с помощью функции `publish-directory`.

```
WEB> (publish-directory :prefix "/" :destination "/tmp/html/")
#<NET.ASERVE::DIRECTORY-ENTITY @ #x72625aa2>
```

В данном случае, аргумент `:prefix` указывает начало пути адресов URL, которые будут обрабатываться данной сущностью. Так что, если сервер получает запрос <http://localhost:2001/foo/bar.html>, то путь будет `/foo/bar.html`, который начинается с `/`. Затем этот путь транслируется в имя файла путем замены префикса (`/`) на аргумент `:destination` (`/tmp/html/`). Так что URL <http://localhost:2001/hello.html> будет преобразован в запрос файла `/tmp/html/hello.html`.

Генерирование динамического содержимого с помощью AllegroServe

Публикация сущностей, генерирующих динамическое содержимое, практически также проста, как и публикация статического содержимого. Функции `publish` и `publish-prefix` являются "динамическими" аналогами `publish-file` и `publish-directory`. Основная их идея заключается в том, что вы публикуете функцию, которая будет вызываться для генерирования ответа на запрос либо к определенному адресу (URL), либо к любому адресу с заданным префиксом. Эта функция будет вызвана с двумя аргументами: объектом, представляющим запрос, и опубликованной сущностью. Большую часть времени нам не нужно будет ничего делать с опубликованной сущностью за исключением ее передачи набору макросов, которые вскоре будут описаны. С другой стороны, мы будем использовать объект запроса для получения информации, переданной браузером: параметров запроса, переданных в строке URL, или данных, посланных формами HTML.

В качестве простого примера использования функции для генерирования динамического содержимого, давайте напишем функцию, которая будет генерировать страницу с различными случайными числами при каждом обращении к ней.

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (format
        (request-reply-stream request)
        "<html>~@
        <head><title>Random</title></head>~@
        <body>~@
        <p>Random number: ~d</p>~@
        </body>~@
        </html>~@"
        "
        (random 1000))))))
```

Макросы `with-http-response` и `with-http-body` являются частью AllegroServe. Первый из макросов начинает процесс генерирования ответа HTTP и может быть использован, также как в нашем примере, для указания таких вещей, как тип возвращаемых данных. Он также обрабатывает различные требования, указанные в стандарте HTTP, такие как обработка запросов `If-Modified-Since`. Макрос же `with-http-body` фактически отправляет заголовки HTTP ответа, а затем вычисляет свое тело, которое должно содержать код, генерирующий содержимое ответа. Внутри `with-http-response`, но перед `with-http-body`, вы можете добавить или изменить значение заголовков HTTP, которые будут отправлены в ответе. Функция `request-reply-stream` также является частью AllegroServe и возвращает поток, в который вы должны записывать данные предназначенные для отправления браузеру.

Как видно из этой функции, вы можете просто использовать **FORMAT** для вывода HTML в поток, возвращенный вызовом `request-reply-stream`. В следующем разделе я покажу вам более удобные способы программного генерирования HTML.

Теперь мы готовы к публикации данной функции.

```
WEB> (publish :path "/random-number" :function 'random-number)
#<COMPUTED-ENTITY @ #x7262bab2>
```

Также, как и в функции `publish-file`, аргумент `:path` указывает "путевую часть" (path part) адреса, указание которой будет приводить к вызову данной функции. Аргумент `:function` указывает либо имя функции, либо сам функциональный объект. Использование имени функции, как показано в этом примере, позволяет вам в дальнейшем переопределить функцию не выполняя заново процесс публикации для того, чтобы AllegroServe стал использовать новое определение. После вычисления данного вызова вы можете указать в браузере адрес <http://localhost:2001/random-number> для получения страницы со случайным числом, как это показано на рисунке 26-3.

Рисунок 26-3. <http://localhost:2001/random-number>

Генерирование HTML

Хотя использование **FORMAT** для генерирования HTML вполне приемлемо для генерирования простых страниц, наподобие приведенной выше, по мере того, как вы начнете создавать более сложные, было бы лучше иметь более краткий способ генерирования HTML. Для генерирования HTML из представления в виде s-выражений существует несколько библиотек, включая `htmlgen`, которая поставляется вместе с AllegroServe. В этой главе мы будем использовать библиотеку `FOO`, которая использует примерно ту же модель, что и `htmlgen`, и чью реализацию мы рассмотрим более подробно в главах 30 и 31. Сейчас, однако, нам нужно лишь знать как использовать `FOO`.

Генерирование HTML из Lisp вполне естественна, так как s-выражения и HTML по своему существу изоморфны. Мы можем представить HTML-элементы с помощью s-выражений, рассматривая каждый элемент HTML как список, "помеченный" соответствующим первым элементом, таким как ключевым символом с таким же именем, что имеет тег HTML. Таким образом, HTML `<p>foo</p>` представляется s-выражением `(:p "foo")`. Так как элементы HTML также, как и списки в s-выражениях, могут быть вложенными, эта схема распространяется и на более сложный HTML. Для примера вот такой HTML:

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

может быть представлен в виде следующего s-выражения:

```
(:html
  (:head (:title "Hello"))
  (:body (:p "Hello, world!")))
```

Элементы HTML с атрибутами несколько усложняют дело, но не создают непреодолимых проблем. FOO предоставляет два способа включения атрибут в тег. Одним из них является добавление после первого элемента списка пар ключ-значение. Первый же элемент, который следует за парами ключ-значение, и который сам не является ключевым символом, обозначает начало содержимого элемента. Таким образом, такой HTML:

```
<a href="foo.html">This is a link</a>
```

будет представляться следующим s-выражением:

```
(:a :href "foo.html" "This is a link")
```

Другой предоставляемый FOO синтаксис заключается в группировке имени тега и его атрибутов в отдельный список следующим образом:

```
((:a :href "foo.html") "This is link.")
```

FOO может использовать представление HTML в виде s-выражений двумя способами. Функция `emit-html` получает представляющее HTML s-выражение и выводит соответствующий HTML.

```
WEB> (emit-html '(:html (:head (:title "Hello")) (:body (:p "Hello, world!"))))
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
T
```

Однако, `emit-html` не всегда является наиболее эффективным способом генерирования HTML, так как ее аргументом должно быть законченное s-выражение, представляющее HTML, который нужно сгенерировать. Хотя генерировать такое представление легко, действовать так не всегда эффективно. Например, представим, что нам нужно сгенерировать страницу HTML, содержащую список из 10,000 случайных чисел. Мы можем построить соответствующее s-выражение используя шаблон квазигенерации, а затем передать его в функцию `emit-html`:

```
(emit-html
  `(:html
    (:head
      (:title "Random numbers"))
    (:body
      (:h1 "Random numbers")
      (:p ,@(loop repeat 10000 collect (random 1000) collect " "))))
```

Однако этот код должен построить дерево, содержащее 10000-элементный список, перед тем как он сможет даже начать генерировать HTML, и все это s-выражение станет мусором как только HTML будет сгенерирован. Для избежания такой неэффективности FOO также предоставляет макрос `html`, позволяющий вам встраивать произвольный код Lisp в середину s-выражения, по которому будет генерироваться HTML.

Литеральные значения, такие как строки и числа, входя html будут подставлены в генерируемый HTML. Также, символы интерпретируются как ссылки на переменные, которые они именуют, и

сгенерированный код будет брать их значения во время выполнения. Таким образом оба следующих выражения:

```
(html (:p "foo"))  
(let ((x "foo")) (html (:p x)))
```

сгенерируют следующее:

```
<p>foo</p>
```

Формы списков, которые не начинаются с ключевых символов, рассматриваются как код и встраиваются в генерируемый код. Любые значения, возвращаемые встроенным кодом, будут проигнорированы, но такой код сам может генерировать HTML путем вызова `html`. Например, для генерирования содержимого списка в виде HTML, мы можем написать следующее:

```
(html (:ul (dolist (item (list 1 2 3)) (html (:li item)))))
```

что сгенерирует следующий HTML:

```
<ul>  
  <li>1</li>  
  <li>2</li>  
  <li>3</li>  
</ul>
```

Если вы захотите выдать значение формы, вы должны обернуть его в псевдотег `:print`. Таким образом, выражение:

```
(html (:p (+ 1 2)))
```

сгенерирует такой HTML после вычисления и отбрасывания значения 3:

```
<p></p>
```

Для выдачи 3 вы должны написать такой код:

```
(html (:p (:print (+ 1 2))))
```

Или же вы можете вычислить значение и сохранить его в переменной вне вызова `html` следующим образом:

```
(let ((x (+ 1 2))) (html (:p x)))
```

Таким образом вы можете использовать макрос `html` для генерирования списка случайных чисел следующим образом:

```
(html
  (:html
    (:head
      (:title "Random numbers"))
    (:body
      (:h1 "Random numbers")
      (:p (loop repeat 10 do (html (:print (random 1000)) " "))))))
```

Версия, использующая макрос `html`, будет эффективнее, чем использующая `emit-html`. И не только из-за того, что теперь не нужно генерировать `s`-выражение, представляющее страницу целиком, но и из-за того, что большая часть работы по интерпретации `s`-выражения, которую `emit-html` осуществляет во время выполнения, будет сделана однократно, во время раскрытия макроса, а не каждый раз при выполнении кода.

Вы можете управлять тем, куда будет отправлен вывод `html` и `emit-html`, с помощью макроса `with-html-output`, который также является частью библиотеки `FOO`. Вы можете использовать макросы `with-html-output` и `html` для переписывания `random-number` следующим образом:

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:html
            (:head (:title "Random"))
            (:body
              (:p "Random number: " (:print (random 1000))))))))))
```

Макросы HTML

Другой возможностью `FOO` является то, что он позволяет вам определять "макросы" HTML, которые могут преобразовывать произвольные формы в представляющие HTML `s`-выражения, которые понимает макрос `html`. Например, предположим, что вы заметили, что часто создаете страницы следующего вида:

```
(:html
  (:head (:title "Some title"))
  (:body
    (:h1 "Some title")
    ... stuff ...))
```

Вы можете определить макрос HTML, представляющий этот образец, следующим образом:

```
(define-html-macro :standard-page ((&key title) &body body)
  `(:html
    (:head (:title ,title))
    (:body
      (:h1 ,title)
      ,@body)))
```

Теперь вы можете использовать "тег" `:standard-page` в ваших представляющих HTML `s`-выражениях, и он будет раскрыт перед интерпретацией или компиляцией этих выражений. Например, следующий код:

```
(html (:standard-page (:title "Hello") (:p "Hello, world.")))
```

сгенерирует такой HTML:

```
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>Hello, world.</p>
  </body>
</html>
```

Параметры запроса

Конечно, генерирование HTML является только половиной темы web-программирования. Еще одной вещью, которую вы должны уметь делать, является получение ввода от пользователя. Как я рассказывал в разделе "30-секундное введение в web-программирование на стороне сервера", когда браузер запрашивает страницу у web-сервера, он может послать параметры запроса в адресе URL или POST-данные, и оба этих источника рассматриваются как ввод для кода на стороне сервера.

AllegroServe, как и большинство других каркасов web-программирования, берет на себя заботу о разборе обоих этих источников ввода для вас. В то время когда ваша опубликованная функция вызывается, все пары ключ-значение из строки запроса и/или POST-данных уже декодированы и помещены в ассоциативный список (alist), который вы можете получить из объекта запроса с помощью функции `request-query`. Следующая функция возвращает страницу, отображающую все полученные ею параметры запроса:

```
(defun show-query-params (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
            (:title "Query Parameters")
            (if (request-query request)
              (html
                (:table :border 1
                  (loop for (k . v) in (request-query request)
                    do (html (:tr (:td k) (:td v))))))
              (html (:p "No query parameters."))))))
        (publish :path "/show-query-params" :function 'show-query-params))
```

Если вы укажете своему браузеру URL со строкой запроса подобной такой:

```
http://localhost:2001/show-query-params?foo=bar&baz=10
```

вы получите страницу, подобную показанной на рисунке 26-4.

Рисунок 26-4. <http://localhost:2001/show-query-params?foo=bar&baz=10>

Для генерирования POST-данных нам нужна форма HTML. Следующая функция генерирует простую форму, которая посылает свои данные `show-query-params`:


```
(defun simple-form (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let ((*html-output* (request-reply-stream request)))
        (html
          (:html
            (:head (:title "Simple Form"))
            (:body
              (:form :method "POST" :action "/show-query-params"
                (:table
                  (:tr (:td "Foo")
                    (:td (:input :name "foo" :size 20)))
                  (:tr (:td "Password")
                    (:td (:input :name "password" :type "password" :size 20)))
                  (:p (:input :name "submit" :type "submit" :value "Okay")
                    (:input :type "reset" :value "Reset"))))))))))))
    (publish :path "/simple-form" :function 'simple-form))
```

Перейдите в вашем браузере по адресу <http://localhost:2001/simple-form>; вы должны увидеть страницу, подобную изображенной на рисунке 26-5.

Если вы заполните форму значениями "abc" и "def", щелкните на кнопку Okay, то получите страницу, сходную с изображенной на рисунке 26-6.

Рисунок 26-5. <http://localhost:2001/simple-form>

Рисунок 26-6. Результат отправки простой формы

Однако, чаще всего вам не нужно проходить по всем параметрам запроса: обычно вам нужно просто получить определенный параметр. Например, вы можете захотеть модифицировать `random-number` так, чтобы предельное значение, передаваемое в функцию **RANDOM**, предоставлялось как параметр запроса. В таком случае используется функция `request-query-value`, получающая объект запроса и имя параметра, значение которого вы хотите получить. Она возвращает либо значение параметра в виде строки, либо **NIL**, если такой параметр не предоставлен. "Параметризованная" версия `random-number` может выглядеть следующим образом:

```
(defun random-number (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (let* ((*html-output* (request-reply-stream request))
        (limit-string (or (request-query-value "limit" request) ""))
        (limit (or (parse-integer limit-string :junk-allowed t) 1000)))
        (html
          (:html
            (:head (:title "Random"))
            (:body
              (:p "Random number: " (:print (random limit))))))))))
```

Так как `request-query-value` может возвращать как **NIL**, так и пустую строку, мы должны обрабатывать оба этих случая при разборе параметра и его преобразования в число, которое будет передано **RANDOM**. Мы можем обработать значение **NIL** при связывании переменной `limit-string`, связывая ее с "" если параметра "limit" нет. Затем мы можем использовать аргумент функции **PARSE-INTEGER** `:junk-allowed` для гарантии того, что она вернет либо **NIL** (если не сможет разобрать целое число из переданной строки), либо целое число. В разделе "Небольшой каркас приложений" мы разработаем несколько макросов для облегчения работы

по получению параметров запроса и преобразования их в различные типы.

Cookies

В AllegroServe вы можете послать заголовок Set-Cookie, который укажет браузеру сохранить cookie и посылать его со всеми последующими запросами, путем вызова функции `set-cookie-header` внутри тела `with-http-response`, но перед вызовом `with-http-body`. Первый аргумент этой функции должен быть объектом запроса, а остальные аргументы — ключевые аргументы, используемые для установки различных свойств cookie. Обязательными являются лишь два аргумента: `:name` и `:value`, оба из которых являются строками. Остальные возможные аргументы, влияющие на посылаемый браузеру cookie: `:expires`, `:path`, `:domain` и `:secure`.

Из этих аргументов нам следует обратить внимание лишь на `:expires`. Он управляет тем, как долго браузер должен сохранять cookie. Если `:expires` равен **NIL** (по умолчанию), браузер сохранит cookie только до завершения своей работы. Другие возможные значения: `:never`, что означает, что cookie должен сохраняться навсегда, или всемирное (`universal`) время, как возвращается **GET-UNIVERSAL-TIME** или **ENCODE-UNIVERSAL-TIME**. Значение `:expires` равное нулю указывает клиенту немедленно удалить существующие cookie.

После того как вы установили cookie, вы можете использовать функцию `get-cookie-values` для получения ассоциативного списка (`alist`), содержащего по паре имя-значение на каждый cookie, посланный браузером. Из этого списка вы можете получить значения отдельных cookie с помощью **ASSOC** и **CDR**.

Следующая функция отображает имена и значения всех cookie, посланных браузером:

```
(defun show-cookies (request entity)
  (with-http-response (request entity :content-type "text/html")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
            (:title "Cookies")
            (if (null (get-cookie-values request))
              (html (:p "No cookies."))
              (html
                (:table
                  (loop for (key . value) in (get-cookie-values request)
                      do (html (:tr (:td key) (:td value))))))))))
        (publish :path "/show-cookies" :function 'show-cookies))
```

При первой загрузке страницы <http://localhost:2001/show-cookies> она должна отобразить сообщение "No cookies" как показано на рисунке 26-7, так как вы еще ничего не установили.

Рисунок 26-7. <http://localhost:2001/show-cookies> без установленных cookie

Для установки cookie нам понадобится другая функция, такая как эта:

```
(defun set-cookie (request entity)
  (with-http-response (request entity :content-type "text/html")
    (set-cookie-header request :name "MyCookie" :value "A cookie value")
    (with-http-body (request entity)
      (with-html-output ((request-reply-stream request))
        (html
          (:standard-page
            (:title "Set Cookie")
            (:p "Cookie set.")
            (:p (:a :href "/show-cookies" "Look at cookie jar."))))))
      (publish :path "/set-cookie" :function 'set-cookie))
```

Если вы откроете в браузере <http://localhost:2001/set-cookie>, он должен отобразить страницу, показанную на рисунке 26-8. Вдобавок сервер пошлет заголовок Set-Cookie с cookie по имени "MyCookie" со значением "A cookie value". Если вы нажмете на ссылку *Look at cookie jar*, вы попадете на страницу /show-cookies, где увидите новый cookie, как показано на рисунке 26-9. Так как вы не задали аргумент :expires, браузер продолжит посылать cookie при каждом запросе пока вы не выйдете из него.

Рисунок 26-8. <http://localhost:2001/set-cookie>

Рисунок 26-9. <http://localhost:2001/show-cookies> после установки cookie

Небольшой каркас приложений

Хотя AllegroServe предоставляет достаточно прямой доступ ко всем базовым возможностям, необходимым для написания кода, выполняющегося на стороне сервера (доступ к параметрам запроса как из строки запроса, так и из POST-данных; возможность установки cookie и получения их значений; и, конечно же, возможность генерирования ответа для отправки браузеру), приходится писать некоторое количество раздражающе повторяющегося кода.

Например, каждая генерирующая HTML функция, которую вы будете писать, будет получать в качестве аргументов запрос и сущность, а затем будет содержать вызовы with-http-response, with-http-body и, если вы будете использовать FOO для генерирования HTML, with-html-output. Далее, функции, которым нужно получать параметры запроса, будут содержать множество вызовов request-query-value и еще больше кода для преобразования получаемых строк к нужным типам. И наконец вам нужно не забыть опубликовать функции.

Для уменьшения повторяющегося кода, который вам нужно писать, мы можем реализовать небольшой каркас поверх AllegroServe в целях облегчения определения функций, обрабатывающих запросы по определенным URL.

Базовым приближением будет определение макроса define-url-function, который мы будем использовать для определения функций, которые будут автоматически публиковаться с помощью publish. Этот макрос будет раскрываться в **DEFUN**, содержащий соответствующий шаблонный код, а также в код публикации функции под URL с таким же, как у функции, именем. Он также возьмет на себя заботу по генерации кода извлечения значений параметров запроса и cookies и связывания их с переменными, объявленными в списке параметров функции. Таким образом, базовой формой определения define-url-function будет такая:

```
(define-url-function name (request query-parameter*)
  body)
```

где body будет кодом, выдающим код HTML страницы. Он будет обернут в вызов макроса html

из FOO, и поэтому для простых страниц может не содержать ничего, кроме представляющего HTML s-выражения.

Внутри тела переменные параметров запроса будут связаны со значениями параметров запроса с такими же именами или значениями cookie. В простейшем случае значением параметра запроса будет строка, полученная из параметра запроса или поля POST-данных с таким же именем. Если же параметр запроса задается списком, вы также можете задать автоматическое преобразование типа, значение по умолчанию, и то, нужно ли получать значение параметра из и сохранять его в cookie. Полный синтаксис для параметра запроса выглядит так:

```
name | (name type [default-value] [stickiness])
```

type должен быть именем, распознаваемым define-url-function. Мы скоро обсудим как определять новые типы. default-value должно быть значением данного типа. И, наконец, stickiness, если предоставлен, указывает, что значение параметра должно быть взято из cookie с соответствующим именем в случае, если параметр запроса не предоставлен, а также что в ответе должен быть послан заголовок Set-Cookie, который сохранит значение cookie с этим именем. Таким образом, сохраняемый параметр (sticky parameter), после явного предоставления значения в параметре запроса, сохранит это значение при последующих запросах страницы даже если параметр запроса не предоставляется.

Имя используемого cookie зависит от значения stickiness: при значении :global cookie будет иметь то же имя, что и параметр. Таким образом, различные функции, использующие глобальные сохраняемые параметры с одинаковым именем, будут разделять значение. Если stickiness равно :package, то имя cookie будет сконструировано из имен параметра и пакета, в котором находится имя функции; это позволит функциям одного пакета разделять значения не заботясь о возможных конфликтах с параметрами функций других пакетов. И, наконец, параметр со значением stickiness равным :local будет использовать имя cookie, составленное из имен параметра, пакета, в котором находится имя функции, и самого имени функции, что делает его уникальным для этой функции.

Например, вы можете использовать define-url-function для замены предыдущего 11-строчного определения random-page 5-строчной версией:

```
(define-url-function random-number (request (limit integer 1000))
  (:html
    (:head (:title "Random"))
    (:body
      (:p "Random number: " (:print (random limit))))))
```

Если вы хотите, чтобы аргумент limit сохранялся, вы должны изменить объявление limit следующим образом: (limit integer 1000 :local).

Реализация

Я разъясню реализацию define-url-function "сверху вниз". Сам макрос выглядит следующим образом:

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params)))
      `(progn
         (defun ,name (,request ,entity)
           (with-http-response (,request ,entity :content-type "text/html")
             (let* (,@(param-bindings name request params))
               ,@(set-cookies-code name request params)
               (with-http-body (,request ,entity)
                 (with-html-output ((request-reply-stream ,request))
                   (html ,@body))))))
         (publish :path ,(format nil "~/(~a~)" name) :function ',name))))))
```

Давайте рассмотрим ее по шагам, начиная с первых строк.

```
(defmacro define-url-function (name (request &rest params) &body body)
  (with-gensyms (entity)
    (let ((params (mapcar #'normalize-param params)))
```

Up to here you're just getting ready to generate code. Мы генерируем с помощью **GENSYM** символ для дальнейшего использования в качестве имени параметра сущности в **DEFUN**. Затем мы нормализуем параметры, преобразуя обычные символы в списочную форму с помощью следующей функции:

```
(defun normalize-param (param)
  (etypecase param
    (list param)
    (symbol `(,param string nil nil))))
```

Другими словами, объявления параметра как просто символа — это тоже самое, что и объявление несохраняемого строкового параметра без значения по умолчанию.

Затем идет **PROGN**. Мы должны раскрывать макрос в **PROGN** так как нам нужно сгенерировать код, осуществляющий две вещи: определение функции с помощью **DEFUN** и вызов `publish`. Определение функции должно идти первым: таким образом, если в ее определении будет ошибка, то функция не будет опубликована. Первые две строки **DEFUN** являются уже привычным нам шаблонным кодом:

```
(defun ,name (,request ,entity)
  (with-http-response (,request ,entity :content-type "text/html")
```

Теперь мы можем приступить к настоящей работе. Следующие две строки генерируют привязки параметров, заданных в `define-url-function` (кроме `request`), а также код, вызывающий `set-cookie-header` для сохраняемых параметров. Конечно же реальная работа осуществляется во вспомогательных функциях, которые мы вскоре увидим.

```
(let* (,@(param-bindings name request params))
  ,@(set-cookies-code name request params)
```

Оставшаяся часть кода более шаблонна: мы помещаем тело из определения `define-url-function` в соответствующий контекст `with-http-body`, `with-html-output` и макроса `html`. Затем идет вызов `publish`.

```
(publish :path ,(format nil "~/(~a~)" name) :function ',name)
```

Выражение `(format nil "~(~a~)" name)` вычисляется во время раскрытия макросов, генерируя строку, состоящую из `/`, за которым следует преобразованное к нижнему регистру имя функции, почти определенной нами. Эта строка становится аргументом `:path` функции `publish`, а имя функции – аргументом `:function`.

Теперь давайте взглянем на вспомогательные функции, используемые при генерировании формы **DEFUN**. Для генерирования привязок параметров нам нужно пройти по параметрам и собрать код, сгенерированный `param-binding` для каждого из них. Такой код для каждого параметра будет списком, содержащим имя связываемой переменной и код, который вычисляет ее значение. Точный код для вычисления значения будет зависеть от типа параметра, того, является ли он сохраняемым, и от наличия значения по умолчанию. Так как мы уже нормализовали параметры, мы можем использовать **DESTRUCTURING-BIND** для их разбора в `param-binding`.

```
(defun param-bindings (function-name request params)
  (loop for param in params
    collect (param-binding function-name request param)))
(defun param-binding (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (let ((query-name (symbol->query-name name))
          (cookie-name (symbol->cookie-name function-name name sticky)))
      `(,name (or
                (string->type ',type (request-query-value ,query-name ,request))
                ,@(if cookie-name
                      (list `(string->type ',type (get-cookie-value ,request
                                                                    ,cookie-name))))
                ,default)))))
```

Функция `string->type`, используемая для преобразования к желаемым типам полученных из параметров запроса и cookies строк, является обобщенной функцией со следующей сигнатурой:

```
(defgeneric string->type (type value))
```

Для того, чтобы иметь возможность использования определенного имени в качестве имени типа параметра запроса, нам нужно просто определить метод `string->type`. Нам нужно определить по меньшей мере метод, специализированный по строковому типу, так как это тип по умолчанию. Конечно же это очень просто. Так как браузеры порой посылают формы с пустыми строками для индикации отсутствия значения, нам нужно преобразовывать пустые строки в **NIL**, как и делает следующий метод:

```
(defmethod string->type ((type (eql 'string)) value)
  (and (plusp (length value)) value))
```

Мы можем добавить преобразования для других типов, нужных нашему приложению. Например, чтобы иметь возможность использования в качестве типа параметров запроса `integer`, а следовательно возможность обработки параметра `limit` функции `random-page`, мы можем определить следующий метод:

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))
```

Еще одной вспомогательной функцией, используемой кодом, генерируемым `param-binding`, является `get-cookie-value`, которая является небольшим синтаксическим сахаром вокруг

функции `get-cookie-values`, предоставляемой `AllegroServe`. Она выглядит следующим образом:

```
(defun get-cookie-value (request name)
  (cdr (assoc name (get-cookie-values request) :test #'string=)))
```

Функции, вычисляющие имена параметров запроса и cookies, довольно прямолинейны:

```
(defun symbol->query-name (sym)
  (string-downcase sym))
(defun symbol->cookie-name (function-name sym sticky)
  (let ((package-name (package-name (symbol-package function-name))))
    (when sticky
      (ecase sticky
        (:global
         (string-downcase sym))
        (:package
         (format nil "~(~a:~a~)" package-name sym))
        (:local
         (format nil "~(~a:~a:~a~)" package-name function-name sym))))))
```

Для генерирования кода, устанавливающего cookies для сохраняемых параметров, нам снова нужно пройти по списку параметров, на этот раз собирая код для каждого сохраняемого параметра. Мы можем использовать формы **LOOP** `when` и `collect it` для собирания только не-**NIL** значений, возвращенных `set-cookie-code`.

```
(defun set-cookies-code (function-name request params)
  (loop for param in params
        when (set-cookie-code function-name request param) collect it))
(defun set-cookie-code (function-name request param)
  (destructuring-bind (name type &optional default sticky) param
    (declare (ignore type default))
    (if sticky
      `(when ,name
         (set-cookie-header
          ,request
          :name ,(symbol->cookie-name function-name name sticky)
          :value (princ-to-string ,name))))))
```

Одним из преимуществ определения макросов в терминах вспомогательных функций, как здесь, является то, что так легко удостовериться, что отдельные части генерируемого кода выглядят правильно. Например, вы можете проверить, что такой вызов `set-cookie-code`:

```
(set-cookie-code 'foo 'request '(x integer 20 :local))
```

генерирует такой код:

```
(WHEN X
  (SET-COOKIE-HEADER REQUEST
    :NAME "com.gigamonkeys.web:foo:x"
    :VALUE (PRINC-TO-STRING X)))
```

Подразумевая, что этот код находится в контексте, в котором `x` является именем переменной, все выглядит хорошо.

И еще раз, макросы позволяют нам свести код, которые необходимо писать, к его сути: в нашем

случае это данные, которые нам нужно извлечь из запроса, и HTML, который мы хотим сгенерировать. Этот каркас не претендует на то, чтобы являться наивысшим достижением в области создания каркасов построения web-приложений, — он является просто небольшим синтаксическим сахаром, немного упрощающим написание простых приложений, наподобие такого, что мы напишем в главе 29.

Но перед тем, как приступить к этому, нам нужно написать "внутренности" приложения, для которых приложение, которое мы напишем в главе 29, будет пользовательским интерфейсом. Мы начнем в следующей главе с написания улучшенной версии базы данных, написанной нами ранее в главе 3, которую мы будем использовать для хранения данных ID3, извлеченных из файлов MP3.

Практика: База данных для MP3

В этой главе мы заново рассмотрим идею, впервые упомянутую в главе Chapter 3 – построение базы данных, расположенной в памяти, на основе базовых типов данных Lisp. Сейчас нашей целью является хранение информации, которую вы извлечете из коллекции файлов в формате MP3 при помощи библиотеки ID3v2 из главы 25. Вы затем будете использовать эту базу данных в главах 28 и 29 как часть потокового MP3-сервера с Web-интерфейсом. Конечно сейчас вы уже можете использовать некоторые из языковых конструкций, изученных со времени изучения главы 3, чтобы создать более совершенный код.

База данных

Основной проблемой базы данных из главы 3 является то, что есть только одна таблица – список, сохраненный в переменной `*db*`. Другой проблемой является то, что код ничего не знает о типах значений, сохраненных в разных колонках. В главе 3 вы просто использовали функцию общего назначения `EQUAL` для сравнения значений в колонках при выборе строк из базы данных, но у вас были бы проблемы, если бы вы хотели бы сохранить значения, которые не сравниваются с помощью `EQUAL`, или если бы вы хотели сортировать строки в базе данных, поскольку не такой функции сравнения, похожей на `EQUAL`.

Сейчас вы будете решать обе проблемы путем определения класса `table`, который будет описывать отдельные таблицы базы данных. Каждый экземпляр класса `table` будет состоять из двух слотов – один для хранения данных, а второй – для хранения информации о колонках таблицы, которую смогут использовать функции для работы с базой данных. Класс выглядит примерно вот так:

```
(defclass table ()
  ((rows :accessor rows :initarg :rows :initform (make-rows))
   (schema :accessor schema :initarg :schema)))
```

Также как и в главе 3, вы можете представлять отдельные строки в виде списков свойств, но сейчас время вы создадите абстракцию, которая позволит вам изменять внутреннее представление без особых трудностей. И в данной версии вы будете сохранять данные в векторе, а не в списке свойств, поскольку некоторые операции, которые вы будете поддерживать, например, произвольный доступ к строкам по числовому индексу, и возможность сортировки таблицы, могут быть более эффективно реализованы с помощью векторов.

Функция `make-rows`, используемая для инициализации слота `rows` может быть простой оберткой для функции `MAKE-ARRAY`, которая создает пустой вектор с изменяемым размером и указателем заполнения.

FIXME this should be inside of table/box Пакет

Объявление пакета для разрабатываемого вами в этой главе кода будет выглядеть следующим образом:

```
(defpackage :com.gigamonkeys.mp3-database
  (:use :common-lisp
        :com.gigamonkeys.pathnames
        :com.gigamonkeys.macro-utilities
        :com.gigamonkeys.id3v2)
  (:export :*default-table-size*
           :*mp3-schema*
           :*mp3s*
           :column
           :column-value
           :delete-all-rows
           :delete-rows
           :do-rows
           :extract-schema
           :in
           :insert-row
           :load-database
           :make-column
           :make-schema
           :map-rows
           :matching
           :not-nullable
           :nth-row
           :random-selection
           :schema
           :select
           :shuffle-table
           :sort-rows
           :table
           :table-size
           :with-column-values))
```

Раздел `:use` дает возможность доступа к функциям и макросам, чьи имена экспортированы из пакетов, созданных в главах 15, 8 и 25, а секция `:export` используется для объявления функций, реализуемых данным пакетом, и которые будут использоваться в главе 29.

FIXME end of table/box

```
(defparameter *default-table-size* 100)
(defun make-rows (&optional (size *default-table-size*))
  (make-array size :adjustable t :fill-pointer 0))
```

Для представление схемы таблицы, вам необходимо определить еще один класс — `column`, каждый экземпляр которого будет содержать информацию об одной колонке в таблице: ее название, способ сравнения значений в колонке на равенство и порядок расположения, значение по умолчанию, а также функцию, которая будет использоваться для нормализации значения при вставке данных в таблицу и при запросе данных из таблицы. Слот `schema` будет хранить список объектов типа `column`. Определение класса будет выглядеть примерно вот так:

```
(defclass column ()
  ((name
    :reader name
    :initarg :name)
   (equality-predicate
    :reader equality-predicate
    :initarg :equality-predicate)
   (comparator
    :reader comparator
    :initarg :comparator)
   (default-value
    :reader default-value
    :initarg :default-value
    :initform nil)
   (value-normalizer
    :reader value-normalizer
    :initarg :value-normalizer
    :initform #'(lambda (v column) (declare (ignore column)) v))))
```

Слоты `equality-predicate` и `comparator` объекта `column` хранят функции, которые будут использоваться для сравнения значений данной колонки на равенство и порядок расположения. Например, для колонки, которая будет хранить строковые значения, мы можем использовать функции `STRING=` в качестве значения `equality-predicate` и `STRING<` для `comparator`, тогда как колонки, хранящие числа, могут использовать функции `=` и `<`.

Слоты `default-value` и `value-normalizer` используются при вставке и при запросе данных (слот `value-normalizer`). Когда вы вставляете строку в базу данных, и для определенной колонки не указано значение, то вы можете использовать значение, хранящееся в слоте `default-value` данной колонки. Затем, значение (значение по умолчанию или указанное пользователем) нормализуется путем передачи его и объекта, описывающего колонку в БД, в качестве параметров функции, указанной в слоте `value-normalizer`. Вы передаете объект типа `column` поскольку для функции `value-normalizer` может понадобиться некоторые данные, связанные с объектом `column`. (Вы увидите пример такого использования в следующем разделе). Вы также должны нормализовывать значения, передаваемые в запросах, до их сравнения с объектами в базе данных.

Таким образом, `value-normalizer` отвечает за возврат значения, которое может быть спокойно передано функциям `equality-predicate` и `comparator`. Если `value-normalizer` не может найти подходящее возвращаемое значение, то она сгенерирует ошибку.

Другой причиной для нормализации значений до их сохранения в БД является возможность уменьшить потребление памяти и процессора. Например, если у вас есть колонка, которая должна хранить строковые значения, но количество значений, которые будут сохранены является ограниченным – например, колонка `genre` (жанр) в базе данных МРЗ-файлов, то вы можете уменьшить потребление памяти и увеличить скорость работы, путем использования функции `value-normalizer` для интернирования (`FIXME intern` - написать что это такое) строк (преобразовать все вызовы `STRING=` к одному объекту-строке). Так что вам нужно будет иметь столько строковых объектов, сколько у вас имеется различающихся строк, вне зависимости от того, сколько строк у вас в таблице, и вы тогда сможете использовать для сравнения функцию `EQL`, а не `STRING=`, которая является более медленной.

Определение схемы базы данных

Таким образом, чтобы создать экземпляр таблицы, вам необходимо создать список объектов `column objects`. Вы можете создать такой список вручную, используя функции `LIST` и `MAKE-`

INSTANCE. Но вы скоро заметите, что вы часто создаете множество объектов `column` с одинаковыми комбинациями функций `comparator` и `equality-predicate`. Это происходит оттого, что комбинация функций сравнения по существу определяет тип колонки. Было бы хорошо, если бы был способ определить имена для этих типов, что позволит вам просто указывать, что конкретная колонка является строковой, вместо того, чтобы указывать `STRING<` и `STRING=` в качестве функций сравнения. Одним из способов решения этой проблемы является определение обобщенной функции, `make-column`, например, вот так:

```
(defgeneric make-column (name type &optional default-value))
```

Теперь вы можете определять методы данной обобщенной функции, специализированные для типа, с использованием EQL, которые будут возвращать объекты `column` со слотами, заполненными соответствующими значениями. Вот определения для методов, которые определяют типы колонок с именами `string` и `number`:

```
(defmethod make-column (name (type (eql 'string)) &optional default-value)
  (make-instance
    'column
    :name name
    :comparator #'string<
    :equality-predicate #'string=
    :default-value default-value
    :value-normalizer #'not-nullable))
(defmethod make-column (name (type (eql 'number)) &optional default-value)
  (make-instance
    'column
    :name name
    :comparator #'<
    :equality-predicate #'=
    :default-value default-value))
```

Следующая функция – `not-nullable`, используется в качестве значения `value-normalizer` для строковых колонок, и просто возвращает переданное значение для всех случаев, кроме тех, когда ей передают значение `NIL`, когда она сигнализирует об ошибке:

```
(defun not-nullable (value column)
  (or value (error "Column ~a can't be null" (name column))))
```

Это важно, поскольку вызовы `STRING<` и `STRING=` будут выдавать ошибку, если им будут передан `NIL`; лучше перехватить неправильные значения до того, как они будут вставлены в таблицу, а не тогда, когда мы будем их использовать.

Еще одним типом колонки, который понадобится для базы данных MP3 является `interned-string`, чьи значения интернируются, как это обсуждалось выше. Поскольку вам нужна хэш-таблица, в которую вы будете интернировать значения, вы должны определить подкласс `column` – `interned-values-column`, который добавит еще один слот, чьим значением будет хэш-таблица, которая будет использоваться для интернирования.

Для реализации интернирования, вам потребуется указать в качестве `:initform` для слота `value-normalizer` функцию, которая будет интернировать значение в хэш-таблицу, которая хранится в колонке `interned-values`. И поскольку, одна из самых главных причин интернирования значений – возможность использования EQL в качестве функции равенства, то вы также должны добавить `#'eql` в качестве значения `:initform` для слота `equality-predicate`.

```
(defclass interned-values-column (column)
  ((interned-values
    :reader interned-values
    :initform (make-hash-table :test #'equal))
   (equality-predicate :initform #'eql)
   (value-normalizer :initform #'intern-for-column)))
(defun intern-for-column (value column)
  (let ((hash (interned-values column)))
    (or (gethash (not-nullable value column) hash)
        (setf (gethash value hash) value))))
```

Затем вы можете определить метод `make-column` специализированный для имени `interned-string`, который будет возвращать экземпляры `interned-values-column`.

```
(defmethod make-column (name (type (eql 'interned-string)) &optional default-value)
  (make-instance
    'interned-values-column
    :name name
    :comparator #'string<
    :default-value default-value))
```

С помощью данных методов, определенных для `make-column`, вы теперь можете определить функцию, `make-schema`, которая создаст список объектов типа `column` из списка описаний колонок, каждое из которых содержит имя колонки, имя типа колонки, и, необязательно, значение по умолчанию.

```
(defun make-schema (spec)
  (mapcar #'(lambda (column-spec) (apply #'make-column column-spec)) spec))
```

Например, с помощью следующего кода вы можете определить схему для таблицы, которая будет использоваться для хранения данных, извлеченных из файлов MP3:

```
(defparameter *mp3-schema*
  (make-schema
    '(:file string)
      (:genre interned-string "Unknown")
      (:artist interned-string "Unknown")
      (:album interned-string "Unknown")
      (:song string)
      (:track number 0)
      (:year number 0)
      (:id3-size number))))
```

Чтобы создать саму таблицу для хранения информации о файлах MP3, вы должны передать `*mp3-schema*` в качестве аргумента `:schema` функции `MAKE-INSTANCE`.

```
(defparameter *mp3s* (make-instance 'table :schema *mp3-schema*))
```

Вставка значений

Сейчас вы готовы к тому, чтобы определить вашу первую операцию для работы с таблицами – `insert-row`, которая получает список свойств (plist) имен и значений, и таблицу, и добавляет строку к таблице. Большая часть работы выполняется в дополнительной функции `normalize-row`, которая создает список свойств для всех колонок таблицы, используя нормализованные

значения и значения по умолчанию, которые получаются из слотов `names-and-values`, если значение было указано, или `default-value` если значение для конкретной колонки не было указано.

```
(defun insert-row (names-and-values table)
  (vector-push-extend (normalize-row names-and-values (schema table)) (rows table)))
(defun normalize-row (names-and-values schema)
  (loop
    for column in schema
    for name = (name column)
    for value = (or (getf names-and-values name) (default-value column))
    collect name
    collect (normalize-for-column value column)))
```

Создание дополнительной функции `normalize-for-column`, которая получает значение, и объект `column` и возвращает нормализованное значение, оправдано тем, что вам будет проводить нормализацию значений при запросах к таблице.

```
(defun normalize-for-column (value column)
  (funcall (value-normalizer column) value column))
```

Теперь вы готовы к объединению кода базы данных с кодом из предыдущих глав, чтобы построить базу данных, содержащую информацию выделенную из файлов MP3. Вы можете определить функцию `file->row`, которая будет использовать функцию `read-id3` из библиотеки `ID3v2` для выделения тегов ID3 из файла, и превращения их в список свойств, который будет передан функции `insert-row`.

```
(defun file->row (file)
  (let ((id3 (read-id3 file)))
    (list
      :file (namestring (truename file))
      :genre (translated-genre id3)
      :artist (artist id3)
      :album (album id3)
      :song (song id3)
      :track (parse-track (track id3))
      :year (parse-year (year id3))
      :id3-size (size id3))))
```

Вам не нужно беспокоиться о нормализации значений, поскольку это будет сделано в `insert-row`. Однако, вы должны сконвертировать строки, возвращенные функциями `track` и `year` в числа. Число `track` (номер композиции) – это тег ID3, который иногда сохраняется как число в виде строки, и иногда как число, за которым следует (через знак слеш) еще одно число, обозначающее количество композиций в альбоме. Поскольку нам нужен только номер композиции, то вы должны использовать аргумент `:end` при вызове функции `PARSE-INTEGER` для того, чтобы указать что разбор должен осуществляться только до знака слеш, если он есть.

```
(defun parse-track (track)
  (when track (parse-integer track :end (position #\/ track))))
(defun parse-year (year)
  (when year (parse-integer year)))
```

В заключение, вы можете собрать все эти функции вместе с `walk-directory` из библиотеки переносимых имен файлов, а также функцией `mp3-p` из библиотеки `ID3v2` чтобы определить

функцию, которая загружает в базу данных MP3 информацию извлеченную из файлов MP3, которые были найдены в определенном каталоге (и всех его подкаталогах).

```
(defun load-database (dir db)
  (let ((count 0))
    (walk-directory
      dir
      #'(lambda (file)
          (princ #\.)
          (incf count)
          (insert-row (file->row file) db))
      :test #'mp3-p)
    (format t "~&В базу данных загружено ~d файлов." count)))
```

Выполнение запросов к базе данных

После того, как загрузите данные в базу данных, вам необходимо найти способ выполнять запросы к ней. Для приложения работающего с файлами MP3 вам понадобятся более сложные функции выполнения запросов чем те, которые были использованы в главе 3. Сейчас вам нужна не только возможность извлекать строки отвечающие определенным критериям, но также и возможность делать выборку только определенных колонок, и возможно, сортировать строки по определенной колонке. В соответствии с теорией реляционных баз данных, результатом запроса будет новая таблица, содержащая строки и колонки.

В качестве образца для функции выполнения запросов – `select`, был взят оператор `SELECT` из языка SQL. Эта функция принимает пять именованных параметров: `:from`, `:columns`, `:where`, `:distinct` и `:order-by`. Аргумент `:from` указывает объект `table` для которого вы хотите выполнить запрос. Аргумент `:columns` указывает то, какие колонки должны быть включены в результат. В качестве значения должен быть указан список имен колонок, имя одной колонки или `T` (значение по умолчанию), указывающее, что должны быть включены все колонки. Аргумент `:where` (если он указан), должен быть функцией, которая получает строку, и возвращает истинное значение, если эта строка должна быть включена в результаты. Немного спустя, вы напишете две функции – `matching` и `in`, которые возвращают функции, допустимые для использования в качестве аргумента `:where`. Аргумент `:order-by` (если он указан), должен быть списком имен колонок; результаты будут отсортированы по соответствующим колонкам. Также как и для аргумента `:columns`, вы можете указать лишь одну колонку, просто используя ее имя, что эквивалентно списку из одного элемента. В заключение, аргумент `:distinct` является логическим значением, которое указывает – должны ли мы удалять дублирующиеся строки из результата. Значением по умолчанию для `:distinct` является `NIL`.

Вот несколько примеров использования `select`:

```
;; Выбрать все строки где колонка :artist равна "Green Day"
(select :from *mp3s* :where (matching *mp3s* :artist "Green Day"))
;; Получить отсортированный список артистов, исполняющих песни в жанре "Rock"
(select
  :columns :artist
  :from *mp3s*
  :where (matching *mp3s* :genre "Rock")
  :distinct t
  :order-by :artist)
```

Реализация `select` вместе со вспомогательными функциями выглядит примерно так:

```

(defun select (&key (columns t) from where distinct order-by)
  (let ((rows (rows from))
        (schema (schema from)))
    (when where
      (setf rows (restrict-rows rows where)))
    (unless (eql columns 't)
      (setf schema (extract-schema (mklist columns) schema))
      (setf rows (project-columns rows schema)))
    (when distinct
      (setf rows (distinct-rows rows schema)))
    (when order-by
      (setf rows (sorted-rows rows schema (mklist order-by))))
    (make-instance 'table :rows rows :schema schema)))

(defun mklist (thing)
  (if (listp thing) thing (list thing)))

(defun extract-schema (column-names schema)
  (loop for c in column-names collect (find-column c schema)))

(defun find-column (column-name schema)
  (or (find column-name schema :key #'name)
      (error "No column: ~a in schema: ~a" column-name schema)))

(defun restrict-rows (rows where)
  (remove-if-not where rows))

(defun project-columns (rows schema)
  (map 'vector (extractor schema) rows))

(defun distinct-rows (rows schema)
  (remove-duplicates rows :test (row-equality-tester schema)))

(defun sorted-rows (rows schema order-by)
  (sort (copy-seq rows) (row-comparator order-by schema)))

```

Конечно, самыми интересными частями `select` является реализация функций `extractor`, `row-equality-tester` и `row-comparator`.

Как вы можете заключить из того, как эти функции используются, каждая из этих функций должна возвращать новую функцию. Например, функция `project-columns` использует значение, возвращенное функцией `extractor` в качестве аргумента функции `MAP`. Поскольку `project-columns` предназначена для возврата набора строк с только определенными значениями колонок, вы можете заключить, что `extractor` возвращает функцию, которая получает строку в качестве аргумента, и возвращает новую строку, которая содержит только колонки, указанные в переданной ей схеме. Вот как мы можем реализовать эту функцию:

```

(defun extractor (schema)
  (let ((names (mapcar #'name schema)))
    #'(lambda (row)
        (loop for c in names collect c collect (getf row c)))))

```

Отметьте то, как вы можете выполнить задачу по извлечению имен из схемы за пределами тела замыкания — поскольку замыкание может быть вызвано несколько раз, вы захотите чтобы в нем выполнялось как можно меньше действий при каждом вызове.

Функции `row-equality-tester` и `row-comparator` реализуются аналогичным образом. Для того, чтобы принять решение о равенстве двух строк, вам необходимо применить соответствующие функции сравнения каждой из колонок к значениям соответствующих колонок. Из материала главы 22 мы знаем, что `LOOP` всегда возвращает `NIL` когда пара значений не проходит тест, в противном случае `LOOP` вернет `T`.


```
(defun row-equality-tester (schema)
  (let ((names (mapcar #'name schema))
        (tests (mapcar #'equality-predicate schema))))
    #'(lambda (a b)
      (loop for name in names and test in tests
            always (funcall test (getf a name) (getf b name))))))
```

Расположение двух строк по порядку – более сложная задача. В Lisp функции сравнения возвращают истинное значение, если первый аргумент должен быть расположен перед вторым аргументом, или NIL в противном случае. Таким образом, NIL может означать, что второй аргумент должен быть расположен перед первым аргументом, или оба аргумента равны. Мы также хотим, чтобы функции сравнения строк вели себя точно также – возвращали T если первая строка должна быть перед второй, и NIL в противном случае.

Таким образом, для сравнения двух строк, вы должны сравнить значения тех колонок, по которым вы будете проводить сортировку, используя соответствующую функцию сравнения для каждой из колонок. Сначала вызывается функция сравнения со значением из первой строки в качестве первого аргумента. Если функция сравнения вернет истинное значение, то это означает, что первая колонка точно должна располагаться перед второй колонкой, так что вы можете сразу вернуть значение T.

Но если функция сравнения вернула NIL, то вам нужно определить почему это произошло – либо второе значение должно быть поставлено перед первым, или они равны. Так что вам необходимо снова вернуть функцию сравнения, но поменять аргументы местами. Если функция сравнения вернет истинное значение, то это означает, что вторая строка должна стоять перед первой, и вы можете сразу вернуть NIL. В противном случае, значения в данной колонке равны, и вам необходимо перейти к следующей колонке. Если вы проверите все колонки и не получите однозначного сравнения в пользу одной из строк, то эти строки равны, и вы должны вернуть NIL. Функция, которая реализует такой алгоритм, будет выглядеть следующим образом:

```
(defun row-comparator (column-names schema)
  (let ((comparators (mapcar #'comparator (extract-schema column-names schema))))
    #'(lambda (a b)
      (loop
        for name in column-names
        for comparator in comparators
        for a-value = (getf a name)
        for b-value = (getf b name)
        when (funcall comparator a-value b-value) return t
        when (funcall comparator b-value a-value) return nil
        finally (return nil)))))
```

Функции отбора (FIXME Matching)

Аргумент `:where` функции `select` может быть любой функцией, которая принимает в качестве аргумента строку и возвращает истинное значение, если она должна быть включена в результаты. Однако на практике, вам редко понадобится вся мощь вручную написанного кода для выражения критериев запроса. Так что вы должны лишь реализовать две функции: `matching` и `in`, которые будут создавать функции запроса, которые позволят вам создавать общие виды запросов, а также возьмут на себя заботу об использовании соответствующих функций равенства и нормализации для каждой из колонок.

FIXME The workhouse query-function constructor will be функция `matching`, которая возвращает функцию, которая будет сравнивать строку с конкретными значениями колонок. Вы

увидели как она может быть использована в предыдущих примерах `select`. Например, такой вызов `matching`:

```
(matching *mp3s* :artist "Green Day")
```

вернет функцию, которая будет выбирать строки, в которых значение колонки `:artist` равно `"Green Day"`. Вы также можете передавать множество имен колонок и значений – возвращаемая функция будет возвращать истинное значение только тогда, когда все колонки имеют заданные значения. Например, следующий вызов вернет замыкание, которое будет принимать строки, в которых артист равен `"Green Day"` и альбом равен `"American Idiot"`:

```
(matching *mp3s* :artist "Green Day" :album "American Idiot")
```

Вам необходимо передать функции `matching` объект `table`, поскольку функции необходим доступ к схеме таблицы для получения функций сравнения и нормализации для тех колонок, для которых выполняется отбор данных.

Вы строите функцию, возвращаемую функцией `matching` из меньших функций, каждая из которых отвечает за проверку значения одной из колонок. Для того, чтобы создать эти функции, вы должны определить функцию `column-matcher`, которая получает объект `column` и не нормализованное значение и возвращает функцию, которая получает строку и возвращает истинное значение в том случае, если значение заданной колонки соответствует нормализованному значению заданного аргумента.

```
(defun column-matcher (column value)
  (let ((name (name column))
        (predicate (equality-predicate column))
        (normalized (normalize-for-column value column)))
    #'(lambda (row) (funcall predicate (getf row name) normalized))))
```

Затем вы создаете список функций `column-matching` для заданных имен и значений, переданных функции `column-matchers`:

```
(defun column-matchers (schema names-and-values)
  (loop for (name value) on names-and-values by #'cddr
        when value collect
        (column-matcher (find-column name schema) value)))
```

Теперь вы можете реализовать `matching`. Снова, заметьте, что вы делаете как можно больше работы за пределами замыкания, чтобы выполнить эти операции один раз при создании замыкания, а не при его вызове для каждой из строк таблицы.

```
(defun matching (table &rest names-and-values)
  "Build a where function that matches rows with the given column values."
  (let ((matchers (column-matchers (schema table) names-and-values)))
    #'(lambda (row)
        (every #'(lambda (matcher) (funcall matcher row)) matchers))))
```

Эта функция выглядит как небольшой клубок замыканий, но стоит пристальней посмотреть на нее для того, чтобы получить "наслаждение" (FIXME flavor) от возможности программирования с функциями как объектами первого класса(FIXME?)

Задачей `matching` является возврат функции, которая будет выполняться для каждой строки в таблице для того, чтобы определить – должна ли эта строка быть включена в результат, или нет.

Так что функция `matching` возвращает замыкание принимающее один параметр – строку `row`.

Теперь вспомните, что функция `EVERY` принимает функцию-предикат в качестве первого аргумента, и возвращает истинное значение, только если функция будет возвращать истинное значение для каждого из элементов списка, который передан `EVERY` в качестве второго аргумента. Однако, в нашем случае список, переданный `EVERY` является списком функций – функций отбора для конкретных колонок. Все что вам нужно знать – это то, что каждая функция отбора колонки, при запуске для строки, для которой вы проводите проверку, возвращает истинное значение. Так что в качестве функции-предиката для `EVERY` вы передаете еще одно замыкание, которое применит `FUNCALL` к функции отбора колонки, передав ей параметр `row`.

Другой полезной функцией отбора является `in`, которая возвращает функцию, которая отбирает строки, где значение определенной колонки входит в заданный набор значений. Функция `in` будет принимать два аргумента – имя колонки, и таблицу, которая содержит значения, с которыми вы будете сравнивать. Предположим, например, что вы хотите найти все песни в базе данных MP3, у которых названия совпадают с названиями песен исполняемых *Dixie Chicks*. Вы можете написать это выражение `where` используя функцию `in` и вспомогательный запрос, например, вот так:

```
(select
  :columns '(:artist :song)
:from *mp3s*
:where (in :song
  (select
    :columns :song
    :from *mp3s*
    :where (matching *mp3s* :artist "Dixie Chicks"))))
```

Хотя запросы более сложные, но реализация `in` намного проще чем реализация `matching`.

```
(defun in (column-name table)
  (let ((test (equality-predicate (find-column column-name (schema table))))
        (values (map 'list #'(lambda (r) (getf r column-name)) (rows table))))
    #'(lambda (row)
      (member (getf row column-name) values :test test))))
```

Работа с результатами выполнения запросов

Поскольку `select` возвращает другую таблицу, вам необходимо немного подумать о том, как вы будете осуществлять доступ к отдельным значениям. Если вы уверены, что вы никогда не измените способ представления данных в таблице, то вы можете просто сделать структуру таблицы частью API и указать, что класс `table` имеет слот `rows` который является вектором содержащим списки свойств, и для доступа к данным в таблице использовать стандартные функции Common Lisp для работы с векторами и списками свойств. Но представление данных – это внутренняя деталь, которую вы можете захотеть изменить. Также вы можете не захотеть, чтобы другие разработчики напрямую работали с данными, например, вы можете захотеть, чтобы никто не мог с помощью `SETF` вставить в строку ненормализованное значение. Так что хорошей идеей может быть определение нескольких абстракций, которые будут обеспечивать нужные вам операции. Так что, если вы захотите изменить внутреннее представление данных, то вам нужно будет изменить только реализацию этих функций и макросов. И хотя Common Lisp не позволяет вам полностью запретить доступ к "внутренним" данным, путем предоставления официального API вы по крайней мере сможете указать где проходит граница, разграничивающая внешнее и внутреннее представление.

Вероятно, наиболее часто используемой операцией с результатами запроса будет итерация по отдельным строкам, и выделение значений конкретных колонок. Так что вам нужно предоставить возможность для выполнения этих операций без прямого доступа к вектору строк или использования GETF для получения значения колонки внутри строки.

Реализация этих операций является тривиальной – эти функции являются лишь wrappers вокруг кода, который бы вы написали, если у вас не было этих абстракций. Вы можете предоставить два способа итерации по строкам таблицы: макрос `do-rows`, который обеспечивает базовый способ организации циклов, и функцию `map-rows`, которая создает список, содержащий результаты применения заданной функции к каждой строке таблицы.

```
(defmacro do-rows ((row table) &body body)
  `(loop for ,row across (rows ,table) do ,@body))
(defun map-rows (fn table)
  (loop for row across (rows table) collect (funcall fn row)))
```

Для получения значения конкретной колонки внутри строки, вы должны реализовать функцию `column-value`, которая будет принимать в качестве аргументов строку и название колонки, и будет возвращать соответствующее значение. Опять же, это лишь тривиальный wrapper вокруг кода, который бы вы и так написали. Но если вы позже измените внутреннее представление данных, то пользователи `column-value` не будут об этом знать.

```
(defun column-value (row column-name)
  (getf row column-name))
```

Хотя `column-value` является достаточной абстракцией доступа к значениям колонок, вам достаточно часто нужно получать одновременный доступ к значениям сразу нескольких колонок. Так что мы реализуем макрос `with-column-values`, который будет связывать набор переменных со значениями извлеченными из строки используя соответствующие именованные параметры. Так что вместо использования такого кода:

```
(do-rows (row table)
  (let ((song (column-value row :song))
        (artist (column-value row :artist))
        (album (column-value row :album)))
    (format t "~a by ~a from ~a~%" song artist album)))
```

вы можете просто написать следующим образом:

```
(do-rows (row table)
  (with-column-values (song artist album) row
    (format t "~a by ~a from ~a~%" song artist album)))
```

И снова, реализация не является очень сложной, если вы используете макрос `once-only` из главы 8.

```
(defmacro with-column-values ((&rest vars) row &body body)
  (once-only (row)
    `(let ,(column-bindings vars row) ,@body)))
(defun column-bindings (vars row)
  (loop for v in vars collect `(,v (column-value ,row ,(as-keyword v)))))
(defun as-keyword (symbol)
  (intern (symbol-name symbol) :keyword))
```

И в заключение, вы должны предоставить функции для получения количества строк в таблице, а также для доступа к конкретной строке используя числовой индекс.

```
(defun table-size (table)
  (length (rows table)))
(defun nth-row (n table)
  (aref (rows table) n))
```

Другие операции с базой данных

И в заключение, вы реализуете несколько дополнительных операций с базой данных, которые будут использованы в главе 29. Первые две из них являются аналогами выражения DELETE языка SQL. Функция `delete-rows` используется для удаления из таблицы строк, соответствующих некоторому критерию. Также как и `select` она принимает именованные аргументы `:from` и `:where`. Но в отличие от `select`, эта функция не возвращает новую таблицу – вместо этого, она изменяет таблицу, переданную в качестве аргумента the `:from`.

```
(defun delete-rows (&key from where)
  (loop
    with rows = (rows from)
    with store-idx = 0
    for read-idx from 0
    for row across rows
    do (setf (aref rows read-idx) nil)
    unless (funcall where row) do
      (setf (aref rows store-idx) row)
      (incf store-idx)
    finally (setf (fill-pointer rows) store-idx)))
```

В интересах повышения производительности, вы можете также реализовать отдельную функцию для удаления всех строк из таблицы.

```
(defun delete-all-rows (table)
  (setf (rows table) (make-rows *default-table-size*)))
```

Оставшиеся функции для работы с базой данных не имеют аналогов среди операций с реляционными базами данных, но они будут полезны при написании приложения использующего базу данных MP3. Первой среди них является функция, которая сортирует строки таблицы изменяя ее.

```
(defun sort-rows (table &rest column-names)
  (setf (rows table) (sort (rows table) (row-comparator column-names (schema table)))))
table)
```

С другой стороны, в приложении, работающем с базой данных MP3, вам может понадобиться функция, которая перемешивает строки в таблице, используя функцию `nshuffle-vector` из главы 23.

```
(defun shuffle-table (table)
  (nshuffle-vector (rows table)
    table))
```

И в заключение, снова для приложения работающего с базой данных MP3, вы должны

реализовать функцию которая будет выбирать N произвольных строк, возвращая результат в виде новой таблицы. Эта функция также использует `nshuffle-vector` вместе с версией `random-sample`, основанной на *Алгоритме S* из книги "Искусство программирования, т.2. Получисленные алгоритмы, 3 изд." Дональда Кнута, и который мы обсуждали в главе 20.

```
(defun random-selection (table n)
  (make-instance
    'table
    :schema (schema table)
    :rows (nshuffle-vector (random-sample (rows table) n))))
(defun random-sample (vector n)
  "Based on Algorithm S from Knuth. TAOCP, vol. 2. p. 142"
  (loop with selected = (make-array n :fill-pointer 0)
    for idx from 0
    do
      (loop
        with to-select = (- n (length selected))
        for remaining = (- (length vector) idx)
        while (>= (* remaining (random 1.0)) to-select)
        do (incf idx))
      (vector-push (aref vector idx) selected)
      when (= (length selected) n) return selected))
```

Имея данный код, вы будете готовы создать (в главе 29) веб-интерфейс для просмотра коллекции файлов в формате MP3. Но до этого вам необходимо реализовать часть сервера, которая будет транслировать поток музыки в формате MP3 используя протокол Shoutcast, что и является темой следующей главы.

Практика. Сервер Shoutcast

В этой главе вы разработаете еще одну важную часть Web-приложения для потокового вещания музыки в формате MP3, а именно – сервер, реализующий протокол Shoutcast, который выполняет потоковое вещание в формате MP3 пользовательским клиентам, таким как iTunes, XMMS, или Winamp.

Протокол Shoutcast

Протокол Shoutcast был создан сотрудниками компании Nullsoft, создателя программы Winamp. Он был спроектирован для поддержки потокового вещания в Internet – Shoutcast DJ отправляет аудио файлы с персональных компьютеров на центральный сервер Shoutcast, который затем отправляет эти данные в виде потока любому из подключенных слушателей.

Сервер, который вы напишите, в действительности реализует только половину функциональности настоящего сервера Shoutcast – вы будете использовать протокол, который используют сервера Shoutcast для потокового вещания MP3 слушателям, но ваш сервер будет способен передавать только те песни, которые уже загружены на тот компьютер, на котором выполняется ваш сервер.

Вам необходимо знать только две части протокола Shoutcast: формат запроса, который клиент делает для того, чтобы начать получать поток данных, и формат ответа, включая механизм, который используется для вставки данных о проигрываемой композиции в поток данных.

Начальный запрос от клиента MP3 к серверу Shoutcast выглядит также как обычный запрос протокола HTTP. В ответе сервер Shoutcast отправляет ответ ICY, который выглядит также как и ответ HTTP, за исключением строки ICY вместо обычной строки версии HTTP, и немного отличающимися заголовками. После отправки заголовков и пустой строки, сервер начинает отправлять потенциально бесконечный поток данных в формате MP3.

Единственной сложной (FIXME tricky) вещью в протоколе Shoutcast является способ вставки информации о песне в данные, отправляемые клиенту. Проблемой, с которой столкнулись дизайнеры протокола Shoutcast, заключалась в нахождении возможности передачи клиенту новой информации о песне сервером Shoutcast при начале проигрывания новой песни, так что клиент мог бы отображать эту информацию в интерфейсе. (Возвращаясь к главе 25, вспоминаем что формат MP3 не обеспечивает механизмов для кодирования метаданных). Хотя одной из целей создания ID3v2 было обеспечение лучшей совместимости с потоковой передачей файлов MP3, сотрудники Nullsoft решили идти своим путем и изобрели новую схему, которую проще реализовать и клиенту и серверу. Это конечно было идеальным случаем, поскольку они сами были авторами клиента для проигрывания MP3.

Их решение заключалось в простом игнорировании структуры данных MP3 и вставке метаданных каждые n байт. И клиент принимал на себя ответственность за удаление метаданных из потока, так чтобы они не рассматривались как данные MP3. Поскольку отправка метаданных клиенту, который не готов к их приему, может вызывать проблемы с воспроизведением звука, то сервер должен отправлять метаданные только если запрос содержит специальный заголовок `Icy-Metadata`. И для того, чтобы клиент знал как часто метаданные будут передвигаться, сервер должен отправить клиенту заголовок `Icy-MetaInt` чьим значением является число байт данных в формате MP3, которые будут переданы между двумя пакетами с метаданными.

Основное содержание метаданных – строка вида `StreamTitle='title';` где `title` является заголовком текущей песни, и не может содержать знак одинарной кавычки. Это содержимое закодировано как массив байт разделенный указателями длины: сначала отправляет одиночный байт, показывающий сколько 16-байтовых блоков будет отправлено, за которым следуют эти

блоки. Они содержат саму строку в кодировке ASCII, и последний блок дополнен нулевыми байтами до 16-байтовой границы.

Таким образом, наименьшим допустимым блоком метаданных является единственный байт, равный нулю, что означает что за ним не следует ни одного блока. Если сервер не нуждается в обновлении метаданных, то он может отправить такой пустой блок, но он должен отправить как минимум один байт, так что клиент не будет отбрасывать данные MP3.

Источники песен

Поскольку сервер Shoutcast должен продолжать передавать поток данных клиенту все время пока он подключен к нему, то вам необходимо обеспечить ваш сервер источником песен из которых он сможет брать данные. В Web-приложении, каждый подключенный клиент будет иметь список песен, с которым он сможет работать через Web-интерфейс. Но для того, чтобы избежать излишней зависимости между модулями, вы должны определить интерфейс, который сможет использовать сервер Shoutcast для получения списка проигрываемых песен. Вы можете сейчас написать простую реализацию этого интерфейса, и заменить ее на более сложную при написании Web-приложения, которое вы будете создавать в главе 29.

FIXME this is embdedded table, will fixed in latex

Пакет

Объявление разрабатываемого вами пакета будет выглядеть примерно так:

```
(defpackage :com.gigamonkeys.shoutcast
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.id3v2)
  (:export :song
           :file
           :title
           :id3-size
           :find-song-source
           :current-song
           :still-current-p
           :maybe-move-to-next-song
           :*song-source-type*))
```

FIXME end of table

Основной идеей для создания интерфейса является то, что сервер Shoutcast будет находить источник песен, основываясь на идентификаторе, выделенном из объекта AllegroServe, представляющего запрос. Затем можно сделать следующие три действия над выделенным источником песен.

- * Получить текущую песню из источника песен
- * Сообщить источнику песен, что мы закончили работу над текущей песней
- * Запросить у источника песен о том, все еще является ли текущей та песня, которую мы запрашивали ранее.

Последняя операция необходима, поскольку могут быть такие случаи, и мы увидим их в главе 29, когда мы работаем с источником песен вне сервера Shoutcast. Вы можете выразить операции, необходимые серверу Shoutcast, с помощью следующих обобщенных функций:


```
(defgeneric current-song (source)
  (:documentation "Return the currently playing song or NIL."))
(defgeneric maybe-move-to-next-song (song source)
  (:documentation
    "If the given song is still the current one update the value
    returned by current-song."))
(defgeneric still-current-p (song source)
  (:documentation
    "Return true if the song given is the same as the current-song."))
```

Функция `maybe-move-to-next-song` определена таким способом, что за одну операцию проверяется – является ли данная песня текущей, и если это так, то источник песен перемещается к следующей песне. Это будет важным в следующей главе, когда вам нужно будет реализовать источник песен, который будет доступен из двух потоков выполнения.

Для представления информации о песне, которая необходима серверу Shoutcast, вы можете определить класс `song`, со слотами, которые будут хранить имя файла MP3, заголовок, который будет отправлен в качестве метаданных Shoutcast, и размер тага ID3, так что он может быть пропущен во время передачи файла.

```
(defclass song ()
  ((file :reader file :initarg :file)
   (title :reader title :initarg :title)
   (id3-size :reader id3-size :initarg :id3-size)))
```

Значение, возвращенное `current-song` (оно же и является первым аргументом функций `still-current-p` и `maybe-move-to-next-song`) будет экземпляром класса `song`.

В добавок к этому, вам необходимо определить обобщенную функцию, которую сервер сможет использовать для нахождения источника песен, основываясь на желательном типе источника и объекте, представляющем запрос. Методы будут специализировать параметр `type` для того, чтобы возвращать разные виды источников песен, и будут вытягивать различную информацию, в которой они нуждаются для определения какой источник песен необходимо возвращать, из объекта `request`.

```
</code> (defgeneric find-song-source (type request)
```

```
  (:documentation "Find the song-source of the given type for the given request."))
```

```
</code>
```

Однако, в данной главе вы можете использовать самую простую реализацию этого интерфейса, которая будет всегда возвращать один и тот же объект – простую очередь объектов `song`, которой вы сможете управлять через строку ввода команд. Вы можете начать эту реализацию путем определения класса `simple-song-queue`, и глобальной переменной `*songs*`, которая содержит экземпляр данного класса.

```
(defclass simple-song-queue ()
  ((songs :accessor songs :initform (make-array 10 :adjustable t :fill-pointer
0))
   (index :accessor index :initform 0)))
(defparameter *songs* (make-instance 'simple-song-queue))
```

Затем вы можете определить метод `find-song-source` специализированный через EQL для

символа `singleton`, который будет возвращать экземпляр объекта, хранимый в переменной `*songs*`.

```
(defmethod find-song-source ((type (eql 'singleton)) request)
  (declare (ignore request))
  *songs*)
```

Теперь вам всего-лишь надо реализовать методы для трех обобщенных функций, которые будут использоваться сервером Shoutcast.

```
(defmethod current-song ((source simple-song-queue))
  (when (array-in-bounds-p (songs source) (index source))
    (aref (songs source) (index source))))
(defmethod still-current-p (song (source simple-song-queue))
  (eql song (current-song source)))
(defmethod maybe-move-to-next-song (song (source simple-song-queue))
  (when (still-current-p song source)
    (incf (index source))))
```

И в целях тестирования, вам необходимо обеспечить возможность добавления песен в очередь.

```
(defun add-file-to-songs (file)
  (vector-push-extend (file->song file) (songs *songs*)))
(defun file->song (file)
  (let ((id3 (read-id3 file)))
    (make-instance
      'song
      :file (namestring (truename file))
      :title (format nil "~a by ~a from ~a" (song id3) (artist id3) (album id3))
      :id3-size (size id3))))
```

Реализация сервера Shoutcast

Теперь вы готовы к реализации сервера Shoutcast. Поскольку протокол Shoutcast практически основан на HTTP, вы можете реализовать сервер в виде функции внутри AllegroServe. Однако, поскольку вам нужно будет взаимодействовать с некоторыми низкоуровневыми функциями AllegroServe, то вы не сможете использовать макрос `define-url-function` из главы 26. Вместо этого, вам нужно написать обычную функцию, которая будет выглядеть примерно так:

```
(defun shoutcast (request entity)
  (with-http-response
    (request entity :content-type "audio/MP3" :timeout *timeout-seconds*)
    (prepare-icy-response request *metadata-interval*)
    (let ((wants-metadata-p (header-slot-value request :icy-metadata)))
      (with-http-body (request entity)
        (play-songs
          (request-socket request)
          (find-song-source *song-source-type* request)
          (if wants-metadata-p *metadata-interval*))))))
```

Затем опубликуйте эту функцию для пути `/stream.mp3`, например вот так:

```
(publish :path "/stream.mp3" :function 'shoutcast)
```

В вызове `with-http-response`, в добавление к стандартным параметрам `request` и `entity`, вам необходимо передать аргументы `:content-type` и `:timeout`. Аргумент `:content-type`

сообщает AllegroServe как установить значение заголовка Content-Type. А аргумент `:timeout` указывает количество времени (в секундах), которое дает AllegroServe функции для генерации ответа. По умолчанию AllegroServe отменяет каждый запрос через пять минут. Поскольку вы собираетесь передавать поток практически бесконечно, то вам необходимо указать большее значение. Не существует способа указать AllegroServe чтобы он не отменял запрос, так что вы должны установить подходящее большое значение в переменной `*timeout-seconds*`, например, 10 лет, переведенных в секунды.

```
(defparameter *timeout-seconds* (* 60 60 24 7 52 10))
```

Затем, внутри тела `with-http-response` и до вызова `with-http-body`, который выполнит отправку заголовков ответа, вам необходимо напрямую поработать с ответом, который отправит AllegroServe. Функция `prepare-icy-response` выполняет все необходимые действия: изменение строки протокола со значения по умолчанию – "HTTP" на "ICY", и добавление заголовков, специфических для Shoutcast. Вам также необходимо добавить код для обхода ошибки в iTunes, который заставит AllegroServe не использовать FIXME chunked transfer-encoding. Функции `request-reply-protocol-string`, `request-uri` и `reply-header-slot-value` являются частью of AllegroServe.

```
(defun prepare-icy-response (request metadata-interval)
  (setf (request-reply-protocol-string request) "ICY")
  (loop for (k v) in (reverse
    `(:|icy-metaint| , (princ-to-string metadata-interval))
      (:|icy-notice1| "<BR>This stream blah blah blah<BR>")
      (:|icy-notice2| "More blah")
      (:|icy-name| "MyLispShoutcastServer")
      (:|icy-genre| "Unknown")
      (:|icy-url| , (request-uri request))
      (:|icy-pub| "1")))
    do (setf (reply-header-slot-value request k) v))
  ;; iTunes, despite claiming to speak HTTP/1.1, doesn't understand
  ;; chunked Transfer-encoding. Grrr. So we just turn it off.
  (turn-off-chunked-transfer-encoding request))
(defun turn-off-chunked-transfer-encoding (request)
  (setf (request-reply-strategy request)
    (remove :chunked (request-reply-strategy request))))
```

Внутри выражения `with-http-body` функции `shoutcast`, вы выполняете потоковое вещание в формате MP3. Функция `play-songs` берет поток, в который вы должны писать данные, источник песен и интервал передачи метаданных, или NIL, если клиент не хочет получать метаданные. Поток – это сокет, полученный из объекта `request`, источник песен получается при помощи функции `find-song-source`, а интервал передачи метаданных берется из глобальной переменной `*metadata-interval*`. Тип источника песен контролируется переменной `*song-source-type*`, который сейчас должен быть установлен в значение `singleton` для того, чтобы использовать `simple-song-queue`, которую мы уже реализовали.

```
(defparameter *metadata-interval* (expt 2 12))
(defparameter *song-source-type* 'singleton)
```

Сама функция `play-songs` не делает ничего сложного – она в цикле вызывает функцию `play-current`, которая берет на себя всю тяжесть задачи по отправке содержимого отдельного файла MP3, пропускания тегов ID3 и вставки метаданных ICY. Единственной трудностью является отслеживание момента отправки метаданных.

Поскольку вы должны отправлять блоки метаданных через фиксированные интервалы,

независимо от того, когда вы переключаетесь с отправки одного файла на другой, то каждый раз когда вы вызываете `play-current`, то вам необходимо указать когда следующие метаданные должны быть переданы, и при возврате, эта функция должна вернуть аналогичное значение, так что вы сможете передать эти данные в следующем вызове `play-current`. Если `play-current` получает `NIL` от источника песен, то она также вернет `NIL`, что позволяет завершить цикл `LOOP` внутри `play-songs`.

В дополнение к выполнению цикла, `play-songs` также использует `HANDLER-CASE` для перехвата ошибок, которые будут выданы когда клиент MP3 отключится от сервера, и одна из процедуры записи в `play-current` приведет к выдаче ошибки. Поскольку `HANDLER-CASE` находится вне `LOOP`, то обработка ошибки приведет к прерыванию цикла, позволяет выполнить выход из `play-songs`.

```
(defun play-songs (stream song-source metadata-interval)
  (handler-case
    (loop
      for next-metadata = metadata-interval
      then (play-current
            stream
            song-source
            next-metadata
            metadata-interval)
      while next-metadata)
    (error (e) (format *trace-output* "Caught error in play-songs: ~a" e))))
```

И теперь вы готовы к реализации функции `play-current`, которая выполняет отставку данных Shoutcast. Основной идея заключается в том, что вы получаете текущую песню от источника песен, открываете файл, содержащий ее, и затем выполняете цикл в котором читаете данные из файла и записываете их в сокет, до тех пор, пока вы не достигните конца файла, или текущая песня не перестанет быть текущей.

Имеется только две трудности: одна из них заключается в том, что вы должны быть уверены, что вы отправляете метаданные через заданный интервал. Другой является то, что если файл начинается с тага ID3, то вам нужно пропустить его. Если вы не особо беспокоитесь об эффективности ввода-вывода, то вы можете реализовать `play-current` вот так:

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song))))
        (with-open-file (mp3 (file song))
          (unless (file-position mp3 (id3-size song))
            (error "Can't skip to position ~d in ~a" (id3-size song) (file
song)))
          (loop for byte = (read-byte mp3 nil nil)
            while (and byte (still-current-p song song-source)) do
              (write-byte byte out)
              (decf next-metadata)
              when (and (zerop next-metadata) metadata-interval) do
                (write-sequence metadata out)
                (setf next-metadata metadata-interval))
              (maybe-move-to-next-song song song-source)))
          next-metadata)))
```

Эта функция получает текущую песню из источника песен, и затем получает буфер, содержащий метаданные путем передачи названия песни функции `make-icy-metadata`. Затем она

открывает файл и пропускает тег ID3 используя функцию `FILE-POSITION` с двумя аргументами. Затем она начинает читать байты из файла и записывать их в сокет.

Эта функция прервет цикл когда достигнет конца файла, или когда источник песен изменит текущую песню. Между тем, когда `next-metadata` будет равен нулю (если вы вообще будете отправлять метаданные), эта функция записывает метаданные в поток и сбрасывает `next-metadata` в начальное значение. После завершения цикла, она проверяет – является ли песня все еще текущей в источнике песен, и если это так, то это значит что мы вышли из цикла из-за того, что прочитали весь файл, и в этом случае она сообщает источнику песен о необходимости перемещения к следующей песне. В противном случае, цикл прерван из-за того, что кто-то изменил текущую песню, и функция просто выполняет возврат без дополнительных действий. В любом случае, она возвращает число байт, оставшихся до отправки следующей порции метаданных, так что это значение может быть использовано при следующем вызове `play-current`.

Реализация функции `make-icy-metadata`, которая получает название текущей песни и формирует массив байт, содержащий правильно отформатированный блок метаданных ICY, также проста.

```
(defun make-icy-metadata (title)
  (let* ((text (format nil "StreamTitle='~a'" (substitute #\Space #' title)))
        (blocks (ceiling (length text) 16))
        (buffer (make-array (1+ (* blocks 16))
                            :element-type '(unsigned-byte 8)
                            :initial-element 0)))
    (setf (aref buffer 0) blocks)
    (loop
      for char across text
      for i from 1
      do (setf (aref buffer i) (char-code char)))
    buffer))
```

В зависимости от того, как конкретная реализация Lisp работает с потоками, и от того, сколько клиентов MP3 вы хотите обрабатывать одновременно, простая версия `play-current` может быть достаточно эффективной или нет.

Потенциальной проблемой простой реализации может быть то, что она использует `READ-BYTE` и `WRITE-BYTE` для передачи каждого байта. Возможно, что каждый вызов может приводить к относительно затратному системному вызову чтения или записи одного байта. И даже если в вашем Lisp реализованы потоки с внутренней буферизацией, так что не каждый вызов `READ-BYTE` и `WRITE-BYTE` будет приводить к системному вызову, то все равно, вызов функции не является дешевой операцией. В частности, в реализациях, которые предоставляют потоки, расширяемые пользователем, используя так называемые "серые потоки" (Gray Streams), вызовы `READ-BYTE` и `WRITE-BYTE` могут приводить к вызову обобщенных функций, которые будут приводить к неявной диспатчеризации вызова в зависимости от класса потока. Хотя диспатчеризация обобщенной функции является достаточно быстрой операцией и вы можете сильно не волноваться об этом, но все равно ее вызов более затратен чем вызов обычной функции, и это не та вещь, которую вы захотите выполнять несколько миллионов раз за несколько минут, особенно если вы можете избежать этого.

Более эффективный, но чуть более сложный способ реализации `play-current` – читать и записывать данные блоками используя функции `READ-SEQUENCE` и `WRITE-SEQUENCE`. Это также дает вам шанс привести чтение данных в соответствие с размером блока данных файловой системы, что обеспечит вам лучшую производительность диска. Конечно, вне зависимости от того, какой размер блока вы будете использовать, отслеживание точки отправки метаданных

станет более сложной задачей. Более эффективная версия `play-current` использующая функции `READ-SEQUENCE` и `WRITE-SEQUENCE` может выглядеть вот так:

```
(defun play-current (out song-source next-metadata metadata-interval)
  (let ((song (current-song song-source)))
    (when song
      (let ((metadata (make-icy-metadata (title song)))
            (buffer (make-array size :element-type '(unsigned-byte 8))))
        (with-open-file (mp3 (file song))
          (labels ((write-buffer (start end)
                     (if metadata-interval
                         (write-buffer-with-metadata start end)
                         (write-sequence buffer out :start start :end end)))
            (write-buffer-with-metadata (start end)
              (cond
                ((> next-metadata (- end start))
                 (write-sequence buffer out :start start :end end)
                 (decf next-metadata (- end start)))
                (t
                 (let ((middle (+ start next-metadata)))
                   (write-sequence buffer out :start start :end middle)
                   (write-sequence metadata out)
                   (setf next-metadata metadata-interval)
                   (write-buffer-with-metadata middle end)))))))
        (multiple-value-bind (skip-blocks skip-bytes)
          (floor (id3-size song) (length buffer)))
        (unless (file-position mp3 (* skip-blocks (length buffer)))
          (error "Couldn't skip over ~d ~d byte blocks."
                 skip-blocks (length buffer)))
        (loop for end = (read-sequence buffer mp3)
              for start = skip-bytes then 0
              do (write-buffer start end)
              while (and (= end (length buffer))
                         (still-current-p song song-source)))
        (maybe-move-to-next-song song song-source))))
  next-metadata)))
```

Теперь вы готовы собрать все части вместе. В следующей главе вы напишите Web-интерфейс для сервера Shoutcast, разработанного в данной главе, и использующего базу данных MP3 из главы 27 в качестве источника песен.

29. Практика. Браузер MP3 файлов

Заключительным шагом в построении приложения для потокового вещания MP3 является разработка Web-интерфейса, который позволит пользователям найти песни, которые они хотят слушать, и добавлять их в списки песен которые будут использоваться сервером Shoutcast при получении запроса от MP3-клиента пользователя. Для этого компонента приложения вы соберете вместе несколько компонентов, разработанных в предыдущих главах: базу данных MP3, макрос `define-url-function` из главы Chapter 26 и, конечно, сам сервер Shoutcast.

Списки песен

Основная идея интерфейса заключается в том, что каждый MP3-клиент, который подключается к серверу Shoutcast, получает отдельный список песен, который служит источником песен для сервера Shoutcast. Список песен также реализует дополнительные функции, не нужные серверу Shoutcast: используя Web-интерфейс, пользователь сможет добавлять песни в список, удалять песни из него, или изменять порядок проигрывания путем сортировки и перемешивания.

Вы можете определить класс для представления списка песен следующим образом:

```
(defclass playlist ()
  ((id :accessor id :initarg :id)
   (songs-table :accessor songs-table :initform (make-playlist-table))
   (current-song :accessor current-song :initform *empty-playlist-song*)
   (current-idx :accessor current-idx :initform 0)
   (ordering :accessor ordering :initform :album)
   (shuffle :accessor shuffle :initform :none)
   (repeat :accessor repeat :initform :none)
   (user-agent :accessor user-agent :initform "Unknown")
   (lock :reader lock :initform (make-process-lock))))
```

Идентификатор списка песен (`id`) является ключем, который вы извлекаете из объекта `request`, переданного `find-song-source` когда происходит поиск списка песен. Вам не нужно сохранять его в объекте `playlist`, но это сделает отладку более простой, если вы сможете для произвольного объекта `playlist` определить его идентификатор.

Самым главным объектом `playlist` является слот `songs-table`, который будет хранить объект `table`. Схема этого объекта будет той же самой, что и схема для основной базы данных MP3. Функция `make-playlist-table`, которую вы используете для инициализации `songs-table`, очень проста:

```
(defun make-playlist-table ()
  (make-instance 'table :schema *mp3-schema*))
```

FIXME Start of block

The Package

Вы можете определить пакет для кода данной главы с помощью следующего определения `DEFPACKAGE`:

```
(defpackage :com.gigamonkeys.mp3-browser
  (:use :common-lisp
        :net.aserve
        :com.gigamonkeys.html
        :com.gigamonkeys.shoutcast
        :com.gigamonkeys.url-function
        :com.gigamonkeys.mp3-database
        :com.gigamonkeys.id3v2)
  (:import-from :acl-socket
                :ipaddr-to-dotted
                :remote-host)
  (:import-from :multiprocessing
                :make-process-lock
                :with-process-lock)
  (:export :start-mp3-browser))
```

FIXME End of block

Поскольку это высокоуровневое приложение, оно использует очень много низкоуровневых пакетов. Оно также импортирует три символа из пакета ACL-SOCKET и два из пакета MULTIPROCESSING, поскольку нам необходимы только эти пять символов, и не нужны остальные 139 символов, экспортируемые из этих пакетов.

Из-за сохранения списка песен в виде таблицы, вы можете использовать функции работы с базами данных из главы 27 для работы со списком песен: вы можете добавлять данные в список песен с помощью `insert-row`, удалять песни с помощью `delete-rows`, и изменять порядок проигрывания с помощью `sort-rows` и `shuffle-table`.

Слоты `current-song` и `current-idx` используются для хранения информации о том, какая песня сейчас проигрывается: `current-song` – это объект `song`, в то время как `current-idx` является индексом строки в `songs-table`, относящейся к текущей песне. В разделе "Изменение списка песен" вы увидите как сделать так, чтобы `current-song` обновлялась когда изменяется `current-idx`.

Слоты `ordering` и `shuffle` хранят информацию о том, как песни в `songs-table` должны быть упорядочены. Слот `ordering` хранит ключевое слово, которое описывает то, как таблица `songs-table` должна быть отсортирована, когда она не перемешана. Допустимыми значениями являются `:genre`, `:artist`, `:album` и `:song`. Слот `shuffle` содержит одно из ключевых слов `:none`, `:song` или `:album`, которые определяют как песни в `songs-table` будут перемешаны, если это нужно.

Слот `repeat` также содержит одно ключевых слов `:none`, `:song` или `:all`, которые указывают режим повторения песен в списке проигрывания. Если `repeat` равно `:none`, то после проигрывания последней песни из списка `songs-table`, `current-song` переключается на значение по умолчанию. Когда `repeat` равно `:song`, то все время проигрывается одна и та же песня из `current-song`. И если установлено значение `:all`, то после проигрывания последней песни, сервер начинает играть с начала списка.

Слот `user-agent` хранит значение заголовка `User-Agent`, который отправлен MP3-клиентом при запросе потока. Вам нужно его держать исключительно для использования в Web-интерфейсе – заголовок `User-Agent` идентифицирует программу, которая выполнила запрос, так что вы можете отображать это значение на странице, на который перечислены все списки песен, так что вам будет легче видеть какой из списков песен используется с каким соединением, если к серверу подключено несколько клиентов.

И в заключение, слот `lock` хранит блокировку процесса (FIXME lock) созданную с помощью функции `make-process-lock`, которая является частью пакета `MULTIPROCESSING` из состава Allegro. Вы будете использовать эту блокировку в некоторых функциях, которые изменяют список песен, так что вы будете уверены в том, что только один поток выполнения выполняет изменения списка. Вы можете определить следующий макрос, созданный на основе макроса `with-process-lock` из пакета `MULTIPROCESSING`, чтобы облегчить написание кода, который должен быть выполнен при захвате блокировки данного списка песен:

```
(defmacro with-playlist-locked ((playlist) &body body)
  `(with-process-lock ((lock ,playlist))
    ,@body))
```

Макрос `with-process-lock` получает эксклюзивный доступ к блокировке процесса, и затем выполняет переданные выражения, освобождая блокировку после их выполнения. По умолчанию, `with-process-lock` разрешает выполнять рекурсивные блокировки, что значит, что один и тот же поток выполнения может захватывать одну и ту же блокировку несколько раз.

Списки песен как источники песен

Для того, чтобы использовать списки песен в качестве источника песен для сервера Shoutcast, вам нужно реализовать метод для обобщенной функции `find-song-source` из главы 28. Поскольку у вас будет множество списков песен, то вам необходим способ нахождения нужного списка для конкретного клиента, подключенного к серверу. Первая часть работы достаточно легка – вы можете определить переменную, которая будет хранить хэш-таблицу (с операцией сравнения `EQUAL`), которую вы сможете использовать для отображения из некоторого идентификатора в список песен.

```
(defvar *playlists* (make-hash-table :test #'equal))
```

Вы также можете определить блокировку процесса для защиты доступа к этой хэш-таблице, например, вот так:

```
(defparameter *playlists-lock* (make-process-lock :name "playlists-lock"))
```

Затем определим функцию, которая производит поиск списка песен по заданному идентификатору, создавая новый список песен, если это необходимо, и используя `with-process-lock` для обеспечения доступа к хэш-таблице только из одного потока выполнения.

```
(defun lookup-playlist (id)
  (with-process-lock (*playlists-lock*)
    (or (gethash id *playlists*)
        (setf (gethash id *playlists*) (make-instance 'playlist :id id)))))
```

Затем вы можете реализовать `find-song-source` на основе этой функции, а также функцию `playlist-id`, которая получает объект `request` от `AllegroServe` и возвращает соответствующий идентификатор списка песен. В функции `find-song-source` вы также получаете строку `User-Agent` из объекта `request`, и сохраняете ее в объекте `playlist`.

```
(defmethod find-song-source ((type (eql 'playlist)) request)
  (let ((playlist (lookup-playlist (playlist-id request))))
    (with-playlist-locked (playlist)
      (let ((user-agent (header-slot-value request :user-agent)))
        (when user-agent (setf (user-agent playlist) user-agent))))
      playlist)))
```

Хитрость заключается в том, как вы реализуете функцию `playlist-id`, которая извлекает идентификатор из объекта `request`. У вас имеется несколько возможностей, каждая из которых по-разному влияет на интерфейс пользователя. Вы можете извлечь нужную информацию из объекта `request`, но поскольку вы решили идентифицировать клиента, то вам нужен какой-то способ связывания пользователя веб-интерфейса с соответствующим списком песен.

В данный момент вы можете выбрать тот подход, который "просто работает", поскольку мы подразумеваем, что есть только один MP3-клиент на компьютере, подключающемся к серверу, также как и пользователь, работающий с веб-интерфейсом с компьютера, на котором запущен MP3-клиент: вы будете использовать IP-адрес компьютера пользователя в качестве идентификатора. Таким образом вы можете найти соответствующий список песен для запроса, вне зависимости от того, пришел запрос от MP3-клиента или от веб-браузера. Однако, вы обеспечите в веб-интерфейсе возможность выбора другого списка песен, так что единственным ограничением будет то, что только один MP3-клиент может быть доступен на компьютере пользователя. Реализация `playlist-id` выглядит примерно так:

```
(defun playlist-id (request)
  (ipaddr-to-dotted (remote-host (request-socket request))))
```

Функция `request-socket` является частью `AllegroServe`, а `remote-host` и `ipaddr-to-dotted` являются частью библиотеки `Allegro` для работы с сокетами.

Чтобы позволить использовать списки песен в качестве источников песен для сервера `Shoutcast`, вам необходимо определить методы `current-song`, `still-current-p` и `maybe-move-to-next-song`, которые специализируют параметр `source` для списка песен. Метод `current-song` уже имеет эту функциональность: путем определения процедуры доступа `current-song` для слота `eponymous`, вы автоматически получите метод `current-song` специализированный для списков песен, который будет возвращать значение этого слота. Однако, для того, чтобы сделать доступ к спискам песен безопасным, вам необходимо блокировать доступ к списку песен до доступа к слоту `current-song`. В этом случае, самым простым способом будет определение метода `:around`, например, вот так:

```
(defmethod current-song :around ((playlist playlist))
  (with-playlist-locked (playlist) (call-next-method)))
```

Реализация `still-current-p` также достаточно простая, предполагая, что мы можем быть уверены, что `current-song` будет обновлен на новый объект `song` только тогда, когда текущая песня действительно сменится. Вам снова нужно захватить блокировку процесса для того, чтобы быть уверенным в консистентности состояния списка песен.

```
(defmethod still-current-p (song (playlist playlist))
  (with-playlist-locked (playlist)
    (eql song (current-song playlist))))
```

The trick, then, is to make sure the `current-song` slot gets updated at the right times. However,

the current song can change in a number of ways. The obvious one is when the Shoutcast server calls `maybe-move-to-next-song`. But it can also change when songs are added to the playlist, when the Shoutcast server has run out of songs, or even if the playlist's repeat mode is changed.

Rather than trying to write code specific to every situation to determine whether to update `current-song`, you can define a function, `update-current-if-necessary`, that updates `current-song` if the song object in `current-song` no longer matches the file that the `current-idx` slot says should be playing. Then, if you call this function after any manipulation of the playlist that could possibly put those two slots out of sync, you're sure to keep `current-song` set properly. Here are `update-current-if-necessary` and its helper functions:

```
(defun update-current-if-necessary (playlist)
  (unless (equal (file (current-song playlist))
                (file-for-current-idx playlist))
    (reset-current-song playlist)))
(defun file-for-current-idx (playlist)
  (if (at-end-p playlist)
      nil
      (column-value (nth-row (current-idx playlist) (songs-table playlist))
                    :file)))
(defun at-end-p (playlist)
  (>= (current-idx playlist) (table-size (songs-table playlist))))
```

You don't need to add locking to these functions since they'll be called only from functions that will take care of locking the playlist first.

The function `reset-current-song` introduces one more wrinkle: because you want the playlist to provide an endless stream of MP3s to the client, you don't want to ever set `current-song` to `NIL`. Instead, when a playlist runs out of songs to play—when `songs-table` is empty or after the last song has been played and `repeat` is set to `:none`—then you need to set `current-song` to a special song whose file is an MP3 of silence and whose title explains why no music is playing. Here's some code to define two parameters, `*empty-playlist-song*` and `*end-of-playlist-song*`, each set to a song with the file named by `*silence-mp3*` as their file and an appropriate title:

```
(defparameter *silence-mp3* ...)
(defun make-silent-song (title &optional (file *silence-mp3*))
  (make-instance
    'song
    :file file
    :title title
    :id3-size (if (id3-p file) (size (read-id3 file)) 0)))
(defparameter *empty-playlist-song* (make-silent-song "Playlist empty."))
(defparameter *end-of-playlist-song* (make-silent-song "At end of playlist."))
```

`reset-current-song` uses these parameters when the `current-idx` doesn't point to a row in `songs-table`. Otherwise, it sets `current-song` to a song object representing the current row.

```
(defun reset-current-song (playlist)
  (setf
    (current-song playlist)
    (cond
      ((empty-p playlist) *empty-playlist-song*)
      ((at-end-p playlist) *end-of-playlist-song*)
      (t (row->song (nth-row (current-idx playlist) (songs-table playlist))))))
  (defun row->song (song-db-entry)
    (with-column-values (file song artist album id3-size) song-db-entry
      (make-instance
        'song
        :file file
        :title (format nil "~a by ~a from ~a" song artist album)
        :id3-size id3-size)))
  (defun empty-p (playlist)
    (zerop (table-size (songs-table playlist)))))
```

Now, at last, you can implement the method on `maybe-move-to-next-song` that moves `current-idx` to its next value, based on the playlist's repeat mode, and then calls `update-current-if-necessary`. You don't change `current-idx` when it's already at the end of the playlist because you want it to keep its current value, so it'll point at the next song you add to the playlist. This function must lock the playlist before manipulating it since it's called by the Shoutcast server code, which doesn't do any locking.

```
(defmethod maybe-move-to-next-song (song (playlist playlist))
  (with-playlist-locked (playlist)
    (when (still-current-p song playlist)
      (unless (at-end-p playlist)
        (ecase (repeat playlist)
          (:song) ; nothing changes
          (:none (incf (current-idx playlist)))
          (:all (setf (current-idx playlist)
                     (mod (1+ (current-idx playlist))
                         (table-size (songs-table playlist)))))))
      (update-current-if-necessary playlist)))))
```

Изменение списка песен

The rest of the playlist code is functions used by the Web interface to manipulate playlist objects, including adding and deleting songs, sorting and shuffling, and setting the repeat mode. As in the helper functions in the previous section, you don't need to worry about locking in these functions because, as you'll see, the lock will be acquired in the Web interface function that calls these.

Adding and deleting is mostly a question of manipulating the `songs-table`. The only extra work you have to do is to keep the `current-song` and `current-idx` in sync. For instance, whenever the playlist is empty, its `current-idx` will be zero, and the `current-song` will be the `*empty-playlist-song*`. If you add a song to an empty playlist, then the index of zero is now in bounds, and you should change the `current-song` to the newly added song. By the same token, when you've played all the songs in a playlist and `current-song` is `*end-of-playlist-song*`, adding a song should cause `current-song` to be reset. All this really means, though, is that you need to call `update-current-if-necessary` at the appropriate points.

Adding songs to a playlist is a bit involved because of the way the Web interface communicates which songs to add. For reasons I'll discuss in the next section, the Web interface code can't just give you a simple set of criteria to use in selecting songs from the database. Instead, it gives you the name of a column and a list of values, and you're supposed to add all the songs from the main database where the given column has a value in the list of values. Thus, to add the right songs, you need to first build a ta-

ble object containing the desired values, which you can then use with an in query against the song database. So, add-songs looks like this:

```
(defun add-songs (playlist column-name values)
  (let ((table (make-instance
                  'table
                  :schema (extract-schema (list column-name) (schema *mp3s*)))))
    (dolist (v values) (insert-row (list column-name v) table))
    (do-rows (row (select :from *mp3s* :where (in column-name table)))
      (insert-row row (songs-table playlist)))
    (update-current-if-necessary playlist))
```

Deleting songs is a bit simpler; you just need to be able to delete songs from the songs-table that match particular criteria—either a particular song or all songs in a particular genre, by a particular artist, or from a particular album. So, you can provide a delete-songs function that takes keyword/value pairs, which are used to construct a matching :where clause you can pass to the delete-rows database function.

Another complication that arises when deleting songs is that current-idx may need to change. Assuming the current song isn't one of the ones just deleted, you'd like it to remain the current song. But if songs before it in songs-table are deleted, it'll be in a different position in the table after the delete. So after a call to delete-rows, you need to look for the row containing the current song and reset current-idx. If the current song has itself been deleted, then, for lack of anything better to do, you can reset current-idx to zero. After updating current-idx, calling update-current-if-necessary will take care of updating current-song. And if current-idx changed but still points at the same song, current-song will be left alone.

```
(defun delete-songs (playlist &rest names-and-values)
  (delete-rows
   :from (songs-table playlist)
   :where (apply #'matching (songs-table playlist) names-and-values))
  (setf (current-idx playlist) (or (position-of-current playlist) 0))
  (update-current-if-necessary playlist))
(defun position-of-current (playlist)
  (let* ((table (songs-table playlist))
        (matcher (matching table :file (file (current-song playlist))))
        (pos 0))
    (do-rows (row table)
      (when (funcall matcher row)
        (return-from position-of-current pos))
      (incf pos))))
```

You can also provide a function to completely clear the playlist, which uses delete-all-rows and doesn't have to worry about finding the current song since it has obviously been deleted. The call to update-current-if-necessary will take care of setting current-song to NIL.

```
(defun clear-playlist (playlist)
  (delete-all-rows (songs-table playlist))
  (setf (current-idx playlist) 0)
  (update-current-if-necessary playlist))
```

Sorting and shuffling the playlist are related in that the playlist is always either sorted or shuffled. The shuffle slot says whether the playlist should be shuffled and if so how. If it's set to :none, then the playlist is ordered according to the value in the ordering slot. When shuffle is :song, the playlist will be randomly permuted. And when it's set to :album, the list of albums is randomly permuted, but the songs within each album are listed in track order. Thus, the sort-playlist function, which will be called

by the Web interface code whenever the user selects a new ordering, needs to set ordering to the desired ordering and set shuffle to :none before calling order-playlist, which actually does the sort. As in delete-songs, you need to use position-of-current to reset current-idx to the new location of the current song. However, this time you don't need to call update-current-if-necessary since you know the current song is still in the table.

```
(defun sort-playlist (playlist ordering)
  (setf (ordering playlist) ordering)
  (setf (shuffle playlist) :none)
  (order-playlist playlist)
  (setf (current-idx playlist) (position-of-current playlist)))
```

In order-playlist, you can use the database function sort-rows to actually perform the sort, passing a list of columns to sort by based on the value of ordering.

```
(defun order-playlist (playlist)
  (apply #'sort-rows (songs-table playlist)
    (case (ordering playlist)
      (:genre '(:genre :album :track))
      (:artist '(:artist :album :track))
      (:album '(:album :track))
      (:song '(:song))))))
```

The function shuffle-playlist, called by the Web interface code when the user selects a new shuffle mode, works in a similar fashion except it doesn't need to change the value of ordering. Thus, when shuffle-playlist is called with a shuffle of :none, the playlist goes back to being sorted according to the most recent ordering. Shuffling by songs is simple—just call shuffle-table on songs-table. Shuffling by albums is a bit more involved but still not rocket science.

```
(defun shuffle-playlist (playlist shuffle)
  (setf (shuffle playlist) shuffle)
  (case shuffle
    (:none (order-playlist playlist))
    (:song (shuffle-by-song playlist))
    (:album (shuffle-by-album playlist)))
  (setf (current-idx playlist) (position-of-current playlist)))
(defun shuffle-by-song (playlist)
  (shuffle-table (songs-table playlist)))
(defun shuffle-by-album (playlist)
  (let ((new-table (make-playlist-table)))
    (do-rows (album-row (shuffled-album-names playlist))
      (do-rows (song (songs-for-album playlist) (column-value album-row :album)))
      (insert-row song new-table)))
    (setf (songs-table playlist) new-table)))
(defun shuffled-album-names (playlist)
  (shuffle-table
    (select
      :columns :album
      :from (songs-table playlist)
      :distinct t)))
(defun songs-for-album (playlist album)
  (select
    :from (songs-table playlist)
    :where (matching (songs-table playlist) :album album)
    :order-by :track))
```

The last manipulation you need to support is setting the playlist's repeat mode. Most of the time you

don't need to take any extra action when setting repeat—its value comes into play only in maybe-move-to-next-song. However, you need to update the current-song as a result of changing repeat in one situation, namely, if current-idx is at the end of a nonempty playlist and repeat is being changed to :song or :all. In that case, you want to continue playing, either repeating the last song or starting at the beginning of the playlist. So, you should define an :after method on the generic function (setf repeat).

```
(defmethod (setf repeat) :after (value (playlist playlist))
  (if (and (at-end-p playlist) (not (empty-p playlist)))
      (ecase value
        (:song (setf (current-idx playlist) (1- (table-size (songs-table
playlist))))))
      (none)
      (:all (setf (current-idx playlist) 0)))
      (update-current-if-necessary playlist)))
```

Now you have all the underlying bits you need. All that remains is the code that will provide a Web-based user interface for browsing the MP3 database and manipulating playlists. The interface will consist of three main functions defined with define-url-function: one for browsing the song database, one for viewing and manipulating a single playlist, and one for listing all the available playlists.

But before you get to writing these three functions, you need to start with some helper functions and HTML macros that they'll use.

Query Parameter Types

Since you'll be using define-url-function, you need to define a few methods on the string->type generic function from Chapter 28 that define-url-function uses to convert string query parameters into Lisp objects. In this application, you'll need methods to convert strings to integers, keyword symbols, and a list of values.

The first two are quite simple.

```
(defmethod string->type ((type (eql 'integer)) value)
  (parse-integer (or value "") :junk-allowed t))
(defmethod string->type ((type (eql 'keyword)) value)
  (and (plusp (length value)) (intern (string-upcase value) :keyword)))
```

The last string->type method is slightly more complex. For reasons I'll get to in a moment, you'll need to generate pages that display a form that contains a hidden field whose value is a list of strings. Since you're responsible for generating the value in the hidden field and for parsing it when it comes back, you can use whatever encoding is convenient. You could use the functions WRITE-TO-STRING and READ-FROM-STRING, which use the Lisp printer and reader to write and read data to and from strings, except the printed representation of strings can contain quotation marks and other characters that may cause problems when embedded in the value attribute of an INPUT element. So, you'll need to escape those characters somehow. Rather than trying to come up with your own escaping scheme, you can just use base 64, an encoding commonly used to protect binary data sent through e-mail. AllegroServe comes with two functions, base64-encode and base64-decode, that do the encoding and decoding for you, so all you have to do is write a pair of functions: one that encodes a Lisp object by converting it to a readable string with WRITE-TO-STRING and then base 64 encoding it and, conversely, another to decode such a string by base 64 decoding it and passing the result to READ-FROM-STRING. You'll want to wrap the calls to WRITE-TO-STRING and READ-FROM-STRING in WITH-STANDARD-IO-SYNTAX to make sure all the variables that affect the printer and reader are set to their standard values. However, because you're going to be reading data that's coming in from the network, you'll definitely want to turn off one feature of the reader—the ability to evaluate arbitrary

Lisp code while reading! You can define your own macro `with-safe-io-syntax`, which wraps its body forms in `WITH-STANDARD-IO-SYNTAX` wrapped around a `LET` that binds `*READ-EVAL*` to `NIL`.

```
(defmacro with-safe-io-syntax (&body body)
  `(with-standard-io-syntax
    (let ((*read-eval* nil))
      ,@body)))
```

Then the encoding and decoding functions are trivial.

```
(defun obj->base64 (obj)
  (base64-encode (with-safe-io-syntax (write-to-string obj))))
(defun base64->obj (string)
  (ignore-errors
    (with-safe-io-syntax (read-from-string (base64-decode string)))))
```

Finally, you can use these functions to define a method on `string->type` that defines the conversion for the query parameter type `base64-list`.

```
(defmethod string->type ((type (eql 'base-64-list)) value)
  (let ((obj (base64->obj value)))
    (if (listp obj) obj nil)))
```

Boilerplate HTML

Next you need to define some HTML macros and helper functions to make it easy to give the different pages in the application a consistent look and feel. You can start with an HTML macro that defines the basic structure of a page in the application.

```
(define-html-macro :mp3-browser-page ((&key title (header title)) &body body)
  `(:html
    (:head
      (:title ,title)
      (:link :rel "stylesheet" :type "text/css" :href "mp3-browser.css"))
    (:body
      (standard-header)
      (when ,header (html (:h1 :class "title" ,header)))
      ,@body
      (standard-footer))))
```

You should define `standard-header` and `standard-footer` as separate functions for two reasons. First, during development you can redefine those functions and see the effect immediately without having to recompile functions that use the `:mp3-browser-page` macro. Second, it turns out that one of the pages you'll write later won't be defined with `:mp3-browser-page` but will still need the standard header and footers. They look like this:


```
(defparameter *r* 25)
(defun standard-header ()
  (html
    (:p :class "toolbar")
    "[" (:a :href (link "/browse" :what "genre") "All genres") "]" "
    "[" (:a :href (link "/browse" :what "genre" :random *r*) "Random genres") "]"
  "
    "[" (:a :href (link "/browse" :what "artist") "All artists") "]" "
    "[" (:a :href (link "/browse" :what "artist" :random *r*) "Random artists")
  "]" "
    "[" (:a :href (link "/browse" :what "album") "All albums") "]" "
    "[" (:a :href (link "/browse" :what "album" :random *r*) "Random albums") "]"
  "
    "[" (:a :href (link "/browse" :what "song" :random *r*) "Random songs") "]" "
    "[" (:a :href (link "/playlist") "Playlist") "]" "
    "[" (:a :href (link "/all-playlists") "All playlists") "]""))))
(defun standard-footer ()
  (html (:hr) (:p :class "footer") "MP3 Browser v" *major-version* "." *minor-
version*)))
```

A couple of smaller HTML macros and helper functions automate other common patterns. The `:table-row` HTML macro makes it easier to generate the HTML for a single row of a table. It uses a feature of FOO that I'll discuss in Chapter 31, an `&attributes` parameter, which causes uses of the macro to be parsed just like normal s-expression HTML forms, with any attributes gathered into a list that will be bound to the `&attributes` parameter. It looks like this:

```
(define-html-macro :table-row (&attributes attrs &rest values)
  `(:tr ,@attrs ,@(loop for v in values collect `(:td ,v))))
```

And the `link` function generates a URL back into the application to be used as the `HREF` attribute with an `A` element, building a query string out of a set of keyword/value pairs and making sure all special characters are properly escaped. For instance, instead of writing this:

```
(:a :href "browse?what=artist&genre=Rhythm+%26+Blues" "Artists")
```

you can write the following:

```
(:a :href (link "browse" :what "artist" :genre "Rhythm & Blues") "Artists")
```

It looks like this:

```
(defun link (target &rest attributes)
  (html
    (:attribute
      (:format "~a~@[?~{~(~a~)=~a~^&~}~]" target (mapcar #'urlencode
attributes)))))
```

To URL encode the keys and values, you use the helper function `urlencode`, which is a wrapper around the function `encode-form-urlencoded`, which is a nonpublic function from AllegroServe. This is—on one hand—bad form; since the name `encode-form-urlencoded` isn't exported from `NET.ASERVE`, it's possible that `encode-form-urlencoded` may go away or get renamed out from under you. On the other hand, using this unexported symbol for the time being lets you get work done for the moment; by wrapping `encode-form-urlencoded` in your own function, you isolate the cruddy code to one function, which you could rewrite if you had to.

```
(defun urlencode (string)
  (net.aserve::encode-form-urlencoded string))
```

Finally, you need the CSS style sheet `mp3-browser.css` used by `:mp3-browser-page`. Since there's nothing dynamic about it, it's probably easiest to just publish a static file with `publish-file`.

```
(publish-file :path "/mp3-browser.css" :file filename :content-type "text/css")
```

A sample style sheet is included with the source code for this chapter on the book's Web site. You'll define a function, at the end of this chapter, that starts the MP3 browser application. It'll take care of, among other things, publishing this file.

The Browse Page

The first URL function will generate a page for browsing the MP3 database. Its query parameters will tell it what kind of thing the user is browsing and provide the criteria of what elements of the database they're interested in. It'll give them a way to select database entries that match a specific genre, artist, or album. In the interest of serendipity, you can also provide a way to select a random subset of matching items. When the user is browsing at the level of individual songs, the title of the song will be a link that causes that song to be added to the playlist. Otherwise, each item will be presented with links that let the user browse the listed item by some other category. For example, if the user is browsing genres, the entry "Blues" will contain links to browse all albums, artists, and songs in the genre Blues. Additionally, the browse page will feature an "Add all" button that adds every song matching the page's criteria to the user's playlist. The function looks like this:

```
(define-url-function browse
  (request (what keyword :genre) genre artist album (random integer))
  (let* ((values (values-for-page what genre artist album random))
        (title (browse-page-title what random genre artist album))
        (single-column (if (eql what :song) :file what))
        (values-string (values->base-64 single-column values)))
    (html
      (:mp3-browser-page
        (:title title)
        ((:form :method "POST" :action "playlist")
          (:input :name "values" :type "hidden" :value values-string)
          (:input :name "what" :type "hidden" :value single-column)
          (:input :name "action" :type "hidden" :value :add-songs)
          (:input :name "submit" :type "submit" :value "Add all"))
        (:ul (do-rows (row values) (list-item-for-page what row)))))))
```

This function starts by using the function `values-for-page` to get a table containing the values it needs to present. When the user is browsing by song—when the `what` parameter is `:song`—you want to select complete rows from the database. But when they're browsing by genre, artist, or album, you want to select only the distinct values for the given category. The database function `select` does most of the heavy lifting, with `values-for-page` mostly responsible for passing the right arguments depending on the value of `what`. This is also where you select a random subset of the matching rows if necessary.

```
(defun values-for-page (what genre artist album random)
  (let ((values
        (select
         :from *mp3s*
         :columns (if (eql what :song) t what)
         :where (matching *mp3s* :genre genre :artist artist :album album)
         :distinct (not (eql what :song))
         :order-by (if (eql what :song) '(:album :track) what))))
    (if random (random-selection values random) values)))
```

To generate the title for the browse page, you pass the browsing criteria to the following function, `browse-page-title`:

```
(defun browse-page-title (what random genre artist album)
  (with-output-to-string (s)
    (when random (format s "~:(~r~) Random " random))
    (format s "~:(~a~p~)" what random)
    (when (or genre artist album)
      (when (not (eql what :song)) (princ " with songs" s))
      (when genre (format s " in genre ~a" genre))
      (when artist (format s " by artist ~a " artist))
      (when album (format s " on album ~a" album))))))
```

Once you have the values you want to present, you need to do two things with them. The main task, of course, is to present them, which happens in the `do-rows` loop, leaving the rendering of each row to the function `list-item-for-page`. That function renders `:song` rows one way and all other kinds another way.

```
(defun list-item-for-page (what row)
  (if (eql what :song)
      (with-column-values (song file album artist genre) row
        (html
         (:li
          (:a :href (link "playlist" :file file :action "add-songs") (:b song)) "
from "
          (:a :href (link "browse" :what :song :album album) album) " by "
          (:a :href (link "browse" :what :song :artist artist) artist) " in genre
"
          (:a :href (link "browse" :what :song :genre genre) genre))))
      (let ((value (column-value row what)))
        (html
         (:li value " - "
              (browse-link :genre what value)
              (browse-link :artist what value)
              (browse-link :album what value)
              (browse-link :song what value))))))
  (defun browse-link (new-what what value)
    (unless (eql new-what what)
      (html
       "["
       (:a :href (link "browse" :what new-what what value) (:format "~(~as~)" new-
what))
       "]" ")))
```

The other thing on the browse page is a form with several hidden `INPUT` fields and an "Add all" submit button. You need to use an `HTML` form instead of a regular link to keep the application stateless—to make sure all the information needed to respond to a request comes in the request itself.

Because the browse page results can be partially random, you need to submit a fair bit of data for the server to be able to reconstitute the list of songs to add to the playlist. If you didn't allow the browse page to return randomly generated results, you wouldn't need much data—you could just submit a request to add songs with whatever search criteria the browse page used. But if you added songs that way, with criteria that included a random argument, then you'd end up adding a different set of random songs than the user was looking at on the page when they hit the "Add all" button.

The solution you'll use is to send back a form that has enough information stashed away in a hidden INPUT element to allow the server to reconstitute the list of songs matching the browse page criteria. That information is the list of values returned by values-for-page and the value of the what parameter. This is where you use the base64-list parameter type; the function values->base64 extracts the values of a specified column from the table returned by values-for-page into a list and then makes a base 64-encoded string out of that list to embed in the form.

```
(defun values->base-64 (column values-table)
  (flet ((value (r) (column-value r column)))
    (obj->base64 (map-rows #'value values-table))))
```

When that parameter comes back as the value of the values query parameter to a URL function that declares values to be of type base-64-list, it'll be automatically converted back to a list. As you'll see in a moment, that list can then be used to construct a query that'll return the correct list of songs. When you're browsing by :song, you use the values from the :file column since they uniquely identify the actual songs while the song names may not.

The Playlist

This brings me to the next URL function, playlist. This is the most complex page of the three—it's responsible for displaying the current contents of the user's playlist as well as for providing the interface to manipulate the playlist. But with most of the tedious bookkeeping handled by define-url-function, it's not too hard to see how playlist works. Here's the beginning of the definition, with just the parameter list:

```
(define-url-function playlist
  (request
    (playlist-id string (playlist-id request) :package)
    (action keyword) ; Playlist manipulation action
    (what keyword :file) ; for :add-songs action
    (values base-64-list) ; "
    file ; for :add-songs and :delete-songs actions
    genre ; for :delete-songs action
    artist ; "
    album ; "
    (order-by keyword) ; for :sort action
    (shuffle keyword) ; for :shuffle action
    (repeat keyword)) ; for :set-repeat action
```

In addition to the obligatory request parameter, playlist takes a number of query parameters. The most important in some ways is playlist-id, which identifies which playlist object the page should display and manipulate. For this parameter, you can take advantage of define-url-function's "sticky parameter" feature. Normally, the playlist-id won't be supplied explicitly, defaulting to the value returned by the playlist-id function, namely, the IP address of the client machine on which the browser is running. However, users can also manipulate their playlists from different machines than the ones running their MP3 clients by allowing this value to be explicitly specified. And if it's specified once, define-url-function will arrange for it to "stick" by setting a cookie in the browser. Later you'll define a URL function that

generates a list of all existing playlists, which users can use to pick a playlist other than the one for the machines they're browsing from.

The action parameter specifies some action to take on the user's playlist object. The value of this parameter, which will be converted to a keyword symbol for you, can be :add-songs, :delete-songs, :clear, :sort, :shuffle, or :set-repeat. The :add-songs action is used by the "Add all" button in the browse page and also by the links used to add individual songs. The other actions are used by the links on the playlist page itself.

The file, what, and values parameters are used with the :add-songs action. By declaring values to be of type base-64-list, the define-url-function infrastructure will take care of decoding the value submitted by the "Add all" form. The other parameters are used with other actions as noted in the comments.

Now let's look at the body of playlist. The first thing you need to do is use the playlist-id to look up the queue object and then acquire the playlist's lock with the following two lines:

```
(let ((playlist (lookup-playlist playlist-id)))
  (with-playlist-locked (playlist)
```

Since lookup-playlist will create a new playlist if necessary, this will always return a playlist object. Then you take care of any necessary queue manipulation, dispatching on the value of the action parameter in order to call one of the playlist functions.

```
(case action
  (:add-songs (add-songs playlist what (or values (list file))))
  (:delete-songs (delete-songs
                  playlist
                  :file file :genre genre
                  :artist artist :album album))
  (:clear (clear-playlist playlist))
  (:sort (sort-playlist playlist order-by))
  (:shuffle (shuffle-playlist playlist shuffle))
  (:set-repeat (setf (repeat playlist) repeat)))
```

All that's left of the playlist function is the actual HTML generation. Again, you can use the :mp3-browser-page HTML macro to make sure the basic form of the page matches the other pages in the application, though this time you pass NIL to the :header argument in order to leave out the H1 header. Here's the rest of the function:

```

(html
  (:mp3-browser-page
    (:title (:format "Playlist - ~a" (id playlist)) :header nil)
    (playlist-toolbar playlist)
    (if (empty-p playlist)
      (html (:p (:i "Empty.")))
      (html
        ( (:table :class "playlist")
          (:table-row "#" "Song" "Album" "Artist" "Genre")
          (let ((idx 0)
                (current-idx (current-idx playlist)))
            (do-rows (row (songs-table playlist))
              (with-column-values (track file song album artist genre) row
                (let ((row-style (if (= idx current-idx) "now-playing" "normal")))
                  (html
                    ( (:table-row :class row-style)
                      track
                      (:progn song (delete-songs-link :file file))
                      (:progn album (delete-songs-link :album album))
                      (:progn artist (delete-songs-link :artist artist))
                      (:progn genre (delete-songs-link :genre genre))))))
                  (incf idx)))))))))))))

```

The function `playlist-toolbar` generates a toolbar containing links to playlist to perform the various :action manipulations. And `delete-songs-link` generates a link to playlist with the :action parameter set to :delete-songs and the appropriate arguments to delete an individual file, or all files on an album, by a particular artist or in a specific genre.

```

(defun playlist-toolbar (playlist)
  (let ((current-repeat (repeat playlist))
        (current-sort (ordering playlist))
        (current-shuffle (shuffle playlist)))
    (html
      (:p :class "playlist-toolbar"
        (:i "Sort by:")
        " [ "
        (sort-playlist-button "genre" current-sort) " | "
        (sort-playlist-button "artist" current-sort) " | "
        (sort-playlist-button "album" current-sort) " | "
        (sort-playlist-button "song" current-sort) " ] "
        (:i "Shuffle by:")
        " [ "
        (playlist-shuffle-button "none" current-shuffle) " | "
        (playlist-shuffle-button "song" current-shuffle) " | "
        (playlist-shuffle-button "album" current-shuffle) " ] "
        (:i "Repeat:")
        " [ "
        (playlist-repeat-button "none" current-repeat) " | "
        (playlist-repeat-button "song" current-repeat) " | "
        (playlist-repeat-button "all" current-repeat) " ] "
        "[ " (:a :href (link "playlist" :action "clear") "Clear") " ] ")
      )))
(defun playlist-button (action argument new-value current-value)
  (let ((label (string-capitalize new-value)))
    (if (string-equal new-value current-value)
      (html (:b label))
      (html (:a :href (link "playlist" :action action argument new-value) label))))
(defun sort-playlist-button (order-by current-sort)
  (playlist-button :sort :order-by order-by current-sort))
(defun playlist-shuffle-button (shuffle current-shuffle)
  (playlist-button :shuffle :shuffle shuffle current-shuffle))
(defun playlist-repeat-button (repeat current-repeat)
  (playlist-button :set-repeat :repeat repeat current-repeat))
(defun delete-songs-link (what value)
  (html " [ " (:a :href (link "playlist" :action :delete-songs what value) "x")
    "]" ))

```

Finding a Playlist

The last of the three URL functions is the simplest. It presents a table listing all the playlists that have been created. Ordinarily users won't need to use this page, but during development it gives you a useful view into the state of the system. It also provides the mechanism to choose a different playlist—each playlist ID is a link to the playlist page with an explicit `playlist-id` query parameter, which will then be made sticky by the `playlist` URL function. Note that you need to acquire the `*playlists-lock*` to make sure the `*playlists*` hash table doesn't change out from under you while you're iterating over it.

```
(define-url-function all-playlists (request)
  (:mp3-browser-page
   (:title "All Playlists")
   ( (:table :class "all-playlists")
     (:table-row "Playlist" "# Songs" "Most recent user agent")
     (with-process-lock (*playlists-lock*)
      (loop for playlist being the hash-values of *playlists* do
        (html
         (:table-row
          (:a :href (link "playlist" :playlist-id (id playlist)) (:print (id
playlist))))
          (:print (table-size (songs-table playlist))))
          (:print (user-agent playlist))))))))))
```

Running the App

And that's it. To use this app, you just need to load the MP3 database with the `load-database` function from Chapter 27, publish the CSS style sheet, set `*song-source-type*` to `playlist` so `find-song-source` uses playlists instead of the singleton song source defined in the previous chapter, and start `Alle-groServe`. The following function takes care of all these steps for you, after you fill in appropriate values for the two parameters `*mp3-dir*`, which is the root directory of your MP3 collection, and `*mp3-css*`, the filename of the CSS style sheet:

```
(defparameter *mp3-dir* ...)
(defparameter *mp3-css* ...)
(defun start-mp3-browser ()
  (load-database *mp3-dir* *mp3s*)
  (publish-file :path "/mp3-browser.css" :file *mp3-css* :content-type
"text/css")
  (setf *song-source-type* 'playlist)
  (net.aserve::debug-on :notrap)
  (net.aserve:start :port 2001))
```

When you invoke this function, it will print dots while it loads the ID3 information from your ID3 files. Then you can point your MP3 client at this URL:

```
http://localhost:2001/stream.mp3
```

and point your browser at some good starting place, such as this:

```
http://localhost:2001/browse
```

which will let you start browsing by the default category, Genre. After you've added some songs to the playlist, you can press Play on the MP3 client, and it should start playing the first song.

Obviously, you could improve the user interface in any of a number of ways—for instance, if you have a lot of MP3s in your library, it might be useful to be able to browse artists or albums by the first letter of their names. Or maybe you could add a "Play whole album" button to the playlist page that causes the playlist to immediately put all the songs from the same album as the currently playing song at the top of the playlist. Or you could change the playlist class, so instead of playing silence when there are no songs queued up, it picks a random song from the database. But all those ideas fall in the realm of application design, which isn't really the topic of this book. Instead, the next two chapters will drop back to the level of software infrastructure to cover how the FOO HTML generation library works.

30. Практика: Библиотека для генерации HTML. Интерпретатор.

В этой и следующей главе вы загляните под капот FOO – генератора HTML, который вы использовали в нескольких предыдущих главах. FOO является примером подхода к программированию, достаточно общего для Common Lisp, и сравнительно несвойственного не-Lisp языкам, а именно – FIXME языкоориентированного программирования. Вместо того чтобы определять API, базирующиеся преимущественно на функциях, классах и макросах, FOO реализует FIXME обработчики для языка специального назначения, которые вы можете встроить в ваши программы на Common Lisp.

FOO предоставляет два языковых обработчика для одного и того же языка s-выражений. Первый – это интерпретатор, который получает программу на "FOO" в качестве входных данных и интерпретирует ее, формируя HTML. Второй – это компилятор, который компилирует выражения FOO (возможно со вставками на Common Lisp) в выражения Common Lisp, которые генерируют HTML и запускает внедренный код. Интерпретатор представлен функцией `emit-html`, а компилятор – макросом `html`, который вы использовали в предыдущих главах.

В этой главе мы рассмотрим составляющие части инфраструктуры, разделяемые интерпретатором и компилятором, а также реализацию интерпретатора. В следующей главе я покажу вам, как работает компилятор.

Проектирование языка специального назначения

Проектирование встраиваемого языка выполняется в два этапа: первый – это проектирование языка, который позволит вам FIXME выражать вещи, которые вы хотите выразить, а второй – реализация обработчика, или обработчиков, которые принимают "программу" на этом языке и либо выполняют действия, указанные программой, либо переводят программу в код на Common Lisp, который выполнит эквивалентные действия.

Итак, первым этапом является проектирование языка для формирования HTML. Ключом к проектированию хорошего языка специального назначения является нахождение верного баланса между выразительностью и краткостью. Например, очень выразительный, но не достаточно краткий "язык" для формирования HTML – это язык FIXME литеральных строк HTML. Разрешенными "формами" этого языка являются строки, содержащие литералы HTML. Языковые процессоры для этого "языка" могут обрабатывать такие формы путем их вывода без изменений.

```
(defvar *html-output* *standard-output*)
(defun emit-html (html)
  "Интерпретатор для языка HTML."
  (write-sequence html *html-output*))
(defmacro html (html)
  "Компилятор для языка HTML."
  `(write-sequence ,html *html-output*))
```

Этот "язык" очень выразительный, поскольку он может сформировать любой HTML, который вы захотите сгенерировать. С другой стороны, этот язык не является настолько кратким, насколько хотелось бы, потому что он дает вам нулевую компрессию – FIXME его вход есть его выход.

Для проектирования языка, дающего вам некоторое полезное FIXME сжатие без ощутимого FIXME жертвования выразительностью, вам необходимо определить детали вывода, которые

являются лишними или не представляют интереса. FIXME Затем вы можете сделать эти аспекты вывода неявными в семантике языка.

Например, согласно структуре HTML, каждый открывающий тэг имеет соответствующую пару в виде закрывающего тэга. Когда вы формируете HTML вручную, то вам необходимо писать эти закрывающие тэги но вы можете улучшить краткость вашего языка формирующего HTML путем неявного включения закрывающего тэга.

Другой способ, который поможет вам FIXME выгадать в краткости, не сильно влияя на выразительность, это возложить на обработчики языка ответственность за добавление необходимых разделителей между элементами - пустых строк и отступов. Когда вы генерируете HTML программно, то вы обычно не сильно заботитесь о том, какие элементы должны обрамляться переводами строк или о том, должны ли элементы быть выровнены относительно своих родительских элементов. Вам не придется беспокоиться о разделителях, если дать возможность обработчику языка самому вставлять их согласно некоторым правилам. Заметим здесь, что FOO в действительности поддерживает два режима – один, использующий минимальное количество разделителей, который позволяет генерировать очень эффективный код и компактный HTML, и другой, генерирующий аккуратный форматированный HTML с различными элементами, которые выровнены и отделены друг от друга согласно своим ролей.

Самая важная деталь, которую необходимо поместить в языковой обработчик – это экранирование определенных знаков, которые имеют специальное значение в HTML, таких как <, >, &. Очевидно, что если вы генерируете HTML просто печатая строки в поток, то вы отвечаете за замену всех вхождений этих знаков на соответствующую экранирующую последовательность <;, > и &. Но если обработчик языка знает, какие строки будут формироваться как данные элемента, тогда он может позаботиться об автоматическом экранировании этих знаков за вас.

Язык FOO

Итак, хватит теории. Я дам быстрый обзор языка, реализуемого FOO и затем вы посмотрите на реализацию двух его обработчиков – интерпретатора, который описан в этой главе и компилятора, который описан в следующей.

Подобно самому Lisp, базовый синтаксис языка FOO определен в терминах выражений, созданных из Lisp объектов. Язык определяет то, как каждое выражение FOO переводится в HTML.

Самые простые выражения FOO – это FIXME само-вычисляющиеся Lisp объекты, такие как строки, числа и ключевые символы. Вам понадобится функция `self-evaluating-p`, которая проверяет является ли данный объект FIXME само-вычисляющимся для целей FOO.

```
(defun self-evaluating-p (form)
  (and (atom form) (if (symbolp form) (keywordp form) t)))
```

Объекты, которые удовлетворяют этому предикату будут выведены путем формирования из них строк с помощью `PRINC-TO-STRING` и затем экранирования всех зарезервированных знаков, таких как <, >, или &. При формировании атрибутов знаки " и ' также экранируются. Таким образом, вы можете применить макрос `html` к FIXME само-вычисляющемуся объекту для вывода его в `*html-output*` (которая изначально связанная с `*STANDARD-OUTPUT*`). Таблица 30-1 показывает как несколько различных само-вычисляющихся значений будут выведены.

Таблица 30-1. Выход FOO для само-вычисляющихся объектов

```
FOO Form Generated HTML
"foo" foo
10 10
:foo FOO
"foo & bar" foo & bar
```

Конечно, большая часть HTML состоит из элементов в тэгах. Каждый такой элемент имеет три составляющие: тэг, множество атрибутов, и тело, содержащее текст и/или другие HTML элементы. Поэтому вам нужен способ представлять эти три составляющие в виде Lisp объектов, желательно таких, которые умеет читать Lisp reader. Если на время забыть об атрибутах, можно заметить, что существует очевидное соответствие между списками Lisp и элементами HTML: каждый HTML элемент может быть представлен как список, чей `FIXME FIRST` – это символ, где имя, это название тэга элемента, а `FIXME REST` – это список само-вычисляющихся объектов или списков, представляющих другие HTML элементы. Тогда:

```
<p>Foo</p> <==> (:p "Foo")
<p><i>Now</i> is the time</p> <==> (:p (:i "Now") " is the time")
```

Теперь остается придумать, как повысить краткость записи атрибутов. Так как у многих элементов нет атрибутов, было бы здорово иметь возможность использовать для них упомянутый выше синтаксис. FOO предоставят два способа нотации элементов с атрибутами. Первое, что приходит в голову, это просто включать атрибуты в список сразу же за символом, чередуя ключевые символы, именующие атрибуты, и объекты, представляющие значения атрибутов. Тело элемента начинается с первого объекта в списке, который находится в позиции имени атрибута и не является ключевым символом. Таким образом:

```
HTML> (html (:p "foo"))
<p>foo</p>
NIL
HTML> (html (:p "foo " (:i "bar") " baz"))
<p>foo <i>bar</i> baz</p>
NIL
HTML> (html (:p :style "foo" "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html (:p :id "x" :style "foo" "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

Для тех, кто предпочитает более очевидное разграничение между телом элемента и его атрибутами, FOO поддерживает альтернативный синтаксис: если первый элемент списка сам является списком с ключевым словом в качестве первого элемента, тогда внешний список представляет элемент HTML с этим ключевым словом в качестве тэга, и с `FIXME REST` вложенного списка в качестве атрибутов, и с `FIXME REST` внешнего списка в качестве тела. То есть вы можете написать два предыдущих выражения вот так:

```
HTML> (html ((:p :style "foo") "Foo"))
<p style='foo'>Foo</p>
NIL
HTML> (html ((:p :id "x" :style "foo") "Foo"))
<p id='x' style='foo'>Foo</p>
NIL
```

Следующая функция проверяет, соответствует ли данный объект одному из этих синтаксисов:

```
(defun cons-form-p (form &optional (test #'keywordp))
  (and (consp form)
        (or (funcall test (car form))
              (and (consp (car form)) (funcall test (caar form))))))
```

Функцию `test` следует сделать параметром, потому что позже вам потребуется проверять те же самые два синтаксиса с немного отличающимся именем предиката.

Чтобы полностью абстрагироваться от различий между двумя вариантами синтаксиса, вы можете определить функцию `parse-cons-form`, которая принимает форму и разбивает ее на три элемента: тэг, список свойств атрибутов и список тела, возвращая их как множественные значения (`multiple values`). Код, который непосредственно вычисляет формы, будет использовать эту функцию, и ему не придется беспокоиться о том, какой синтаксис был использован.

```
(defun parse-cons-form (sexp)
  (if (consp (first sexp))
      (parse-explicit-attributes-sexp sexp)
      (parse-implicit-attributes-sexp sexp)))
(defun parse-explicit-attributes-sexp (sexp)
  (destructuring-bind ((tag &rest attributes) &body body) sexp
    (values tag attributes body)))
(defun parse-implicit-attributes-sexp (sexp)
  (loop with tag = (first sexp)
        for rest on (rest sexp) by #'cddr
        while (and (keywordp (first rest)) (second rest))
        when (second rest)
          collect (first rest) into attributes and
          collect (second rest) into attributes
        end
        finally (return (values tag attributes rest))))
```

Теперь, когда у вас есть базовая спецификация языка, вы можете подумать о том, как вы собираетесь реализовать обработчики языка. Как вы получите желаемый HTML из последовательности выражений FOO? Как я упоминал ранее, вы реализуете два языковых обработчика для FOO: интерпретатор, который проходит по дереву выражений FOO и формирует соответствующий HTML непосредственно, и компилятор, который проходит по дереву выражений и транслирует его в Common Lisp код, который будет формировать такой же HTML. И интерпретатор и компилятор будут построены поверх общего фундамента кода, предоставляющего поддержку для таких вещей, как экранирование зарезервированных знаков и формирование аккуратного, выровненного вывода, так что с этого мы и начнем.

Экранирование знаков

Основой, которую вам необходимо заложить, будет код, который знает, как экранировать знаки специального назначения в HTML. Существует три таких знака, и они не должны появляться в тексте элемента или в значении атрибута; вот они: `<`, `>` и `&`. В тексте значения элемента или атрибута эти знаки должны быть заменены на знаки ссылок на сущность (`character reference entities`) `<`, `>` и `&`. Также, в значениях атрибутов знаки кавычек, используемые для разделения значения, должны быть экранированы, `'` в `'` и `"` в `"`. Вдобавок, любой знак может быть представлен в виде числовой ссылки на символ, состоящей из амперсанда, за которым следует знак `FIXME sharp (#)`, за которым следует числовой код в десятичной системе счисления, за которым следует точка с запятой. Эти числовые экранирования иногда используются для формирования не-ASCII знаков в HTML.

FIXME это таблица в тексте

Пакет

Так как FOO это низкоуровневая библиотека, пакет, в котором вы ее разрабатываете, не зависит от внешнего кода, за исключением стандартных имен из пакета COMMON-LISP и, почти стандартных, имен вспомогательных макросов из пакета COM.GIGAMONKEYS.MACRO-UTILITIES. С другой стороны, пакет нуждается в экспорте всех имен, необходимых коду, который использует FOO. Вот DEFPACKAGE из исходных текстов, которые вы можете скачать с Web-сайта книги:

```
(defpackage :com.gigamonkeys.html
  (:use :common-lisp :com.gigamonkeys.macro-utilities)
  (:export :with-html-output
           :in-html-style
           :define-html-macro
           :html
           :emit-html
           :&attributes))
```

FIXME конец таблицы

Следующая функция принимает один знак и возвращает строку, содержащую FIXME ссылочное вхождение знака для этого знака:

```
(defun escape-char (char)
  (case char
    (#\& "&amp;")
    (#\< "&lt;")
    (#\> "&gt;")
    (#\' "&apos;")
    (#\" "&quot;")
    (t (format nil "&#~d;" (char-code char)))))
```

Вы можете использовать эту функцию как основу для функции escape, которая принимает строку и последовательность знаков и возвращает копию первого аргумента, в которой все вхождения знаков из второго аргумента, заменены соответствующими вхождениями знаков, возвращенными функцией escape-char.

```
(defun escape (in to-escape)
  (flet ((needs-escape-p (char) (find char to-escape)))
    (with-output-to-string (out)
      (loop for start = 0 then (1+ pos)
            for pos = (position-if #'needs-escape-p in :start start)
            do (write-sequence in out :start start :end pos)
            when pos do (write-sequence (escape-char (char in pos)) out)
            while pos))))
```

Вы также можете определить два параметра: **element-escapes**, который содержит знаки, которые вам нужно экранировать в данных элемента, и **attribute-escapes**, который содержит множество знаков, которые необходимо экранировать в значениях атрибутов.

```
(defparameter *element-escapes* "<>&")
(defparameter *attribute-escapes* "<>&\"'")
```

Вот несколько примеров:

```
HTML> (escape "foo & bar" *element-escapes*)
"foo & bar"
HTML> (escape "foo & 'bar'" *element-escapes*)
"foo & 'bar'"
HTML> (escape "foo & 'bar'" *attribute-escapes*)
"foo & 'bar'"
```

Наконец, вам нужна переменная `*escapes*`, которая будет связана с множеством знаков, которые должны быть экранированы. Изначально она установлена в значение `*element-escapes*`, но, как вы увидите, при формировании атрибутов, она будет установлена в значение `*attribute-escapes*`.

```
(defvar *escapes* *element-escapes*)
```

ФИХМЕ Принтер отступов

Для формирования аккуратно выровненного вывода, вы можете определить класс `indenting-printer`, который является оберткой вокруг потока вывода, и функции, которые используют экземпляр этого класса для вывода строк в поток и имеют возможность отслеживать начала новых строк. Класс выглядит примерно вот так:

```
(defclass indenting-printer ()
  ((out :accessor out :initarg :out)
   (beginning-of-line-p :accessor beginning-of-line-p :initform t)
   (indentation :accessor indentation :initform 0)
   (indenting-p :accessor indenting-p :initform t)))
```

Главная функция, работающая с `indenting-printer` это `emit`, которая принимает принтер и строку и выводит строку в поток вывода принтера, отслеживая переходы на новую строку, что позволяет ей управлять значением слота `beginning-of-line-p`.

```
(defun emit (ip string)
  (loop for start = 0 then (1+ pos)
        for pos = (position #\Newline string :start start)
        do (emit/no-newlines ip string :start start :end pos)
        when pos do (emit-newline ip)
        while pos))
```

Для непосредственного вывода строки она использует функцию `emit/no-newlines`, которая формирует необходимое количество отступов посредством функции-помощника `indent-if-necessary` и затем записывает строку в поток. Эта функция может также быть вызвана из любого другого кода, для вывода строки, которая заведомо не содержит переводов строк.

```
(defun emit/no-newlines (ip string &key (start 0) end)
  (indent-if-necessary ip)
  (write-sequence string (out ip) :start start :end end)
  (unless (zerop (- (or end (length string)) start))
    (setf (beginning-of-line-p ip) nil)))
```

Функция-помощник `indent-if-necessary` проверяет значения `beginning-of-line-p` и `indenting-p`, чтобы определить, нужно ли выводить отступ, и если они оба имеют истинное значение, выводит столько пробелов, сколько указывается значением `indentation`. Код, использующий `indenting-printer`, может управлять выравниванием, изменяя значения слотов `indentation` и `indenting-p`. Увеличивая или уменьшая значение `indentation`, можно

изменять количество ведущих пробелов, в то время как установка `indenting-p` в `NIL` может временно выключить выравнивание.

```
(defun indent-if-necessary (ip)
  (when (and (beginning-of-line-p ip) (indenting-p ip))
    (loop repeat (indentation ip) do (write-char #\Space (out ip)))
    (setf (beginning-of-line-p ip) nil)))
```

Последние две функции в API `indenting-printer` это `emit-newline` и `emit-freshline`, которые используются для вывода знака новой строки и похожи на `~%` и `~&` директивы `FORMAT`. Единственное различие в том, что `emit-newline` всегда выводит перевод строки, в то время как `emit-freshline` делает это только тогда, когда `beginning-of-line-p` установлено в ложное значение. Таким образом, множественные вызовы `emit-freshline` без промежуточных вызовов `emit` не отразятся на количестве пустых линий. Это удобно, когда один кусок кода хочет сгенерировать некоторый вывод, который должен заканчиваться переводом строки, в то время как другой кусок кода хочет сгенерировать некоторый выход, который должен начаться с перевода строки, но вы не хотите избыточных пустых линий между двумя частями вывода.

```
(defun emit-newline (ip)
  (write-char #\Newline (out ip))
  (setf (beginning-of-line-p ip) t))
(defun emit-freshline (ip)
  (unless (beginning-of-line-p ip) (emit-newline ip)))
```

Теперь вы готовы перейти к внутреннему устройству `FOO` процессора.

Интерфейс HTML процессора

Теперь вы готовы к тому, чтобы определить интерфейс, с помощью которого вы будете использовать процессор языка `FOO` для формирования `HTML`. Вы можете определить этот интерфейс как множество обобщенных функций, потому что вам потребуются две реализации — одна, которая непосредственно формирует `HTML`, и другая, которую макрос `html` может использовать как список инструкций для выполнения, которые затем могут быть оптимизированы и скомпилированы в код, формирующий такой же вывод более эффективно. Я буду называть это множество обобщенных функций интерфейсом выходного буфера. Он состоит из следующих восьми обобщенных функций:

```
(defgeneric raw-string (processor string &optional newlines-p))
(defgeneric newline (processor))
(defgeneric freshline (processor))
(defgeneric indent (processor))
(defgeneric unindent (processor))
(defgeneric toggle-indenting (processor))
(defgeneric embed-value (processor value))
(defgeneric embed-code (processor code))
```

В то время как некоторые из этих функций имеют очевидное соответствие функциям `indenting-printer`, очень важно понять, что эти обобщенные функции определяют абстрактные операции, которые используются обработчиками языка `FOO` и не всегда будут реализованы в терминах вызовов функций `indenting-printer`.

Возможно, самый легкий способ понять семантику этих абстрактных операций, это взглянуть на конкретные реализации специализированных методов в `html-pretty-printer`, классе,

используемом для генерации удобочитаемого HTML.

FIXME backend Выходной буфер аккуратной печати

Вы можете начать реализацию, определив класс с двумя слотами, — одним для хранения экземпляра `indenting-printer` и одним — для хранения размера табуляции — количества пробелов, на которое вы хотите увеличить отступ для каждого вложенного уровня HTML элементов.

```
(defclass html-pretty-printer ()  
  ((printer :accessor printer :initarg :printer)  
   (tab-width :accessor tab-width :initarg :tab-width :initform 2)))
```

Теперь вы можете реализовать методы, специализированные для `html-pretty-printer`, в виде 8 обобщенных функций, которые составляют интерфейс выходного буфера.

Обработчики FOO используют функцию `raw-string` для вывода строк, которые не нуждаются в экранировании знаков, либо потому, что вы действительно хотите вывести зарезервированные знаки как есть, либо потому, что все зарезервированные знаки уже были экранированы. Обычно `raw-string` вызывается для строк, которые не содержат переводов строки, таким образом поведение по умолчанию заключается в использовании `emit/no-newlines` до тех пор, пока клиент не передаст не-NIL значение в качестве аргумента `newlines-p`.

```
(defmethod raw-string ((pp html-pretty-printer) string &optional newlines-p)  
  (if newlines-p  
      (emit (printer pp) string)  
      (emit/no-newlines (printer pp) string)))
```

Функции `newline`, `freshline`, `indent`, `unindent` и `toggle-indenting` реализуют FIXME справедливо прямые манипуляции нижележащего `indenting-printer`. Единственная загвоздка заключается в том, что принтер HTML формирует аккуратный вывод только когда динамическая переменная `*pretty*` имеет истинное значение. Когда она равна NIL, то формируется компактный HTML, без лишних пробелов. Поэтому все эти методы, за исключением `newline`, проверяют значение переменной `*pretty*` перед тем, как что-то сделать:

```
(defmethod newline ((pp html-pretty-printer))  
  (emit-newline (printer pp)))  
(defmethod freshline ((pp html-pretty-printer))  
  (when *pretty* (emit-freshline (printer pp))))  
(defmethod indent ((pp html-pretty-printer))  
  (when *pretty*  
    (incf (indentation (printer pp)) (tab-width pp))))  
(defmethod unindent ((pp html-pretty-printer))  
  (when *pretty*  
    (decf (indentation (printer pp)) (tab-width pp))))  
(defmethod toggle-indenting ((pp html-pretty-printer))  
  (when *pretty*  
    (with-slots (indenting-p) (printer pp)  
      (setf indenting-p (not indenting-p)))))
```

И наконец, функции `embed-value` и `embed-code` используются только компилятором FOO: `embed-value` используется для генерации кода, который будет формировать значение выражений Common Lisp, а `embed-code` используется для внедрения фрагментов кода для запуска и ее результат исключается FIXME. В интерпретаторе вы не можете полностью

вычислять внедренный Lisp код, поэтому вызов этих функций всегда будет сигнализировать об ошибке.

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (error "Can't embed values when interpreting. Value: ~s" value))
(defmethod embed-code ((pp html-pretty-printer) code)
  (error "Can't embed code when interpreting. Code: ~s" code))
```

ФИКСИРОВАНО окно в тексте

Использование Условий. И невинность соблюсти, и капитал приобрести.

В оригинале – To have your cake and eat it too – известная английская пословица, смысл который в том, что нельзя одновременно делать две взаимоисключающие вещи. Почти дословный русский аналог – Один пирог два раза не съешь. Видимо автор хотел подчеркнуть гибкость механизма условий Common Lisp – прим. перев.

Альтернативным подходом является использование EVAL для вычисления Lisp выражений в интерпретаторе. Проблема, связанная с данным подходом заключается в том, что EVAL не имеет доступа к лексическому окружению. Таким образом, не существует способа выполнить что-то, подобное следующему:

```
(let ((x 10)) (emit-html '(:p x)))
```

если *x* это лексическая переменная. Символ *x*, который передается `emit-html` во время выполнения, не связан с лексической переменной, названной этим же символом. Компилятор Lisp создает ссылки на *x* в коде для обращения к переменной, но после того, как код скомпилирован, больше нет необходимости в связи между именем *x* и этой переменной. Это главная причина, по которой когда вы думаете, что EVAL – это решение вашей проблемы, вы вероятно ошибаетесь.

Как бы то ни было, если бы *x* был динамической переменной, объявленной с помощью `DEFFVAR` или `DEFPARAMETER` (и назван **x** вместо *x*), то EVAL могла бы получить доступ к ее значению. То есть, в некоторых ситуациях имеет смысл позволить интерпретатору FOO использовать EVAL. Но использовать EVAL всегда – это плохая идея. Вы можете взять лучшее из каждого подхода, комбинируя идеи использования EVAL и системы условий.

Сначала определим некоторые классы ошибок, которые вы можете просигнализировать, когда `embed-value` и `embed-code` вызываются в интерпретаторе.

```
(define-condition embedded-lisp-in-interpreter (error)
  ((form :initarg :form :reader form)))
(define-condition value-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed values when interpreting. Value: ~s" (form c)))))
(define-condition code-in-interpreter (embedded-lisp-in-interpreter) ()
  (:report
   (lambda (c s)
     (format s "Can't embed code when interpreting. Code: ~s" (form c)))))
```

Потом вы можете реализовать `embed-value` и `embed-code`, используя сигнализирование этих ошибок и предоставление перезапуска, который вычислит форму с помощью EVAL.

```
(defmethod embed-value ((pp html-pretty-printer) value)
  (restart-case (error 'value-in-interpreter :form value)
    (evaluate ()
      :report (lambda (s) (format s "EVAL ~s in null lexical environment."
value)))
    (raw-string pp (escape (princ-to-string (eval value)) *escapes*) t))))
(defmethod embed-code ((pp html-pretty-printer) code)
  (restart-case (error 'code-in-interpreter :form code)
    (evaluate ()
      :report (lambda (s) (format s "EVAL ~s in null lexical environment." code))
      (eval code))))
```

Теперь вы можете делать что-то подобное этому:

```
HTML> (defvar *x* 10)
*X*
HTML> (emit-html '(:p *x*))
```

и вас выкинет в отладчик с таким сообщением:

```
Can't embed values when interpreting. Value: *X*
[Condition of type VALUE-IN-INTERPRETER]
Restarts:
 0: [EVALUATE] EVAL *X* in null lexical environment.
 1: [ABORT] Abort handling SLIME request.
 2: [ABORT] Abort entirely from this process.
```

Если вы вызовете перезапуск `evaluate`, то `embed-value` вызовет `EVAL *x*`, получит значение 10 и сгенерирует следующий HTML:

```
<p>10</p>
```

Для удобства, вы можете предоставить функции перезапуска – функции, которые вызывают `evaluate` перезапуск в определенных ситуациях. Функция `evaluate` перезапуска безусловно вызывает перезапуск, в то время как `eval-dynamic-variables` и `eval-code` вызывают ее только если форма в условии является динамической переменной или потенциальным код.

```
(defun evaluate (&optional condition)
  (declare (ignore condition))
  (invoke-restart 'evaluate))
(defun eval-dynamic-variables (&optional condition)
  (when (and (symbolp (form condition)) (boundp (form condition)))
    (evaluate)))
(defun eval-code (&optional condition)
  (when (consp (form condition))
    (evaluate)))
```

Теперь вы можете использовать `HANDLER-BIND` для установки обработчика для автоматического вызова `evaluate` перезапуска для вас.

```
HTML> (handler-bind ((value-in-interpreter #'evaluate)) (emit-html '(:p *x*)))
<p>10</p>
T
```

И наконец, вы можете определить макрос, чтобы предоставить более приятный синтаксис для

связывания обработчиков для двух видов ошибок.

```
(defmacro with-dynamic-evaluation ((&key values code) &body body)
  `(handler-bind (
    ,@(if values `((value-in-interpreter #'evaluate)))
    ,@(if code `((code-in-interpreter #'evaluate)))
    ,@body))
```

Этот макрос позволяет вам писать следующим образом:

```
HTML> (with-dynamic-evaluation (:values t) (emit-html '(:p *x*)))
<p>10</p>
Т
```

FIXME конец таблицы в тексте

Базовое правило вычисления

Теперь для того, чтобы соединить язык FOO с интерфейсом обработчика, все, что вам нужно, это функция, которая принимает объект и обрабатывает его, вызывая подходящие функции обработчика для генерации HTML. Например, когда дано простое выражение, наподобие такого:

```
(:p "Foo")
```

эта функция может выполнить эту последовательность вызовов обработчика:

```
(freshline processor)
(raw-string processor "<p" nil)
(raw-string processor ">" nil)
(raw-string processor "Foo" nil)
(raw-string processor "</p>" nil)
(freshline processor)
```

Теперь вы можете определить простую функцию, которая просто проверяет, является данное выражение разрешенным выражением FOO, и если это так, передать ее функции `process-sexp-html` для обработки. В следующей главе вы добавите некоторые расширения в эту функцию, чтобы позволить ей обрабатывать макросы и специальные операторы. Но для текущих целей она выглядит так:

```
(defun process (processor form)
  (if (sexp-html-p form)
      (process-sexp-html processor form)
      (error "Malformed FOO form: ~s" form)))
```

Функция `sexp-html-p` определяет, является ли данный объект разрешенным выражением FOO, само-вычисляющимся выражением или корректно сформатированной ячейкой.

```
(defun sexp-html-p (form)
  (or (self-evaluating-p form) (cons-form-p form)))
```

Само-вычисляющиеся выражения обрабатываются просто: преобразуются в строку с помощью `PRINC-TO-STRING`, а затем экранируются знаки, указанные в переменной `*escapes*`, которая, как вы помните, изначально связана со значением `*element-escapes*`. Формы ячеек вы передаете в `process-cons-sexp-html`.

```
(defun process-sexp-html (processor form)
  (if (self-evaluating-p form)
      (raw-string processor (escape (princ-to-string form) *escapes*) t)
      (process-cons-sexp-html processor form)))
```

Функция `process-cons-sexp-html` отвечает за вывод открывающего тэга, всех атрибутов, тела и закрывающего тэга. Главная трудность здесь в том, что для генерирования аккуратного HTML, вам нужно выводить дополнительные линии и регулировать отступы согласно типу выводимого элемента. Вы можете разделить все элементы, определенные в HTML, на три категории: блок, параграф, и встроенные. Элементы блоки – такие как тело и `ul` – выводятся с дополнительными линиями (переводами строк) перед и после открывающих и закрывающих тэгов, и с содержимым, выровненным по одному уровню. Элементы параграфы – такие как `p`, `li` и `blockquote` – выводятся с переводом строки перед открывающим тэгом и после закрывающего тэга. Встроенные элементы просто выводятся в линию. Три следующих параметра являются списками элементов каждого типа:

```
(defparameter *block-elements*
  '(:body :colgroup :dl :fieldset :form :head :html :map :noscript :object
    :ol :optgroup :pre :script :select :style :table :tbody :tfoot :thead
    :tr :ul))
(defparameter *paragraph-elements*
  '(:area :base :blockquote :br :button :caption :col :dd :div :dt :h1
    :h2 :h3 :h4 :h5 :h6 :hr :input :li :link :meta :option :p :param
    :td :textarea :th :title))
(defparameter *inline-elements*
  '(:a :abbr :acronym :address :b :bdo :big :cite :code :del :dfn :em
    :i :img :ins :kbd :label :legend :q :samp :small :span :strong :sub
    :sup :tt :var))
```

Функции `block-element-p` и `paragraph-element-p` проверяют, является ли данный тэг членом соответствующего списка.

```
(defun block-element-p (tag) (find tag *block-elements*))
(defun paragraph-element-p (tag) (find tag *paragraph-elements*))
```

К двум другим категориям со своими собственными предикатами относятся элементы, которые всегда пусты, такие как `br` и `hr` и три элемента `pre`, `style` и `script`, в которых положено сохранение разделителей. Формы обрабатываются особо при формировании регулярного HTML (другими словами, не XHTML), так как в них не предполагаются закрывающие тэги. И при выводе трех тэгов, в которых пробелы сохраняются, вы можете временно выключить выравнивание, и тогда `pretty printer` не добавит каких-либо разделителей, которые не являются частью действительного содержимого элементов.

```
(defparameter *empty-elements*
  '(:area :base :br :col :hr :img :input :link :meta :param))
(defparameter *preserve-whitespace-elements* '(:pre :script :style))
(defun empty-element-p (tag) (find tag *empty-elements*))
(defun preserve-whitespace-p (tag) (find tag *preserve-whitespace-elements*))
```

Последнее, что вам понадобится при генерации HTML, это параметр, указывающий, генерируете ли вы XHTML, так как это влияет на то, как вам нужно выводить пустые элементы.

```
(defparameter *xhtml* nil)
```

Со всей этой информацией, вы готовы к обработке FIXME FOO форм ячеек. Вы используете `parse-cons-form`, чтобы разбить список на три части, символ тэга, возможно пустой список свойств пар ключ/значение атрибутов, и, возможно пустой, список форм тела. Затем вы формируете открывающий тэг, тело и закрывающий тэг с помощью вспомогательных функций `emit-open-tag`, `emit-element-body` и `emit-close-tag`.

```
(defun process-cons-sexp-html (processor form)
  (when (string= *escapes* *attribute-escapes*)
    (error "Can't use cons forms in attributes: ~a" form))
  (multiple-value-bind (tag attributes body) (parse-cons-form form)
    (emit-open-tag processor tag body attributes)
    (emit-element-body processor tag body)
    (emit-close-tag processor tag body)))
```

В `emit-open-tag` вам нужно вызвать `freshline`, когда это необходимо, и затем вывести атрибуты с помощью `emit-attributes`. Вам нужно передать тело элемента в функцию `emit-open-tag`, тогда в случае формирования XHTML, она определит, закончить тэг с `/>` или `>`.

```
(defun emit-open-tag (processor tag body-p attributes)
  (when (or (paragraph-element-p tag) (block-element-p tag))
    (freshline processor))
  (raw-string processor (format nil "<~(~a~)" tag))
  (emit-attributes processor attributes)
  (raw-string processor (if (and *xhtml* (not body-p)) ">" ">")))
```

В `emit-attributes` имена атрибутов не вычисляются, так как они являются ключевыми символами, но вам следует вызывать функцию `process` верхнего уровня для вычисления значений атрибутов, связывая `*escapes*` с `*attribute-escapes*`. Для удобства при спецификации булевских атрибутов, чьи значения должны быть именем атрибута, если это значение равно `T` (не любое истинное значение, а именно `T`), то тогда вы заменяете значение именем атрибута.

```
(defun emit-attributes (processor attributes)
  (loop for (k v) on attributes by #'cddr do
    (raw-string processor (format nil " ~(~a~)=\"" k))
    (let ((*escapes* *attribute-escapes*))
      (process processor (if (eql v t) (string-downcase k) v)))
    (raw-string processor "'")))
```

Формирование тела элемента похоже на формирование значений атрибута: вы можете циклически проходить по телу, вызывая `process` для вычисления каждого выражения. Основа кода заключена в выводе переводов строк и регулирования отступов подходящим образом в соответствии с типом элемента.

```
(defun emit-element-body (processor tag body)
  (when (block-element-p tag)
    (freshline processor)
    (indent processor))
  (when (preserve-whitespace-p tag) (toggle-indenting processor))
  (dolist (item body) (process processor item))
  (when (preserve-whitespace-p tag) (toggle-indenting processor))
  (when (block-element-p tag)
    (unindent processor)
    (freshline processor)))
```

Наконец `emit-close-tag`, как вы вероятно ожидаете, выводит закрывающий тэг (до тех пор,

пока в нем нет необходимости, например когда тело пустое и вы либо формируете XHTML, либо элемент является одним из специальных пустых элементов). Независимо от того, выводите ли вы закрывающий тэг, вам нужно вывести завершающий перевод строки для элементов блока и параграфа.

```
(defun emit-close-tag (processor tag body-p)
  (unless (and (or *xhtml* (empty-element-p tag)) (not body-p))
    (raw-string processor (format nil "</~(~a~)>" tag)))
  (when (or (paragraph-element-p tag) (block-element-p tag))
    (freshline processor)))
```

Функция `process` – это основа интерпретатора FOO. Чтобы сделать ее немного проще в использовании, вы можете определить функцию `emit-html`, которая вызывает `process`, передавая ей `html-pretty-printer` и форму для вычисления. Вы можете определить и использовать вспомогательную функцию `get-pretty-printer` для получения `pretty printer`, которая возвращает текущее значение `*html-pretty-printer*`, если оно связано; в ином случае, она создает новый экземпляр `html-pretty-printer` с `*html-output*` в качестве выходного потока.

```
(defun emit-html (sexp) (process (get-pretty-printer) sexp))
(defun get-pretty-printer ()
  (or *html-pretty-printer*
      (make-instance
        'html-pretty-printer
        :printer (make-instance 'indenting-printer :out *html-output*))))
```

С этой функцией вы можете выводить HTML в `*html-output*`. Вместо того, чтобы предоставлять переменную `*html-output*` как часть открытого API FOO, вам следует определить макрос `with-html-output`, который берет на себя заботу о связывании потока для вас. Он также позволяет вам определить, хотите ли вы использовать аккуратный HTML вывод, выставляя по умолчанию значение переменной `*pretty*`.

```
(defmacro with-html-output ((stream &key (pretty *pretty*)) &body body)
  `(let* ((*html-output* ,stream)
         (*pretty* ,pretty))
     ,@body))
```

Итак, если вы хотите использовать `emit-html` для вывода HTML в файл, вы можете написать следующее:

```
(with-open-file (out "foo.html" :direction output)
  (with-html-output (out :pretty t)
    (emit-html *some-foo-expression*)))
```

Что Дальше?

В следующей главе вы увидите, как реализовать макрос, который компилирует выражения FOO в Common Lisp, что позволит вам внедрить код генерации HTML прямо в ваши Lisp программы. Вы также расширите язык FOO, чтобы сделать его немного более выразительным, путем добавления специальных операторов и макросов.

31. Практика: Библиотека для генерации HTML, Компилятор.

Теперь вы готовы к тому, чтобы увидеть как работает компилятор FOO. Основное различие между компилятором и интерпретатором заключается в том, что интерпретатор обрабатывает программу и сразу производит какие-то действия – генерирует HTML, в случае интерпретатора FOO, а компилятор обрабатывает ту же самую программу, и генерирует код на каком-то другом языке, который будет производить те же самые действия. В языке FOO компилятор является макросом Common Lisp, который транслирует инструкции FOO в код на Common Lisp, так что он может быть встроен в программы на Common Lisp. В общем, компиляторы имеют преимущества над интерпретаторами, поскольку компиляция происходит заранее, так что они могут потратить некоторое количество времени оптимизируя генерируемый код, делая его более эффективным. Это же делает компилятор FOO, объединяя текстовые строки насколько это возможно, чтобы выдавать точно такой же HTML с меньшим количеством операций записи по сравнению с интерпретатором. Когда компилятор является макросом Common Lisp, вы также имеете преимущества, поскольку компилятор сможет обрабатывать вставленные куски кода на Common Lisp – компилятору нужно лишь распознать их и вставить в правильное место генерируемого кода. Компилятор FOO получит некоторые преимущества от использования этой возможности.

Компилятор

Базовая архитектура компилятора состоит из трех уровней. Сначала вы реализуете класс `html-compiler` который имеет один слот, который содержит расширяемый вектор, который используется для накопления кодов операций (`ops`), представляющих вызовы сделанные к обобщенным функциям FIXME backend при выполнении `process`.

Затем вы реализуете методы для обобщенных функций FIXME backend interface, которые будут сохранять последовательность действий в векторе. Каждый код операции представлен списком состоящим из ключевого слова, именующего операцию, и аргументов, переданных функции, которая сгенерировала этот код операции. Функция `sexpr->ops` реализует первую стадию компиляции – преобразование списка выражений FOO путем вызова `process` для каждого выражения с объектом `html-compiler`.

Этот вектор с кодами операций, созданный компилятором, затем передается функции, которая оптимизирует его, объединяя последовательные операции `raw-string` в одну операцию, которая выдаст объединенную строку за один вызов. Функция оптимизации, также может, не обязательно, отбросить операции, которые необходимы только для выдачи хорошо отформатированного кода, что является достаточно важным, поскольку это позволит объединить большее количество операций `raw-string`.

И в заключение, оптимизированный вектор с кодами операций передается третьей функции, `generate-code`, которая возвращает список выражений Common Lisp, выполнение которых приведет к выводу HTML. Когда переменная `*pretty*` имеет истинное значение, то `generate-code` генерирует код, который использует методы, специализированные для `html-pretty-printer`, для того, чтобы вывести хорошо отформатированный HTML. Когда `*pretty*` равна `NIL`, то эта функция генерирует код, который будет выводить данные напрямую в поток `*html-output*`.

Макрос `html` в действительности генерирует тело выражения, которое содержит два раскрытия кода – одно для случая, когда `*pretty*` равно `T`, и второе для случая, когда `*pretty*` равно `NIL`. То, какое выражение будет использоваться, определяется во время выполнения в

зависимости от значения переменной `*pretty*`. Таким образом, любая функция, которая содержит вызов `html`, будет иметь код для генерации компактного и хорошо оформленного вывода.

Другим важным отличием между компилятором и интерпретатором является то, что компилятор может внедрять выражения на Lisp в генерируемый код. Чтобы воспользоваться этим преимуществом, вам необходимо изменить функцию `process` таким образом, чтобы она вызывала функции `embed-code` и `embed-value` в тех случаях, когда ее просят обработать выражение, которое не является выражением `FOO`. Поскольку, все `FIXME self-evaluating` объекты являются допустимыми выражениями `FOO`, единственными выражениями, которое не будет передано `process-sexp-html` являются списки, которые не соответствуют синтаксису выражений-ячеек (`FIXME cons forms`) `FOO` и не-именованным символам – единственным атомам, которые не вычисляются сами в себя `FIXME self-evaluating`. Вы можете предположить, что любой список не относящийся к `FOO` является кодом, который необходимо выполнять, а все символы являются переменными, чьи значения вы должны вставить в генерируемый код.

```
(defun process (processor form)
  (cond
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form) (embed-code processor form))
    (t (embed-value processor form))))
```

Теперь давайте взглянем на код компилятора. Во первых вы должны определить две функции, которые абстрагируют вектор, который вы будете использовать для хранения кодов операций в первых двух стадиях компиляции.

```
(defun make-op-buffer () (make-array 10 :adjustable t :fill-pointer 0))
(defun push-op (op ops-buffer) (vector-push-extend op ops-buffer))
```

Затем, вы можете определить класс `html-compiler` и методы, специализированные для него, реализующие `FIXME backend interface`.

```
(defclass html-compiler ()
  ((ops :accessor ops :initform (make-op-buffer))))
(defmethod raw-string ((compiler html-compiler) string &optional newlines-p)
  (push-op `(:raw-string ,string ,newlines-p) (ops compiler)))
(defmethod newline ((compiler html-compiler))
  (push-op '(:newline) (ops compiler)))
(defmethod freshline ((compiler html-compiler))
  (push-op '(:freshline) (ops compiler)))
(defmethod indent ((compiler html-compiler))
  (push-op `(:indent) (ops compiler)))
(defmethod unindent ((compiler html-compiler))
  (push-op `(:unindent) (ops compiler)))
(defmethod toggle-indenting ((compiler html-compiler))
  (push-op `(:toggle-indenting) (ops compiler)))
(defmethod embed-value ((compiler html-compiler) value)
  (push-op `(:embed-value ,value ,*escapes*) (ops compiler)))
(defmethod embed-code ((compiler html-compiler) code)
  (push-op `(:embed-code ,code) (ops compiler)))
```

После определения этих методов, вы можете реализовать первую стадию компиляции – `sexp->ops`.


```
(defun sexp->ops (body)
  (loop with compiler = (make-instance 'html-compiler)
    for form in body do (process compiler form)
    finally (return (ops compiler))))
```

Во время этой фазы, вам нет необходимости учитывать значение переменной `*pretty*`: просто записывайте все функции вызванные функцией `process`. Вот что `sexp->ops` сделает из простого выражения `FOO`:

```
HTML> (sexp->ops '(:p "Foo"))
#((:FRESHLINE) (:RAW-STRING "<p" NIL) (:RAW-STRING ">" NIL)
  (:RAW-STRING "Foo" T) (:RAW-STRING "</p>" NIL) (:FRESHLINE))
```

На следующей фазе, функция `optimize-static-output` принимает вектор кодов операций, и возвращает новый вектор, содержащий оптимизированную версию. Алгоритм очень прост – для каждой операции `:raw-string`, функция записывает строку во временный строковый буфер. Таким образом, последовательные вызовы `:raw-string` приведут к построению одной строки, содержащих объединение всех строк, которые должны быть выведены. Когда вы встречаете код операции, отличный от кода `:raw-string`, то вы преобразуете созданную строку в последовательность операций `:raw-string` и `:newline` используя вспомогательную функцию `compile-buffer`, и затем добавляете новый код операции. В этой функции вы также отбрасываете "красивое" форматирование, если значением `*pretty*` является `NIL`.

```
(defun optimize-static-output (ops)
  (let ((new-ops (make-op-buffer)))
    (with-output-to-string (buf)
      (flet ((add-op (op)
                (compile-buffer buf new-ops)
                (push-op op new-ops)))
        (loop for op across ops do
          (ecase (first op)
            (:raw-string (write-sequence (second op) buf))
            ((:newline :embed-value :embed-code) (add-op op))
            ((:indent :unindent :freshline :toggle-indenting)
             (when *pretty* (add-op op))))
          (compile-buffer buf new-ops)))
      new-ops))
  new-ops)

(defun compile-buffer (buf ops)
  (loop with str = (get-output-stream-string buf)
    for start = 0 then (1+ pos)
    for pos = (position #\Newline str :start start)
    when (< start (length str))
    do (push-op `(:raw-string ,(subseq str start pos) nil) ops)
    when pos do (push-op '(:newline) ops)
    while pos))
```

Последним шагом является преобразование кодов операций в соответствующий код Common Lisp. Эта фаза также учитывает значение переменной `*pretty*`. Когда `*pretty*` имеет истинное значение, то функция генерирует код, который вызывает функции используя переменную `*html-pretty-printer*`, которая содержит экземпляр класса `html-pretty-printer`. А когда значение `*pretty*` равно `NIL`, то функция генерирует код, который выводит данные прямо в поток, указанный переменной `*html-output*`.

Реализация функции `generate-code` крайне проста.

```
(defun generate-code (ops)
  (loop for op across ops collect (apply #'op->code op)))
```

Вся работа выполняется методами обобщенной функции `op->code` специализированной для аргумента `op` со специализатором EQL для имени операции.

```
(defgeneric op->code (op &rest operands))
(defmethod op->code ((op (eql :raw-string)) &rest operands)
  (destructuring-bind (string check-for-newlines) operands
    (if *pretty*
      `(raw-string *html-pretty-printer* ,string ,check-for-newlines)
      `(write-sequence ,string *html-output*))))
(defmethod op->code ((op (eql :newline)) &rest operands)
  (if *pretty*
    `(newline *html-pretty-printer*)
    `(write-char #\Newline *html-output*)))
(defmethod op->code ((op (eql :freshline)) &rest operands)
  (if *pretty*
    `(freshline *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))
(defmethod op->code ((op (eql :indent)) &rest operands)
  (if *pretty*
    `(indent *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))
(defmethod op->code ((op (eql :unindent)) &rest operands)
  (if *pretty*
    `(unindent *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))
(defmethod op->code ((op (eql :toggle-indenting)) &rest operands)
  (if *pretty*
    `(toggle-indenting *html-pretty-printer*)
    (error "Bad op when not pretty-printing: ~a" op)))
```

Два наиболее интересных метода `op->code` – это те, которые генерируют код для операций `:embed-value` и `:embed-code`. В методе `:embed-value`, вы можете генерировать немного отличающийся код в зависимости от значения аргумента `escapes`, поскольку, если `escapes` равен `NIL`, то вам нет необходимости генерировать вызов `escape`. И когда и `*pretty*`, и `escapes` равны `NIL`, то вы можете сгенерировать код, который будет использовать функцию `PRINC` для вывода значения напрямую в поток.

```
(defmethod op->code ((op (eql :embed-value)) &rest operands)
  (destructuring-bind (value escapes) operands
    (if *pretty*
      (if escapes
        `(raw-string *html-pretty-printer* (escape (princ-to-string ,value) ,escapes) t)
        `(raw-string *html-pretty-printer* (princ-to-string ,value) t))
      (if escapes
        `(write-sequence (escape (princ-to-string ,value) ,escapes) *html-output*)
        `(princ ,value *html-output*))))))
```

Так, что что-то подобное вот такому коду:

```
HTML> (let ((x 10)) (html (:p x)))
<p>10</p>
NIL
```

будет работать, поскольку `html` преобразует `(:p x)` в что-то наподобие вот этого:

```
(progn
  (write-sequence "<p>" *html-output*)
  (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
  (write-sequence "</p>" *html-output*))
```

Так, что когда будет сгенерирован код, заменяющий вызов `html` в контексте `LET`, то вы получите следующий результат:

```
(let ((x 10))
  (progn
    (write-sequence "<p>" *html-output*)
    (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
    (write-sequence "</p>" *html-output*)))
```

и ссылки на `x` в сгенерированном коде превратятся в ссылки на лексическую переменную из выражения `LET`, окружающего выражение `html`.

С другой стороны, метод `:embed-code` интересен, поскольку она крайне примитивен. Поскольку функция `process` передала выражение функции `embed-code`, которая сохранила его в операции `:embed-code`, то все, что вам нужно сделать – извлечь и вернуть это выражение.

```
(defmethod op->code ((op (eql :embed-code)) &rest operands)
  (first operands))
```

Это позволяет использовать, например, вот такой код:

```
HTML> (html (:ul (dolist (x '(foo bar baz)) (html (:li x)))))
<ul>
  <li>FOO</li>
  <li>BAR</li>
  <li>BAZ</li>
</ul>
NIL
```

Внешний вызов `html` раскрывается в код, который делает что-то подобное следующему коду:

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz)) (html (:li x)))
  (write-sequence "</ul>" *html-output*))
```

И затем, если вы раскроете вызов `html` в теле `DOLIST`, то вы получите что-то подобное следующему коду:

```
(progn
  (write-sequence "<ul>" *html-output*)
  (dolist (x '(foo bar baz))
    (progn
      (write-sequence "<li>" *html-output*)
      (write-sequence (escape (princ-to-string x) "<>&") *html-output*)
      (write-sequence "</li>" *html-output*)))
  (write-sequence "</ul>" *html-output*))
```

Этот код, будет генерировать результат, который вы уже видели выше.

Специальные операторы FOO

Вы можете остановиться на этом – язык FOO достаточно выразителен для генерации практически любого HTML какой вы можете придумать. Однако, вы можете добавить две новых возможности к языку с помощью небольшого количества кода, который сделает язык более мощным: специальные операторы и макросы.

Специальные операторы FOO аналогичны специальным операторам в Common Lisp. Специальные операторы обеспечивают возможность выражения тех вещей, которые не могут быть выражены на языке, поддерживающем только базовые правила вычисления выражений. Или, если посмотреть с другой стороны, специальные операторы предоставляют доступ к примитивам, используемым ядром языка, вычисляющим выражения.

В качестве простого примера, в компиляторе FOO, ядро языка использует функцию `embed-value` для генерации кода, который будет вставлять значение переменной в генерируемый HTML. Однако, поскольку `embed-value` передаются только символы, то не существует способа (в том языке, который я описывал) включить значение произвольного выражения Common Lisp; в этом случае функция `process` передает пары значений функции `embed-code`, а не `embed-value`, так что возвращаемые значения игнорируются. Обычно, это то, что нам надо, поскольку главной причиной вставки кода на Lisp в программу на а FOO является возможность использования управляющих конструкций Lisp. Однако, иногда вы захотите вставить значение вычисленного выражения в сгенерированный HTML. Например, вы можете захотеть, чтобы программа на FOO генерировала параграф, содержащий случайное число:

```
(:p (random 10))
```

Но это не будет работать, поскольку код будет вычислен и его значение будет отброшено.

```
HTML> (html (:p (random 10)))  
<p></p>  
NIL
```

В реализованном нами языке, вы можете обойти это путем вычисления значения вне вызова `html`, и затем вставки его используя переменную.

```
HTML> (let ((x (random 10))) (html (:p x)))  
<p>1</p>  
NIL
```

Но это будет раздражать, особенно когда вы считаете, что если бы вы могли бы передать выражение `(random 10)` функции `embed-value` вместо `embed-code`, то это было бы то, что надо. Так что вы можете определить специальный оператор `:print`, который будет обрабатываться ядром языка FOO с использованием правила, отличного от правил для обычных выражений FOO. А именно, вместо генерации элемента `<print>`, он будет передавать выражение, заданное в его теле, функции `embed-value`. Так что вы сможете вывести параграф, содержащий случайное число с помощью вот такого вот кода:

```
HTML> (html (:p (:print (random 10))))  
<p>9</p>  
NIL
```

Понятно, что это специальный оператор полезен только в скомпилированном коде на FOO, поскольку `embed-value` не работает в режиме интерпретации. Еще одним специальным оператором, который может быть использован и в режиме компиляции, и в режиме интерпретации, является оператор `:format`, который позволяет вам генерировать вывод используя функцию `FORMAT`. Аргументами специального оператора `:format` являются строка, управляющая форматом вывода данных, и за ней, любые аргументы. Когда все аргументы `:format` являются само-вычисляемыми FIXME self-evaluating объектами, то строка генерируется путем передачи аргументов функции `FORMAT`, и полученная строка затем выводится также как и любая другая строка. Это позволяет использовать выражения `:format` в выражениях FOO, переданных функции `emit-html`. В скомпилированном коде FOO, аргументами `:format` могут быть любые выражения Lisp.

Другие специальные операторы обеспечивают контроль за тем, какие символы будут автоматически преобразовываться, а также использоваться для вывода символов новой строки: специальный оператор `:noescape` приводит к вычислению всех выражений, но при этом переменная `*escapes*` получает значение `NIL`, в то время, как `:attribute` вычисляет все выражения с `*escapes*` равным `*attribute-escapes*`. А оператор `:newline` преобразуется в код, который выдает явный перевод строки.

Так что, как вы будете определять специальные операторы? Существует два аспекта обработки специальных операторов: как обработчик языка распознает формы, которые используются для представлении специальных операторов и как он будет знать, какой код выполнять для обработки каждого из специальных операторов?

Вы можете изменить функцию `process-sexp-html` чтобы она распознавала каждый их специальных операторов и обрабатывала их соответствующим образом – с логической точки зрения, специальные операторы являются частью реализации языка, и их не будет очень много. Однако, было бы удобно иметь модульный способ добавления новых специальных операторов – не потому-что пользователи FOO будут иметь возможность их добавления, а просто для нашего собственного удобства.

Определим специальное выражение как список, чьим значением CAR является символ, представляющий имя специального оператора. Вы можете пометить имена специальных операторов путем добавления не-NIL значения к списку свойств символов принадлежащем пункту FIXME `key html-special-operator`. Так что вы можете определить функцию, которая проверяет, является ли данное выражение, выражением специального оператора, примерно вот так:

```
(defun special-form-p (form)
  (and (consp form) (symbolp (car form)) (get (car form) 'html-special-operator)))
```

Код, реализующий каждый из специальных операторов, также ответственен за получение оставшейся части списка FIXME операндов?? и выполнения того, чего требует семантика специального оператора. Предполагая, что вы также определили функцию `process-special-form`, которая принимает в качестве аргументов обработчик язык и выражение со специальным оператором, и выполняет соответствующий код для генерации последовательности вызовов для объекта `processor`, то вы можете расширить функцию `process` обработкой специальных операторов следующим образом:

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form) (embed-code processor form))
    (t (embed-value processor form))))
```

Вы должны в начале добавить вызов `special-form-p` поскольку специальные операторы могут выглядеть также как обычные выражения `FOO`, точно также как специальные операторы Common Lisp выглядят также как вызовы обычных функций.

Теперь вам нужно лишь реализовать `process-special-form`. Вместо того, чтобы определять одну монолитную функцию, которая реализует все специальные операторы, вы должны определить макрос, который позволит вам определять специальные операторы, практически также как обычные функции, и который возьмет на себя заботу о добавлении записи `html-special-operator` в список свойств имен специальных операторов. В действительности, значением которое вы сохраняете в списке свойств может быть функция, которая реализует специальный оператор. Вот определение соответствующего макроса:

```
(defmacro define-html-special-operator (name (processor &rest other-parameters)
&body body)
  `(eval-when (:compile-toplevel :load-toplevel :execute)
    (setf (get ',name 'html-special-operator)
      (lambda (,processor ,@other-parameters) ,@body))))
```

Это достаточно сложный вид макроса, но если вы будете изучать по одной строке за раз, то вы не найдете ничего сложного. Для того, чтобы увидеть как он работает, рассмотрим простое использование этого макроса – определение специального оператора `:noescape`, и посмотрим на раскрытие этого макроса. Если вы напишите вот так:

```
(define-html-special-operator :noescape (processor &rest body)
  (let ((*escapes* nil))
    (loop for exp in body do (process processor exp))))
```

то это приведет к получению следующего кода:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (setf (get ':noescape 'html-special-operator)
    (lambda (processor &rest body)
      (let ((*escapes* nil))
        (loop for exp in body do (process processor exp))))))
```

Специальный оператор `EVAL-WHEN`, как обсуждалось в главе 20, используется для того, чтобы быть уверенными, что данный код будет виден во время компиляции с помощью функции `COMPILE-FILE`. Это нужно, если вы захотите определить `define-html-special-operator` в файле, и затем использовать только что определенный специальный оператор в том же самом файле.

Затем, выражение `SETF` устанавливает значение для свойства `html-special-operator` символа `:noescape` чтобы оно содержало анонимную функцию, с тем же списком параметров, как это было определено в `define-html-special-operator`. За счет того, что для `define-html-special-operator` параметры разбиваются на две части – `processor` и все остальное, вы будете уверены в том, что все специальные аргументы будут принимать как минимум один аргумент.

Тело анонимной функции является выражением, передаваемым `define-html-special-operator`. Задачей анонимной функции является реализация действия специального оператора путем вызова соответствующих функций интерфейса `FIXME backend` для генерации корректного HTML или кода, который будет генерировать этот HTML. Она также использует `process` для вычисления выражения как выражения `FOO`.

Специальный оператор `:noescape` является достаточно простым – все что он делает, это передача выражения в функцию `process` с переменной `*escapes*` установленной в `NIL`. Другими словами, этот специальный оператор запрещает стандартное маскирование символов, выполняемое `process-sexp-html`.

При использовании специальных операторов определенных таким образом, все что нужно делать `process-special-form` – всего лишь найти анонимную функцию в списке свойств символа с именем оператора, и применить ее (с помощью `APPLY`) к списку из обработчика и оставшейся части выражения.

```
(defun process-special-form (processor form)
  (apply (get (car form) 'html-special-operator) processor (rest form)))
```

Теперь вы готовы к тому, чтобы определить пять оставшихся специальных операторов `FOO`. Похожим на `:noescape` является `:attribute`, который вычисляет заданные выражения с переменной `*escapes*` равной `*attribute-escapes*`. Этот специальный оператор полезен, если вы хотите написать вспомогательную функцию, которая будет выдавать значения атрибутов. Если вы напишите вот такую вот функцию:

```
(defun foo-value (something)
  (html (:print (frob something))))
```

то `macrohtml` сгенерирует код, который выполнит маскирование символов, указанных в `*element-escapes*`. Но если вы планируете использовать `foo-value` следующим образом:

```
(html (:p :style (foo-value 42) "Foo"))
```

то вы захотите, чтобы генерировался код, который бы использовал данные из переменной `uses *attribute-escapes*`. Так что вместо этого, вы можете написать нечто подобное:

```
(defun foo-value (something)
  (html (:attribute (:print (frob something)))))
```

Определение `:attribute` выглядит следующим образом:

```
(define-html-special-operator :attribute (processor &rest body)
  (let ((*escapes* *attribute-escapes*))
    (loop for exp in body do (process processor exp))))
```

Два других специальных оператора – `:print` и `:format`, используются для вывода значений. Специальный оператор `:print`, как обсуждалось ранее, используется в скомпилированных программах на `FOO` для вставки значения произвольного выражения `Lisp`. Специальный оператор `:format` соответствует операции генерации строки с помощью выражения `(format nil ...)` и последующей вставки этой строки в вывод. Основной причиной определения `:format` как специального оператора является удобство. Так:

```
(:format "Foo: ~d" x)
```

лучше выглядит чем:

```
(:print (format nil "Foo: ~d" x))
```

Есть также небольшое преимущество если вы используете `:format` с само-вычисляемыми аргументами FIXME self-evaluating, то FOO может вычислить `:format` во время компиляции, а не ждать выполнения программы. Определения для `:print` и `:format` выглядят вот так:

```
(define-html-special-operator :print (processor form)
  (cond
    ((self-evaluating-p form)
     (warn "Redundant :print of self-evaluating form ~s" form)
     (process-sexp-html processor form))
    (t
     (embed-value processor form))))
(define-html-special-operator :format (processor &rest args)
  (if (every #'self-evaluating-p args)
      (process-sexp-html processor (apply #'format nil args))
      (embed-value processor `(format nil ,@args))))
```

Специальный оператор `:newline` приводит к выводу знака новой строки, что иногда удобно.

```
(define-html-special-operator :newline (processor)
  (newline processor))
```

В заключение, специальный оператор `:progn` аналогичен специальному оператору `PROGN` в Common Lisp. Он просто последовательно обрабатывает выражения внутри своего тела.

```
(define-html-special-operator :progn (processor &rest body)
  (loop for exp in body do (process processor exp)))
```

Другими словами, следующий код:

```
(html (:p (:progn "Foo " (:i "bar") " baz")))
```

сгенерирует тот же код, что и:

```
(html (:p "Foo " (:i "bar") " baz"))
```

Это может быть показаться странным, поскольку обычное выражение FOO может иметь любое количество выражений внутри своего тела. Однако специальный оператор удобен в одной ситуации – при написании макросов FOO, что приводит нас к последней возможности языка, которую нам надо реализовать.

Макросы FOO

Макросы FOO аналогичны по духу макросам Common Lisp. Макрос FOO является отрывком кода, который принимает в качестве аргумента выражение на FOO, и возвращает в качестве результата новое выражение на FOO, которое затем вычисляется в соответствии со стандартными правилами вычисления выражений на FOO. Реализация очень похожа на реализацию специальных операторов.

Также как и для специальных операторов вы можете определить функцию-предикат, которая будет проверять – является ли заданное выражение макросом.

```
(defun macro-form-p (form)
  (cons-form-p form #'(lambda (x) (and (symbolp x) (get x 'html-macro))))))
```

Тут мы используем функцию `cons-form-p`, определенную выше, поскольку мы хотим позволить использовать любой синтаксис FOO выражений. Однако, вам нужно передать другую функцию-предикат, которая будет проверять – является ли имя выражения символом с не-NIL свойством `html-macro`. Также, как и при реализации специальных операторов, мы определим макрос для определения макросов FOO, которая будет отвечать за сохранение функции в списке свойств символа с именем макроса (имя свойства будет равно `html-macro`). Однако, определение макроса немного более сложное, поскольку FOO поддерживает использование двух видов макросов. Некоторые из макросов, которые вы будете определять будут вести себя как обычные элементы HTML, и вам может понадобиться упрощенный доступ к списку атрибутов. Другие макросы будут требовать упрощенного доступа к элементам их тела.

Вы можете сделать различие между двумя видами макросов неявным: когда вы определяете макрос FOO, то список параметров может включать параметр `&attributes`. Если он будет указан, то макро-выражение будет рассматриваться как обычное выражение-ячейка, и макро-функция будет получать два значения – список свойств-атрибутов, и список выражений из которых состоит тело выражения. Макро-выражение без параметра `&attributes` будет разбираться как не имеющее атрибутов, и макро-функция будет принимать один параметр – список, содержащий выражения составляющие тело макроса. Первый вид полезен для шаблонов HTML. Например:

```
(define-html-macro :mytag (&attributes attrs &body body)
  `((:div :class "mytag" ,@attrs) ,@body))
```

```
HTML> (html (:mytag "Foo"))
<div class='mytag'>Foo</div>
NIL
HTML> (html (:mytag :id "bar" "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
HTML> (html ([:mytag :id "bar"] "Foo"))
<div class='mytag' id='bar'>Foo</div>
NIL
```

Последний вид макросов более полезен для написания макросов, которые манипулируют выражениями, составляющими их тело. Этот тип макросов может работать как управляющие конструкции HTML. В качестве простого примера рассмотрим следующий макрос, который реализует конструкцию `:if`:

```
(define-html-macro :if (test then else)
  `(if ,test (html ,then) (html ,else)))
```

Этот макрос позволит вам писать так:

```
(:p (:if (zerop (random 2)) "Heads" "Tails"))
```

вместо такой, более многословной версии:

```
(:p (if (zerop (random 2)) (html "Heads") (html "Tails")))
```

Для того, чтобы определить какой тип макроса вы должны генерировать, вам необходима функция, которая выполнит разбор списка параметров, переданных `define-html-macro`. Эта функция возвращает два значения: имя параметра `&attributes`, или `NIL`, если он не указан, и список всех элементов `args` оставшихся после удаления маркера `&attributes` и последующих элементов списка.

```
(defun parse-html-macro-lambda-list (args)
  (let ((attr-cons (member '&attributes args)))
    (values
     (cadr attr-cons)
     (nconc (ldiff args attr-cons) (cddr attr-cons)))))
```

```
HTML> (parse-html-macro-lambda-list '(a b c))
NIL
(A B C)
HTML> (parse-html-macro-lambda-list '(&attributes attrs a b c))
ATTRS
(A B C)
HTML> (parse-html-macro-lambda-list '(a b c &attributes attrs))
ATTRS
(A B C)
```

Элемент, следующий за `&attributes` в списке параметров, также может быть списком параметров.

```
HTML> (parse-html-macro-lambda-list '(&attributes (&key x y) a b c))
(&KEY X Y)
(A B C)
```

Теперь у вас все готово для написания `define-html-macro`. В зависимости от того, были ли указан параметр `&attributes` вам нужно сгенерировать один или другой из видов HTML макросов, так что главный макрос просто определяет что он должен генерировать, и затем вызывает вспомогательную функцию, которая будет генерировать нужный код.

```
(defmacro define-html-macro (name (&rest args) &body body)
  (multiple-value-bind (attribute-var args)
    (parse-html-macro-lambda-list args)
    (if attribute-var
        (generate-macro-with-attributes name attribute-var args body)
        (generate-macro-no-attributes name args body))))
```

Функции, которые генерируют соответствующий код выглядят вот так:

```
(defun generate-macro-with-attributes (name attribute-args args body)
  (with-gensyms (attributes form-body)
    (if (symbolp attribute-args) (setf attribute-args `(&rest ,attribute-args)))
    `(eval-when (:compile-toplevel :load-toplevel :execute)
      (setf (get ',name 'html-macro-wants-attributes) t)
      (setf (get ',name 'html-macro)
        (lambda (,attributes ,form-body)
          (destructuring-bind (,@attribute-args) ,attributes
            (destructuring-bind (,@args) ,form-body
              ,@body)))))))
(defun generate-macro-no-attributes (name args body)
  (with-gensyms (form-body)
    `(eval-when (:compile-toplevel :load-toplevel :execute)
      (setf (get ',name 'html-macro-wants-attributes) nil)
      (setf (get ',name 'html-macro)
        (lambda (,form-body)
          (destructuring-bind (,@args) ,form-body ,@body))))))
```

Функции, которые вы определите, принимают либо один, либо два аргумента, и затем используют DESTRUCTURING-BIND для их разделения, и связывания их с параметрами, определенными в вызове к define-html-macro. В обоих раскрытиях выражений вам необходимо сохранить макро-функции в списке свойств символа, используя имя свойства равное html-macro, а также логическое значение, указывающее на то, принимает ли макрос параметр &attributes, в свойстве html-macro-wants-attributes. Вы используете это свойство в следующей функции, expand-macro-form, для того, чтобы определить как макро-функция должна быть запущена:

```
(defun expand-macro-form (form)
  (if (or (consp (first form))
        (get (first form) 'html-macro-wants-attributes))
      (multiple-value-bind (tag attributes body) (parse-cons-form form)
        (funcall (get tag 'html-macro) attributes body))
      (destructuring-bind (tag &body body) form
        (funcall (get tag 'html-macro) body))))
```

Нам надо сделать последний шаг для интеграции макросов в наш язык путем добавления соответствующей ветки в COND в функции process.

```
(defun process (processor form)
  (cond
    ((special-form-p form) (process-special-form processor form))
    ((macro-form-p form) (process processor (expand-macro-form form)))
    ((sexp-html-p form) (process-sexp-html processor form))
    ((consp form) (embed-code processor form))
    (t (embed-value processor form))))
```

Это окончательная версия process.

Публичный интерфейс разработчика (API)

Теперь вы готовы к реализации макроса html – основной точке входа компилятора FOO. Другими частями публичного интерфейса разработчика являются emit-html и with-html-output, которые мы обсуждали в предыдущей главе, и define-html-macro, которую мы обсуждали в предыдущем разделе. Макрос define-html-macro должен быть частью интерфейса разработчика, поскольку пользователи FOO захотят писать свои собственные макросы. С другой стороны, define-html-special-operator не является частью

интерфейса, поскольку он требует слишком глубоко знания внутреннего устройства FOO для определения нового специального оператора. И должно быть очень мало вещей которые не смогут быть сделаны при наличии существующих возможностей языка и специальных операторов.

Последним элементом публичного интерфейса, который мы рассмотрим до `html`, является еще один макрос – `in-html-style`. Этот макрос контролирует то, должен ли FOO генерировать XHTML или простой HTML путем установки переменной `*xhtml*`. Причиной того, что вам нужен макрос, является то, что вы можете захотеть обернуть код, который устанавливает `*xhtml*` в `EVAL-WHEN`, так что вы можете установить его в файл, и это будет влиять на поведение макроса `html` находящегося в том же файле.

```
(defmacro in-html-style (syntax)
  (eval-when (:compile-toplevel :load-toplevel :execute)
    (case syntax
      (:html (setf *xhtml* nil))
      (:xhtml (setf *xhtml* t)))))
```

И в заключение, давайте рассмотрим `html`. Единственная нестандартность в реализации `html` возникает из необходимости генерировать код, который будет использоваться для генерации и компактного, и "красивого" `FIXME pretty` вывода, в зависимости от значения переменной `*pretty*` во время выполнения. Таким образом в `html` требуется генерировать раскрытие, которое будет содержать выражение `IF` и две версии кода – одну скомпилированную с `*pretty*` равным истине, и одну – для значения переменной равной `NIL`. Также составляет сложность то, что достаточно часто один вызов `html` содержит вложенные вызовы `html`, например вот так:

```
(html (:ul (dolist (item stuff)) (html (:li item)))))
```

Если внешний вызов `html` раскрывается в выражение `IF` с двумя версиями кода, одним для случая, когда переменная `*pretty*` имеет истинное значение, и вторым, когда она имеет ложное, то будет глупо, если вложенные выражения `html` также будут раскрываться в две версии. В действительности это будет вести к экспоненциальному росту кода, поскольку вложенные `html` уже будут раскрыты дважды – один раз для ветви `*pretty*-is-true`, и один раз для ветви `*pretty*-is-false`. Если каждое из раскрытий сгенерирует две версии, то вы будете иметь 4 версии кода. А если вложенное выражение `html` содержит еще одно вложенное выражение `html`, то вы получите восемь версий. Если компилятор достаточно умен, то он может распознать, что большая часть кода не будет использована, и удалит ее, но распознавание таких ситуаций займет достаточно большое время, замедляя компиляцию любой функции, которая использует вложенные вызовы `html`.

К счастью вы можете легко избежать это разрастание ненужного кода путем генерации раскрытия, которое локально переопределяет макрос `html` используя `MACROLET`, для того, чтобы генерировать только нужный вид кода. Сначала вы определяете вспомогательную функцию, которая получает вектор кодов операций, возвращаемы `sexpr->ops` и прогоняет его через функции `optimize-static-output` и `generate-code` (две стадии, на которые влияет значение переменной `*pretty*`) с переменной `*pretty*` установленной в нужное значение, и затем собирает результирующий код в `PROGN`. (`PROGN` возвращает `NIL` лишь для унификации результатов.).

```
(defun codegen-html (ops pretty)
  (let ((*pretty* pretty))
    `(progn ,@(generate-code (optimize-static-output ops)) nil)))
```

Используя эту функцию, вы можете определить html следующим образом:

```
(defmacro html (&whole whole &body body)
  (declare (ignore body))
  `(if *pretty*
      (macrolet ((html (&body body) (codegen-html (sexp->ops body) t)))
        (let ((*html-pretty-printer* (get-pretty-printer))) ,whole))
      (macrolet ((html (&body body) (codegen-html (sexp->ops body) nil)))
        ,whole)))
```

Параметр `&whole` представляет оригинальное выражение `html`, и поскольку он интерполируется в раскрытие в теле двух `MACROLET`, то он будет обрабатываться с каждым из определений `html` – для кода, выдающий "красивый" и обычный результат. Заметьте, что переменная `*pretty*` используется и при раскрытии макроса и при выполнении сгенерированного кода. Она используется при раскрытии макроса в `codegen-html` для того, чтобы заставить `generate-code` генерировать нужный вид кода. И она используется во время выполнения в выражении `IF` сгенерированном макросом `html` самого верхнего уровня, для того, чтобы определить, какая из ветвей – `pretty-printing` и `non-pretty-printing` будет выполнена.

Конец работы

Как обычно, вы можете продолжить работу над этим кодом для расширения его функциональности. Одной из интересных задач может быть использование системы генерации вывода для создания других видов выходных данных. В реализации `FOO`, которую вы можете скачать с сайта посвященной книге, вы можете найти код, который реализует вывод `CSS`, который может быть интегрирован в `HTML`, и в компиляторе и в интерпретаторе. Это интересный случай, поскольку синтаксис `CSS` не может быть также просто отображен в `s-выражения`, как это можно сделать для `HTML`. Однако, если вы посмотрите в этот код, то вы увидите, что все равно возможно определить синтаксис, основанный на `s-выражениях`, для представления разных конструкций, доступных в `CSS`.

Более амбициозной задачей будет добавление поддержки генерации `JavaScript`. При правильном подходе, добавление поддержки `JavaScript` в `FOO` может привести к двум большим победам. Во первых, если вы определите синтаксис, основанный на `s-выражениях`, так что вы сможете отобразить его на синтаксис `JavaScript`, то вы сможете начать писать макросы (на `Common Lisp`) для добавления новых конструкций к языку, который вы используете для написания кода, исполняемого на стороне пользователя, который затем будет компилироваться в `JavaScript`. Во вторых, при переводе `s-выражений` `FOO` с поддержкой `JavaScript` в обычный `JavaScript`, вы можете столкнуться небольшими, но раздражающими различиями в реализации `JavaScript` в разных браузерах. Так что код `JavaScript` генерируемый `FOO` либо может содержать соответствующие условия для выполнения одних операций в одном браузере, и других в другом браузере, либо может генерировать разный код в зависимости от того, какой браузер вы хотите поддерживать. Так что, если вы используете `FOO` в динамически генерируемых страницах, то вы можете использовать информацию из заголовка `User-Agent`, заставляя функцию `request` генерировать правильный код `JavaScript` для конкретного браузера.

Но если это вас интересует, то вы можете это реализовать сами, поскольку это конец последней практической главы данной книги. В следующей главе я подведу итоги, сделаю короткий обзор некоторых тем, которые я не затрагивал в этой книге, такие как искать библиотеки, как оптимизировать код на `Common Lisp`, и как распространять приложения на `Lisp`.

32. Заключение: что дальше ?

Я надеюсь, что теперь вы убеждены, что в названии этой книги нет противоречия. Однако вполне вероятно что есть какая-то область программирования, которая очень практически важна для вас, и которую я совсем не обсудил. Например, я ничего не сказал о том, как разрабатывать графический пользовательский интерфейс (GUI), как связываться с реляционными базами данных, как разбирать XML, или как писать программы, которые являются клиентами различных сетевых протоколов. Так же я не обсудил две темы, которые станут важными, когда вы начнете писать реальные приложения на языке Common Lisp: оптимизацию кода и сборку приложений для поставки.

Должно быть очевидно, что я не собираюсь глубоко охватить эти темы в этой финальной главе. Вместо этого я дам вам несколько направлений в которых вы можете двигаться, занимаясь теми аспектами программирования на LISP, которые интересуют вас больше.

Поиск библиотек LISP

В то время как стандартная библиотека функций, типов данных и макросов, поставляемая с языком Common Lisp, достаточно велика, она служит лишь целям общего программирования. Специализированные задачи, такие, как создание графического пользовательского интерфейса (GUI), общение с базами данных и разбор XML требуют библиотек, не обеспечиваемых стандартом этого языка.

Простейшим путем найти библиотеку, делающую что-то нужное вам, может быть просто проверить ее наличие в составе вашей реализации Lisp. Большинство реализаций предоставляет по крайней мере некоторые возможности, не указанные в стандарте языка. Коммерческие поставщики, как правило, особенно усиленно работают над предоставлением дополнительных библиотек для своих реализаций с целью оправдать их стоимость. Например, Franz's Allegro Common Lisp, Enterprise Edition, поставляется среди прочего с библиотеками для разбора XML, общения по протоколу SOAP, генерирования HTML, взаимодействия с реляционными базами данных и построения графического интерфейса пользователя различными путями. Другая выдающаяся коммерческая реализация, LispWorks, также предоставляет несколько подобных библиотек, включая пользующуюся уважением переносимую библиотеку CAPI, которая может использоваться для разработки GUI-приложений, работающих на любой операционной системе, где есть LispWorks.

Свободные реализации и реализации с открытым кодом обычно не включают такую большую связку библиотек, вместо этого полагаясь на переносимые свободные и открытые библиотеки. Но даже эти реализации обычно заполняют такие из наиболее важных областей, не охваченных стандартом языка, как работа с сетями и многопоточность.

Единственный недостаток использования специфичных для реализации библиотек – это то, что они привязывают вас к той реализации, которая их предоставляет. Если вы поставляете приложения конечным пользователям или развёртываете ваши приложения на серверах, которыми вы сами управляете, это может быть не очень важно. Однако если вы хотите писать код для повторного использования другими программистами на Лисп или просто не хотите быть привязанными к конкретной реализации, это может быть будет вас раздражать.

Переносимые библиотеки (переносимость означает либо что они написаны целиком на стандартном Common Lisp, либо что они содержат необходимые макросы чтобы работать с несколькими реализациями) лучше всего искать в Интернете.

Вот три лучших места, откуда можно начать поиски (с обычными предосторожностями,

связанными с тем, что все URL-и устаревают сразу же, как только они напечатаны на бумаге):

- * Common-Lisp.net (<http://www.common-lisp.net/>) – сайт, размещающий свободные и открытые проекты на Common Lisp, предоставляя средства контроля версий исходного кода, списки рассылки и размещение WEB-документов проектов. В первые полтора года работы сайта было зарегистрировано около сотни проектов.
- * Коллекция открытого кода Common Lisp (The Common Lisp Open Code Collection, CLOCC) (<http://clocc.sourceforge.net/>) – немного более старая коллекция библиотек свободного ПО, которые, как подразумевается, должны быть переносимы между разными реализациями языка Common Lisp и самодостаточными, то есть не зависящими от каких-то других библиотек, не включенных в эту коллекцию.
- * Cliki (Common Lisp Wiki) (<http://www.cliki.net/>) – Wiki-сайт, посвященный свободному программному обеспечению на языке Common Lisp. В то время как и любой другой сайт, основанный на Wiki, он может изменяться в любое время, обычно на нем есть довольно много ссылок на библиотеки и реализации Common Lisp с открытым кодом. Система редактирования документов, на которой работает сайт, и давшая ему имя, также написана на языке Common Lisp.

Пользователи Linux, работающие на системах Debian или Gentoo, также могут легко устанавливать постоянно растущее число библиотек для языка Lisp, которые распространяются с помощью средств распространения и установки этих систем: apt-get на Debian и emerge на Gentoo.

Я не буду сейчас рекомендовать какие-то конкретные библиотеки, поскольку ситуация с ними меняется каждый день – после многих лет зависти к библиотекам языков Perl, Python и Java, программисты на Common Lisp в последние пару лет приняли вызов к созданию такого набора библиотек, которого заслуживает Common Lisp, и коммерческих, и с открытым кодом.

Одна из областей, в которой в последнее время было много активности – это фронт разработки графического интерфейса приложений. В отличие от Java и C#, но как и в языках Perl, Python и C, в языке Common Lisp не существует единственного пути для разработки графического интерфейса. Способ разработки зависит от реализации Common Lisp, с которой вы работаете, а также от операционной системы, которую вы хотите поддерживать.

Коммерческие реализации Common Lisp обычно обеспечивают какой-то способ разработки графического интерфейса для платформ, на которых они работают. В дополнение к этому, LispWork предоставляет библиотеку CAPI, уже упоминавшуюся, для разработки переносимых графических приложений.

Из программного обеспечения с открытым кодом у вас есть несколько возможных вариантов. На системах Unix вы можете писать низкоуровневые приложения для системы X Windows используя библиотеку CLX, реализацию X Windows протокола на чистом языке Common Lisp, примерно такая же, как xlib для языка C. Или вы можете использовать различные обертки для высокоуровневых API и библиотек, таких как GTK или Tk, также, как вы можете делать это в языках Perl или Python.

Или если вы ищите чего-то совсем необычного, вы можете посмотреть на библиотеку Common Lisp Interface Manager (CLIM). Будучи наследником графической библиотеки символических LISP-машин, CLIM является мощным, но сложным средством. Хотя многие коммерческие реализации языка LISP поддерживают его, не похоже что он очень сильно использовался. Но в последние несколько лет новая реализация CLIM с открытым кодом, McCLIM, набирает обороты (она располагается на сайте Common-Lisp.net), так что возможно мы на грани нового расцвета этой библиотеки.

Взаимодействие с другими языками программирования

В то время как много полезных библиотек может быть написано на чистом Common Lisp, используя только возможности, указанные в стандарте языка, и гораздо больше может быть написано с использованием нестандартных средств, предоставляемых какой-то из реализаций, иногда проще использовать существующую библиотеку, написанную на другом языке, таком, как C.

Стандарт языка не дает механизма для вызова из кода на языке LISP кода, написанного на другом языке программирования, и не требует, чтобы реализация языка предоставляла такие возможности. Но в наше время почти все реализации поддерживают то, что называется Foreign Function Interface, или кратко – FFI.

Основная задача FFI – позволить вам дать языку LISP достаточно информации для того, чтобы прилинковать чужой код, написанный не на LISP. Таким образом, если вы собираетесь вызывать функции из какой-то библиотеки для языка C, вам нужно рассказать языку LISP о том, как транслировать объекты LISP, передаваемые этой функции, в типы данных языка C, а значение, возвращаемое функцией, – обратно в объекты языка LISP. Однако каждая реализация предоставляет свой собственный механизм FFI, со своими отличными от других возможностями и синтаксисом. Некоторые реализации FFI позволяют делать функции обратного вызова (callback functions), другие – нет. Библиотека Universal Foreign Function Interface (UFFI) предоставляет слой для переносимости приложений, написанных с использованием FFI, доступный на более полудюжины реализаций Common Lisp. Библиотека определяет макросы, которые раскрываются в вызовы соответствующих функций интерфейса FFI данной реализации. Библиотека UFFI применяет подход "наименьшего общего знаменателя", то есть она не может использовать преимущества всех возможностей разных реализаций FFI, но все же обеспечивает хороший способ построения оберток API для языка C.

Сделать чтоб работало, сделать чтоб работало правильно, сделать чтоб работало быстро

Как уже было сказано много раз, по разным сведениям, Дональдом Кнутом, Чарльзом Хоаром и Эдсгером Дейкстрой, преждевременная оптимизация является первопричиной всех зол. Common LISP – замечательный язык в этом отношении, если вы хотите следовать этой практике и при этом вам нужна высокая производительность. Эта новость может быть для вас сюрпризом, если вы уже слышали традиционное мнение о том, что LISP – медленный язык. В ранние года языка LISP, когда компьютеры еще программировали с помощью перфокарт, этот язык с его высокоуровневыми чертами был обречен быть медленнее, чем его конкуренты, а именно, ассемблер и Фортран. Но это было давно. В то же время LISP использовался для всего, от создания сложных систем искусственного интеллекта, до написания операционных систем, и было проделано много работы, чтобы выяснить, как компилировать LISP в эффективный код. В этом разделе мы поговорим о некоторых из причин, почему Common LISP является прекрасным языком для написания высокопроизводительных программ, и о том, как это достигается.

Первой причиной того, что LISP является прекрасным языком для написания высокопроизводительного кода, является, как ни парадоксально, динамический характер программирования на этом языке – та самая вещь, которая сначала мешала довести производительность программ на языке LISP до уровней, достигнутых компиляторами языка Фортран. Причина того, что динамичность LISP-а упрощает создание высокопроизводительного кода, заключается в том, что первый шаг к эффективному коду – это всегда поиск правильных алгоритмов и структур данных.

Динамические свойства языка LISP сохраняют код гибким, что упрощает опробование разных

подходов. Имея ограниченное время на создание программы, вы, в конечном итоге, более вероятно придете к более высокопроизводительной версии, если не будете проводить много времени, забираясь в тупики и выбираясь из них обратно. В языке Common LISP вы можете опробовать идею, увидеть, что она ведет в никуда, и двигаться дальше, не тратя много времени убеждая компилятор, что ваш код все же достоин того, чтобы он наконец запустился, и ожидая, когда же он наконец закончит его компилировать. Вы можете написать простую, но эффективную версию функции – набросок кода –, чтобы определить, работает ли ваш основной код правильно, а затем заменить эту функцию на более сложную и более эффективную ее реализацию, если все хорошо. Но если общий подход к решению оказался с изъяном, то вы не потратите кучу времени на отладку функции, которая в конечном итоге не нужна, что означает, что у вас остается больше времени для поиска лучшего подхода к решению задачи.

Следующая причина того, что Common LISP является хорошим языком для разработки высокопроизводительного программного обеспечения, заключается в том, что большинство реализаций Common LISP обладают зрелыми компиляторами, которые генерируют достаточно эффективный машинный код. Мы остановимся сейчас на том, как помочь компиляторам сгенерировать код, который был бы сравним с кодом, генерируемым компиляторами языка C, но эти реализации и так уже немного быстрее, чем те языки программирования, реализации которых менее зрелы, и которые используют более простые компиляторы или интерпретаторы. Также, поскольку компилятор LISP доступен во время выполнения программы, программист на LISP имеет некие возможности, которые было бы трудно эмулировать в других языках – ваша программа может генерировать код на LISP во время выполнения, который затем может быть скомпилирован в машинный код и запущен. Если сгенерированный код должен работать много раз, то это – большой выигрыш. Или, даже без использования компилятора во время выполнения, замыкания дают вам другой способ объединять код и данные времени выполнения. Например, библиотека регулярных выражений CL-PPCRE, работающая под CMUCL, быстрее, чем библиотека регулярных выражений языка Perl, на некоторых тестах, несмотря даже на то, что библиотека языка Perl написана на высоко оптимизированном языке C. Это вероятно происходит от того, что в Perl регулярные выражения транслируются в байт-код, который затем интерпретируется средствами поддержки регулярных выражений Perl, в то время как библиотека CL-PPCRE транслирует регулярные выражения в дерево скомпилированных функций, использующих замыкание, (FIXME closures?), которые вызывают друг друга средствами нормальных вызовов функций.

Однако, даже с правильным алгоритмом и высококачественным компилятором, вы можете не достичь нужной вам скорости. Значит пришло время подумать об оптимизации и профайлере. Основной подход здесь в языке LISP, как и в других языках, – сначала применить профайлер, чтобы найти места, где ваша программа реально тратит много времени, а уже потом озаботиться ускорением этих частей.

Есть несколько подходов к профилированию. Стандарт языка предоставляет несколько простейших средств измерения времени выполнения каких-то форм. В частности, макросом TIME можно обернуть любую форму и он вернет возвращаемое формой значение, напечатав сообщение в поток *TRACE_OUTPUT* о том, как долго выполнялась форма, и сколько памяти она использовала. Конкретный вид сообщения определяется конкретной реализацией.

Вы также можете использовать TIME для довольно быстрого и грубого профилирования, чтобы сузить ваши поиски узкого места. Например, предположим у вас есть долго работающая функция, которая вызывает две другие функции, примерно так:

```
(defun foo ()  
  (bar)  
  (baz))
```

Если вы хотите увидеть, какая из функций работает дольше, вы можете изменить определение функции таким образом:

```
(defun foo ()  
  (time (bar))  
  (time (baz)))
```

Теперь вы можете вызвать `foo`, и LISP напечатает два отчета, один для `bar`, другой для `baz`. Формат отчета зависит от реализации, вот как это выглядит в Allegro Common Lisp:

```
CL-USER> (foo)  
; cpu time (non-gc) 60 msec user, 0 msec system  
; cpu time (gc) 0 msec user, 0 msec system  
; cpu time (total) 60 msec user, 0 msec system  
; real time 105 msec  
; space allocation:  
; 24,172 cons cells, 1,696 other bytes, 0 static bytes  
; cpu time (non-gc) 540 msec user, 10 msec system  
; cpu time (gc) 170 msec user, 0 msec system  
; cpu time (total) 710 msec user, 10 msec system  
; real time 1,046 msec  
; space allocation:  
; 270,172 cons cells, 1,696 other bytes, 0 static bytes
```

Конечно, было бы немного проще это читать, если бы вывод включал какие-то пометки. Если вы часто пользуетесь этим приемом, было бы полезно определить ваш собственный макрос вот так:

```
(defmacro labeled-time (form)  
  `(progn  
    (format *trace-output* "~2&~a" ',form)  
    (time ,form)))
```

Если вы замените `TIME` на `labeled-time` в `foo`, вы увидите следующий вывод:

```
CL-USER> (foo)  
(BAR)  
; cpu time (non-gc) 60 msec user, 0 msec system  
; cpu time (gc) 0 msec user, 0 msec system  
; cpu time (total) 60 msec user, 0 msec system  
; real time 131 msec  
; space allocation:  
; 24,172 cons cells, 1,696 other bytes, 0 static bytes  
(BAZ)  
; cpu time (non-gc) 490 msec user, 0 msec system  
; cpu time (gc) 190 msec user, 10 msec system  
; cpu time (total) 680 msec user, 10 msec system  
; real time 1,088 msec  
; space allocation:  
; 270,172 cons cells, 1,696 other bytes, 0 static bytes
```

С этой формой отчета сразу ясно, что большее время выполнения `foo` тратится в `baz`.

Конечно, вывод от `TIME` становится немного неудобным, если форма, которую вы хотите профилировать, вызывается последовательно много раз. Вы можете создать свои средства измерения, используя функции `GET-INTERNAL-REAL-TIME` и `GET-INTERNAL-RUN-TIME`, которые возвращают число, увеличиваемое на величину константы `INTERNAL-TIME-UNITS-`

PER-SECOND каждую секунду.

GET-INTERNAL-REAL-TIME измеряет абсолютное время, реальное время, прошедшее между событиями, в то время как GET-INTERNAL-RUN-TIME дает некое специфичное для конкретной реализации значение, равное времени, которое Лисп-программа тратит реально на выполнение именно пользовательского кода, исключая внутренние затраты Лисп-машины на поддержку программы, такие, как выделение памяти и сборка мусора.

Вот достаточно простой, но полезный вспомогательный инструмент профилирования, построенный с помощью нескольких макросов и функции GET-INTERNAL-RUN-TIME:

```
(defparameter *timing-data* ())
(defmacro with-timing (label &body body)
  (with-gensyms (start)
    `(let ((,start (get-internal-run-time)))
      (unwind-protect (progn ,@body)
        (push (list ',label ,start (get-internal-run-time)) *timing-data*))))))
(defun clear-timing-data ()
  (setf *timing-data* ()))
(defun show-timing-data ()
  (loop for (label time count time-per %-of-total) in (compile-timing-data) do
    (format t "~3d% ~a: ~d ticks over ~d calls for ~d per.~%"
      %-of-total label time count time-per)))
(defun compile-timing-data ()
  (loop with timing-table = (make-hash-table)
    with count-table = (make-hash-table)
    for (label start end) in *timing-data*
    for time = (- end start)
    summing time into total
    do
      (incf (gethash label timing-table 0) time)
      (incf (gethash label count-table 0))
    finally
      (return
        (sort
          (loop for label being the hash-keys in timing-table collect
            (let ((time (gethash label timing-table))
                  (count (gethash label count-table)))
              (list label time count (round (/ time count)) (round (* 100 (/
time total)))))))
          #'> :key #'fifth))))
```

Этот профайлер позволяет вам обернуть вызов любой формы в макрос with-timing. Каждый раз, когда форма будет выполняться, время начала и конца выполнения будет записываться в список, связываясь с меткой, которую вы указываете. Функция show-timing-data выводит таблицу, в которой показано, как много времени было потрачено в различно помеченных секциях кода, следующим образом:

```
CL-USER> (show-timing-data)
 84% BAR: 650 ticks over 2 calls for 325 per.
 16% FOO: 120 ticks over 5 calls for 24 per.
NIL
```

Очевидно, вы могли бы сделать этот профайлер более сложным во многих отношениях. С другой стороны, каждая реализация Лиспа часто предоставляет свои средства профилирования, которые могут выдавать информацию, часто недоступную пользовательскому коду, поскольку они имеют доступ к внутренним механизмам реализации.

Как только вы нашли узкое место в вашем коде, вы начинаете оптимизацию. Первое, что вам следует попробовать, – это, конечно, попытаться найти более эффективный базовый алгоритм, это всегда приносит большую выгоду FIXME (that's where the big gains are to be had). Но если предположить, что вы уже используете эффективный алгоритм, то тогда как раз время начинать рихтовать код, то есть оптимизировать код, чтобы он не делал абсолютно ничего, кроме необходимой работы.

Главное средство при этом – дополнительные объявления LISP. Основная идея объявлений в языке LISP в том, что они дают компилятору информацию, которую он может использовать различным образом, чтобы генерировать более хороший код.

Например, рассмотрим простую функцию:

```
(defun add (x y) (+ x y))
```

Как я упоминал в главе 10, если вы сравните производительность этой LISP-функции с вроде бы эквивалентной функцией на языке C,

```
int add (int x, int y) { return x + y; }
```

вы возможно обнаружите, что LISP-функция заметно медленнее, даже если ваша реализация Common Lisp обладает высококачественным компилятором в машинный код.

Это происходит потому, что версия функции на Common Lisp выполняет гораздо больше всего – компилятор Common LISP даже не знает, что значения *x* и *y* являются числами, и таким образом вынужден сгенерировать код для проверки типов во время выполнения. И как только он определил, что это – числа, он должен также определить, какого типа эти числа – целые, рациональные, с плавающей точкой или комплексные, и перенаправить вызов соответствующей вспомогательной процедуре, обрабатывающей соответствующие типы. И даже если *x* и *y* являются целыми числами, то процедура сложения должна позаботиться о том, что результат сложения может быть слишком большим для представления в FIXNUM-числе, числе, которое может быть представлено в одном машинном слове, и, таким образом, должна выделить число типа BIGNUM.

В языке C, с другой стороны, поскольку типы всех переменных объявлены, компилятор знает точно, какой тип значений будет храниться в *x* и *y*. И, поскольку арифметика языка C просто вызывает переполнение, когда результат сложения слишком большой, чтобы быть представленным в возвращаемом типе, в функции нет ни проверки на переполнение, ни выделения длинного целого значения, если результат не помещается в машинное слово.

Таким образом, поскольку поведение кода на Common Lisp гораздо более близко к математической корректности, версия на языке C, возможно, может быть скомпилирована в одну или две машинных инструкции. Но если вы пожелаете дать компилятору Common Lisp ту же информацию, которую имеет компилятор языка C о типах аргументов и возвращаемом значении, и принять некоторые компромиссы, подобные используемым в языке C, относительно общности кода и проверки ошибок, функция на Common Lisp также может компилироваться в пару машинных инструкций.

Именно для этого и служат объявления. Главная польза объявлений в том, чтобы сказать компилятору о типах переменных и других выражений. Например, вы можете указать компилятору, что складываемые аргументы являются FIXNUM-значениями, написав функцию следующим образом:

```
(defun add (x y)
  (declare (fixnum x y))
  (+ x y))
```

Выражение `DECLARE` не является формой языка `Common LISP`, это – часть синтаксиса макроса `DEFUN`, где оно должно быть указано до тела функции, если оно используется. Данное объявление объявляет, что аргументы функции `x` и `y` будут всегда `FIXNUM`-значениями. Другими словами, это обещание компилятору, и компилятору разрешено генерировать код в предположении, что все, что вы пообещали ему, будет истинным.

Чтобы объявить тип возвращаемого значения, вы можете обернуть основное выражение функции `(+ x y)` в специальный оператор `THE`. Этот оператор принимает спецификатор типа, такой как `FIXNUM`, и какую-то форму, и говорит компилятору, что эта форма будет возвращать значение указанного типа. Таким образом, чтобы дать компилятору `Common Lisp` всю ту же информацию, которую имеет компилятор языка `C`, вы можете написать следующее:

```
(defun add (x y)
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Однако даже эта версия нуждается в еще одном объявлении, чтоб дать компилятору `Common Lisp` те же разрешения, что есть и у компилятора языка `C`, генерировать быстрый, но опасный код. Объявление `OPTIMIZE` используется для того, чтобы указать компилятору как распределить свое внимание между пятью целями:

- * скоростью генерируемого кода
- * количеством проверок времени выполнения
- * использованием памяти, как в терминах размера кода, так и в терминах размера памяти данных
- * количеством отладочной информации, хранимой вместе с кодом
- * скоростью процесса компиляции.

Объявление `OPTIMIZE` содержит один или более списков, каждый из которых содержит один из символов: `SPEED`, `SAFETY`, `SPACE`, `DEBUG` или `COMPILE-SPEED` и число от нуля до трех включительно. Число указывает относительный вес, который компилятор должен дать соответствующему параметру, причем 3 означает наиболее важное направление, а 0 – что это не имеет значения вообще. Таким образом, чтобы заставить компилятор `Common Lisp` компилировать функцию `add` более-менее так же, как это бы делал компилятор языка `C`, вы можете переписать ее так:

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Конечно, теперь `LISP`-версия функции страдает многими из слабостей `C`-версии: если переданные аргументы не `FIXNUM`-значения или если сложение вызывает переполнение, результаты будут математически некорректны или даже хуже. Также если кто-то вызовет функцию `add` с неправильным количеством параметров, будет мало хорошего. Таким образом, вам следует использовать объявления этого вида только после того, как ваша программа стала работать правильно, и вы должны добавлять их только в местах, где профилирование показывает необходимость этого. Если вы имеете достаточную производительность без этих объявлений, пропускайте их. Но если профайлер показывает вам какое-то проблемное место в вашем коде и вам нужно оптимизировать его – вперёд. Поскольку вы можете использовать объявления таким

образом, редко когда нужно переписывать код на языке С только из соображений производительности. Для доступа к существующему коду на С используется FFI, но когда нужна производительность, подобная языку С, используют объявления. И конечно, то, как близко вы захотите приблизить производительность данной части кода на языке Common Lisp к коду на С или С++, зависит главным образом от вашего желания.

Другое средство оптимизации, встроенное в язык Lisp, – это функция `DISASSEMBLE`. Точное поведение этой функции зависит от реализации, потому что оно зависит от того, как реализация компилирует код: в машинный код, байт-код или какую-то другую форму. Но основная идея в том, что она показывает вам код, сгенерированный компилятором, когда он компилировал данную функцию.

Таким образом, вы можете использовать `DISASSEMBLE`, чтобы увидеть, возымели ли ваши объявления какой-то эффект на генерируемый код. Если ваша реализация языка LISP использует компиляцию в машинный код, и если вы знаете язык ассемблера вашей платформы, вы сможете достаточно хорошо представить себе, что реально происходит, когда вы вызываете одну из ваших функций. Например, вы можете использовать `DISASSEMBLE`, чтобы понять, в чем различия между нашей первой версией `add` и окончательной версией. Сначала определите и скомпилируйте исходную версию

```
(defun add (x y) (+ x y))
```

Затем, в сессии REPL вызовите `DISASSEMBLE` с именем этой функции. В Allegro это выведет следующий ассемблероподобный листинг сгенерированного компилятором кода:

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y
;; code start: #x737496f4:
 0: 55 pushl ebp
 1: 8b ec movl ebp,esp
 3: 56 pushl esi
 4: 83 ec 24 subl esp,$36
 7: 83 f9 02 cmpl ecx,$2
10: 74 02 jz 14
12: cd 61 int $97 ; SYS::TRAP-ARGERR
14: 80 7f cb 00 cmpb [edi-53],$0 ; SYS::C_INTERRUPT-PENDING
18: 74 02 jz 22
20: cd 64 int $100 ; SYS::TRAP-SIGNAL-HIT
22: 8b d8 movl ebx,eax
24: 0b da orl ebx,edx
26: f6 c3 03 testb bl,$3
29: 75 0e jnz 45
31: 8b d8 movl ebx,eax
33: 03 da addl ebx,edx
35: 70 08 jo 45
37: 8b c3 movl eax,ebx
39: f8 clc
40: c9 leave
41: 8b 75 fc movl esi,[ebp-4]
44: c3 ret
45: 8b 5f 8f movl ebx,[edi-113] ; EXCL::+_2OP
48: ff 57 27 call *[edi+39] ; SYS::TRAMP-TWO
51: eb f3 jmp 40
53: 90 nop
; No value
```

Очевидно, что здесь полно всякой всячины. Если вы знакомы с ассемблером процессоров архитектуры x86, вы может быть это поймете. Теперь скомпилируйте версию функции `add` со всеми объявлениями:

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Теперь дизассемблируйте функцию `add` снова, и посмотрите, был ли какой-то толк от этих объявлений.

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y
;; code start: #x7374dc34:
0: 03 c2 addl eax,edx
2: f8 clc
3: 8b 75 fc movl esi,[ebp-4]
6: c3 ret
7: 90 nop
; No value
```

Похоже, что был.

Поставка приложений

Другая практически важная тема, о которой я не говорил нигде в этой книге, – это как поставлять приложения, написанные на языке Lisp. Главная причина, по которой я пренебрегал этой темой, заключается в том, что есть много разных способов это делать, и который из них лучше вам подходит, зависит от того, какой вид программного обеспечения вам нужно распространять, кому его надо распространять, и какой реализацией Common Lisp этот кто-то пользуется. В этом разделе я дам обзор некоторых из возможностей.

Если вы написали код, которым хотите поделиться со своими коллегами, которые тоже пишут на языке LISP, самый простой путь распространения – это распространять исходный код. Вы можете поставлять простую библиотеку как один исходный файл, который программисты могут загрузить в их образ LISP-машины командой `LOAD`, возможно, после компиляции с помощью `COMPILE-FILE`.

Более сложные библиотеки или приложения, разбитые на несколько исходных файлов, ставят дополнительный вопрос: для того, чтобы загрузить сложный код, файлы исходного кода должны быть загружены и откомпилированы в правильном порядке. Например, файл, содержащий определения макросов, должен быть загружен до файлов, которые используют эти макросы, а файл, содержащий инструкцию определения пакета `DEFPACKAGE` должен быть загружен до того, как все файлы, использующие этот пакет, будут просто читаться `READ`-ом. Программисты на языке Lisp называют это проблемой определения систем (*system definition problem*) и обычно разрешают ее с помощью средств, называемых средствами определения систем (*system definition facilities*) или утилитами определения систем (*system definition utilities*), которые являются чем-то вроде аналогов билд-системам, таким, как утилиты `make` и `ant`. Как и эти средства, средства определения систем позволяют указать зависимости между разными файлами и берут на себя заботу о загрузке и компиляции этих файлов в правильном порядке, стараясь при этом выполнять только ту работу, которая необходима – например, перекомпилировать только те файлы, которые изменились.

В наши дни наиболее широко используется система определения систем, называемая ASDF, что означает Another System Definition Facility (Еще одно средство определения систем). Основная идея ASDF в том, что вы описываете вашу систему в специальном файле - ASD, а ASDF предоставляет возможности по выполнению некоторого набора операций с описанными таким образом системами, такие, как загрузка и компиляция систем. Система может быть объявлена зависимой от других систем, которые в таком случае будут корректно загружены при необходимости. Например, вот как выглядит содержимое файла "html.asd", содержащего описание системы для ASDF для библиотеки FOO из глав 31 и 32:

```
(defpackage :com.gigamonkeys.html-system (:use :asdf :cl))
(in-package :com.gigamonkeys.html-system)
(defsystem html
  :name "html"
  :author "Peter Seibel <peter@gigamonkeys.com>"
  :version "0.1"
  :maintainer "Peter Seibel <peter@gigamonkeys.com>"
  :license "BSD"
  :description "HTML and CSS generation from sexps."
  :long-description ""
  :components
  ((:file "packages")
   (:file "html" :depends-on ("packages")))
   (:file "css" :depends-on ("packages" "html")))
  :depends-on (:macro-utilities))
```

Если вы добавите символьную ссылку на этот файл в каталог, указанный в переменной `asdf:*central-registry*`, то затем вы можете набрать следующую команду

```
(asdf:operate 'asdf:load-op :html)
```

чтобы скомпилировать и загрузить файлы "packages.lisp", "html.lisp" и "html-macros.lisp" в правильном порядке после того, как система `:macro-utilities` будет скомпилирована и загружена. Примеры других файлов определения систем ASDF вы можете найти в прилагаемом к книге коде – код из каждой практической главы определён как система с соответствующими межсистемными зависимостями, определенными в формате ASDF.

Большинство свободного ПО и ПО с открытым кодом на языке Common Lisp, которое вы найдёте, будет поставляться с ASD файлом. Некоторые библиотеки могут поставляться с другими системами определения, такими, как немного более старая `МК:DEFSYSTEM` или даже системами, изобретенными авторами этой библиотеки, но общая тенденция, кажется все же, в использовании ASDF.

Конечно, ASDF позволяет программистам легко устанавливать библиотеки, но это никак не помогает поставлять приложения конечным пользователям, которые не имеют представления о языке LISP. Если вы поставляете приложения для конечного пользователя, по-видимому, вы бы хотели предоставить пользователям нечто, что он бы мог загрузить, установить и запустить, не зная ничего о языке Lisp. Вы не можете ожидать, что пользователи будут отдельно устанавливать реализацию Lisp-а и вы бы наверное хотели, чтобы приложения, написанные на языке Lisp запускались так же, как и все прочие приложения в вашей операционной системе – двойным нажатием кнопки мыши на иконке приложения или набором имени приложения в командной строке интерпретатора команд.

Однако, в отличие от программ, написанных на языке C, которые обычно могут рассчитывать на присутствие неких совместно используемых библиотек, которые составляют так называемый "С-рантайм" и присутствуют как часть операционной системы, программы на языке LISP должны

включать ядро языка LISP, то есть ту же часть LISP-системы, которая используется при разработке, хотя и возможно без каких-то модулей, не требуемых во время выполнения программы.

И даже все еще более сложно – понятие "программа" не очень хорошо определено в языке LISP. Как вы видели во время чтения этой книги, процесс разработки программ на языке LISP – это инкрементальный процес, подразумевающий постоянные изменения определений и данных, находящихся внутри исполняемого образа LISP-машины. Поэтому "программа" – это всего лишь определенное состояние этого образа, которое достигнуто в результате загрузки файлов `.lisp` или `.fasl`, которые содержат код, который создает соответствующие определения и данные. Вы могли бы соответственно распространять приложение на языке LISP как рантайм LISP-машины, плюс набор файлов `.fasl` и исполняемый модуль, который запускал бы рантайм LISP-машины, загружал бы все файлы `.fasl` и как-то вызывал бы соответствующую начальную функцию вашего приложения. Однако, поскольку в реальности загрузка файлов `.fasl` может занимать значительное время, особенно если они должны выполнить какие-то вычисления для установки начального состояния данных, большинство реализаций Common LISP предоставляют способ выгрузить исполняемый образ LISP-машины в файл, называемый файл образа (image file) или иногда `core file`, чтобы сохранить все состояние LISP-машины. Когда рантайм LISP-машины запускается, первым делом он загружает файл образа, что у него получается гораздо быстрее, чем если бы то же состояние восстанавливалось с помощью загрузки файлов `.fasl`.

Обычно файл образа – это базовый образ LISP-машины, содержащий только стандартные пакеты, определенные стандартом языка и какие-то дополнительные пакеты, предоставляемые данной реализацией. Но в большинстве реализаций вы можете указать другой, свой файл образа. Таким образом, вместо поставки приложения в виде рантайма LISP-машины и набора файлов `.fasl`, вы можете поставлять рантайм LISP-машины и только один файл образа, содержащий все определения кода и данных, которые составляют ваше приложение. В таком случае вам будет надо только запустить рантайм с этим файлом образа и вызвать функцию, служащую точкой входа в ваше приложение.

Здесь всё становится зависящим от реализации и операционной системы. Некоторые реализации Common Lisp, в особенности коммерческие, как например Allegro или LispWorks, предоставляют средства для создания таких файлов образа. Например, Allegro Enterprise Edition предоставляет функцию `excl:generate-application`, которая создаёт каталог, содержащий рантайм языка Lisp как разделяемую библиотеку, файл образа, и исполняемый файл, который запускает рантайм с данным образом. Похожим образом механизм LispWorks Professional Edition, называемый "поставка" (delivery), позволяет вам строить однофайловый исполняемый файл из ваших программ. В различных реализациях Unix вы можете делать фактически то же самое, хотя возможно там проще использовать командный файл для запуска всего, что угодно.

В Mac OS X всё ещё лучше: поскольку все приложения в этой ОС упакованы в `.app`-пакеты, которые по сути являются каталогами с определённой структурой, очень легко упаковать все части приложения на LISP в `.app`-пакет, который можно запустить двойным нажатием клавиши мыши. Утилита Bosco Микеля Эвинса (Mikel Evins) делает создание `.app`-пакетов простым для приложений, работающих на OpenMCL.

Конечно, в наши дни другой популярный способ поставки приложений – это серверные приложения. Это – та ниша, где Common Lisp может превосходно использоваться: вы можете выбрать сочетание операционной системы и реализации Common Lisp, которая вас устраивает, и вам не нужно заботиться о поставке приложения в том виде, в котором его мог бы установить пользователь. А возможность языка Common Lisp интерактивно разрабатывать и отлаживать программы позволяет вам отлаживаться и обновлять версии вашей программы без остановки сервера, что либо вообще было бы невозможно в менее динамичных языках программирования,

либо требовало бы построения довольно сложной инфраструктуры поддержки.

Что дальше

Ну вот и все. Добро пожаловать в чудесный мир языка LISP. Лучшее, что вы можете сейчас сделать (если вы уже это не сделали) – это начать писать свои собственные программы на языке LISP. Выберите проект, который вам интересен, и сделайте его в Common LISP. Потом сделайте еще один. Как говорить, намылить, прополоскать, повторить ...

Однако, если вам нужна дальнейшая помощь, этот раздел даст вам несколько мест, где ее можно найти. Для начинающих полезно посмотреть сайт этой книги "Practical Common Lisp Web site" по ссылке <http://www.gigamonkeys.com/book>, где вы найдете исходный код из практических глав книги, список опечаток, и ссылки на другие ресурсы в сети Интернет.

В добавок к сайтам, которые я упомянул в разделе "Поиск библиотек LISP", вы возможно заходите изучить Common Lisp HyperSpec (известную также как HyperSpec или CLHS), которая является HTML-версией ANSI-стандарта языка, подготовленной Кентом Питманом (Kent Pitman) и сделанной общедоступной компанией LispWorks по ссылке <http://www.lisp-works.com/documentation/HyperSpec/index.html>. HyperSpec ни в коем случае не является учебником, но это – самое надёжное руководство по языку, которое вы только можете достать, не покупая печатную версию стандарта языка у комитета ANSI, и при том гораздо более удобное в повседневном использовании.

Если вы хотите связаться с другими лисперами, Usenet-группа (группа новостей) "comp.lang.lisp" и IRC-канал "#lisp" в чат-сети Freenode (<http://www.freenode.net>) – вот два основных собрания лисперов в сети. Существует также некоторое количество блогов, связанных с языком LISP, большее количество которых собрано на сайте "Planet Lisp" ("Планета ЛИСП") по адресу <http://planet.lisp.org>.

И не пропускайте также во всех этих форумах анонсы собраний местных групп пользователей языка ЛИСП – в последние несколько лет собрания лисперов возникают спонтанно во всех городах мира, от Нью-Йорка до Окленда, от Кёльна до Мюнхена, от Женева до Хельсинки.

Если же вы хотите продолжить изучение книг, вот вам несколько советов. В качестве хорошего толстого справочника можете держать на вашем столе книгу "The ANSI Common Lisp Reference Book" под редакцией Дэвида Марголиса (David Margolies)(издательство Apress, 2005 г.).

Для детального изучения объектной системы языка LISP вы можете для начала прочитать книгу Сони Кин (Sonya E. Keene) "Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS" ("Объектно-ориентированное программирование в Common LISP: Руководство программиста по CLOS.") (издательство Addison-Wesley, 1989 г.). Затем, если вы хотите стать настоящим мастером, или просто чтобы расширить кругозор, прочитайте книгу авторов Грегора Кикзалеса, Джима де Ривьереса и Даниэля Боброва (Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow) "The Art of the Metaobject Protocol" (издательство MIT Press, 1991 г.). Эта книга, также известная как АМОП, является как объяснением, что такое метаобъектный протокол и зачем он нужен, так и фактически стандартом метаобъектного протокола, поддерживаемого многими реализациями языка Common Lisp.

Две книги, которые охватывают общие приёмы программирования на Common LISP, – это "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp" Питера Норвига (Peter Norvig) (издательство Morgan Kaufmann, 1992 г) и "On Lisp: Advanced Techniques for Common Lisp" Пола Грэма (Paul Graham) (издательство Prentice Hall, 1994 г.). Первая даёт твёрдую основу приёмов искусственного интеллекта, и в то же время учит немного, как писать хороший код на Common Lisp. Вторая особенно хороша в рассмотрении макросов.

Если вы – человек, который любит знать, как всё работает до последнего винтика, книга Кристиана Квене (Christian Queinnec) "Lisp in Small Pieces" (издательство Cambridge University Press, 1996 г) сможет дать вам удачное сочетание теории языков программирования и практических методик реализации языка LISP. Несмотря на то, что она изначально сконцентрирована на реализации языка Scheme, а не Common Lisp, принципы, изложенные в книге, применимы и для него.

Для людей, которые хотят более теоретического взгляда на вещи, или для тех, кто просто хочет попробовать, каково это – быть новичком-студентом компьютерных наук в Массачусеттском технологическом институте – следующая книга авторов Харольда Абельсона, Геральда Джея Сусмана и Джули Сусман (Harold Abelson, Gerald Jay Sussman, and Julie Sussman) – "Structure and Interpretation of Computer Programs, Second Edition" ("Структура и интерпретация компьютерных программ") (издательство M.I.T. Press, 1996 г), классическая работа по компьютерным наукам, которая использует язык Scheme для обучения важным концепциям программирования. Любой программист может много узнать из этой книги, не забывая только, что у языков Scheme и Common Lisp есть важные отличия.

Как только вы охватите своим умом язык Lisp, вам возможно захочется добавить чего-то объектного. Поскольку никто не может заявлять, что он действительно понимает объекты без знания хотя-бы чего-то о языке Smalltalk, вы, возможно, захотите начать знакомство с этим языком с книги Адели Голдберг и Дэвида Робсона (Adele Goldberg, David Robson) "Smalltalk-80: The Language" (издательство Addison Wesley, 1989 г), которая является стандартным введением в язык Smalltalk. После этого – книга Кента Бека (Kent Beck) "Smalltalk Best Practice Patterns" (издательство Prentice Hall, 1997), которая полна хороших советов для программистов на этом языке, большинство из которых также применимо и к любому другому объектно-ориентированному языку.

На другом конце спектра – книга Бертрана Мейера (Bertrand Meyer), изобретателя часто незамечаемого наследника Симулы и Алгола – языка Eiffel, "Object-Oriented Software Construction" (Prentice Hall, 1997), прекрасная демонстрация склада ума людей, мыслящих в стиле статических языков программирования. Эта книга содержит много пищи для размышлений даже для программистов, работающих с динамическими языками, каковым является язык Common Lisp. В частности, идеи Мейера о контрактном программировании могут помочь понять, как нужно использовать систему условий языка Common Lisp.

Хотя и не о компьютерах как таковых, книга Джеймса Суrowьеки "Мудрость масс: почему те, кого много, умнее тех, кого мало, и как коллективная мудрость формирует бизнес, экономику, общество и нации". ("The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies, and Nations", James Surowiecki, Doubleday, 2004) содержит великолепный ответ на вопрос: "почему если LISP такой замечательный, его все не используют?" Смотрите раздел "Лихорадка досчатой дороги", начиная со страницы 53.

И в заключение для удовольствия и чтобы понять, какое влияние LISP и лисперы оказали на развитие культуры хакеров, просмотрите или прочитайте от корки до корки "Новый словарь хакеров" Эрика реймонда ("The New Hacker's Dictionary, Third Edition, compiled by Eric S. Raymond, MIT Press, 1996) которое базируется на исходном словаре хакеров, редактируемом Гаем Стиллом (Harper & Row, 1983).

Но не давайте всем этим книгам мешать вашему программированию, потому что единственный путь изучить язык программирования – это использовать его. Если вы дошли до этого места, вы безусловно в состоянии заняться этим. Счастливого покодить!