Предисловие

Эта книга предназначена для каждого, кто хочет улучшить свои навыки программирования на Лисп. Подразумевается наличие некоторого знакомства читателя с Лисп, но наличие большого опыта программирования не требуется. Первые несколько глав содержат развернутое введение. Надеюсь, что они также будут интересны и более опытным программистам на Лисп, поскольку представляют знакомые темы в новом свете.

Довольно трудно выразить сущность языка программирования в одном предложении, но Джон Фодераро подобрался к этому вплотную:

> Лисп – это программируемый язык программирования.

На самом деле Лисп это много большее, но способность подстроить под себя Лисп в значительной степени отличает профессионала от новичка. Опытные программисты на Лисп не только приспосабливают программы к языку, но и создают язык для нужд своих программ. Эта книга учит, как программировать в восходящем стиле, для которого Лисп, по сути, хорошо подходит.

Восходящее проектирование

По мере возрастания сложности программного обеспечения восходящее проектирование становится все более значимым. Сегодня программы могут иметь чрезвычайно сложные или незавершенные спецификации. При подобных обстоятельствах традиционное нисходящее проектирование иногда дает сбой. В этом случае как раз и применяется стиль программирования, совершенно отличающийся от того, что сейчас изучают в большинстве курсов программирования: стиль <<снизу-вверх>>, согласно которому программы пишутся в виде последовательности слоев, каждый из которых выступает в роли своего рода языка программирования для вышестоящего слоя. Например, X Windows и TeX – программы, написанные в этом стиле.

Основными темами книги являются следующие две: то, что Лисп – это естественный язык для программ, написанных в восходящем стиле, а также то, что восходящий стиль является естественным для написания программ на Лисп. Поэтому, книга <<К вопросу о Лисп>> будет интересна следующим двум категориям читателей. Тем, кто заинтересован в написании расширяемых програм, эта книга покажет, что вы сможете сделать, используя правильный язык. Лисп-программистам книга даст практическое объяснение того, как использовать Лисп с наибольшей отдачей.

Название раздела подчеркивает важность метода программирования снизу-вверх в языке Лисп. Вместо того, чтобы просто написать свою программу на Лисп, можно реализовать собственный язык на Лисп, и написать на нем свою программу.

Писать программы в восходящем стиле можно при помощи любого языка, но Лисп является наиболее естественной движущей силой для такого стиля программированя. В Лисп восходящее проектирование не является какой-то особой техникой, применяемой исключительно для больших или сложных задач. Любая достаточно солидная программа будет отчасти написана в таком стиле. Лисп с самого начала был задуман как расширяемый язык. Сам по себе он, в большой степени, просто коллекция Лисп-функций, мало отличающихся от тех что вы определяете сами. И что более важно - функции Лисп можно рассматривать как списки, которые в свою очередь являются

структурами данных Лисп. А это значит что можно писать Лисп-функции, которые генерируют код на Лисп.

Хороший программист на Лисп должен понимать, как извлечь выгоду из этой возможности. Для этого обычно определяют специального вида операторы, называемые макросами. Овладение макросами – одна из важнейших ступеней на пути от написания корректных программ на Лисп до создания прекрасных программ. Обычно в книгах, знакомящих читателя с Лисп, хватает места не более чем на небольшой обзор макросов. Читателю лишь объясняют, что такое макросы, и показывают пару примеров, намекающих на странные и удивительные вещи, к которым они открывают доступ. Этим странным и удивительным вещам уделяется особое внимание в этой книге. Одна из её целей состоит в том, чтобы собрать вместе все знания и опыт по работе с макросами.

Разумеется, вводные книги о Лисп не уделяют должного внимания отличию Лисп от других языков. Их задача – донести знание до студентов, которые, в большинстве своем, привыкли думать о программировании в терминах языка Паскаль. Такой подход больше запутывает, чем объясняет: хотя defun и похож на объявление процедуры, он представляет собой программу для создания программ, генерирующую код, который построит функциональный объект и проиндексирует его под символом, переданным в качестве первого аргумента.

Объяснить отличия Лисп от других языков – так же одна из целей этой книги. В начале было известно то, что при прочих равных, я бы стал писать программы на Лисп нежели на Си, Паскале или Фортране. Так же было известно, что это не только и не столько дело вкуса. Но так же было ясно, что если я и правда соберусь сказать что Лисп это лучший язык в том или ином роде, то лучше быть готовым объяснить почему.

Когда кто-то спросил Луи Армстронга о том, что такое джаз, он ответил: «Если вы спрашиваете, что такое джаз, то этого никогда не узнаете». Но он дал ответ иным способом: он показал людям, что такое джаз. Один из способов объяснить сильные стороны Лисп – это показать техники, которые будет сложно или невозможно реализовать в других языках. Большинство книг по программированию, даже книги по программированию на Лисп, рассматривают класс таких программ, которые могут быть написаны на любом языке. В книге <<К вопросу о Лисп>> рассматриваются программы, которые можно написать только на Лисп. Расширяемость, восходящий стиль программирования, интерактивная разработка, преобразование исходного кода, встраиваемые языки – это области, где Лисп показывает свое преимущество.

Конечно, в принципе любой Тьюринг-эквивалентный язык программирования способен делать то же самое, что и любой другой. Но это – не главная особенность языков программирования. Теоретически, все, что возможно сделать с помощью языка программирования, возможно сделать также и с помощью машины Тьюринга; на практике-же, программирование машины Тьюринга не стоит затраченных усилий.

Итак, говоря, что эта книга о том как сделать то, что невозможно в других языках, я не имею в виду «невозможно» в математическом смысле, а в том который имеет значение для языков программирования. То есть, если потребуется написать несколько программ из этой книги на C, можно – было бы это сделать, написав сначала компилятор Лисп. Например, встраивание Пролог в Си – вы можете представить

себе возможный объем работы? Глава 24 показывает, как сделать это в 180 строк на Лисп.

Тем не менее, была надежда сделать больше, чем просто показать силу Лисп. Мне также хотелось объяснить, почему Лисп иной. Это оказалось довольно сложным вопросом – слишком сложным, чтобы можно было ответить фразой вроде «символьные вычисления». Все, что мне было известно, я попытался изложить настолько ясно, насколько смог.

План книги

Так как функции – это основа Лисп-программ, книга начинается с нескольких глав посвящённых функциям. Глава 2 объясняет, что такое функции в Лисп, а также раскрывает предоставляемые ими возможности. Затем, в главе 3 рассматриваются преимущества функционального программирования – основного стиля программ на Лисп. Глава 4 показывает как использовать функции для расширения Лисп. Далее, в главе 5 представлены новые виды абстракций, которые могут быть определены при помощи функций, возвращающих другие функции. И, наконец, глава 6 показывает, как использовать функции вместо традиционных структур данных.

Оставшаяся часть книги более посвящена макросам, нежели функциям. Макросам уделяется больше внимания отчасти потому, что о них можно говорить больше, и отчасти потому, что до настоящего момента им не уделялось достаточно внимания в печати. Главы 7-10 представляют собой полное учебное пособие по технике макросов. К концу вы узнаете большую часть того, что опытный Лисп-программист знает о макросах: как они работают; как их определять, тестировать и отлаживать; когда использовать макросы, а когда – нет; основные типы макросов; как писать программы, генерирующие макрорасширения; основные отличия стиля макросов от стиля Лисп; а также, как диагностировать и исправить любую проблему из тех, которым подвержены макросы.

Главы 11-18, следующие за этим учебным пособием, показывают кое-что из сложных абстракций, которые можно создать, применяя макросы. Глава 11 показывает, как написать классический макрос – тот, который создает контекст, реализует циклы или условия. Глава 12 объясняет роль макросов в операциях с обобщенными переменными. Глава 13 показывает, как макросы позволяют программам работать быстрее за счет переноса вычислений на время компиляции. Глава 14 знакомит с анаморфными макросами, которые позволяют использовать местоимения в программах. Глава 15 демонстрирует, как предоставить более удобный интерфейс функциям-конструкторам, определенным в главе 5. Глава 16 показывает, как заставить Лисп писать программы, используя макроопределяемые макросы. Глава 17 рассматривает макросы чтения, а глава 18 — макросы-деструкторы.

Глава 19 открывает четвертую часть книги, посвященную встраиваемым языкам. Она знакомит с темой, реализуя одну и ту же программу для ответа на запросы к базе данных, сначала как интерпретатор, а затем как полноценный встраиваемый язык. Глава 20 демонстрирует, как включить в программы на Коммон Лисп механизм «продолжений», представления объектов, как остатка вычислений. Механизм продолжений – очень мощный иструмент, который можно использовать как для реализации многопроцессности, так и для недетерминированного выбора. Встраивания этих управляющих структур в Лисп рассматриваются в главах 21 и 22 соответственно.

Недетерминизм, который позволяет писать програмы так, будто они способны предвидеть, звучит как демонстрация невиданной силы. Главы 23 и 24 представляют два встраиваемых языка, показывающих что недетрменизм отлично себя оправдывает: полный синтаксический анализатор ATN (augmented transition network - расширенная сеть переходов) и встраиваемый Пролог, включающие в себя общей сложностью около 200 строчек кода.

Глава 19 открывает четвертую часть книги, посвященную встраиваемым языкам. Она знакомит с темой, реализуя одну и ту же программу для ответа на запросы к базе данных, сначала как интерпретатор, а затем как полноценный встраиваемый язык. Глава 20 демонстрирует, как включить в программы на Коммон Лисп механизм «продолжений», представления объектов, как остатка вычислений. Механизм продолжений – очень мощный иструмент, который можно использовать как для реализации многопроцессности, так и для недетерминированного выбора. Встраивания этих управляющих структур в Лисп рассматриваются в главах 21 и 22 соответственно. Недетерминизм, который позволяет писать програмы так, будто они способны предвидеть, звучит как демонстрация невиданной силы. Главы 23 и 24 представляют два встраиваемых языка, показывающих что недетрменизм отлично себя оправдывает: полный синтаксический анализатор ATN (augmented transition network - расширенная сеть переходов) и встраиваемый Пролог, включающие в себя общей сложностью около 200 строчек кода.

Книга завершается рассмотрением объектно-ориентированного программирования, в частности CLOS (Common Lisp Object System - объектная система Коммон Лисп). Оставив эту тему напоследок, мы увидим более ясно, что объектно-ориентированное программирование является расширением идей, уже представленных в Лисп. Это одна из множества абстракций, которые могут быть реализованы поверх Лисп.

Примечания к главам начинаются на странице 387. Примечания содержат ссылки, дополнительный или альтернативный код, или описания особенностей лисп, напрямую не связанные с предметом обсуждения. Примечания выделены маленьким кружком на полях, как этот. Там также есть приложение (страница 381) о модулях.

Точно так же, как экскурсия по Нью-Йорку может быть экскурсией по большинству мировых культур, так и изучение Лисп, как программируемого языка программирования, охватывает большую часть технических приемов Лисп. Большая часть приемов, описанных здесь, общеизвестна Лисп-сообществу, но многие до настоящего момента так и не были нигде опубликованы. А некоторые вопросы, такие как роль макросов, или сущность захвата переменных, лишь смутно понятны даже многим опытным Лисп-программистам.

Примеры

Лисп – это семейство языков. Так как Коммон Лисп продолжает оставаться широко используемым диалектом, большинство примеров в этой книге реализованы на нем. Язык изначально был сформулирован 1984 в публикации Гая Стила (Guy Steel) \Common Lisp: the Language" (CLTL1). Это описание было заменено в 1990 году публикацией нового издания (CLTL2), которое в свою очередь уступит место грядущему стандарту ANSI.

Эта книга содержит сотни примеров, начиная небольшими выражениями, и заканчивая работающей реализацией Пролога. Код в этой книге везде, где это возможно,

писался для работы под любой версией Коммон Лисп. Те несколько примеров, которым требуются функциональность, не содержащаяся в реализациях CLTL1, явно выделены в тексте. Последние главы содержат несколько примеров на языке Схема. Они также явно выделены.

Код доступен через анонимный FTP с сервера `endor . harvard . edu`, в директории pub/onlisp. Вопросы и комментарии можно отправлять по адресу `onlisp@das.harvard.edu`.

Благодарности

Я особенно благодарен Роберту Моррису (Robert Morris) за оказанную помощь во время работы над этой книгой. Я обращался к нему постоянно за советами, и всегда был этому рад. Несколько примеров в этой книге заимствованы из кода, изначально написанного им, включая версию for на странице 127, версию aand на странице 191, match на странице 239, true-choose алгоритма поиска в ширину на странице 304, и интерпретатор Prolog в разделе 24.2. На самом деле, вся книга отражает (а временами даже пересказывает) мои беседы с Робертом за последние семь лет. (Спасибо, читайте руководство FIXME!)

Хотелось бы отдельно выразить благодарность Дэвиду Муну (David Moon), он очень внимательно прочитал большие части рукописи и дал очень ценные комментарии. Глава 12 была полностью переписана по его совету, а также пример захвата переменной на странице 199 был предоставлен им.

Мне повезло, что техническими рецинзентами книги были Дэвид Турецки (David Touretzky) и Скона Бриттен (Skona Brittain). Несколько разделов были добавлены или переписаны по их совету. Альтернативный настоящий недетерминированный оператор выбора на странице 397 основан на совете, данным Дэвидом Турецки.

Еще нескольких человек, кто согласился прочитать всю рукопись целиком или ее часть, включая Тома Читхэма (Tom Cheatham), Ричарда Дрэйвса (Richard Draves) (он также переписал alambda и propmacro в 1985 FIXME), Джона Фодераро (John Foderaro), Дэвида Хендлера (David Hendler), Джорджа Люгера (George Luger), Роберта Мюллера (Robert Muller), Марка Ницберга (Mark Nitzberg) и Гая Стила (Guy Steele).

Я признателен профессору Читхэму, и Гарвардскому университету вообще, за предоставленные средства, использованные при написании этой книги. Также выражаю благодарность сотрудникам лаборатории Эйкена, включая Тони Хартмана (Tony Hartman), Януша Джуду (Janusz Juda), Гарри Вочнера (Harry Bochner) и Джоанн Клес (Joanne Klys).

Все люди в Prentice Hall проделали отличную работу. Мне посчастливилось работать с Аланом Аптом (Alan Apt), отличным редактором и отличным парнем. Благодарю также Мону Помпили (Mona Pompili), Ширли Майклс (Shirley Michaels) и Ширли Макгир (Shirley McGuire) за их организованность и хороший юмор.

Несравненного Джино Ли (Gino Lee) из издательства Bow and Arrow, Кембридж, сделавшего обложку. Дерево на обложке упоминается, в частности, на странице 27.

Эта книга была набрана с использованием LATEX, языка, написанного Лесли Лэмпортом (Leslie Lamport) поверх TEX Дональда Кнута (Donald Knuth) с дополнительными макросами Л. А. Карра (L. A. Carr), Вана Джэкобсона (Van Jacobson) и Гая

Стила (Guy Steele). Диаграммы были выполнены при помощи Idraw Джона Влиссидеса (John Vlissides) и Скотта Стэнтона (Scott Stanton). Текст целиком просматривался с помощью Ghostview Тима Тэйсена (Tim Theisen), который основан на Ghostscript Л. Питера Дойча (L. Peter Deutsch). Гэри Бисби (Gary Bisbee) из Chiron Inc., изготовившего оригинал-макет.

Я также в долгу у многих других, включая Пола Беккера (Paul Becker), Фила Чапника (Phil Chapnick), Элис Хартли (Alice Hartley), Гленна Холловея (Glenn Holloway), Мейчун Хсу (Meichun Hsu), Крзыштова Ленка (Krzysztof Lenk), Армана Магболе (Arman Maghbouleh), Говарда Муллингса (Howard Mullings), Нэнси Пармет (Nancy Parmet), Роберта Пенни (Robert Penny), Гэри Сабота(Gary Sabot), Патрика Слэйни (Patrick Slaney), Стива Страссмана (Steve Strassman), Дэйва Уоткинса (Dave Watkins), Вейкеров (Weickers), и Билла Вудса (Bill Woods).

Больше всего благодарю моих родителей за их пример и поддержку, и Джеки (Jackie), научившей меня тому, что я должен был бы усвоить, слушая их.

Надеюсь, что чтение этой книги будет развлечением. Из всех языков, которые я знаю, мне больше всего нравится Lisp просто потому, что он наиболее красивый. Эта книга о Lisp, и она сама лисповая. Мне было очень радостно ее писать, и я надеюсь, что это пройдет сквозь текст.

Пол Грэм(Paul Graham)

1 1 Расширяемый язык (The Extensible Language)

Не так давно, если бы вы спросили, для чего нужен Лисп, то многие бы ответили «для искусственного интеллекта». На самом деле, связь между Лисп и ИИ – просто историческая случайность. Лисп был изобретен Джоном Маккарти (John McCarthy), который также придумал термин «искусственный интеллект». Его студенты и коллеги писали свои программы на Лисп, поэтому о Лисп стали говорить как о языке для задач ИИ. Во времена бума ИИ в 1980-х эта идея обсуждалась и повторялась настолько часто, что стала практически непременным атрибутом.

К счастью, стали ходить разговоры о том, что ИИ – это не единственная область применения Лисп. Недавние успехи в области аппаратного и программного обеспечения сделали Лисп коммерчески жизнеспособным: сейчас он используется в Gnu Emacs – лучшем текстовом редакторе для Unix; Autocad – индустриальном стандарте среди САПР для настольных компьютеров и Interleaf – распространенной профессиональной издательской программе. Использование Лисп в этих программах совершенно никак не связано с ИИ.

Если Lisp не язык ИИ, тогда что это? Вместо того, чтобы судить о Лисп по использующим его компаниям, давайте посмотрим на сам язык. Что вы можете сделать в Лисп и не сможете в других языках? Одна из наиболее отличительных черт Лисп – гибкость – позволяет ему лучше подстраиваться под задачу FIXME. Сам по себе Лисп является программой на Лисп, а программы на Лисп могут быть выражены в виде списков, которые в свою очередь в Лисп являются структурами данных. Вместе эти два принципа означают то, что любой пользователь может добавить операторы в Лисп так, что они не будут отличаться от встроенных в язык.

1.1 1-1 Эволюция дизайна

Поскольку Лисп дает свободу в определении операторов, вы можете преобразовать его в тот язык, который вам нужен. Если вы пишете текстовый редактор, вы можете переделать Лисп в язык для написания текстовых редакторов. Если вы пишете САПР, то можете превратить Лисп в язык для написания САПР. И даже если вы пока не уверены насчет того, какую программу вы пишете, то беспроигрышный вариант – написать ее на Лисп. В какой бы вид программы она ни превратилась, Лисп в процессе ее написания будет эволюционировать в язык, предназначенный для написания такого вида программ.

Вы все еще не уверены, какого рода программу пишете? Для кого-то этот вопрос прозвучит странно. Такой разительный контраст с известной моделью разработки, где вы (1) тщательно планируете то, что собираетесь делать, а затем (2) делаете это. Если Лисп вдохновляет вас на написание программы до того, как вы определите способ ее работы, то согласно этой модели, такой подход лишь порождает хаотичное мышление.

Ну, это совсем не так. Метод «спланируй и реализуй», возможно, был хорош для строительства плотин или совершения военных вторжений, но опыт не показал, что это хороший способ написания программ. Почему? Возможно, потому что он слишком точный. Может быть, между программами больше различий, чем между дамбами или вторжениями. Или, наверное, старые методы не работают, поскольку старая идея избыточности не имеет аналогов в области разработки программного обеспечения:

если дамба содержит бетона на 30% выше нормы, то это граница допустимого предела, но если программа делает на 30% больше работы, чем нужно, то это недопустимо.

Сложно сказать наверняка, почему старые методы не работают, но любой может видеть, что именно это и происходит. Когда программы сдавались вовремя? Опытные программисты знают, что вне зависимости от того, насколько тщательно вы планировали программу, в процессе ее написания планы в какой-то степени нарушатся. Иногда планы становятся безнадежно неверными. Но лишь немногие жертвы метода «спланируй и реализуй» ставят под сомнение его обоснованность. Вместо этого они начинают винить человеческие недостатки: если бы планы были более дальновидными, то всех этих неприятностей можно было бы избежать. Поскольку даже лучшие программисты, переходя к реализации, сталкиваются с проблемами, возможно, было бы слишком надеяться на то, что люди когда-нибудь станут такими прозорливыми. Вероятно, метод «спланируй и реализуй» может быть заменен другим подходом, более подходящим нашим критериям.

Если у нас есть правильные инструменты, то мы можем подойти к программированию по-другому. Для чего планировать перед реализацией? Большая опасность в чрезмерном планировании состоит в возможности быть загнанным в угол. Мы делаем, вот и все. Гибкость Лисп породила полностью новый стиль программирования. В Лисп многое можно спланировать в процессе написания программы.

Зачем ждать оценок прошлого? Как заметил Монтень, ничто так не проясняет наши идеи, как попытка записать их. Однажды вы освободитесь от беспокойства, которе загоняет вас в угол; вы сможете извлечь всю выгоду из этой возможности. Способность планировать программы по мере их написания имеет два важных последствия: на написание программ уходит меньше времени, потому что, когда вы пишете и планируете одновременно, у вас есть настоящая программа, чтобы сосредоточить на ней свое внимание; они становятся лучше, потому что окончательная разработка – это всегда продукт эволюции. Таким образом в процессе написания программы вы постоянно переписываете ошибочные части, как только становится очевидна их несостоятельность, в итоге конечный продукт будет более элегантным решением, чем если бы вы потратили недели, планируя его заранее.

Приспособляемость Лисп делает этот вид программирования практической альтернативой. Действительно, самая большая опасность Лисп состоит в том, что он может избаловать вас. Один раз попробовав Лисп, вы можете стать настолько чувствительным к соответствию между языком и приложением, что будете не в состоянии вернуться назад к другому языку без постоянного ощущения того, что он не дает в полной мере ту гибкость, которая вам необходима.

1.2 1-2 Программирование снизу-вверх

Давно известное правило методологии программирования состоит в том, что функциональные элементы программы не должны быть слишком большими. Если какой-то из элементов программы вырастает настолько, что перестает быть легким для понимания, то он превращается в запутанный клубок, скрывающий ошибки так же легко, как большой город скрывает беглецов. Такое программное обеспечение будет трудно читать, трудно проверять и трудно отлаживать.

В соответствии с этим правилом большая программа должна быть разбита на части, и чем больше программа, тем больше частей должно быть. Как же разбить программу? Традиционный подход называется нисходящим проектированием. Вы скажете: "Цель программы состоит в том, чтобы сделать семь таких-то вещей, значит, я разбиваю программу на семь главных подпрограмм. Первая подпрограмма должна сделать четыре таких-то вещи, поэтому у нее, в свою очередь, будет четыре собственные подпрограммы, и так далее". Этот процесс продолжается до тех пор, пока вся программа не достигнет надлежащего уровня модульности – каждая часть достаточно велика, чтобы сделать что-то важное, но и достаточно мала для того, чтобы представлять собой отдельный модуль.

Опытные Лисп-программисты разбивают свои программы по-другому. Так же, как и в нисходящем проектировании, они следуют принципу, который можно было бы назвать восходящим проектированием – изменением языка для решения задачи. В Лисп вы не просто подгоняете свою программу к языку, вы также создаете язык для нее. По мере написания программы вам может прийти в голову: "Мне бы хотелось, чтобы в Лисп был такой-то оператор". Тогда вы идете и пишете его. Позже вы понимаете, что использование нового оператора упростило бы разработку другой части программы и так далее. Язык и программа развиваются вместе. Как граница между двумя враждующими государствами, граница между языком и программой чертится и перечерчивается, пока в конечном счете черта не остается вдоль гор и рек – естественных границ вашей задачи. В итоге ваша программа будет выглядеть так, как будто язык был разработан специально для нее. И когда язык и программа будут хорошо соответствовать друг другу, вы получите ясный, маленький и эффективный код.

Стоит подчеркнуть, что восходящее проектирование не означает всего лишь написание той же программы в другом порядке. Когда вы работаете в восходящем стиле, то в итоге, как правило, получите другую программу. Вместо единственной, монолитной программы вы получите более объемный язык с более абстрактными операторами и более короткую программу, написанную на нем. Вместо перемычки у вас будет арка.

В типичном коде, как только вы обобщаете части, являющиеся просто бухгалтерией, то что остается намного короче FIXME; чем выше вы создаете язык, тем меньше расстояния вы должны будете проходить сверху вниз. Это дает несколько преимуществ:

1. Заставляя язык быть производительнее, восходящее проектирование помогает создать более шустрые и менее объемные программы. Небольшую программу ни к чему разбивать на слишком большое число компонент, ведь меньшее число компонент означает, что программы легче прочитать или изменить. Также, чем меньше число компонент, тем меньше связей между ними и, таким образом, меньше шанс на ошибки в этих местах. Как промышленные проектировщики стремятся сократить количество движущихся частей в машине,так и опытные программисты Лисп используют восходящее проектирование с целью уменьшения размера и сложности своих программ.
2. Восходящая разработка способствует многократному использованию кода. Когда вы пишете две или более программы, многие из утилит, написанных вами для первой программы, пригодятся и для последующих. Собрав однажды большое

количество утилит, на создание новой программы вы потратите лишь часть тех усилий, которые бы потребовались, начни вы писать на Лисп с нуля.

3. Восходящая разработка делает программы более легкими для чтения. В случае абстракции типа экземпляра требуется, чтобы читатель понял оператор общего назначения; в случае функциональной абстракции нужно, чтобы читатель понял подпрограмму специального назначения.[1]
4. Заставляя вас всегда быть в поисках шаблонов в вашем коде, восходящее проектирование помогает прояснить ваши идеи о модели программы. Если два отдаленных компонента программы похожи по форме, то вы первым заметите сходство и, возможно, перепроектируете программу более простым способом.

Восходящее проектирование на языках, отличных от Лисп, осуществимо лишь до определенной степени. Всякий раз, когда вы видите библиотечные функции, происходит восходящее проектирование. Однако Лисп дает вам намного более широкие возможности в этом случае, и расширяемый язык играет соответственно большую роль в технике программирования на Лисп, настолько большую, что Лисп – это не просто другой язык, а совершенно иной способ программирования.

Это правда, что этот способ разработки лучше подходит для программ, которые могут быть написаны небольшими группами. Однако, в то же самое время, он расширяет пределы того, что может быть сделано небольшой группой. В книге "Мифический человеко-месяц" Фредерик Брукс (Frederick Brooks) предположил, что производительность группы программистов не растет линейно с ее размером. С увеличением размера группы понижается производительность отдельных программистов. Опыт программирования на Лисп предлагает более жизнерадостную формулировку этого закона: с уменьшением размера группы производительность индивидуальных программистов повышается. Небольшая группа побеждает, собственно говоря, просто потому, что она меньше. Когда небольшая группа воспользуется еще и методиками, которые делает доступными Лисп, то она может победить вчистую.

1.3 1-3 Расширяемое программное обеспечение

Именно техника программирования на Лисп становиться более важной по мере роста сложности программного обеспечения. Искушенные пользователи теперь требуют так много от ПО, что мы не имеем возможности предугадывать все их потребности. Они и сами не могут их предугадать. Но если мы не можем дать им софт, который делает всё, что они хотят, сразу из коробки, то мы можем дать им расширяемое программное обеспечение. Мы преобразуем наше ПО из обычных программ в язык программирования, и продвинутые пользователи могут надстраивать поверх него необходимые им дополнительные функции.

Восходящая разработка естественным образом ведет к расширяемым программам. Простейшие восходящие программы состоят из двух уровней: язык и программа. Сложные программы могут быть написаны как последовательность уровней, каждый из которых выступает в качестве языка для уровня лежащего над ним. Если эту философию пронести через весь процесс до самого верхнего уровня, то тот уровень и

[1] "Но никто не может читать программу без понимания всех ваших новых утилит." Чтобы понять, почему такие утверждения обычно ошибочны, см Раздел 4-8.

станет программным языком для пользователя. Такая программа, где расширяемость проходит сквозь каждый уровень, вероятно, создает более хороший язык программирования, нежели системы, которые были написаны как традиционный черный ящик, и лишь с запозданием сделаны расширяемыми.

X Windows и TEX – примеры первых программ, основанных на этом принципе.

В 1980-е более хорошее оборудование сделало возможным новое поколение программ, которые использовали Лисп в качестве собственного языка расширений. Первым был Gnu Emacs – популярный текстовый редактор Unix. Позже появился Autocad – первый крупномасштабный коммерческий продукт, давший возможность использовать Лисп как расширяемый язык. В 1991 году Interleaf представила новую версию своего продукта, который не только использовал Лисп как расширяемый язык, но и был в значительной степени на нем же и реализован.

Лисп – особенно хороший язык для написания расширяемых программ, потому что он сам расширяем. Если вы напишете свою программу на Лисп так, чтобы передать эту расширяемость пользователю, вы фактически получите расширяемый язык бесплатно. Различие между расширением Лисп-программы в Лисп и тем же самым, но на традиционном языке, равносильно различию между личной встречей с кем-нибудь и почтовой перепиской. В программе, которая сделана расширяемой просто за счет предоставления доступа к внешним программам, лучшее, на что мы можем рассчитывать – это два черных ящика, соединенных один с другим при помощи какого-нибудь заранее определенного протокола. В Лисп же расширения могут иметь прямой доступ ко всем подпрограммам. Это не означает, что вы должны предоставить пользователям доступ к каждой части своей программы – просто теперь у вас есть выбор, давать ли им доступ или нет.

Когда такой уровень доступа встречается с интерактивным окружением, достигается наилучшая расширяемость. Любая программа, которую можно использовать как основу для собственных расширений – это, вероятно, довольно большая программа, возможно, слишком большая для того, чтобы вы держали в уме ее полный образ. Что происходит, когда вы в чем-то не уверены? Если программа написана на Лисп, вы можете испытать ее в диалоговом режиме: вы можете изучить ее структуры данных, можете вызвать ее функции, даже можете посмотреть на изначальный исходный код. Такой вид обратной связи дает вам возможность программировать с большой уверенностью: писать более мощные расширения и писать их быстрее. Диалоговое окружение всегда делает программирование проще, но оно приобретает имеет особую ценность при написании расширений.

Расширяемая программа – это обоюдоострый меч, тем не менее последний опыт показал, что пользователи предпочитают обоюдоострый меч тупому. Расширяемые программы, похоже, одерживают верх, несмотря на присущие им опасности.

1.4 1-4 Расширение Лисп

Существует два способа добавления в Лисп новых операторов: функции и макрос. В Лисп определяемые вами функции имеют тот же статус, что и встроенные. Если вы хотите новый вариант mapcar, вы можете сами его определить и использовать точно так же, как использовали бы mapcar. Например, если вам нужен список значений,

возвращаемых некой функцией, примененной ко всем целым числам от 1 до 10, вы могли бы создать новый список и передать его mapcar:

```
(mapcar fn
       (do* ((x 1 (1+ x))
             (result (list x) (push x result)))
                      ((= x 10) (nreverse result))))
```

но такой подход и уродливый и неэффективный.[2]

вместо этого вы могли бы определить новую отображающую функцию map1-n (см. страницу 54) и затем ее вызвать следующим образом:

```
(map1-n fn 10)
```

Функции определяются относительно просто. Макросы предоставляют более широкое, но и менее понятное средство определения новых операторов. Макросы – это программы, которые пишут программы. Это утверждение имеет глубокий смысл, и его раскрытие – одна из основных целей книги.

Осмысленное использование макросов ведет к программам, удивляющим ясностью и изящностью. Эти сокровища не даются просто так. В итоге макросы, наверное, самая естественная в мире вещь, но вначале они могут быть трудными для понимания. Частично это из-за того, что они имеют больше возможностей, чем функции, поэтому, когда их пишешь, больше приходится держать в голове. Но основная причина, почему макросы трудны для понимания, это потому, что они незнакомы. Никакой другой язык не имеет ничего, подобного Лисп-макросам. Таким образом, изучение макросов позволит рассеять предрассудки, неосторожно приобретенные от других языков. Самый главный из которых – это представление программы в виде чего-то, подверженного трупному окоченению. Почему структуры данных должны быть гибкими и изменяемыми, а программы нет? В Лисп программы – это данные, но требуется время, чтобы вникнуть в смысл этого факта.

Если привыкание к макросам и займет некоторое время, то оно стоит затраченных усилий. Даже в таком простом применении, как итерация, макросы могут сделать программы существенно меньше и чище. Предположим, что программа должна произвести итерацию с некоторым участком кода для x от a до b. Встроенный в Лисп оператор do предназначен для более общих случаев. Для простой итерации он не даст наиболее читаемый код:

```
(do ((x a (+ 1 x)))
        ((> x b))
  (print x))
```

Вместо этого, допустим, мы могли бы сказать всего лишь:

```
(for (x a b)
  (print x))
```

Макросы делают это возможным. При помощи шести строчек кода (см. стр. 154) мы можем добавить к языку for, как будто бы он там был с самого начала. И, как

[2] Вы могли бы написать это более красиво при помощи последовательности новых макросов Коммон Лисп, но это лишь доказывает ту же точку зрения, так как эти макросы сами по себе являются расширением Лисп.

покажут последующие главы, написание for только начало того, что вы можете делать с макросами.

Вы не ограничены расширять Лисп за раз только одной функцией или макросом. Если потребуется, вы можете создать целый язык поверх Лисп и писать на нем свои программы. Лисп – это превосходный язык для написания компиляторов и интерпретаторов, но он предлагает иной способ описания нового языка, который зачастую более изящен и, несомненно, менее трудоемок – описать новый язык как модификацию Лисп. В таком случае части Лисп, которые могут появиться в новом языке без изменений (например, арифметика или ввод/вывод), могут быть использованы как есть, и вам нужно реализовать компоненты, которые отличаются (например, управляющую структуру). Язык, реализованный таким образом, называется встраиваемым языком.

Встраиваемые языки – естественный результат восходящего программирования. Коммон Лисп уже включает в себя несколько. Наиболее известный из них – CLOS – рассматривается в последней главе. Но вы также можете описать и свои собственные встраиваемые языки. Вы можете получить язык, который подходит к программе, даже если он станет выглядеть совершенно отличным от Лисп.

1.5 1-5 Почему Lisp (или Когда)

Эти новые возможности не происходят от единственного волшебного ингридиента. В этом отношении Lisp как арка. Какой из клинообразных камней (voussoirs) является тем, что поддерживает арку? Вопрос неверен сам по себе: они все ими и являются. Как и арка, Лисп – это набор взаимосвязанных функций(возможностей). Мы можем перечислить некоторые из этих функций: динамическое распределение памяти и сборка мусора, динамическая типизация, функции как объекты, встроенный систаксический анализатор, генерирующий списки, компилятор, принимающий программы, представленные в виде списков, диалоговое окружение, и так далее, но ни в одной из них нельзя усмотреть способности Лисп. Это сочетание, делающее программирование на Лисп таким, какое есть.

На протяжении двадцати последних лет подход к программированию изменился. Многие из этих изменений: диалоговые окружения, динамическое связывание, даже объектно-ориентированное программирование – были частичными попытками передать другим языкам немного гибкости Лисп. Метафора арки как бы намекает, насколько хорошо они преуспели.

Широко известно, что Лисп и Фортран – два, пока еще используемых, старейших языка. Что, наверное, наиболее важно, так это то, что они представляют противоположные стороны философии дизайна языков программирования. Фортран был изобретен как расширение ассемблера. Лисп был изобретен как язык для выражения алгоритмов. Такие разные стремления привели к совершенно разным языкам. Фортран облегчает жизнь автору компилятора, Лисп облегчает жизнь программиста. Большинство языков программирования впоследствии распределились где-то между двумя полюсами. Фортран и Лисп переместились ближе к центру. Фортран сейчас выглядит больше, как Алгол, а Лисп лишился некоторых скверных привычек своей юности.

Изначально Фортран и Лисп характеризовались как, своего рода, два враждующих лагеря. В одном из них кричали: "Эффективность!" (За исключением того, что было

бы слишком трудно реализовать). В другом лагере кричали: "Абстракция!" (В любом случае это непромышленное ПО). И боги определили из далеких исходов древнегреческих баталий, что исход этой битвы будет определен аппаратным обеспечением. С каждым годом обстоятельства складываются лучше для Лисп. Аргументы против Лисп сейчас стали звучать очень похоже на аргументы программистов на ассемблере, направленные против высокоуровневых языков в середине 1970-х. Вопрос, который сейчас ставится, в том не "Почему Лисп?", а "Когда?".

2 2 Функции

Функции - стандартные блоки программ Лисп. Они также и стандартные блоки Лисп. В большинстве языков оператор "+" – это нечто, совершенно отличное от определяемых пользователем функций. Но у Лисп есть единая модель для описания всех вычислений, сделанных программой – применение функций. В Лисп оператор "+" – это точно такая же функция, какую вы можете определить сами.

В действительности, за исключением небольшого количества операторов, называемых специальными формами, ядро Лисп – это коллекция функций Лисп. Что же остановит вас от пополнения коллекции? Совершенно ничего. Если вы думаете о чем-нибудь, что хотели бы, чтобы делал Лисп, то можете написать это сами, и ваша новая функция будет работать точно так же, как встроенная. Этот факт имеет важные следствия для программиста. Это означает, что любую новую функцию можно рассматривать или как дополнение к Лисп, или как часть определенного приложения. Как правило, опытный программист Лисп напишет всего понемногу, корректируя границу между языком и приложением, пока они не подойдут друг другу идеально. Эта книга о том, как достигнуть хорошей подгонки между языком и приложением. Поскольку все, что мы делаем в этом направлении, в конечном счете, зависит от функций, функция – это естественная точка отсчета.

2.1 2-1 Функции в качестве данных

Функции Лисп отличаются двумя вещами. Первая, упомянутая выше, это то, что сам Лисп – коллекция функций. Это означает, что мы можем добавить к Лисп свои новые операторы. Другая важная вещь, которую нужно знать о функциях, это то, что они являются объектами Лисп.

В Лисп можно найти большинство типов данных, которые имеются в других языках. Такие, как целые числа и числа с плавающей запятой, строки, массивы, структуры и так далее. А еще Лисп поддерживает один тип данных, который может поначалу вызвать удивление: функция. Почти все языки программирования в том или ином виде поддерживают функции или процедуры. Что значит, когда говорят, что Лисп представляет функции как тип данных? Это значит, что в Лисп мы можем делать с ними то же самое, что могли бы делать с известными типами данных, например, целыми числами: создавать новые во время выполнения, сохранять их в переменных и в структурах, передавать их как параметры другим функциям и возвращать их как результаты.

Способность создавать и возвращать функции во время выполнения особенно полезна. Сначала кажется, что это сомнительное преимущество, подобно такому, как самомодифицирующиеся программы на машинном языке, которые могут работать на некоторых компьютерах. Но создание новых функций во время выполнения, оказывается, обычная методика программирования на Лисп.

2.2 2-2 Определение функций.

Многие сначала узнают, как создавать функции при помощи defun. Следующее выражение определяет функцию с именем double, которая возвращает удвоение аргумента.

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

```
DOUBLE
```

Скормив это Lisp, мы можем вызывать double из других функций или из верхнего уровня:

```
> (double 1)
2
```

Файл с исходным кодом Лисп, как правило, состоит из подобных defun и этим походит на файл с определением процедур на языке Си или Паскаль. Но происходит кое-что совершенно другое. Эти defun – это не только определения процедур, но и вызовы Лисп. Это различие станет более ясным, когда мы увидим, что же происходит внутри defun.

Функции – это полноправные объекты. На самом деле, defun выстраивает такой объект и сохраняет его под именем, заданным первым аргументом. Помимо этого, по имени double, мы можем выяснить, что за функция его реализует. Обычно это делают при помощи оператора #'. Этот оператор следует понимать как отображение имен на существующие функциональные объекты. Применяя его к имени double,

```
> #'double
#<Interpreted-Function C66ACE>
```

мы получим реально существующий объект, созданный определением выше. Хотя это напечатанное представление меняется от реализации к реализации, функция Коммон Лисп – это объект первого класса с полностью такими же правами, как и более привычные объекты, вроде чисел и строк. Так что мы можем передавать эту функцию как аргумент, возвращать ее, сохранять в структуре данных и так далее:

```
> (eq #'double (car (list #'double)))
T
```

Нам даже не нужен defun, чтобы создавать функции. Как и на большинство объектов Лисп, мы можем прямо на них ссылаться. Когда мы хотим сослаться на целое число, мы прямо используем само целое число. Для представления строки мы используем последовательность символов, окруженных двойными кавычками. Для представления функции мы используем так называемое лямбда-выражение. Лямбда-выражение – это список, состоящий из трех частей: символа лямбда, списка параметров и тела из нуля или более выражений. Лямбда-выражение ссылается на функцию, эквивалентную double:

```
(lambda (x) (* x 2))
```

Оно описывает функцию, принимающую один аргумент x и возвращающую 2x.

Лямбда-выражение также может рассматриваться как имя функции. Если double – имя собственное, как \Микеланджело", то (lambda (x) (* x 2)) – точное определение, как \человек, расписавший свод Сикстинской капеллы". Помещая диез-кавычку перед лямбда-выражением, мы получим соответствующую функцию:

```
> #'(lambda (x) (* x 2))
#<Interpreted-Function C674CE>
```

— : After #\# is #\RIGHT_SINGLE_QUOTATION_MARK an undefined dispatch macro Измените, пожалуйста, апострофы на прямые '. При копировании примеров в интерпретатор Lisp ошибки выполнения форм. — lispuser

Эта функция ведет себя точно так же, как и double, но это два различных объекта.

При вызове функции имя функции идет в начале, а за ним следуют аргументы:

```
> (double 3)
6
```

Так как лямбда-выражения – это также имена функций, то при вызове функций они также могут появиться в начале:

```
> ((lambda (x) (* x 2)) 3)
6
```

В Коммон Лисп у нас одновременно может быть функция с именем double и переменная с именем double.

```
> (setq double 2)
2> (double double)
4
```

Когда имя встречается первым при вызове функции или ему предшествует диезкавычка, то оно воспринимается как ссылка на функцию. В противном случае рассматривается как имя переменной.

Именно поэтому говорят, что Коммон Лисп имеет различные пространства имен для переменных и функций. У нас может быть переменная с именем foo и функция с именем foo, и им необязательно быть одинаковыми. Ситуация может сбивать с толку и приводить к определенному уродству в коде, но это что-то, с чем программисты Common Lisp вынуждены сосуществовать.

Если необходимо, Коммон Лисп предоставляет две функции, отображающие символы в значения или функции, которые они представляют. Функция symbol-value принимает символ и возвращает значение соответствующей специальной переменной:

```
> (symbol-value 'double)
2
```

в то время как symbol-function делает то же самое для глобально определенной функции:

```
> (symbol-function 'double)
#<Interpreted-Function C66ACE>
```

Обратите внимание: так как функции являются обычными объектами данных, переменные могут иметь функцию в качестве значения:

```
> (setq x #'append)
#<Compiled-Function 46B4BE>
> (eq (symbol-value 'x) (symbol-function 'append))
T
```

За кулисами defun устанавливает symbol-function от первого аргумента на функцию, состоящую из последующих аргументов. Следующие два выражения делают примерно то же самое:

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

(setf (symbol-function 'double)
          #'(lambda (x) (* x 2)))
```

Итак, defun делает то же, что и описание процедуры в других языках – связывает имя с участком кода. Но лежащий в основе механизм иной. Нам не нужен defun

для создания функций, и функции не обязаны сохраняться как значение некоторого символа. В основе defun, который аналогичен описанию процедуры в любом другом языке, лежит более общий механизм: создание функции и связывание ее с определенным именем – это две различные операции. Когда нам не требуется полная общность представления функций в Лисп, defun делает определение функции таким же простым, как и в более ограниченных языках.

2.3 2-3 Функциональные аргументы

Представление функций как объектов данных означает, помимо прочего, что мы можем передавать их в качестве аргументов другим функциям. Эта возможность отчасти отвечает за важность восходящего программирования в Лисп.

Язык, допускающий функции в качестве объектов данных, должен также предоставить определенные способы их вызова. В Лисп это функция apply. Вообще, мы вызываем apply с двумя аргументами: функцией и списком ее аргументов. Все последующие четыре выражения выполняют одно и то же действие:

```
(+ 1 2)

(apply #'+ '(1 2))

(apply (symbol-function '+) '(1 2))

(apply #'(lambda (x y) (+ x y)) '(1 2))
```

В Коммон Лисп apply может принимать любое число аргументов, и функция, заданная первой, будет применена к списку, полученному путем синтеза в него оставшихся аргументов, заданному в конце. Так, выражение

```
(apply #'+ 1 '(2))
```

эквивалентно четырем предыдущим. Если задавать аргументы в виде списка неудобно, то мы можем использовать funcall, который отличается от apply только в этом отношении. Это выражение

```
(funcall #'+ 1 2)
```

выполняет то же действие, что и все предыдущие.

Множество встроенных в Коммон Лисп функций принимает функциональные аргументы. Среди наиболее широко используемых находятся отображающие функции. Например, mapcar принимает два или более аргументов, функцию и один или более списков (один на каждый параметр функции) и применяет функцию последовательно к элементам каждого списка:

```
> (mapcar #'(lambda (x) (+ x 10))
                '(1 2 3))
(11 12 13)
> (mapcar #'+
                '(1 2 3)
                '(10 100 1000))
(11 102 1003)
```

Программы на Лисп часто испытывают необходимость делать что-то с каждым аргументом списка и вернуть назад список результатов. Первый пример, приведенный выше, показывает обычный способ это реализовать: создать функцию, которая выполняет то, что вам нужно, и при помощи mapcar применить ее к списку.

Мы уже видим, насколько удобно иметь возможность обращаться с функциями, как с данными. Во многих языках, даже если мы могли бы передать функцию в качестве аргумента чему-то, вроде mapcar, она бы, тем не менее, была функцией, определенной заранее в каком-нибудь исходном файле. Если бы требовался именно цельный код, добавляющий 10 к каждому элементу списка, мы бы определили функцию с именем plus ten или каким-то подобным только для этого единственного применения. С помощью лямбда выражений мы можем ссылаться на функции непосредственно.

Одно из существенных отличий между Коммон Лисп и предшествовавшими ему диалектами состоит в огромном количестве встроенных функций, принимающих функциональные аргументы. Две из наиболее используемых после вездесущего mapcar – это sort и remove-if. Первая из них является функцией сортировки общего назначения. Она принимает список аргументов и предикат, а возвращает список, отсортированный пропусканием через предикат каждой пары элементов.

```
> (sort '(1 4 2 5 6 7 3) #'<)
(1234567)
```

Чтобы понять, как работает sort, полезно осознать, что если вы сортируете список без повторяющихся элементов с помощью <, то < вернет true, если его применить к получившемуся списку.

Если бы remove-if не был бы включен в Коммон Лисп, то это, возможно, была бы первая утилита, которую бы вы написали. Она принимает функцию и список, а возвращает все элементы списка, для которых функция вернула false.

```
> (remove-if #'evenp '(1 2 3 4 5 6 7))
(1 3 5 7)
```

В качестве примера функции, принимающей функциональные аргументы, здесь приводится определение ограниченной версии remove-if:

```
(defun our-remove-if (fn lst)
  (if (null lst)
            nil
            (if (funcall fn (car lst))
                   (our-remove-if fn (cdr lst))
                   (cons (car lst) (our-remove-if fn (cdr lst))))))
```

Заметим, что в этом определении fn используется без диеза-кавычки. Так как функции являются объектами данных, то переменные могут содержать функцию в качестве обычного значения. Вот что здесь происходит. Диез-кавычка служит только для ссылки на функцию, имеющую символьное имя, как правило, глобально определенное с помощью defun.

Как покажет глава 4, написание новых утилит, принимающих функциональные аргументы, является важным элементом программирования снизу-вверх. В Коммон Лисп есть так много встроенных утилит, что та, которая вам требуется, возможно, уже существует. Но используете ли вы встроенные функции, как sort, или пишете

собственные утилиты, принцип тот же. Вместо того, чтобы вписывать функциональность, передавайте функциональный аргумент.

2.4 2-4 Функции в качестве свойств

То обстоятельство, что функции Лисп являются объектами, также позволяет нам писать программы, которые могут быть расширены на лету для работы в изменившихся условиях. Допустим, мы хотим написать функцию, принимающую аргумент – вид животного и ведущую себя соответственно. В большинстве языков естественным средством для этого будет оператор case, и мы точно так же можем это сделать и в Lisp:

```
(defun behave (animal)
  (case animal
        (dog (wag-tail)
             (bark))
        (rat (scurry)
             (squeak))
        (cat (rub-legs)
             (scratch-carpet))))
```

Что, если нам потребуется добавить новый тип животного? Если бы мы хотели добавить новых животных, то было бы лучше определить behave следующим образом:

```
(defun behave (animal)
  (funcall (get animal 'behavior)))
```

и определить поведение отдельного животного как функцию, сохраненную, к примеру, в списке свойств его имени:

```
(setf (get 'dog 'behavior)
            #'(lambda ()
               (wag-tail)
               (bark)))
```

В таком случае, все, что потребуется сделать для добавления нового животного это определить новое свойство. Никаких функций переписывать не нужно.

Другой подход, несмотря на большую гибкость кажется медленным. Так и есть. Если скорость была бы критична, то мы использовали бы структуры вместо списков свойств и, главным образом, компилированные, вместо интерпретируемых функций. (Раздел 2.9 объясняет, как это сделать). Более гибкий вид кода, со структурами и компилированными функциями , может приблизиться или обогнать по скорости версии, использующие оператор case.

Такое использование функций передаёт концепцию методов в объектно-ориентированном программировании. Собственно, метод – это функция, являющаяся свойством объекта, и это всё. Если мы добавим наследование в эту модель, то получим все элементы объектно-ориентированного программирования. Глава 25 покажет, как сделать это с удивительно маленьким кодом.

Одним из больших преимуществ объектно-ориентированного программирования является расширяемость. Такая перспектива не особо удивительна в мире Лиспа, где расширяемость всегда была в порядке вещей. Если она не очень сильно зависит от наследования, то простоты Лиспа уже может хватать.

2.5 2-5 Область видимости(Scope)

Common Lisp — Лисп с лексическим связыванием. Scheme является старейшим диалектом с лексическим связыванием; до Scheme динамическое связывание считалось одним из особенностей Лиспа.

Разница между лексической и динамической областью действия связываемых переменных заключается в том, как реализация поступает со свободными переменными. Символ связывается в выражении, если он используется в роли переменной, например как аргумент или при помощи операторов связывания таких, как let и do. Символы, которые остаются несвязанными, называются свободными. В данном примере связывание проявляет себя:

```
(let ((y 7))
      (defun scope-test (x)
       (list x y)))
```

Внутри defun x является связанной переменной а y - свободной. Интерес к свободным переменным появляется потому, что неясно, каким должно быть их значение. Нет никакой неопределенности в значении связанной переменной – когда scope-test будет вычисляться значение x будет тем, что передадут в качестве аргумента, но каким должно быть значение y? На этот вопрос могут ответить правила связывания переменных конкретного диалекта.

В лиспе с динамической областью видимости для того, чтобы найти значение свободной переменной во время выполнения scope-test мы смотрим назад по цепочке вызовов функций, когда мы находим окружение, в котором y - связана, мы используем эту привязку в scope-test, если же такого окружения нет, будет взято значение глобальной переменной y. Таким образом в лиспе с динамическим связыванием y будет содержать то значение, которое оно содержало в вызывающем выражении:

```
> (let ((y 5))
        (scope-test 3))
(3 5)
```

При динамическом связывании ничего не значит тот факт, что y было связано со значением 7 при объявлении scope-test. Имеет значение только то, что y содержало значение 5 когда scope-test была вызвана.

В лиспе с лексическим связыванием вместо просматривания всей цепочки вызовов функций мы смотрим назад, на окружение в тот момент, когда функция была объявлена. В лиспе с лексическим связыванием наш пример захватит значение y в тот момент, когда scope-test была объявлена. Так что в Common Lisp произойдет следующее:

```
> (let ((y 5))
        (scope-test 3))
(3 7)
```

Здесь связывание y со значением 5 во время вызова никак не повлияло на возвращаемое значение.

Несмотря на то, что вы можете получить динамическое связывание, объявляя переменную специальной, лексические связывание являются стандартными для Common Lisp. В основном комьюнити относится к динамическому связыванию с недоверием.

Так, например, оно может быть причиной опасных неуловимых ошибок, но лексические замыкания - это больше чем просто способ избежать ошибки. В следующем параграфе показано, как с его помощью можно использовать некоторые новые техники.

2.6 2-6 Замыкания

Поскольку Common Lisp является языком с лексическим связыванием, когда мы определяем функцию, содержащую свободные переменные, система должна сохранить копии привязок к этим переменным, в то время когда функция определяется. Такое сочетание функции и набора привязок переменных называется - Замыканием. Замыкания оказываются полезными в широком спектре приложений.

Замыкания настолько распространены в обычных программах на Lisp, что можно использовать их даже не подозревая об этом. Каждый раз, когда вы отдаёте mapcar лямбда-выражение с диез-кавычкой, вы используете замыкания. Например, предположим, что мы хотим написать функцию, которая принимает список номеров и добавляет определённую сумму к каждому. Функция list+

```
(defun list+ (lst n)
      (mapcar #'(lambda (x) (+ x n))
                lst))
```

будет делать то, что мы хотим:

```
> (list+ '(1 2 3) 10)
(11 12 13)
```

Если мы посмотрим внимательно на функцию, передаваемую в mapcar внутри list+, то поймём, что это замыкание. Экземпляр n является свободным и будет связан со значением из внешнего окружения. С лексическим связыванием, каждое использование присваивающей функции приводит к появлению замыкания.[1]

Замыкания играют важную роль в стиле программирования, предлагаемом в книге Абельсона и Сассмана структура и интерпретация компьютерных программ (SICP). Замыкания - это функции с локальным состоянием. Простейший способ применения показан ниже:

```
(let ((counter 0))
      (defun new-id ()   (incf counter))
      (defun reset-id () (setq counter 0)))
```

Эти две функции используют одну переменную(разделяют ее меж собой), являющуюся счетчиком. Первая возвращает следующее значение, вторая обнуляет счетчик. Аналогичный результат можно получить сделав счетчик глобальной переменной, но применение замыканий защищает от случайных ссылок.

Замыкания также удобно использовать для создания функций с локальным состоянием. Например функция make-adder

```
(defun make-adder (n)
      #'(lambda (x) (+ x n)))
```

[1] С динамическим связыванием та же идиома будет работать по другой причине и до тех пор, пока ни один из параметров mapcar не назван x.

в качестве аргумента принимает число и возвращает замыкание, которое, если его вызвать, складывает это число со своим аргументом. Мы может создать столько таких замыканий, сколько нам нужно:

```
> (setq add2 (make-adder 2)
             add10 (make-adder 10))
#<Interpreted-Function BF162E>
> (funcall add2 5)
7> (funcall add10 3)
13
```

В замыкании возвращаемом функцией make-adder внутреннее состояние фиксированно, но можно сделать замыкание, которое можно попросить менять свое состояние

```
(defun make-adderb (n)
     #'(lambda (x &optional change)
          (if change
                (setq n x)
                (+ x n))))
```

Эта новая версия make-adder возвращает замыкание, которое, при вызове его с одним аргументом, действует так же как и его предыдущая версия.

```
> (setq addx (make-adderb 1))
#<Interpreted-Function BF1C66>
> (funcall addx 3)
4
```

Тем не менее если новую версию вызвать со вторым аргументом, не равным nil, то его внутренняя копия переменной n будет установлена в значение, переданное первым аргументом:

```
> (funcall addx 100 t)
100
> (funcall addx 3)
103
```

Более того, можно вернуть набор замыканий, которые разделяют одно состояние. Рисунок 2.1 содержит функцию, которая создает примитивную базу данных. В качестве аргумента она принимает ассоциативный список (база данных) и возвращает список из трех замыканий для выборки, добавления и удаления данных соответственно.

Каждый вызов make-dbms создает новую базу данных - новый набор функций, лексически замкнутых относительно разделяемого ими ассоциативного списка.

```
> (setq cities (make-dbms '((boston . us) (paris . france))))
(#<Interpreted-Function 8022E7>
     #<Interpreted-Function 802317>
     #<Interpreted-Function 802347>)
```

```
(defun make-dbms (db)
      (list
        #'(lambda (key)
               (cdr (assoc key db)))
        #'(lambda (key val)
               (push (cons key val) db)
               key)
        #'(lambda (key)
               (setf db (delete key db :key #'car))
               key)))
```

Рисунок 2-1: Три замыкания совместно используют список

Настоящий ассоциативный список внутри базы данных невидим для внешнего мира, мы даже не можем сказать что это ассоциативный список, но мы может взаимодействовать с ним при помощи функций:

```
> (funcall (car cities) 'boston)
US
> (funcall (second cities) 'london 'england)
LONDON
> (funcall (car cities) 'london)
ENGLAND
```

Вызов (car list) - не самое красивое решение. В настоящих программах функции для доступа могут быть, например, элементами структуры. Их использование может быть проще - база данных может быть доступна не напрямую, через функции, например:

```
(defun lookup (key db)
      (funcall (car db) key))
```

Тем не менее поведение замыканий не зависит от подобных тонкостей.

В настоящих программах замыкания и структуры данных могут быть гораздо сложней тех, что мы видели в make-adder или make-dbms. Вместо одной разделяемой переменной может быть любой набор переменных, каждая из которых связана с какой-нибудь структурой данных.

Замыкания - одно из явных, осязаемых достоинств лиспа. Некоторые программы на лиспе могут быть, с некоторыми усилиями, перенесены на менее мощный язык, но попробуйте перевести программу, использующую замыкания, например, как приведенная выше, и станет очевидно насколько меньше работы нам пришлось проделать имея такую абстракцию как замыкание. В следующих главах мы рассмотрим замыкания более детально. Глава 5 покажет нам, как использовать их для построения еще более сложных функций

2.7 2-7 Локальные функции

Когда мы определяли функции при помощи лямбда-выражений, то столкнулись с ограничением, которого нет при использовании defun: у функции определенной как лямбда нет имени, так что нет возможности ссылаться на себя. Это означает что в Common Lisp мы неможем использовать лямбды для определения рекурсивных функций.

Если мы хотим приенить некоторую функцию ко всем элементам списка, мы используем наиболее привычный для лиспа стиль:

```
> (mapcar #'(lambda (x) (+ 2 x))
                '(2 5 7 3))
(4795)
```

Но что поделать, если мы хотим передать первым аргументом mapcar рекурсивную функцию? Если бы функция была определена через defun, мы могли бы просто сослаться на нее по имени:

```
> (mapcar #'copy-tree '((a b) (c d e)))
((A B) (C D E))
```

Но предположим теперь, что функция должны быть замыканием, содержащим некоторые свящанные переменные из окружения в котором будет работать mapcar. В нашем примере list+:

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
                lst))
```

первый аргумент mapcar, #'(lambda (x) (+ x n)), должен быть определен внутри list+, потому что ему нужно связанное значение n. Пока что все хорошо, но что поделать, если нам нужно передать в mapcar и локально связанную переменную и рекурсивную функцию? Мы не можем использовать функцию, определенную где-то при помощи defun, потому что нам нужны локально связанные переменные и мы не можем использовать лямбду для определения рекурсивной функции, потому что она неможет ссылаться на себя.

Common Lisp дает нам labels для разрешения этой дилеммы. С одной важной оговоркой, labels должны быть описаны в форме похожей на let. Каждое описание связывания в выражении должно быть следующего вида

```
( name  parameters  .  body )
```

Внутри labels выражение <name> ссылаться на функцию, эквивалентную следующей:

```
#'(lambda  parameters  .  body )
```

В качестве примера:

```
> (labels ((inc (x) (1+ x)))
       (inc 3))
4
```

Тем не менее есть важное различие между let и labels. Внутри let выражения значение одной переменной неможет зависеть от какой-либо другой объявленной внутри одной let формы, это значит что вы НЕ можете написать

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

и ожидать что значение новой переменной y будет содержать значение новой x. С другой стороны тело функции f определенной внутри выражения labels может ссылаться на любую другую функцию, определенную там же, влючая саму себя, что делает возможным описание рекурсивных функций.

Используя labels мы можем написать функцию, аналогичную list+, но в которой первый аргумент mapcar будет рекурсивной функцией:

```
(defun count-instances (obj lsts)
      (labels ((instances-in (lst)
                  (if (consp lst)
                       (+ (if (eq (car lst) obj) 1 0)
                           (instances-in (cdr lst)))
                       0)))
       (mapcar #'instances-in lsts)))
```

Эта функция берет объект и список и возвращает список чисел, соответствующих числу вхождений объекта в каждый элемент:

```
> (count-instances 'a '((a b c) (darpa)(dar)(aa)))
(1212)
```

2.8 2-8 Хвостовая Рекурсия

Рекурсивными называют функции вызывающие сами себя. Хвостовой рекурсией называется рекурсия, если в вызывающей не остается никакой работы после вызова себя рекурсивно. Следующая функция НЕ является функцией с хвостовой рекурсией

```
(defun our-length (lst)
      (if (null lst)
            0(1+ (our-length (cdr lst)))))
```

поскольку, вернувшись из рекурсивыного вызова, мы должны передать результат в 1 +. Следующая функция является функцией с хвостовой рекурсией

```
(defun our-find-if (fn lst)
   (if (funcall fn (car lst))
               (car lst)
               (our-find-if fn (cdr lst))))
```

потому что значение рекурсивного вызова сразу же возвращается.

Хвостовая рекурсия более эффективна, потому что многие реализации Common Lisp могут трасформировать хвостовую рекурсию в цикл. С такими компиляторами вы можете получить элегантность рекурсии в исходном коде без накладных расходов на вызов функций. Прирост скорости обычно настолько высок, что программисты делают многое, чтобы сделать функцию с хвостовой рекурсией.

Функция, не обладающая свойством хвостовой рекурсии, часто может быть трансформированна в таковую, при помощи включения в нее локальной функции, использующей аккумулятор. В нашем контексте аккумулятор - это параметр, представляющий значение, которое нужно вычислить в процессе рекурсии. Например наша функция our-length может быть трансформирована в

```
(defun our-length (lst)
   (labels ((rec (lst acc)
                    (if (null lst)
                          acc
                          (rec (cdr lst) (1+ acc)))))
         (rec lst 0)))
```

где колличество элементов списка, которые будут пройдены содержится во втором параметре acc. Когда рекурсивная функция достигнет конца списка, значение acc будет его длиной, которое сразу может быть возвращено. Аккумулируя значения при спуске по дереву вызовов вместо того чтобы конструировать его поднимаясь мы можем сделать rec функцией с хвостовой рекурсией.

Многие компиляторы Common Lisp делают оптимизацию хвостовой рекурсии, но не все делают это по умолчанию. Так что после написания функции с хвостовой рекурсией непомешает так же добавить

```
(proclaim '(optimize speed))
```

в начале файла, чтобы компилятор смог извлечь пользу из ваших стараний.[2]

Используя хвостовую рекурсию и декларацию типов существующие реализации компиляторов Common Lisp могут генерировать код такой же бытрый, или даже быстрее кода на C. Ричард Гэбриэл приводит пример следующей функции, которая возвращает сумму целых чисел от 1 до n:

```
(defun triangle (n)
      (labels ((tri (c n)
                  (declare (type fixnum n c))
                  (if (zerop n)
                      c(tri (the fixnum (+ n c))
                             (the fixnum (- n 1))))))
       (tri 0 n)))
```

Это пример того, как выглядит быстрый код на Common Lisp. С первого взгляда не кажется естественным писать код в таком виде. Хорошей идеей будет написать функцию в том виде, в котором она выражается на лиспе наиболее естественно, и затем, если возникнет необходимость, трансформировать ее в вариант с хвостовой рекурсией.

2.9 2-9 Компиляция

Функции в лиспе могут быть скомпилированы либо индивидуально, либо в составе файла. Если вы просто напечатаете выражение defun в toplevel

```
> (defun foo (x) (1+ x))
FOO
```

то многие реализации создадут интерпретируемую функцию. Вы можете проверить, является ли данная функция скомпилированной скормив ее compiled-function-p:

```
> (compiled-function-p #'foo)
NIL
```

Мы можем получить скомпилированную версию, передав ее имя функции compile

```
> (compile 'foo)
FOO
```

[2] Декларация (optimize speed) является аббревиатурой для (optimize (speed 3)). Тем не менее некоторые компиляторы Common Lisp делают оптимизацию хвостовой рекурсии при первом варианте но не втором.

которая скомпилирует определение функции foo и заменит интерпретируемую версию на скомпилированную.

```
> (compiled-function-p #'foo)
T
```

Скомпилированная и интерпретируемая версии являются объектами в лиспе и ведут себя одинаково за исключением функции compiled-function-p. Лямбда функции так же могут быть скомпилированы: compile ожидает в качестве первого аргумента имя функции, но если передать nil, то она скомпилирует лямбда выражение, переданое вторым аргументом.

```
> (compile nil '(lambda (x) (+ x 2)))
#<Compiled-Function BF55BE>
```

Если передать и имя и лямбда выражение, то compile будет работать как компилирующий defun:

```
> (progn (compile 'bar '(lambda (x) (* x 3)))
                 (compiled-function-p #'bar))
T
```

Наличие compile в языке означает, что программа может строить и компилировать новые функции на лету. Тем не менее вызов compile в явном виде является очень сильным средством, сравнимым с eval и должно применяться с такой же осторожностью.[3] Когда в секции 2.1 См. Раздел 2.1 [2-1 Functions as Data], страница 15, говорилось, что создание новых функций во время выполнения является привычной техникой - имелось ввиду создание новых замыканий, вроде тех, которые создавались при помощи make-adder, но не функции созданные при помощи вызова compile на сыром списке. Вызов compile - не обычная техника, это крайне редкий случай. Так что избегайте применения ее без необходимости. И все же если вы реализуете другой язык над лиспом (и даже в таком случае) то что вам нужно может быть возможно при помощи макросов.

Существует два вида функций, которые вы не можете передать compile в качестве аргумента. Согласно CLTL2 (стр. 677), вы не можете скомпилировать функцию "объявленную интерпретируемо в ненулевом лексическом окружении". Тоесть если в toplevel определить foo внутри let

```
> (let ((y 2))
        (defun foo (x) (+ x y)))
```

то (compile 'foo) необязательно сработает.[4] Вы так же несможете скомпилировать функцию, которая уже была скомпилирована. В этой ситуации CLTL2 отвечает туманно "результат... не определен."

Привычный способ копмилирования лисп кода - не компиляция отдельных функций, а компиляция всего файла при помощи compile-file. Это функция берет имя файла и создает скомпилированную версию исходного файла - как правило с таким же именем и другим расширением. Когда скомпилированный файл загружен, compile-function-p должен вернуть true для всех функций в нем.

---

[3] Объяснение, почему плохо вызывать eval в явном виде есть на 278 странице.

[4] Не страшно держать такой код в файле и затем компилировать его. Ограничение распространяется на интерпретируемый код по причине реализации, но не из-за того, что что-то не так с определением функций в различных окружениях.

В следующих главах будем опираться на еще один эффект компиляции: когда одна функция встречается внутри другой, и внешняя функция компилируется, то внутренняя функция так же будет скомпилирована. CLTL2 явно не оговаривает этот момент, но вы можете расчитывать на него в основных реализациях.

Компиляция внутренней функции становится явной в функциях, которые возвращают функции. Когда make-adder (стр. 18) будет скомпилирована, она вернет скомпилированную фунуцию:

```
> (compile 'make-adder)
MAKE-ADDER
> (compiled-function-p (make-adder 2))
T
```

В следующих главах будет показано, что этот факт играет важнейшую роль в разработке встроенного языка. Если новый язык реализован как трансформация и трансформирующий код скомпилирован, то будет получен скомпилированный результат - и таким образом получаем эффективный компилятор для нового языка. (Простой пример описан на странице 81.)

Если у нас есть очень маленькая функция, мы можем попросить компилятор сделать ее встроенной. Иначе издержки на вызов функции могут быть больше, чем ее полезная работа. Если мы определим функцию:

```
(defun 50th (lst) (nth 49 lst))
```

и добавим декларацию:

```
(proclaim '(inline 50th))
```

то ссылка на функцию 50th внутри скомпилированной функции больше не должно требовать реального вызова функции. Если мы определим и скомпилируем функцию, которая вызывает 50th,

```
(defun foo (lst)
      (+ (50th lst) 1))
```

то когда foo будет скомпилирована, код 50th будет вкомпилирован непосредственно в нее, как быдто мы написали

```
(defun foo (lst)
      (+ (nth 49 lst) 1))
```

в первом случае. Недостаток в том, что если мы переопределим 50th то придется перекомпилировать foo, или она все еще будет работать со старым определением. Ограничение на встраиваемые функции то же, что и на макросы (см Глава 7.9).

2.10 2-10 Функции из списков

В некоторых ранних диалектах лиспов функции были представлены как списки. Это давало лиспу выдающуюся способность писать и выполнять свои собственные программы на лисп. В Common Lisp функции более не списки, хорошие реализации сразу компилируют их в машинный код. Но вы все еще можете писать программы, которые пишут программы, потому что списки можно отдать компилятору.

Нельзя переоценить насколько важен факт того, что программы на лиспе могут писать программы на лиспе, особенно учитывая то как часто эту возможность недооценивают. Даже опытные программисты редко осознают какой выигрыш они получают

от этой возможности языка. Это причина того, почему макросы лиспа настолько могущественны. Большинство техник в этой книге зависят от возможности писать программы, манипулирующие лисп выражениями.

3 3 Функциональное программирование

В предыдущей главе рассказывалось о том, что Lisp и программы на нем созданы из единственного материала: функций. Как и любому строительному материалу, им присуще оказывать влияние как и на свойства того, что мы создаем, так и на способ создания.

Эта глава описывает методы разработки, преобладающие в мире Lisp. Изощренность этих методов позволит нам попытаться писать более сложные программы. В следующей главе будет рассмотрен особенно важный класс программ, который становится возможно реализовать в Лисп: программы, которые эволюционируют вместо того, чтобы разрабатываться старым способом <<спланируй и реализуй>>.

3.1 3-1 Функциональное проектирование

На свойства объекта влияют элементы, из которых он создан. Например, деревянное здание на вид отличается от каменного. Даже находясь от него на далеком расстоянии и не имея возможности отличить на взгляд камень от дерева, вы сможете определить материал, из которого оно выстроено, исходя из общей формы этого здания. Особенности функций Lisp оказывают аналогичное влияние на структуру Lisp программ.

Функциональное программирование означает написание программ, которые работают за счет возврата значений, а не за счет побочных действий. Побочные действия включают в себя деструктивные изменения объектов (напр. rplaca) и присвоения переменным (напр. setq). Если побочные действия немногочисленны и локализованы, то программы становится легче читать, тестировать и отлаживать. Lisp-программисты не всегда писали в таком стиле, но со временем Lisp и функциональное программирование постепенно стали неразделимы.

Пример покажет, как функциональное программирование отличается от того, что можно делать при помощи другого языка. Допустим, нам нужно с какой-то целью

```
(defun bad-reverse (lst)
       (let* ((len (length lst))
                 (ilimit (truncate (/ len 2))))
          (do ((i 0 (1+ i))
                 (j (1- len) (1- j)))
                ((>= i ilimit))
            (rotatef (nth i lst) (nth j lst)))))
```

Рисунок 3-1: Функция, инвертирующая списки.

изменить порядок элементов списка. Вместо написания функции, инвертирующей списки, мы напишем функцию, которая принимает список, а возвращает список с теми же элементами в обратном порядке.

На рисунке 3.1 представлена функция, инвертирующая списки. Она обрабатывает списки как массивы, сразу переворачивая их; ее возвращаемое значение неважно:

```
> (setq lst '(a b c))
(ABC)
```

```
> (bad-reverse lst)
NIL
> lst
(CBA)
```

Ее имя подсказывает, что bad-reverse далека от хорошего Lisp-стиля. Кроме того, ее уродство заразно: так как она работает при помощи побочных эффектов, то также уводит в сторону от функционального идеала вызвавшие ее функции.

Хотя, выступая в роли злодея, у bad-reverse есть одно достоинство: она демонстрирует идиому Common Lisp для перестановки двух значений. Макрос rotatef переворачивает значения любого числа обобщенных переменных, то есть выражения, которые можно передать setf в качестве первого аргумента. Результатом применения непосредственно к двум аргументам будет их перестановка.

С другой стороны, на рисунке 3.2 показана функция, возвращающая инвертированные списки. С помощью good-reverse мы получаем инвертированный список в качестве возвращаемого значения; исходный список остается прежним.

```
> (setq lst '(a b c))
(ABC)
> (good-reverse lst)
(CBA)
> lst
(ABC)
```

```
(defun good-reverse (lst)
      (labels ((rev (lst acc)
                    (if (null lst)
                           acc
                           (rev (cdr lst) (cons (car lst) acc)))))
         (rev lst nil)))
```

Рисунок 3-2: Функция, возвращающая инвертированные списки

Раньше считалось, что можно судить о чьем-то характере, глядя на форму его головы. Относится ли это в самом деле к людям или нет, но как правило, применимо к Lisp-программам. Внешний вид функциональных программ отличается от внешнего вида императивных. Структурно функциональная программа – это всегда композиция аргументов внутри выражений, а так как аргументы можно размещать с отступом, то функциональный код оказывается более разнообразным в плане наглядности. Функциональный код выглядит гибким[1], а императивный код выглядит сплошным и блочным, как Basic.

Мы видим, не вдаваясь в детали, исходя из внешнего вида bad- и good-reverse, какая из программ лучше. Несмотря на то, что good-reverse короче, она также более эффективна: O(n) вместо O(n^2).

Мы избежали проблемы написания reverse, поскольку Common Lisp имеет встроенную. Имеет смысл ненадолго взглянуть на эту функцию, потому что она часто

[1] Типичный пример приведен на странице 242.

выявляет неправильные представления о функциональном программировании. Как и good-reverse, встроенная reverse возвращает значение, не изменяя аргументов. Но изучающие Lisp могут подумать, что она, как и bad-reverse, работает за счет побочных эффектов. Если в какой-либо части программы потребуется инвертировать список, можно написать

```
(reverse lst)
```

но к удивлению это не сработает. В действительности, если нам нужны действия от такой функции, то придется сделать это самим в вызывающем коде. Вот, что вместо этого необходимо написать:

```
(setq lst (reverse lst))
```

Такие операторы, как reverse, предназначены для вызова с целью получения значений, а не побочных действий. Писать собственные программы в таком стиле стоит не только из-за обязательно присущих ему преимуществ, но и потому, что если этого не делать, вы будете работать против языка.

Сравнивая bad- и good-reverse, мы не придали значения одной из особенностей, заключающейся в том, что bad-reverse не создает новый список. Вместо создания новой структуры списка, она работает с исходным списком. Это может быть опасно: список может потребоваться где-либо еще в программе, но это иногда бывает необходимо ради производительности. Для таких случаев Common Lisp предоставляет O(n) деструктивную инвертирующую функцию, которая называется nreverse.

Деструктивной является та функция, которая может изменять передаваемые ей аргументы. Тем не менее даже деструктивные функции работают при помощи возвращаемых значений: вы должны считать, что nreverse переработает списки, переданные ей в качестве аргументов, но по-прежнему нельзя полагаться на то, что она их инвертирует. Как и раньше, инвертированный список будет содержаться в возвращаемом значении. Вы не можете написать

```
(nreverse lst)
```

в середине функции, полагая, что после этого lst будет инвертирован. Вот, что происходит в большинстве реализаций:

```
> (setq lst '(a b c))
(ABC)
> (nreverse lst)
(CBA)
> lst
(A)
```

Для инвертирования lst можно было бы присвоить lst возвращаемое значение, как в случае с обычным reverse.

Если функция обозначена как деструктивная, то это вовсе не означает, что ее предполагалось вызывать для побочных действий. Опасность в том, что некоторые деструктивные функции создают впечатление таковых. Например,

```
(nconc x y)
```

почти всегда производит те же действия, что и

```
(setq x (nconc x y))
```

Если вы написали код, зависящий от приведенной выше конструкции, то он, вероятно, может работать некоторое время. Однако он не будет делать то, что вы ожидали, когда x равно nil.

Лишь несколько операторов Lisp предназначены для вызова с целью получения побочных эффектов. Вообще, встроенные операторы задумывались для вызова ради возвращаемых ими значений. Такие названия, как sort, remove или substitute, не должны вводить в заблуждение. Если вам нужны побочные эффекты, применяйте setq к возвращаемому значению.

Это самое правило предполагает, что побочные действия(эффекты) неизбежны. Придерживаться функционального программирования как идеала, не означает того, что программы никогда не должны осуществлять побочных эффектов. Это означает, что они должны быть, но не более, чем это необходимо.

Чтобы развить такую привычку, возможно, потребуется время. Один из способов состоит в том, чтобы использовать следующие операторы так, как если бы был налог на их использование:

set setq setf psetf psetq incf decf push pop pushnew rplaca rplacd rotatef shiftf remf remprop remhash, а также let*

в которых нередко скрываются императивные программы. Применение этих операторов, как облагаемых налогом, лишь помогает приблизиться к хорошему Lisp-стилю, а не служит для него критерием. Тем не менее, лишь одно это позволит вам удивительно далеко зайти.

В других языках одной из наиболее частых причин возникновения побочных эффектов является необходимость возврата функцией нескольких значений. Если функции способны вернуть лишь одно значение, то они должны <<вернуть>> оставшиеся путем изменения собственных параметров. К счастью, это не требуется в Common Lisp, так как любая функция может вернуть несколько значений.

Встроенная функция truncate возвращает два значения, например, округленное целое и то, что было отрезано с целью его создания. Типичная реализация напечатает оба, когда truncate вызывается из верхнего уровня:

```
> (truncate 26-21875)
26
0-21875
```

Когда вызывающий код требует одно значение, будет использовано первое:

```
> (= (truncate 26-21875) 26)
T
```

Вызывающий код может получить оба возвращаемых значения при помощи multiple-value-bind. Этот оператор принимает список переменных, вызова, а также программный код. Код выполняется с привязкой переменных к соответствующим возвращаемым вызовом значениям:

```
> (multiple-value-bind (int frac) (truncate 26-21875)
       (list int frac))
(26 0-21875)
```

Наконец, чтобы возвращать несколько значений, мы используем оператор values:

```
> (defun powers (x)
```

```
       (values x (sqrt x) (expt x 2)))
POWERS
> (multiple-value-bind (base root square) (powers 4)
       (list base root square))
(4 2-0 16)
```

Вообще, функциональное программирование – хорошая идея. И особенно хорошая в Lisp, так как Lisp развивался для того, чтобы её поддерживать. Встроенные операторы, такие как reverse и nreverse, предполагается использовать в таком ключе. Другие операторы, как values и multiple-value-bind, были разработаны как раз для того, чтобы сделать функциональное программирование легче.

3.2 3-2 Императивное программирование наизнанку

Возможно, цели функционального программирования станут более понятными в процессе сравнения с другим, более распространенным подходом – императивным программированием. Функциональная программа говорит вам то, что она хочет; императивная же говорит вам, что делать. Функциональная программа говорит: <<Вернуть список, состоящий из a и квадрата первого элемента x>>:

```
(defun fun (x)
  (list 'a (expt (car x) 2)))
```

Императивная программа говорит: <<Взять первый элемент x, затем возвести его в квадрат, затем вернуть список, состоящий из a и квадрата>>:

```
(defun imp (x)
  (let (y sqr)
       (setq y (car x))
       (setq sqr (expt y 2))
       (list 'a sqr)))
```

Пользователям Lisp повезло в том, что они могут написать эту программу любым из двух способов. Некоторые языки приспособлены только для императивного программирования, среди большинства языков программирования особенно выделяется Basic. В действительности, определение imp по внешнему виду напоминает машинный код, который большинство компиляторов генерируют, вероятно, ради забавы.

Зачем писать такой код, который может сделать за вас компилятор? Для многих программистов этот вопрос даже никогда не встает. Язык накладывает шаблон на наши мысли: кто-то, кто привык программировать на императивном языке, скорее всего, начнет планировать программу в императивных выражениях и наверняка обнаружит, что написание императивных программ легче, чем функциональных. Имеет смысл перешагнуть через этот стереотип, если у вас есть язык, позволяющий это сделать.

Те, кто изучил другие языки, начав использовать лисп, будут походить на ступивших в первый раз на каток. Вообще-то, если использовать коньки, то намного легче выйти на лед, чем на землю. До тех пор вам останется лишь гадать о том, что люди находят в этом спорте.

Функциональное программирование означает для Lisp то же, что и коньки для льда. Вместе они позволят передвигаться более изящно и с меньшими усилиями. Но

если вам привычен иной способ перемещения, то этот не будет вашим первым опытом. Одно из препятствий к изучению Lisp в качестве второго языка заключается в том, чтобы научиться программировать в функциональном стиле.

К счастью, есть хитрость, позволяющая преобразовать императивные программы в функциональные. Можно начать с применения этой хитрости к законченному коду. Вскоре вы начнете себя предвосхищать и преобразовывать код по мере его написания. Вслед за этим, вы начнете думать о программах в функциональном плане с самого начала.

Хитрость заключается в том, чтобы понять, что императивная программа представляет собой вывернутую на изнанку функциональную. Чтобы отыскать запрятанную функциональную программу, мы просто вывернем ее еще раз. Давайте попробуем этот метод на imp.

Первое, что мы замечаем – это создание y и sqr в начальном let. Это признак того, что плохие вещи впереди. Как eval во время выполнения, неинициализированные переменные необходимы настолько редко, что они в большинстве случаев должны рассматриваться как симптом некоторых заболеваний программы. Такие переменные часто используются как булавки, удерживающие программу от скручивания в естественную форму.

Несмотря на это, мы на время оставим их в покое и перейдем прямо в конец функции. То, что в императивной программе происходит последним, в функциональной происходит вначале. Таким образом, наш первый шаг – захватить последний вызов в список и начать набивать туда оставшуюся часть программы в точности, как и выворачивание рубашки наизнанку. Мы продолжим многократно применять такое же преобразование так же, как поступили бы с рукавами рубашки, а затем и с манжетами.

Начиная с конца, мы переместим sqr вместе с (expt y 2):

```
(list 'a (expt y 2)))
```

Затем заменим y на (car x):

```
(list 'a (expt (car x) 2))
```

Теперь мы можем выкинуть оставшийся код, который полностью вошел в последнее выражение. В ходе этого процесса мы убрали необходимость в переменных y и sqr, так что мы можем также избавиться и от let.

Окончательный результат короче, чем тот, с которого мы начинали, и легче для понимания. В исходном коде мы столкнулись с последним выражением (list 'a sqr), но не сразу понятно, откуда берется значение sqr. Теперь источник возвращаемого значения открыт нашему взору, как дорожная карта.

Пример в этом разделе был коротким, но технология масштабируется. В самом деле, она становится более ценной, если применяется к функциям большего размера. Даже функции, осуществляющие побочные действия, можно разобрать на части, которые не будут этого делать.

3.3 3-3 Функциональные интерфейсы

Некоторые побочные действия хуже, чем другие. Например, хотя эта функция вызывает nconc

```
(defun qualify (expr)
  (nconc (copy-list expr) (list 'maybe)))
```

она оставляет ссылочную прозрачность. .[2] Если вы вызовете ее с заданным аргументом, то она всегда будет возвращать такое же (равное) значение. С точки зрения вызывающего, qualify также может считаться чисто функциональным кодом. Но нельзя сказать того же о bad-reverse (стр. 29), которая фактически модифицирует свои аргументы.

Вместо того, чтобы рассматривать побочные действия как одинаково плохие, было бы полезно, если бы мы каким-то образом могли находить различия между такими случаями. Неформально можно сказать, что для функции безвредно модифицировать что-то, чем никто не владеет. Например, nconc в qualify является безвредной потому, что список, переданный в качестве первого аргумента, только что создан (consed). Больше никто не может быть его владельцем.

В общем случае, следует говорить не о том, чем владеют сами функции, а о том, чем владеют вызовы функций. Хотя здесь никто не владеет переменной x:

```
(let ((x 0))
  (defun total (y)
        (incf x y)))
```

эффекты одного вызова будут видны в последующих. Поэтому правило должно быть таким: данный вызов может безопасно изменять лишь то, чем владеет только он.

Кто владеет аргументами и возвращаемыми значениями? По соглашению в Lisp считается, что вызов владеет объектами, полученными в качестве возвращаемых значений, но не объектами, переданными ему в качестве аргументов. Функции, изменяющие свои аргументы, определяются с помощью метки "деструктивный", но нет никакого специального названия для функций, изменяющих возвращенные им объекты.

Например, эта функция придерживается соглашения:

```
(defun ok (x)
  (nconc (list 'a x) (list 'c)))
```

Она вызывает nconc, который не придерживается [соглашения], но поскольку список, сформированный nconc будет всегда новым, в отличие, например, от списка, переданного ok в качестве аргумента, ok сам по себе в порядке.

Тем не менее, если она была бы написана несколько иначе,

```
(defun not-ok (x)
  (nconc (list 'a) x (list 'c)))
```

вызов nconc изменил бы аргумент, переданный not-ok.

Многие Lisp программисты нарушают это соглашение, по крайней мере локально. Однако, как мы видели в ok, локальные нарушения не должны делать негодной вызывающую функцию. И функции, действительно отвечающие предыдущим условиям, будут содержать в себе множество преимуществ чисто функционального кода.

[2] Определение ссылочной прозрачности приводится на странице 198

Чтобы писать программы, действительно не отличимые от функционального кода, мы должны добавить еще одно условие. Функции не могут совместно использовать объекты с остальным кодом, который не следует следующим правилам. Например, хотя эта функция и не имеет побочных эффектов,

```
(defun anything (x)
      (+ x *anything*))
```

ее значение зависит от глобальной переменной *anything*. Так, если любая другая функция может изменять значение этой переменной, то anything может возвращать что угодно.

Код, написанный таким образом, что каждый вызов изменяет лишь только то, чем он владеет, почти так же хорош, как и чисто функциональный код. Функция, удовлетворяющая всем приведенным выше условиям, представляет миру функциональный интерфейс: если вызвать ее дважды с теми же аргументами, то вы должны получить одинаковые результаты. И это, как будет показано в следующем разделе, является важным ингредиентом в программировании снизу-вверх.

Одна из проблем с деструктивными операциями заключается в том, что, как и глобальные переменные, они могут разрушить локальность программы. Во время написания функционального кода, вы можете сузить свое поле зрения: необходимо рассмотреть лишь функции, которые вызывают ту или вызываются той, которую вы пишете. Данное преимущество исчезает, если вы хотите изменить что-то деструктивно. Это может быть использовано где угодно.

Приведенные выше условия не гарантируют, что вы получите идеальную локальность вместе с чисто функциональным кодом, хотя они несколько улучшают положение вещей. Например, предположим, что f вызывает g, как показано ниже:

```
(defun f (x)
      (let ((val (g x)))
       ; safe to modify val here?
       ))
```

Безопасно ли для f присоединить что-нибудь к val с помощью nconc? Нет, если g тождество(простое переименование переменных): тогда мы бы изменяли что-то, изначально переданное самой f в качестве аргумента.

Поэтому, даже в программах, которые следуют соглашению, нам, возможно, придется выйти за пределы f в том случае, если мы захотим там что-то изменить. Тем не менее, не придется идти слишком далеко: вместо того, чтобы беспокоиться о всей программе, теперь только лишь нужно рассмотреть поддерево, начиная с f.

Следствием приведенного выше соглашения является то, что те функции не должны возвращать ничего, что небезопасно изменять. Таким образом, следует избегать написания функций, возвращаемые значения которых включают в себя квотированные объекты. Если мы определим excalaim так, что его возвращаемые значения включают в себя квотированный список, то

```
(defun exclaim (expression)
  (append expression '(oh my)))
```

Тогда любое последующее деструктивное изменение возвращаемого значения

```
> (exclaim '(lions and tigers and bears))
```

```
(LIONS AND TIGERS AND BEARS OH MY)
> (nconc * '(goodness))
(LIONS AND TIGERS AND BEARS OH MY GOODNESS)
```

может изменить список внутри функции!!!:

```
> (exclaim '(fixnums and bignums and floats))
(FIXNUMS AND BIGNUMS AND FLOATS OH MY GOODNESS)
```

Для того чтобы сделать exclaim защищенной от такого рода проблем, ее следовало написать так:

```
(defun exclaim (expression)
  (append expression (list 'oh 'my)))
```

Существует одно важное исключение из того правила, что функции не должны возвращать квотированные списки: это функции, которые генерируют макрорасширения. Макрорасширители могут без проблем включать квотированные списки в порождаемые ими расширения в том случае, если макрорасширения прямо передаются компилятору.

С другой стороны, можно было бы также с подозрением относиться к квотированным спискам в целом. Можно придумать им и множество других применений, которые скорее всего будут выполнены с помощью макросов наподобие in.

3.4 3-4 Интерактивное программирование

Предыдущие разделы представляли функциональное программирование как хороший способ организации программ. Но это нечто большее. Lisp-программисты приняли функциональный стиль не только из эстетических соображений. Они используют его потому, что он облегчает им работу. В динамическом окружении Lisp функциональные программы могут быть написаны с необыкновенной быстротой и в то же время необычайно надёжны.

В Lisp относительно легко отлаживать программы. Очень много информации, помогающей найти причины возникновения ошибок, можно получить во время выполнения. Но еще более важным является легкость, с которой вы можете тестировать программы. Нет необходимости компилировать программу и тестировать все целиком за один раз. Вы можете протестировать функции по отдельности, вызывая их из цикла верхнего уровня.

Инкрементное тестирование настолько значимо, что для того, чтобы получить выигрыш от его использоывания, Lisp стиль эволюционировал. Программы, написанные в функциональном стиле, можно воспринимать по одной функции за раз, и с точки зрения читающего, это и есть его главное достоинство. Тем не менее, функциональный стиль прекрасно приспособлен к инкрементному тестированию: программы, написанные в этом стиле, можно также протестировать по одной функции за раз. Когда функция не анализирует и не изменяет внешнего состояния, любые ошибки немедленно проявятся. Такая функция может влиять на внешний мир только через свои возвращаемые значения. В тех же пределах, в каких вы полагаетесь на эти значения, вы можете доверять вернувшему их коду.

Опытные Lisp-программисты фактически делают свои программы удобными для тестирования:

1. Они пытаются отделить побочные действия в несколько функций, позволяющих большей части программы быть написанной в чисто функциональном стиле.
2. Если функция должна осуществлять побочные действия, они по крайней мере пытаются дать ей функциональный интерфейс.
3. Они дают каждой функции единственную, четко определенную цель.

Когда функция написана, они могут проверить ее на выборке из типичных случаев, а затем перейти к следующей. Если каждый кирпич делает то же, что и предполагалось делать, то стена будет стоять.

С таким же успехом в Lisp стену можно спроектировать лучшим образом. Представьте себе, что если бы вы разговаривали с кем-то, находящимся настолько далеко, что задержки передачи составляли бы одну минуту. Теперь представьте разговор с кем-то, находящимся в соседней комнате. Это бы не сделало прежний разговор быстрее, это был бы другой вид разговора. В Lisp разработка программ похожа на разговор лицом к лицу. Можно проверять код по мере его написания. И стоит вам отвернуться, и это возымеет такие же резкие последствия на разработку, какие оказало бы на беседу. Вы не просто пишете ту же программу быстрее, вы пишете программу иного вида.

Как так? Когда тестирование быстрее, его можно делать чаще. В Lisp, как и в любом языке, разработка - это цикл написания и тестирования. Но в Lisp этот цикл очень короткий: одиночная функция или части функций. А если проверять все по мере написания, то в случае возникновения ошибки, вы будете знать, где искать: в конце написанного. Как бы просто это ни выглядело, но именно этот принцип, в большей степени, позволяет программированию снизу-вверх быть возможным. Это приносит дополнительную степень уверенности, позволяющую Lisp-программистам свободно отдыхать, по крайней мере часть времени, от старого стиля разработки программного обеспечения <<спланируй и реализуй>>.

В раздел 1.1 отмечено, что проектирование снизу-вверх – эволюционный процесс. Вы создаете язык по мере написания на нем программы. Такой подход может работать, только если вы доверяете нижним слоям кода. Если вы действительно хотите использовать этот слой как язык, то вы должны уметь допускать, как и в случае с любым языком, что любые обнаруженные ошибки – это ошибки приложения, а не самого языка.

Итак, ваши новые абстракции должны нести это тяжелое бремя ответственности, и все же вы собираетесь создавать на их основе новые? Именно так, в Lisp это возможно одновременно. Когда вы пишете программы в функциональном стиле и инкрементно тестируете их, можно получить гибкость создания вещей под влиянием момента вдобавок к некоторой надежности, обычно ассоциируемой с тщательным планированием.

4 4 Сервисные Функции

Операторы Common Lisp бывают трех типов: функции и макросы, которые вы можете написать, и специальные формы, которые вы написать не можете. Эта глава описывает способы расширения Lisp новыми функциями. Но ‘способы‘ здесь значат нечто отличное от того, что они обычно значит слово ‘способы‘. Важно знать о функциях не то, как они написаны, но откуда они появились. Расширения Lisp должны быть написаны в-основном тем же способом, каким обычно пишутся любые другие функции в Lisp. Сложность написания расширением не в том, как писать их, а в том, что бы решить что именно писать.

4.1 4-1 Рождение Утилиты(сервисной функции)

В простейшем виде, программирование снизу вверх подразумевает попытку обмануть создателей Lisp. В то же время, когда вы пишите вашу программу вы так же добавляете в Lisp новые операторы, которые делают вашу программу более простой в написании. Эти новые операторы называются утилитами (стандартные либы/API внутри программы)

Термин ‘утилита‘ не имеет точного определения. Кусок кода может быть назван утилитой, если он слишком маленький, чтобы быть отдельным приложением и при этом слишком общеупотребителен, чтобы быть частью одной программы. Например, база данных не может быть утилитой, но функция, которая производит некую операцию над списком может. Большинство утилит напоминают функции и макросы, которые уже есть в Lisp. Фактически, многие встроенные в CL операторы начали свою жизнь как утилиты. Функция remove-if-not, которая возвращает все элементы, удовлетворяющие некоторому условию, была определена программистами за годы до того, как она стала частью CL.

Обучение написанию утилит может быть лучше описано как привитие привычки писать их, а не описание способов написания их. Программирование снизу вверх означает одновременное написание программы и языка программирования. Для того, чтобы сделать это хорошо, вы должны ясно представить, каких операторов в программе не хватает. Вы должны быть способны взглянуть на программу и сказать: ‘Ого, да то, что ты имел ввиду, заключается в таком-то алгоритме¡

Например, представьте что никнэйм - это функция которая получает имена и преобразует их в список никнэймов. Имея эту функцию, как мы соберём все никнеймы, полученные из списка имен? Некто изучающий Lisp может написать эту функцию подобным образом:

```
(defun all-nicknames (names)
  (if (null names)
          nil
          (nconc (nicknames (car names))
                 (all-nicknames (cdr names)))))
```

Более опытный Lisp-программист может посмотреть на подобную функцию и сказать ‘Кхм..., то что тебе действительно нужно так это mapcan.‘ В результате вместо написания и вызова новой функции нахождения всех никнеймов группы людей, ты сможешь использовать одно выражение:

```
(mapcan #'nicknames people)
```

Определение all-nicknames это изобретение велосипеда. Кстати, это не единственный косяк этой функции: она также прячет в специальной функции нечто, что может быть сделано оператором общего назначения.

В этом случае оператор, mapcan, уже существует. Любой, кто знает о mapcan чувствует себя некомфортно, смотря на all-nicknames. Хорошо программировать снизу вверх значит чувствовать такой же дискомфорт, когда необходимый оператор ещё не написан в стандартной библиотеке. Вы должны уметь сказать ‘то что тебе действительно нужно, это X‘ и в тоже время знать, что из себя представляет X.

Программирование на Lisp, кроме всего прочего, влечёт за собой водоворот новых утилит, создаваемых при первой же необходимости. Цель этого раздела - показать как такие утилиты рождаются. Представим, что towns - список близлежащих городов, отсортированных от ближнего к дальнему и bookshops - функция, которая возвращает список всех книжных магазинов в городе. Если мы хотим найти ближайший город, в котором есть хоть один книжный магазин, и вернуть полученную информацию о городе и магазинах, мы можем начать с:

```
(let ((town (find-if #'bookshops towns)))
  (values town (bookshops town)))
```

Но это немного коряво: когда find-if находит элемент, для которого bookshops возвращает не nil значение, значение выбрасывается наружу, а затем мы вновь производим операцию bookshops. Если вызов bookshops требует больших ресурсов, эта идиома будет неэффективна и убога. Для избежания такого, мы можем использовать следующую функцию:

```
(defun find-books (towns)
      (if (null towns)
           nil
           (let ((shops (bookshops (car towns))))
             (if shops
                   (values (car towns) shops)
                   (find-books (cdr towns))))))
```

Вызов (find-books towns) будет возвращать как минимум то, что нам надо, без лишних расчётов. Но подождите, мы же наверняка в будущем опять захотим выполнить подобный тип поиска? Да, то, что действительно нужно, так это утилита, которая, совмещая find-if и нечто, возвращает искомый элемент и значение проверочной функции. Вот как может выглядеть такая утилита:

```
(defun find2 (fn lst)
      (if (null lst)
           nil
           (let ((val (funcall fn (car lst))))
             (if val
                   (values (car lst) val)
                   (find2 fn (cdr lst))))))
```

Необходимо отметить сходство между find-books и find2. Фактически, find2 можно воспринимать как скелетон find-books. Сейчас, используя новую утилиту, мы можем добиться нашей изначальной цели в одно выражение:

```
(find2 #'bookshops towns)
```

Одно из уникальных свойств Lisp программирования состоит в важной роли использования функции в качестве аргумента. Это часть причины, по которой Lisp хорошо адаптирован к программированию снизу вверх. Проще написать каркас функции, когда ты можешь передать часть начинки в неё в качестве функции-аргумента.

Вводные курсы программирования ранее учили, что абстрагирование позволяет избежать дублирования кода. Один из первых уроков: не будьте прямолинейным. Например, вместо определения двух функций, которые делают одно и тоже, но отличаются одной-двумя константами, определите одну функцию и передавайте константы как аргументы.

В Lisp мы можем развить эту идею дальше, потому как в качестве аргумента мы можем передавать не только данные, но и код (функции). В предыдущих примерах мы проходим путь от конкретной функции к более общей, которая в качестве аргумента получает другую функцию. В первом случае мы используем предопределённый mapcan; во втором мы пишем новую утилиту, find2, но общий принцип такой же: вместо смешивание общего и частного, определяем общее и передаём частное в качестве аргумента.

При аккуратном использовании этот принцип порождает более элегантные программы. Это не единственная сила, поддерживающая архитектуру снизу вверх, но одна из основных. Из 32 утилит, определённых в этой главе, 18 получают аргументом функцию.

4.2 4-2 Инвестируй в абстракции

Если краткость - сестра таланта, тогда эффективность это сущность хорошего софта. Цена написания и поддержки программы растет вместе с её размером. При прочих равных условиях более короткая программа лучше.

С этой точки зрения, написание утилит можно рассматривать как капитальные расходы. Заменой find-books на find2 мы получили столько же строк кода. Но мы сделали программу короче в некотором смыссле, потому как длина утилиты не добавляется повторно в текущую программу.

Это не только трюк подсчета, считать расширения Lisp капитальными вложениями. Утилиты могут быть расположены в отдельном файле; они не будут засорять наш взор пока мы работаем над программой, да и не будут мешать впоследствии, если мы вернемся к программе позже.

Как капитальные вложения, конечно же, утилиты требуют дополнительного внимания. Это особенно важно, что бы они были хорошо написаны. Они будут использоваться неоднократно, а значит любая некорректность или неэффективность будет преумножена. Дополнительное внимание так же должно быть уделено их проектированию: новая утилита должна быть написана для общего случая, не для текущей проблемы. В конце-концев, как и с любым другим капитальное вложением, мы не должны спешить с ней. Если Вы думаете о каком-то новом операторе, но не уверены что он понадобится где-то еще, напишите его все равно, но оставьте его в той программе, в которой он используется. Позже если вы будете использовать его в других программах вы можете преобразовать его в утилиту и открыть к нему общий доступ.

Утилита find2 кажется неплохой инвестицией. Сделав вложение в 7 строк мы тут же сохранили столько же. Утилита окупила себя при первом использовании. Язык, написанный Гаем Стилом должен "соотносится с нашим природным стремлением к краткости:"

> ...мы склонны полагать, что издержки программной конструкции пропорциональны количеству усилий писателя ("верить, полагать" - здесь я имею в виду, неосознанную тенденцию, а не горячую убежденность). Действительно, это неплохой психологический принцип для разработчиков языков, помнить о нем. Мы думаем о сложении, как о дешевом, отчасти потому, что мы можем записать его одним символьным знаком: "+". Даже если мы считаем, что эта конструкция стоит дорого, мы часто предпочитаем ее, если она наполовину сократит наши усилия по написанию кода.

В любом языке "тенденция к краткости" будет вызывать проблемы, если язык не позволяет выражать себя в новых утилитах. Самые краткие идиомы редко бывают самыми эффективными. Если мы хотим знать какой из двух списков длиннее другого, чистый Lisp склоняет нас к написанию

```
(> (length x) (length y))
```

Если мы хотим map функцию для нескольких списков, мы должны таким же образом написать:

```
(mapcar fn (append x y z))
```

Такие примеры показывают, что очень важно писать утилиты для ситуаций, которые мы иным способом можем решить неэффективно. Язык, расширенный правильными утилитами должен направлять нас к написанию более абстрактных программ. Если эти утилиты правильно определены, это так же будет способствовать к написанию более эффективных программ.

Набор утилит несомненно сделает программирование проще. Но утилиты также могут сделать больше: вы можете начать писать лучшие программы. Музы(Вдохновения), подобно тому как, повара переходят к делу при виде ингридиенты. По этой причине люди искусства любят иметь большое количество инструментов и материалов в своих студиях. Они знают, что проще начать что-то новое, если они имеют то, что им нужно. Тот же феномен происходит с программами, которые написаны снизу вверх. Однажды написав новую утилиту, вы можете обнаружить, что вы используете её чаще, чем вы ожидали.

Следующие разделы описывают некоторые классы функций-утилит. Это ни в коем случае не значит, что они показывают все возможные типы функций, которые вы можете добавить в Lisp. Как бы то ни было, все эти утилиты даны как образцы функций, которые доказали свою пользу на практике.

4.3 4-3 Операции над списками

Списки изначально главная структура данных Lisp. Не зря же ‘Lisp‘ расшифровывается как "LISt Processing". Однако не стоит обманываться этим историческим фактом. Lisp по своей сути предназначен для обработки списков не более чем рубашка поло для игры в поло. Хорошо оптимизированная программа на Common Lisp может не использовать списки.

Это будут списки хотя бы во время компиляции. Самые изощренные программы, которые используют списки меньше во время выполнения используют их пропорционально больше

```
(proclaim '(inline last1 single append1 conc1 mklist))

(defun last1 (lst)
       (car (last lst)))

(defun single (lst)
       (and (consp lst) (not (cdr lst))))

(defun append1 (lst obj)
       (append lst (list obj)))

(defun conc1 (lst obj)
       (nconc lst (list obj)))

(defun mklist (obj)
       (if (listp obj) obj (list obj)))
```

Рисунок 4-1: Маленькие функции работающие со списками.

во время компиляции, когда происходит разворачивание макросов. Так что несмотря на то, что роль списков уменьшается в новейших диалектах, операции над списками может всё еще составлять большую часть программы на Lisp

Figures 4.1 и 4.2 содержат выборку функций, которые создают или просматривают списки. Те, которые в Figure 4.1 самые маленькие из полезных. Для эффективности, они должны все быть определены как inline (стр. 26)

Первая функция, last1, возвращает последний элемент в списке. Встроеная функция last возвращает последний cons в списке, но не последний элемент. Большую часть времени last используется для получения последнего элемента путем (car (last ...)). Если толк в написании новой утилиты для такого случая? Да, когда он эффективно заменяет один из встроенных операторов.

Отметьте, что last1 не проводит проверок на ошибки. В общем, в этой книге нет кода, который проверяет на ошибки. Частично это потому, что это делает примеры яснее. Но в коротких утилитах имеет смысл вообще не делать проверок на ошибки. Если мы запустим:

```
> (last1 "blub")
>>Error: "blub" is not a list.
Поломка в LAST...
```

ошибка будет вызвана в last. Когда утилиты маленькие, слой абстракции настолько тонок, что становится прозрачным. Как возможно видеть сквозь тонкий слой льда, так же можно видеть сквозь утилиты для интерпретации ошибок, возникающих во внутренних вызовах.

Функция single проверяет состоит ли список из одного элемента. Программы на Lisp зачастую делают эту проверку часто. Первая попытка может быть попыткой перевести single с английского:

```
(= (length lst) 1)
```

Написанная таким образом, проверка будет очень неэффективной. Мы узнаем всё, что нам нужно как только попытаемся заглянуть далее первого элемента.

Следующими идут append1 and conc1. Оба добавляют новый элемент в конец списка, второй деструктивен. Эти функции маленькие, но так часто нужны что их стоит определить. Кстати, append1 была предопределена в предыдущих диалектах Lisp.

Так же есть mklist, которые уже есть (как минимум) в Interlisp. Его назначение в том, что бы гарантировать, что нечто - список. Многие функции Lisp возвращают и список или одно значение. Предположим, что lookup такая функция, и мы хотим собрать результаты её вызова для всех элементов списка ‘data‘. Мы можем сделать это написав:

```
(mapcan #'(lambda (d) (mklist (lookup d)))
               data)
```

Figure 4.2 содержит другие большие примеры списочных утилит. Первая, longer, удобна с точки зрения эффективности как и абстракции. Она сравнивает две последовательности и возвращает true если первая длиннее второй. Сравнивая длины двух списков, есть соблазнительная идея сделать это в лоб:

```
(> (length x) (length y))
```

Эта идиома неэффективна так как она требует прохождения обоих списков для нахождения длины. Если один из списков сильно длиннее другого, проход разницы в длине списков будет лишним. Быстрее проходить их параллельно.

Встроенная в longer рекурсивная функция compare сравнивает длины списков. Так как longer предназначен для сравнения длин, он должен работать со всем, что вы можете передать в length в качестве аргумента. Но возможность сравнения длин параллельно возможно только для списков, следовательно compare вызывается только если оба аргумента списки.

Следующая функция, filter, как remove-if-not к find-if. Встроенный метод remove-if-not возвращает все значения, которые могли бы быть возвращены, если вы вызвали find-if с той же функцией на последующие cdrs списка. Аналогично, filter возвращает то, что должно быть возвращено для последующих crds в списке:

```
(defun longer (x y)
      (labels ((compare (x y)
                    (and (consp x)
                           (or (null y)
                                (compare (cdr x) (cdr y))))))
        (if (and (listp x) (listp y))
              (compare x y)
              (> (length x) (length y)))))

(defun filter (fn lst)
      (let ((acc nil))
        (dolist (x lst)
           (let ((val (funcall fn x)))
              (if val (push val acc))))
        (nreverse acc)))

(defun group (source n)
      (if (zerop n) (error "zero length"))
      (labels ((rec (source acc)
                    (let ((rest (nthcdr n source)))
                       (if (consp rest)
                            (rec rest (cons (subseq source 0 n) acc))
                            (nreverse (cons source acc))))))
        (if source (rec source nil) nil)))
```

Рисунок 4-2: Большие функции работающие со списками.

```
> (filter #'(lambda (x) (if (numberp x) (1+ x)))
                '(a12b3cd4))
(2345)
```

Вы даете filter функцию и список, и получаете назад список не nil значений возвращенных функцией, применённой к элементам входного списка.

Заметьте, что filter использует аккумулятор тем же способом, как функции с хвостовой рекурсией, описанной в разделе 2.8. Цель написания функции с хвостовой рекурсией в том, что бы компилятор генерировал код в форме фильтра. Для filter, простая итеративная реализация проще, чем версия с хвостовой рекурсией. Комбинация push и nreverse в filter является стандартной идиомой Lisp для накопления значений в списке.

Последняя функция в Figure 4-2 предназначена для группировки списков в подсписки. Вы даете group список l и число n, и она вернет новый список, в котором элементы l сгруппированы в подсписки длины n. Остальная часть помещается в последний подсписок. Таким образом, если мы дадим 2 в качестве второго аргумента, мы получим ассоциативный список(assoc-list):

```
> (group '(abcdefg)2)
((A B) (C D) (E F) (G))
```

Эта функция написана довольно запутанно, чтобы сделать ее хвостовой рекусией (Section 2-8). Принцип быстрого прототипирования применим к индивидуальным

функциям, а также к целым программам. При написании такой функции, как flatten, может быть хорошей идеей начать с самой простой реализации. Потом, когда простая версия работает, вы можете заменить ее, при необходимости на более эффективной версией с хвостовой рекурсией или итеративной. Если она достаточно коротка, начальная версия может быть оставлена в качестве комментария для описания поведения ее замены. (Более простые версии group и некоторых других функций на Рисунках 4-2 и 4-3 включены в примечания на странице 389.)

Определение group необычно тем, что оно проверяет, по крайней мере одну ошибку: второй аргумент на 0, который в противном случае отправил бы функцию в бесконечную рекурсию.

В одном отношении, примеры в этой книге отличаются от обычной практики Лисп: сделать главы независимыми друг от друга, примеры кода возможными к написанию на чистом Лиспе. Поскольку эти функции так полезны при определении макросов, group как исключеченение, появиться в нескольких местах в следующих главах.

Все функции на рисунке 4-2 работают по структуре верхнего уровня списка. На Рисунке 4-3 показано два примера , которые спускаются во вложенные списки. Первый, flatten, также предопределен в Interlisp. Он возвращает список всех атомов, которые являются элементами списка, или элементами его элементов и так далее:

```
> (flatten '(a (b c) ((d e) f)))
(ABCDEF)
```

Другая функция на рисунке 4-3, prune, это как remove-if работающий вместе copy-tree при копировании списка. То есть, он рекурсивно спускается вниз в подсписки:

```
> (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
(1 (3 (5)) 7 (9))
```

Каждый лист для которого функция возвращает истину - удаляется.

4.4 4-4 Поиск

В этом разделе приведены некоторые примеры функий для поиска в списках. Common Lisp предоставляет для этого богатый набор встроенных операторов. но некоторые задачи

```
(defun flatten (x)
      (labels ((rec (x acc)
                    (cond ((null x) acc)
                          ((atom x) (cons x acc))
                          (t (rec (car x) (rec (cdr x) acc))))))
         (rec x nil)))

(defun prune (test tree)
      (labels ((rec (tree acc)
                    (cond ((null tree) (nreverse acc))
                          ((consp (car tree))
                            (rec (cdr tree)
                                 (cons (rec (car tree) nil) acc)))
                          (t (rec (cdr tree)
                                  (if (funcall test (car tree))
                                      acc
                                      (cons (car tree) acc))))))))
         (rec tree nil)))
```

Рисунок 4-3: Утилиты двойной рекурсии для работы со списками.

все еще трудно, или по крайней мере, трудно выполнить эффективно. Мы видели это в гипотетическом случае описанном на стр. 41. Первая утилита на рисунке 4-4, find2, так которую мы определеили в в ответ на это.

Следующая утилита, before, написана с похожими намерньями. Она говорит вам, если один объект найден перед другим в списке:

```
> (before 'b 'd '(abcd))
(BCD)
```

Это довольно легко сделать в чистом Лиспе:

```
(< (position 'b '(a b c d)) (position 'd '(a b c d)))
```

Но последняя идиома неэффективна и подвержена ошибкам: неэффективна, поскольку нам не нужно искать оба объекта, только тот который появляется первым; и подвержена ошибкам, поскольку если какого-либо объекта нет в списке, nil будет передан в качестве аргумента <. Использование before решает обе эти проблемы.

Так как before похож на проверку для membership, он написан похоже на встроенную функцию member. Подобно member он принимает необязательный аргумент test, который по умолчанию является eql. Кроме того, вместо простого возврата t, он пытается

```
(defun find2 (fn lst)
      (if (null lst)
          nil
          (let ((val (funcall fn (car lst))))
             (if val
                  (values (car lst) val)
                  (find2 fn (cdr lst))))))

(defun before (x y lst &key (test #'eql))
      (and lst
           (let ((first (car lst)))
              (cond ((funcall test y first) nil)
                       ((funcall test x first) lst)
                       (t (before x y (cdr lst) :test test))))))

(defun after (x y lst &key (test #'eql))
      (let ((rest (before y x lst :test test)))
       (and rest (member x rest :test test))))

(defun duplicate (obj lst &key (test #'eql))
      (member obj (cdr (member obj lst :test test))
                 :test test))

(defun split-if (fn lst)
      (let ((acc nil))
       (do ((src lst (cdr src)))
             ((or (null src) (funcall fn (car src)))
              (values (nreverse acc) src))
          (push (car src) acc))))
```

Рисунок 4-4: Функции которые ищут списки.

вернуть потенциально полезную информацию: cdr начинающийся с объекта, заданного как первый аргумент.

Обратите внимание, что before возвращает истину, если мы находим первый аргумент перед вторым. Таким образом, он вернет истину, если второй аргумент не появляется вообще в списке:

```
> (before 'a 'b '(a))
(A)
```

Мы можем выполнить более точный тест, вызывав after, которые требует чтобы оба его аргуемента были в списке:

```
> (after 'a 'b '(b a d))
(A D)
> (after 'a 'b '(a))
NIL
```

Если (member o l) находит o в списке l, он также возвращает cdr из l начинающийся с o. Это возвращаемое значение можно использовать, например, для проверки

на дублирование. Если о дублируется в l, тогда он также будет найден в cdr списка возвращенного функцией member. Эта идиома воплощена в следующей утилите, duplicate:

```
> (duplicate 'a '(abcad))
(A D)
```

Другие утилиты для проверки на дублирование могут быть написаны по тому же принципу.

Более привередливые разработчики языка шокриованы тем, что Common Lisp использует nil для представления как лжи так и пустого списка. Иногда это вызывает проблемы (см Раздел 14-2), но это удобно в таких функциях, как duplicate. В запросах о членстве в последовательности, кажется естественным представлять лож как пустую последовательность.

Последняя функция на рисунке 4-4 также является своего рода обобщением member. Пока member возвращает cdr списка, начиная с найденного элемента, split-if возвращает обе половины. Эта утилита в основном используется со списками, которые в некотором отношении упорядочены:

```
> (split-if #'(lambda (x) (> x 4))
                  '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4)
(5 6 7 8 9 10)
```

Рисунок 4-5 содержит функции поиска другого типа: те которые сравнивают элементы друг против друга. Первая, most, смотрит на один элемент за раз. Она принимает список и функцию оценки, и возвращает элемент с наибольшим количеством очков. В случае ничьей, элемент, встреченный первым, выигрывает.

```
> (most #'length '((a b) (a b c) (a) (e f g)))
(ABC)
3
```

Для удобства, most также возвращает и счет победителя.

Более общий вид поиска обеспечивается функцией best. Эта утилита также принимает функцию и список, но здесь функция должна быть предикатом двух аргументов. Она возвращает элемент, который согласно предикато превосходит все остальные.

```
(defun most (fn lst)
 (if (null lst)
     (values nil nil)
     (let* ((wins (car lst))
              (max (funcall fn wins)))
       (dolist (obj (cdr lst))
            (let ((score (funcall fn obj)))
             (when (> score max)
               (setq wins obj
                     max score))))
       (values wins max))))

(defun best (fn lst)
 (if (null lst)
     nil
     (let ((wins (car lst)))
       (dolist (obj (cdr lst))
            (if (funcall fn obj wins)
               (setq wins obj)))
       wins)))

(defun mostn (fn lst)
 (if (null lst)
     (values nil nil)
     (let ((result (list (car lst)))
              (max (funcall fn (car lst))))
       (dolist (obj (cdr lst))
            (let ((score (funcall fn obj)))
             (cond ((> score max)
                    (setq max        score
                          result (list obj)))
                   ((= score max)
                    (push obj result)))))
       (values (nreverse result) max))))
```

Рисунок 4-5: Функции поиска которые сравнивают элементы.

```
> (best #'> '(1 2 3 4 5))
5
```

Мы можем думать о best как о эквиваленте car после sort, но более эффективном.

Вызывающий должен предоставить предикат, который задает общий порядок на элементах списка. В противном случае порядок элементов будет влиять на результат; как и для before, в случае ничьей выигрывает первый встреченный элемент.

Наконец, mostn берет функцию и список и возвращает список всех элементов для которых функция выдает высший бал (вместе с самим баллом):

```
> (mostn #'length '((a b) (a b c) (a) (e f g)))
((A B C) (E F G))
```

```
3
```

4.5 4-5 Отображение

Другим широко используемым классом функций Lisp являются функции отображения (mapping functions), которые применяют функцию к последовательности аргументов. Рисунок 4-6 показывает несколько примеров новых функций отображения. Первые три предназначены для применения функции к диапазону чисел без необходимости составлять список, чтобы хранить их. Первые две, map0-n и map1-n, работают для диапазона положительных целых чисел:

```
> (map0-n #'1+ 5)
(123456)
```

Обе написаны с использованием более общей функции mapa-b, которая работает для любого диапазона номеров:

```
> (mapa-b #'1+ -2 0 .5)
(-1 -0-5 0-0 0-5 1-0)
```

Следующая за mapa-b более общая функция map->, которая работает для последовательностей объектов любого вида. Последовательность начинается с объекта, данного как второй аргумент, конец последовательности определяется функцией заданной как третий аргумент, приемники (последующие объекты) генерируются функцией, заданной в качестве четвертого аргумента. С map-> можно перемещаться по произвольным структурам данных, а также оперировать последовательностями чисел. Мы могли бы определить mapa-b в терминах map-> следующим образом:

```
(defun mapa-b (fn a b &optional (step 1))
     (map-> fn
               a#'(lambda (x) (> x b))
               #'(lambda (x) (+ x step))))
```

```
(defun map0-n (fn n)
     (mapa-b fn 0 n))

(defun map1-n (fn n)
     (mapa-b fn 1 n))

(defun mapa-b (fn a b &optional (step 1))
     (do ((i a (+ i step))
          (result nil))
         ((> i b) (nreverse result))
       (push (funcall fn i) result)))

(defun map-> (fn start test-fn succ-fn)
     (do ((i start (funcall succ-fn i))
          (result nil))
         ((funcall test-fn i) (nreverse result))
       (push (funcall fn i) result)))

(defun mappend (fn &rest lsts)
     (apply #'append (apply #'mapcar fn lsts)))

(defun mapcars (fn &rest lsts)
     (let ((result nil))
       (dolist (lst lsts)
         (dolist (obj lst)
            (push (funcall fn obj) result)))
       (nreverse result)))

(defun rmapcar (fn &rest args)
     (if (some #'atom args)
         (apply fn args)
         (apply #'mapcar
                #'(lambda (&rest args)
                    (apply #'rmapcar fn args))
                args)))
```

Рисунок 4-6: Отображающие функции.

Для эффективности, встроенный mapcan является разрушающим. Он может быть дублирован как:

```
(defun our-mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))
```

Поскольку mapcan объединяет списки с помощью nconc, списки возвращаемые первым аргументом должны быть заново созданы, или в когда в следующий раз мы посмотрим на них, они могут быть изменены. Вот почему функция nicknames (стр 41) была определена как функция которая "строит список" псевдонимов(nicknames). Если она просто вернет хранящийся где-либо список, было бы не безопасно использовать mapcan. Вместо этого нам пришлось бы склеить возвращенные списки с использова-

нием append. Для таких случаев mappend предлагает неразрушающую альтернативу mapcan.

Следующая утилита, mapcars, предназначена для случаев, когда мы хотим отобразить mapcar функцию по нескольким спискам. Если у нас есть два списка чисел и мы хотим получить один список содержащий квадратный корни чисел из обоих этих списков, используя чистый Lisp мы могли бы сказать

```
(mapcar #'sqrt (append list1 list2))
```

но это создание нового списка append бессмысленно. Мы склеиваем list1 и list2 вместе только для того чтобы отменить результат немедленно. С mapcars мы можем получить тот же результат из:

```
(mapcars #'sqrt list1 list2)
```

и не делает ненужных созданий списков.

Последняя функция на рисунке 4-6 это версия mapcar для деревьев. Ее имя, rmapcar, сокращение для "рекурсивный mapcar," и то что mapcar делает на плоских списках, она делает на деревьях:

```
> (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
123456789
(12(34(5)6)7(89))
```

Как и mapcar, она может принимать более одного аргумента списка.

```
> (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
(11 (22 (33) 44))
```

Некоторые из функций, которые появяться позже, должны на самом деле вызывать rmapcar, включая rep на стр. 324.

В некоторой степени, традиционные функции отображения списков могут быть устаревшими макросами новой серии, представленныой в CLTL2. Например,

```
(mapa-b #'fn a b c)
```

можно представить

```
(defun readlist (&rest args)
     (values (read-from-string
                 (concatenate 'string "("
                                                  (apply #'read-line args)
                                                  ")"))))

(defun prompt (&rest args)
     (apply #'format *query-io* args)
     (read *query-io*))

(defun break-loop (fn quit &rest args)
     (format *query-io* "Entering break-loop.~%")
     (loop
        (let ((in (apply #'prompt args)))
          (if (funcall quit in)
                (return)
                (format *query-io* "~A~%" (funcall fn in))))))
```

Figure 4-7: Функции ввода/вывода(I/O functions).

```
(collect (#Mfn (scan-range :from a :upto b :by c)))
```

Тем не менее, есть еще некоторые вызовы для функций отображения. Функция отображения может в некоторых случаях быть более ясной и более элегантной. Некоторые вещи, которые можно выразить с map-> может быть трудно выразить с помощью series. Наконец, функции отображения, могут быть переданы в качестве аргументов.

4.6 4-6 Ввод/Вывод

Рисунок 4-7 содержит три примера утилит ввода/вывода. Необходимость такого рода утилит варьируется от программы к программе. Те что на Рисунке 4-7 являются просто представительными образцами. Первая - для случая, когда вы хотите, чтобы пользователи могли вводить выражения без скобок; она читает строку ввода и возвращает ее в виде списка:

```
> (readlist)
Call me "Ed"
(CALL ME "Ed")
```

Вызов values гарантирует что мы получаем только одно значение назад (read-from-string само возвращает второе значение которое не уместно в этом случае).

Функция prompt сочетает печать вопроса и чтение ответа. Она принимает аргументы для функции format, кроме начального аргумента потока(stream).

```
> (prompt "Enter a number between ~A and ~A.~%>> " 1 10)
Enter a number between 1 and 10.
>> 3
3
```

Наконец, break-loop предназначен для ситуаций, когда вы хотите имитировать верхний урвоень Lisp. Она принимает две функции и аргумент &rest, который по-

вторно передается функции prompt. Пока вторая функция возвращает ложь для полученного ввода, первая функция применяется к нему(к введенному значению). Так, например, мы могли бы имитировать настоящий верхний уровень Lisp использовав вызов:

```
> (break-loop #'eval #'(lambda (x) (eq x :q)) ">> ")
Entering break-loop.
>> (+ 2 3)
5>> :q
:Q
```

Это кстати причина, по которой производители Common Lisp обычно настаивают на лицензии времени выполнения. Если вы можете вызвать eval во время выполнения, тогда любая Lisp программа может включать весь Lisp.

4.7 4-7 Символы и Строки

Символы и Строки тесно связаны. С помощью функций печати и чтения мы можем идти вперед и назад между этими двум представлениями. Рисунок 4-8 содержит примеры утилит, которые работают на этой границе. Первая, mkstr, принимает любое количество аргументов и объединяет их печатные представления в строку:

```
> (mkstr pi " pieces of " 'pi)
"3.141592653589793 pieces of PI"
```

На ней(mkstr) строиться symb, которая в основном используется для создания символов. Требует один или больше аргументов и возвращает символ (создавая его при необходимости), чье имя для печати это их(аргументов) объединение. Она может принимать в качестве аргумента любой объект, который имеет печатное представление: символы, строки, числа и даже списки.

```
> (symb 'ar "Madi" #\L #\L 0)
|ARMadiLL0|
```

```
(defun mkstr (&rest args)
  (with-output-to-string (s)
     (dolist (a args) (princ a s))))

(defun symb (&rest args)
  (values (intern (apply #'mkstr args))))

(defun reread (&rest args)
  (values (read-from-string (apply #'mkstr args))))

(defun explode (sym)
  (map 'list #'(lambda (c)
                 (intern (make-string 1:initial-element c)))
       (symbol-name sym)))
```

Рисунок 4-8: Функции которые оперирует с символами и строками.

После вызова mkstr для объединения всех ее аргументов в одну строку, symb отправляет строку в intern. Эта фунция Lisp традиционно является построителем символов: она принимает строку и либо находит символ, который печатается как эта строка, или создает новый.

Любая строка может быть напечатана как имя символа, даже строка содержащая символьный знаки нижнего регистра или макро-знаки, такие как скобки. Когда имя символа содержит такие странности, оно печатается внутри вертикальных полос, как показано выше. В исходном коде, такие символы должны быть заключены либо в вертикальные черты, либо неправильным символьным знакам должна предшествовать обратная косая черта.:

```
> (let ((s (symb '(a b))))
         (and (eq s '|(A B)|) (eq s '\(A\ B\))))
T
```

Следующая функция reread является обобщением symb. Она получает последовательность объектов и печатает их и перепрочитывает их. Она может возвращать символ подобно symb, но она может возвратить также все, что можно прочитать. Макрос чтения будет вызываться вместо рассматриваемого как часть имени символа(?), и a:b должно читаться как символ b в пакете a, вместо символа |a:b| в текущем пакете.[1] Чем более общая функция, тем она более придирчива: reread вызовет ошибку, если ее аргументы не имеют правильного Lisp синтаксиса.

Последняя функция на рисунке 4-8 была предопределена в некоторых ранних диалектах: explode берет символ и возвращает список символов, составленный из символьных знаков в его имени.

```
> (explode 'bomb)
(BOMB)
```

Не случайно эта функция не была включена в Common Lisp. Если вы хотите разбирать символы на части, то вы вероятно делаете чтото не эффективно. Тем не менее, есть место для такого рода утилиты в прототипах, если она не включается в результирующее программное обеспечение.

4.8 4-8 Компактность

Если в вашем коде используется много новых утилит, некоторые читатели могут жаловаться, что его трудно понимать. Люди которые еще не очень свободно говорят на Lisp , привыкли только чтению чистого Lisp. На самом деле, они не могут привыкнуть к идее расширяемого языка для всего. Когда они смотрят на программу, которая сильно зависит от утилит, для них может показаться, что автор из чистой неординарности, решил написать программу на каком-то частном языке.

Можно утверждать, что все эти новые операторы затрудняют чтение программы. Нужно понять их все, прежде чем можно будет прочитать программу. Чтобы увидеть, что такое утверждение ошибочно, рассмотрим случай описанный на стр. 41, в котором мы хотим найти ближайшие книжные магазины. Если вы написали программу, используя find2, кто то может пожаловаться, что он должен понять определение

[1] Для ознакомления с пакетами, см. Приложения для начинающих на стр. 381.

этой новой утилиты, прежде чем он сможет прочитать вашу программу. Ну предположим, что вы не использовали утилиту find2. Тогда, вместо того чтобы понимать определение find2, читателю пришлось бы понимать определение функции find-books, в которой функция find2 смешена со специфической задачей поиска книжного магазина. Понимание работы find-books не менее трудная задача чем понимани работы find2. И здесь мы имеем только одну новую утилиту. Утилиты предназначены для многократного использования. В реальной программе, это может быть выбором между пониманием find2, и пониманием трех или четырех специализированных алгоритмов поиска. Конечно первое легче.

Так что да, чтение программы спроектированной по восходящему принципу, требует понимания всех новых операторов, определенных автором. Но это почти всегда требует меньшще работы, чем необходимость понимания весего кода, который был бы необходим без них.

Если люди жалуются, что использование утилит затрудняет чтение вашего кода, они возможно не понимают, как бы выглядел код, без использования этих утилит. Восходящее программирование делает то, что иначе выглядело бы как большая программа, маленьким и простым. Поскольку код мал, это может создать впечатление, что программа не делает много, и поэтому должна легко читаться. Когда неопытные читатели изучают ближе код и обнаруживают что это не так, они реагируют с тревогой.

Мы находим такое же явление в других областях, хорошо продуманная машина может иметь меньше частей, и все же выглядеть сложнее, потому что она занимает меньше пространства. Восходящие программы концептуально более плотные. Это может потребовать усилий для их чтения, но не столько, сколько потребовалось бы, если бы они не были написаны таким образом.

Существует один случай, когда вы можете осознанно избегать утилит: если вы должны написать небольшую программу, которая будет распространяться независимо от остального вашего кода. Утилита, обычно окупается после двух или трех использований, но в небольшой программе, утилита может быть недостаточно использована, чтобы оправдать ее включение.

5 5 Возврат Функций

Предыдущая глава показала, как способность передавать функции в качестве аргументов приводит к большим возможностям для абстрагирования. Чем больше мы можем использовать функции, тем болье мы имеем возможностей. Определив функции для построения и возврата новых функций, мы можем увеличить эффект от утилит, которые принимают функции как аргументы.

Утилиты в этой главе работают с функциями. Это было бы более естественно, хотя бы в Common Lisp, чтобы написать множество из них для работы с выражениями, то есть как макросы. Слой макросов будет накладываться на некоторые из этих операторов в главе 15. Тем не менее, важно знать, какая часть задачи может быть сделана функциями, даже если мы в конечном итоге будем вызывать эти функции только через макросы.

5.1 5-1 Common Lisp Развивается

Common Lisp изначально предоставлял несколько пар взаимодополняющих функцй. Функции remove-if и remove-if-not составляют одну такую пару. Если pred является предикатом одного аргумента, то

```
(remove-if-not #'pred lst)
```

является эквивалентным

```
(remove-if #'(lambda (x) (not (pred x))) lst)
```

Изменяя функцию, заданную в качестве аргумента, мы можем продублировать эффект другой функции. В таком случае, почему существуют две функции? CLTL2 включает новую функцию предназначенную для случаев, подобных этому: функция complement принимает предикат p и возвращает функцию, которая всегда возвращает проитвоположное значение. Когда p возвращает истину, complement возвращает ложь, и наоборот. Теперь мы можем заменить

```
(remove-if-not #'pred lst)
```

с эквивалентом

```
(remove-if (complement #'pred) lst)
```

С complement, есть мало оснований для продолжения использования функций -if-not.[1]Действительно, CLTL2 (стр. 391) говорит, что их использование сейчас устарело. Если они остаются в Common Lisp, это делается только ради совместимости.

Новый оператор complement вершина важного айсберга: функций которые возвращают функции. Это давно стало важной частью идиом на Scheme. Scheme была первым Lisp-ом сделавшим функции лексическим замыканием, и это делает интересным использование функций в качестве возвращаемых значений.

Не то чтобы мы не могли возвращать функции в динамческом области охвата связывания Lisp. Следующая функция будет работать одинаково как с использованием динамической области охвата, так и с использованием лексического охвата связывания(контекта).

```
(defun joiner (obj)
```

[1] За исключением, возможно remove-if-not, которая используется чаще чем remove-if.

```
(typecase obj
  (cons       #'append)
  (number #'+)))
```

Она берет объект и в зависимости от его типа, возвращает функцию для добавления таких объектов. Мы могли бы использовать ее для определения полиморфной функции соединения(join), которая бы работала для чисел или списков:

```
(defun join (&rest args)
      (apply (joiner (car args)) args))
```

Тем не менее, возврат постоянных функций является пределом того, что мы можем сделать с динамической областью охвата. Что мы не можем сделать(хорошо), так это строить функции во время выполнения; joiner может вернуть одну из двух функций, но оба варианта фиксированные.

На странице 18 мы видели еще одну функцию для возврата функций, которая основывалась на лексической области охвата(лексическом связывании):

```
(defun make-adder (n)
      #'(lambda (x) (+ x n)))
```

Вызов make-adder приводит к созданию замыкания, поведение которого зависит от значения переданного как аргумент в функцию:

```
> (setq add3 (make-adder 3))
#<Interpreted-Function BF1356>
> (funcall add3 2)
5
```

В лексическом контексте(лексической области охвата), вместо простого выбора из группы константных функций, мы можем создавать новые замыкания во время выполнения. С динамической областью действия привязок эта техника недоступна.[2] Если мы посмотрим, как написан complement, мы увидим, что этот оператор возвращает замыкание:

```
(defun complement (fn)
     #'(lambda (&rest args) (not (apply fn args))))
```

Функция возвращаемая complement использует значение параметра fn, который был передан когда complement был вызван. Так что вместо того, чтобы просто выбирать из группы постоянных функций, complement может создавать обратные функции для любой переданной ей функции:

```
> (remove-if (complement #'oddp) '(123456))
(135)
```

Возможность передавать функции в качестве аргументов является мощным инструментом для абстракции. Возможность писать функции, которые возвращают функции, позволяет нам максимально использовать это. Остальные разделы представляют несколько примеров утилит, которые взвращают функции.

[2] Под динамической областью, мы могли бы написать что то вроде make-adder, но вряд ли это работало. Привязка n будет определяться средой, в которую будет возвращена функция и мы не имеем никакого контроля над этим.

5.2 5-2 Ортогональность

Ортогональный язык это язык, в котором вы можете выразить много, комбинируя маленькое количество операторов во многими различными способами. Игрушечные блоки типа лего, очень ортогональны, пластмассовая модель практически не ортогональна. Основное преимущество complement в том, что он делает язык более ортогональным. До complement-a, Common Lisp были пары функций, такие как remove-if и remove-if-not, subst-if и subst-if-not, и так далее. С complement половина из них становитсья не нужной.

Макрос setf также улучшает ортогональность Lisp. Ранние диалекты Lisp часто имели пары функций для чтений и записи данных. Например, с property - lists(списков свойств), будет одна функция для установки свойств и другая для запроса о нем(свойстве). В Common Lisp, у нас есть только последняя, get. Чтобы

```
(defvar *!equivs* (make-hash-table))

(defun ! (fn)
     (or (gethash fn *!equivs*) fn))

(defun def! (fn fn!)
     (setf (gethash fn *!equivs*) fn!))
```

Рисунок 5-1: Возвращение деструктивных(разрушающих) эквивалентов.

установить(set) свойство, мы использжуем get в сочетании с setf:

```
(setf (get 'ball 'color) 'red)
```

Возможно, мы не сможем сделать Common Lisp меньше, но мы можем кое что сделать почти так же хорошо: использовать меньшее его подмножество. Можем ли мы определить новые операторы, которые смогут, подобно complement и setf, помочь нам в достижении этой цели? Есть хотя бы еще одно направление, по которому функции сгруппированы в пары. Для многих функций существуют их разрушающие(деструктивные) версии: remove-if и delete-if, reverse и nreverse, append и nconc. Определив оператор, чтоыбы возвращать разрушающий аналог функции, мы можем больше не обращаться к разрушающим функциям напрямую.

Рисунок 5-1 содержит код для поддержки понятия деструктивных аналогов. Глобальная хеш-таблица *!equivs* отображает функции на их деструктивные эквиваленты; ! возвращает деструктивный эквивалент; и def! устанавливает их. Имя оператора ! (банг) происходит из Scheme соглашения добавлять ! к имени функции имеющей побочный(сторонний) эффект. Теперь когда мы определили

```
(def! #'remove-if #'delete-if)
```

тогда вместо

```
(delete-if #'oddp lst)
```

мы бы сказали

```
(funcall (! #'remove-if) #'oddp lst)
```

Здесь неловкость Common Lisp скрывает основную элегантность идеи, что было бы более заметно на Scheme:

```
((! remove-if) oddp lst)
```

```
(defun memoize (fn)
  (let ((cache (make-hash-table :test #'equal)))
     #'(lambda (&rest args)
          (multiple-value-bind (val win) (gethash args cache)
             (if win
                  val
                  (setf (gethash args cache)
                           (apply fn args)))))))
```

Рисунок 5-2: Утилита запоминания результатов работы функции(Memoizing).

Кроме великолепной ортогональности, оператор ! приносит пару других преимуществ. Он делает программы понятнее, потому что мы сразу видим, что (! #'foo) это разрушающий эквивалент foo. Кроме того, он придает разрушающим операциям отчетливую, узнаваемую форму в исходном коде, что хорошо, потому что они должны получить особое внимание, когда мы ищем ошибку.

Поскольку связь между функцией и ее разрушающим аналогом будет как правило известна до выполнения программы, было бы наиболее эффективно определить (банг) ! как макрос, или даже предоставить макрос чтения для него.

5.3 5-3 Memoizing(Запоминание предыдущих результатов)

Если какая-то функция является дорогой, в смысле вычилений,и мы ожидаем, что иногда один и тот же вызов более одного раза, то стоит запоминать результаты(memoize): для кэширования возвращаемых значений всех предыдущих значений, и каждый раз, когда функция будет вызываться, в первую очередь будет просматриваться кэш, чтобы найти уже известное значение.

Рисунок 5-2 содержит обобщенную утилиту memoizing. Мы передаем не записывающую функцию, в memoize, и она возвращает эквивалентную записывающую версию - замыкание, содержащее хеш-таблицу для хранения результатов предыдущих вызовов.

```
> (setq slowid (memoize #'(lambda (x) (sleep 5) x)))
#<Interpreted-Function C38346>
> (time (funcall slowid 1))
Elapsed Time = 5-15 seconds
1> (time (funcall slowid 1))
Elapsed Time = 0-00 seconds
1
```

С функцией memoize, повторный вызов это просто поиск в хеш-таблице. Есть конечно дополнительные затраты на поиск при каждом начальном вызове, но поскольку мы

```
(defun compose (&rest fns)
 (if fns
      (let ((fn1 (car (last fns)))
            (fns (butlast fns)))
        #'(lambda (&rest args)
            (reduce #'funcall fns
                    :from-end t
                    :initial-value (apply fn1 args))))
      #'identity))
```

Рисунок 5-3: Оператор для композиции функций.

запоминаем только функцию, достаточно дорогую для вычисления, разумно предположить, что эти затраты сравнительно не велики.

Хотя это подходит для большинства применений, эта реализация memoize имеет несколько ограничений. Она обрабатывает вызовы как идентичные, если они имеют одинаковые списки аргументов; это может быть слишком строгим ограничением, если у функции есть ключевые параметры. Так же она предназначена для работы только с с однозначными функциями и не может хранить или возвращать множественные значения.

5.4 5-4 Композиция Функций(составные функции)

Дополнение(complement) к функции f обозначается ка ~f. Раздел 5-1 показал, что что замыкания делают определение ~ как функции Lisp-a. Еще одна распространенная операция над функциям это композиция(composition, составление функций), обозначаемая оператором *. Если f и g функции, тогда композиция функций f * g также является функцией, и f * g(x) =f * (g(x)). Замыканя также позволяют определеить * как функцию Лиспа.

Рисунок 5-3 определяет функцию compose которая принимает любое количество функций и возвращает их композицию. Например

```
(compose #'list #'1+)
```

возвращает функцию, эквивалентную

```
#'(lambda (x) (list (1+ x)))
```

Все функции, приведенные в качестве аргументов для композиции, должно быть функциями одного аргумента, за исключением последней. На последнюю функцию ограничений нет, и все аргументы, которые она принимает, также принимает функция возвращаемая compose:

```
> (funcall (compose #'1+ #'find-if) #'oddp '(2 3 4))
4
```

```
(defun fif (if then &optional else)
      #'(lambda (x)
          (if (funcall if x)
                 (funcall then x)
                 (if else (funcall else x)))))

(defun fint (fn &rest fns)
      (if (null fns)
          fn
          (let ((chain (apply #'fint fns)))
                #'(lambda (x)
                   (and (funcall fn x) (funcall chain x))))))

(defun fun (fn &rest fns)
      (if (null fns)
          fn
          (let ((chain (apply #'fun fns)))
                #'(lambda (x)
                   (or (funcall fn x) (funcall chain x))))))
```

Рисунок 5-4: Еще построители функций.

Поскольку это не Lisp функция, complement является частным случаем compose. Она могла бы быть определена как:

```
(defun complement (pred)
  (compose #'not pred))
```

Мы можем комбинировать функции другими способами, а не просто составлять(composing) их. Например, мы часто видим такие выражения как

```
(mapcar #'(lambda (x)
                  (if (slave x)
                      (owner x)
                      (employer x)))
            people)
```

Мы могли бы определить оператор для автоматического создания таких функций, как эта. Используя fif из рисунка 5-4, мы моглли бы получить тот же эффект:

```
(mapcar (fif #'slave #'owner #'employer)
            people)
```

Рисунок 5-4 содержин несколько других конструкторов для часто встречающихся типов функций. Вторая, fint, для случаев подобных этому:

```
(find-if #'(lambda (x)
                  (and (signed x) (sealed x) (delivered x)))
               docs)
```

Предикат, переданный как второй аргумент в find-if определяет пересечение трех предикатов вызываемых в нем. С fint, чье имя означает "пересечение функций(function intersection)" мы можем сказать:

```
(find-if (fint #'signed #'sealed #'delivered) docs)
```

Мы можем определить аналогичный оператор, чтобы вернуть объединение набора предикатов. Функция fun похожа на fint но исползует or вместо and.

5.5 5-5 Рекурсия на Cdrs

Рекурсивные функции настолько важны в программах на Lisp, что стоит иметь утилиты для их строительства. Этот раздел и следующий описывают функции, которые строят функции для двух наиболее распространенных типов. В Common Lisp, эти функции немного неудобно использовать. Как только мы перейдем к теме макросов, мы увидим, как поставить более элегантный фасад для этого механизма. Макросы для построения рекурсий обсуждаются в разделах 15-2 и 15-3.

Повторяющиеся куски(образцы) кода в программе являются признаком того, что они могли быть написана на более высоком уровне абстракции. Какой кусок кода чаще всего встречается в программах на Lisp чем подобная функция?:

```
(defun our-length (lst)
      (if (null lst)
            0(1+ (our-length (cdr lst)))))
```

или такая:

```
(defun our-every (fn lst)
      (if (null lst)
            t(and (funcall fn (car lst))
                  (our-every fn (cdr lst)))))
```

Конструктивно эти две функции имеют много общего. Они обе работают рекурсивно поочередно на последовательных cdrs списка, вычисляя одно и то же выражение на каждом шаге,

```
(defun lrec (rec &optional base)
      (labels ((self (lst)
                     (if (null lst)
                           (if (functionp base)
                                 (funcall base)
                                 base)
                           (funcall rec (car lst)
                                        #'(lambda ()
                                              (self (cdr lst)))))))
            #'self))
```

Рисунок 5-5: Функция для определения рекурсивных функций для плоских списков.

исключая базовый случай, когда они возвращают различные значения. Эта картина появляется так часто в программах на Лиспе, что опытные программисты могут читать и воспроизводить, не переставая думать. Действительно, урок настолько быстро усваивается, что вопрос о том, как упаковать шаблон в новую абстракцию, не возникает.

Впрочем, это все таки, образец(шаблон/закономерность). И вместо того, чтобы писать эти функции вручную мы можем написать функцию, которая будет генери-

ровать их для нас. Рисунок 5-5 содержит построитель функций с именем lrec ("list recurser"), который должен быть в состоянии сгенерировать большинство функций, которые используются на последовательных окончаниях(cdrs) списка.

Первый аргумент для lrec должен быть функцией двух аргументов: текущее начало(car) списка и функция, которая может быть вызвана для продолжения рекурсии. Используя lrec мы могли бы выразить our-length как:

```
(lrec #'(lambda (x f) (1+ (funcall f))) 0)
```

Чтобы найти длину списка, нам не нужно смотреть на элементы или на часть останавливающую рекурсию. Кстати, объект x всегда игнорируется, а функция f всегда вызывается. Тем не мение, нам нужно воспользоваться обеими параметрами, чтобы выразить нашу функцию our-every, например oddp:[3]

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f))) t)
```

Определение lrec использует labels для построения локальной рекурсивной функции под названием self. В рекурсивый вариант функции rec передаются два аргумента, текущий car списка, и функция, воплощающая рекурсивный вызов. На функции, такой как our-every, где рекурсивный случай представляет собой and, если первый аргумент возвращает false мы хотим остановиться прямо здесь. Это означает, что аргумент передается в рекурсивном случае, должен быть не значением, а функцией, которую мы могли бы вызвать(если хотим), чтобы получить значение.

```
; copy-list
(lrec #'(lambda (x f) (cons x (funcall f))))

; remove-duplicates
(lrec #'(lambda (x f) (adjoin x (funcall f))))

; find-if, for some function fn
(lrec #'(lambda (x f) (if (fn x) x (funcall f))))

; some, for some function fn
(lrec #'(lambda (x f) (or (fn x) (funcall f))))
```

Рисунок 5-6: Функции выраженные с помощью lrec.

case must not be a value but a function, which we can call (if we want) in order to get a value.

На рисунке 5.6 показаны некоторые функции существующие в Common Lisp, определенные с помощью lrec.[4] Вызов lrec не всегда дает наиболее эффективную реализацию данного функция. Действительно, lrec и другие генераторы рекурсивных функций определеные в этой главе, как правило, уводит нас от хвостовых рекурсивных решений. По этой причине они лучше всего подходят для использования в начальных версиях программы или в частях, где скорость не критична.

[3] В одном широко используемом Common Lisp, функция ошибочно возвращает истину для t и nil. В этой реализации он не будет работать в качестве второго аргумента для lrec.

[4] В некоторых реализациях может потребоваться установить *print-circle* в t перед отображением этих функций

5.6 5-6 Рекурсия на под деревьях(Subtrees)

Существует еще один рекурсивный шаблон, обычно встречающийся в программах на Лиспе: рекурсия на поддеревьях. Эта картина наблюдается в тех случаях, когда вы начинаете с возможно вложенного списка, и вам надо спуститься в его начало(car) и его окончание(CDR).

Список Lisp - это универсальная структура. Списки могут представлять, помимо прочего, последовательности, множества, отображения, массивы и деревья. Есть несколько разных способов интерпретировать список как дерево. Наиболее распространенным является рассматривать список как двоичное дерево, чья левая ветвь - car, и чья правая ветвь - cdr. (На самом деле это обычно внутреннее представление списков.) На рисунке 5-7 показаны три примера списков и деревья, которые они представляют. Каждый внутренний узел в таком дереве соответствует точке в представлении списка парой с точкой(a.b), поэтому древовидная структура может быть

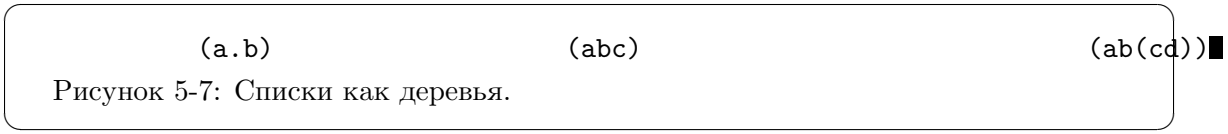

Рисунок 5-7: Списки как деревья.

легче интерпретировать, если списки представлять в такой форме:

```
(a b c)           = (a . (b . (c . nil)))
(a b (c d)) = (a . (b . ((c . (d . nil)) . nil)))
```

Любой список можно интерпретировать как двоичное дерево. Отсюда и различие между парой функций из Common Lisp, таких как copy-list и copy-tree. Первый копирует список как последовательность, т.е если список содержит подсписки, которые являют простыми элементами в последовательности, то они не копируются:

```
> (setq x          '(a b)
          listx (list x 1))
((A B) 1)
> (eq x (car (copy-list listx)))
T
```

Напротив, copy-tree копирует список как дерево подсписков, поскольку подсписки дерева являются поддеревьями, и поэтому должны быть также скопированными:

```
> (eq x (car (copy-tree listx)))
NIL
```

Мы могли бы определить версию copy-tree следующим образом:

```
(defun our-copy-tree (tree)
      (if (atom tree)
          tree
          (cons (our-copy-tree (car tree))
                  (if (cdr tree) (our-copy-tree (cdr tree))))))
```

Это определение оказывается одним из примеров общего паттерна. (Некоторые из следующие функции написаны немного странно, чтобы сделать шаблон очевидным.) Рассмотрим, например, утилиту для подсчета количества листьев в дереве:

```
(defun count-leaves (tree)
```

```
(if (atom tree)
    1(+ (count-leaves (car tree))
       (or (if (cdr tree) (count-leaves (cdr tree)))
             1))))
```

Дерево имеет больше листьев, чем атомов, что вы можете видеть, когда оно представлено в виде списка:

```
> (count-leaves '((a b (c d)) (e) f))
10
```

Листья дерева - это все атомы, которые вы можете видеть, когда вы смотрите на дерево в его представлении в виде пар с точкой. В записи в виде пар с точкой список ((a b (c d)) (e) f) будет иметь четыре nils, которые не видны при представлении в виде списка (по одному для каждой пары круглых скобок), поэтому count-leaf возвращает 10.

В последней главе мы определили несколько утилит, которые работают на деревьях. Например, flatten (стр. 47) берет дерево и возвращает список всех атомов в нем. То есть, если вы укажете вложенный список, вы получите назад список, который выглядит так же за исключением того, что в нём отсутствуют скобки, кроме внешней пары скобок:

```
> (flatten '((a b (c d)) (e) f ()))
(ABCDEF)
```

Эта функция также может быть определена (несколько неэффективно) следующим образом:

```
(defun flatten (tree)
      (if (atom tree)
          (mklist tree)
          (nconc (flatten (car tree))
                 (if (cdr tree) (flatten (cdr tree))))))
```

Наконец, рассмотрим rfind-if, рекурсивную версию find-i, которая работает на деревьях, а также на плоских списках:

```
(defun rfind-if (fn tree)
     (if (atom tree)
           (and (funcall fn tree) tree)
           (or (rfind-if fn (car tree))
                 (if (cdr tree) (rfind-if fn (cdr tree))))))
```

Чтобы обобщить find-if для деревьев, мы должны решить, хотим ли мы искать только для листьев, или еще для поддеревьев. Наш метод rfind-if использует первый подход, поэтому вызывающая сторона может предположить, что функция, указанная в качестве первого аргумента, будет вызываться для атомов:

```
> (rfind-if (fint #'numberp #'oddp) '(2 (3 4) 5))
3
```

Сейчас можно видеть насколько похожи эти четыре функции, copy-tree, count-leaves, flatten, и rfind-if. Действительно, все они являются примерами архетипической функции для рекурсии на поддеревьях. Как и в случае рекурсии на cdrs(плоских списках), нам не нужно предоставлять этот архетип, который может быть неопределенно

изменчивым, мы можем написать функцию для генерации экземпляров рекурсивных функций для него.

Чтобы получить сам архетип, давайте посмотрим на эти функции и увидим что не входит шаблоном. По сути our-copy-tree это два факта:

1. В базовом случае она возвращает свой аргумент.
2. В рекурсивном случае, она применяет cons к рекурсиям нижних поддеревьев левого(car) и правого (cdr).

Таким образом, мы должны иметь возможность выразить это как вызвов построителя с двумя аргументами:

```
(ttrav #'cons #'identity)
```

Определение ttrav ("обходчик деревьев(tree traverser)") показано на рисунке 5-8. Вместо перечачи одного значения в рекурсивном случае, мы передаем два, одно для левого поддерева и одно для правого. Если базовый аргумент является функцией, она будет вызвана для текущего листа. В рекурсии на плоском списке, базовый случай всегда nil, но рекурсии по дереву базовый случай может быть интересным значением, и мы можем захотеть его использовать.

С помощью ttrav мы можем выразить все предыдущие функции, кроме rfind-if. (Они показаны на рисунке 5-9.) Чтобы определить rfind-if нам нужен более общий конструктор рекурсии по дереву, который даст нам контроль когда, и если, делать рекурсивные вызовы. В качестве первого аргумента ttrav мы дали функцию, которая берет результаты рекусивных вызовов. В общем случае, мы хотим использовать вместо этого функцию, которая принимает два замыкания, представляющие сами вызовы. Тогда мы сможем написать рекурсеры, которые проходят по стольким деревьям, сколько им будет надо.

```
(defun ttrav (rec &optional (base #'identity))
     (labels ((self (tree)
                    (if (atom tree)
                         (if (functionp base)
                               (funcall base tree)
                               base)
                         (funcall rec (self (car tree))
                                      (if (cdr tree)
                                            (self (cdr tree)))))))
          #'self))
```

Рисунок 5-8: Функция для рекурсии на деревьях.

```
; our-copy-tree
(ttrav #'cons)

; count-leaves
(ttrav #'(lambda (l r) (+ l (or r 1))) 1)

; flatten
(ttrav #'nconc #'mklist)
```

Рисунок 5-9: Функции выраженные с помощью ttrav.

Функции, построенные ttrav всегда проходят по всему дереву. Это хорошо для функций подобных подсчету листьев(count-leaves) или flatten(создающих плоский список), которые все равно должны проходить по всему дереву. Но мы хотим, чтобы rfind-if останавливало поиск, как только она найдет то, что ищет. Она должна быть построена по более общей схеме, показанной на рисунке 5-10. Второй аргумент trec должен быть функцией трех аргументов: текущего объектиа и двух рекурсеров. Последние два будут замыканиями, представляющими рекурсии по левому и правому поддервьям. С помощью trec мы можем определить flatten как:

```
(trec #'(lambda (o l r) (nconc (funcall l) (funcall r)))
          #'mklist)
```

Теперь мы можем так же выразить rfind-if например для oddp как:

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
          #'(lambda (tree) (and (oddp tree) tree)))
```

```
(defun trec (rec &optional (base #'identity))
      (labels
        ((self (tree)
            (if (atom tree)
                  (if (functionp base)
                       (funcall base tree)
                       base)
                  (funcall rec tree
                                   #'(lambda ()
                                       (self (car tree)))
                                   #'(lambda ()
                                       (if (cdr tree)
                                            (self (cdr tree))))))))
          #'self))
```

Рисунок 5-10: Функция для рекурсии по деревьям.

5.7 5-7 Когда создавать функции

Выражение функций через вызовы конструкторов вместо шарп-квотированных('#) лямбда выражений, к сожалению, может повлечь за собой ненужную работу во время выполнения. шарп-квотированные лямбда выражения являются константами, но вызов функции конструктора будет вычисляться во время выполнения программы.

Если нам действительно нужно сделать этот вызов во время выполнения, он может не стоить выполнения функции конструктора. Тем не менее, по крайней мере, иногда, мы можем вызвать конструктор заранее. Используя макрос чтения: #.(знак решетку совместно со знаком точка), мы можем создавать новые функции построенные во время ЧТЕНИЯ программы. Например, мы можем сказать, что пока читается это выражение определяются compose и ее аргументы:

```
(find-if #.(compose #'oddp #'truncate) lst)
```

Тогда вызов compose будет вычислен читателем, и результирующая функция будет вставлена как константа(постоянная) в наш код. Поскольку обе функции oddp и truncate являются встроенными, можно с уверенностью предположить, что мы можем вычислить compose во время чтения, до тех пор, пока compose уже будет загружена.

В вобщем, составление и объединени функций проще и эффективнее сделать макросами. Это особенно верно в Common Lisp, с его отдельным пространством имен для функций. После введения макросов, мы рассмотрим в главе 15 большую часть описанного здесь материала, но с использоавнием более роскошных инструментов.

6 6 Функции как представление

Как правило, для представления информации используются структуры данных. Массив может быть использован как представление геометрических преобразований; дерево может представлять иерархию команд; граф может представлять систему железнодорожной сети. В Лиспе, иногда мы можем использовать замыкания в роли представления чего-либо. Внутри замыкания, переменные могут хранить информацию, и могут также играют роль, которую играют указатели при построении сложных структур данных. Создав группу замыканий, которые предоставляют доступ к замкнутым переменным, или могут ссылаться друг на друга, мы можем создавать гибридные объекты, которые объединяют в себе преимущества структур данных и программ.

На более низком уровне, разделяемые связанные переменные представляют собой указатели. Замыкания просто предоставляют нам удобства взаимодействия с ними на более высоком уровне абстракции. Используя замыкания для представления чего-либо, где обычно используются статические структуры данных, мы можем получить более элегантный и эффективный код.

6.1 6-1 Сети

Замыкания имеют три полезных свойства: они активны, у них есть локальное состояние, и мы можем создавать много экземпляров при необходимости. Где нам нужно использовать несколько копий активных объектов с локальным состоянием? В приложениях с сетями, среди прочего. Во многих случаях мы можем представить узлы в сети как замыкания. Кроме наличия собственного состояния, замыкания могут ссылаться друг на друга. Таким образом узел сети ввиде замыкания может знать несколько других узлов (замыканий) которым он должен посылать свой вывод. Это означает, что у нас будет возможность переносить некоторые сети непосредственно в код.

```
> (run-node 'people)
Is the person a man?
>> yes
Is he living?
>> no
Was he American?
>> yes
Is he on a coin?
>> yes
Is the coin a penny?
>> yes
LINCOLN
```

Рисунок 6.1: Пример игры "20 вопросов".

В этом и следующем параграфах мы рассмотрим два способа обхода сети. Сначала мы будем следовать традиционному подходу, где узлы определяются как структуры, и существует отдельный код для обхода сети. Затем, в следующем разделе мы покажем, как написать такую же программу на основе одной абстракции.

В качестве примера мы будем использовать самый простой случай: одну из тех программ, которые играют в "20 вопросов". Наша сеть будет представлена бинарным деревом. Каждый нетерминальный узел будет содержать вопрос типа "да/нет", и в зависимости от ответа на вопрос, обход будет продолжаться по левому или правому поддереву. Терминальные узлы (листья) будут содержать результаты. Когда будет достигнут конечный узел, его значение будет возвращено в качестве результата обхода. Взаимодействие с этой программой может выглядеть как на рисунке 6.1.

Традиционный способ начать работу - это определить какую-то структуру данных для представления узлов сети. Узел должен хранить определенную информацию: если это лист дерева, то какое значение возвращать, а если нет, то какой вопрос задать, и куда идти в зависимости от ответа. Удовлетворяющая этим требования структура данных представлена на рисунке 6.2. Она рассчитана на минимальный размер. Поле contents будет содержать вопрос или возвращаемое значение. Если узел не терминальный, поля yes и no скажут, куда идти в зависимости от ответа на вопрос; если узел является листом, мы узнаем это, т.к. эти поля будут пустыми. Глобальная переменная *nodes* будет хэш-таблицей, в которой узлы будут индексированы по имени. Наконец, функция defnode будет создавать новый узел (любого типа) и сохраняет его в *nodes*. Используя эти составные кирпичики, мы можем определить первый узел нашего дерева:

```
(defnode 'people "Is the person a man?"
               'male 'female)
```

```
(defstruct node contents yes no)

(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
     (setf (gethash name *nodes*)
               (make-node :contents conts
                          :yes         yes
                          :no          no)))
```

Рисунок 6.2: Представление и определение узлов.

```
(defnode 'people "Is the person a man?" 'male 'female)

(defnode 'male "Is he living?" 'liveman 'deadman)

(defnode 'deadman "Was he American?" 'us 'them)

(defnode 'us "Is he on a coin?" 'coin 'cidence)

(defnode 'coin "Is the coin a penny?" 'penny 'coins)

(defnode 'penny 'lincoln)
```

Рисунок 6.3: Пример сети.

Рисунок 6.3 показывает все, что необходимо для примера из рисунка 6.1.

Теперь все, что нужно сделать, это написать функцию для обхода этой сети, вывода вопросов и следования указанному пути. Эта функция, run-node, показана на рисунке 6.4. Указывая имя, мы ищем соответствующий узел. Если это не лист, задается вопрос из поля content, и в зависимости от ответа, мы продолжаем обход в одном из двух возможных направлений. Если узел терминальный, run-node просто возвращает значение поля content. При обходе сети с рисунка 6.3, эта функция производит результат, показанный на рисунке 6.1.

```
(defun run-node (name)
      (let ((n (gethash name *nodes*)))
         (cond ((node-yes n)
                  (format t "~A~%>> " (node-contents n))
                  (case (read)
                     (yes (run-node (node-yes n)))
                     (t     (run-node (node-no n)))))
                (t (node-contents n)))))
```

Рисунок 6.4: Функция обхода сети.

```
(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
      (setf (gethash name *nodes*)
              (if yes
                    #'(lambda ()
                           (format t "~A~%>> " conts)
                           (case (read)
                            (yes (funcall (gethash yes *nodes*)))
                            (t     (funcall (gethash no *nodes*)))))
                    #'(lambda () conts))))
```

Рисунок 6.5: Сеть с использованием замыканий.

6.2 6.2 Компиляция(Сборка) сетей

В предыдущем разделе мы написали программу для работы с сетью так, как она могла бы быть написана на любом другом языке. Действительно, программа настолько проста, что кажется странным рассчитывать на то, что ее можно написать по-другому. Но, на самом деле, мы можем реализовать ее гораздо проще.

Код на рисунке 6.5 показывает такой пример. Это все, что нам действительно нужно, чтобы запустить нашу программу. Вместо того, чтобы иметь отдельно структуры данных для узлов, и отдельную функцию для их обхода, мы представляем узлы как замыкания. Данные, ранее хранимые в структурах, сохраняются в связанных переменных внутри замыканий. Теперь нет необходимости в функции run-node; эта функциональность заложена в самих узлах. Для того, чтобы начать обход,

```
(defvar *nodes* nil)

(defun defnode (&rest args)
     (push args *nodes*)
     args)

(defun compile-net (root)
     (let ((node (assoc root *nodes*)))
       (if (null node)
              nil
              (let ((conts (second node))
                        (yes (third node))
                        (no (fourth node)))
                (if yes
                        (let ((yes-fn (compile-net yes))
                              (no-fn (compile-net no)))
                          #'(lambda ()
                              (format t "~A~%>> " conts)
                              (funcall (if (eq (read) 'yes)
                                           yes-fn
                                           no-fn))))
                        #'(lambda () conts))))))
```

Рисунок 6.6: Компиляция со статическими ссылками.

мы просто делаем вызов(funcall) узла-замыкания, в котором мы хотим перейти:

```
(funcall (gethash 'people *nodes*))
Is the person a man?
>>
```

После этого, выполнение будет аналогичным предыдущей реализации.

Представляя узлы как замыкания, мы можем превратить нашу сеть целиком в код. Сейчас, код должен искать узлы по имени во время выполнения. Однако, если мы знаем, что сеть не будет изменяться во время выполнения, мы можем добавить

следующее улучшение: мы можем делать прямые вызовы функций-замыканий, без необходимости проходить по хэш-таблице.

Рисунок 6.6 содержит новую версию программы. Теперь переменная *nodes* представляет собой одноразовый список, а не хэш-таблицу. Все узлы определяются в defnode как и раньше, но в этот раз никаких замыканий не создается. После того как все узлы определены, мы вызываем compile-net для компиляции всей сети сразу. Эта функция рекурсивно проходит свой путь вплоть до листьев дерева, и на пути обратно, возвращает на каждом шаге узел-функцию для каждого из двух поддеревьев.[1] Итак, теперь каждый узел будет совершать прямой вызов обработчиков обоих направлений, а не искать их по имени. Когда первоначальный вызов compile-net завершается, он возвращает функцию, представляющую часть сети, которую мы попросили собрать.

```
> (setq n (compile-net 'people))
#<Compiled-Function BF3C06>
> (funcall n)
Is the person a man?
>>
```

Обратите внимание, что compile-net производит компиляцию в обоих смыслах. Происходит компиляция в общем смысле, путем трансляции абстрактного представления сети в код. Более того, если функция compile-net компилируется, она возвращает скомпилированные функции. (См. стр. 25.)

После компиляции сети, нам больше не нужен список, созданный с помощью defnode. Он может быть выброшен (например, путем установки *nodes* в nil) и удален сборщиком мусора.

6.3 6.3 Заглядывая в будущее

Многие программы, связанные с сетями могут быть реализованы путем компиляции узлов в замыкания. Замыкания представляют собой данные, и они могут быть использованы для представления различных вещей так же, как и структуры. Это требует некоторого нестандартного мышления, но наградой будут более быстрые и элегантные программы.

Макросы могу существенно помочь, когда мы используем замыкания для представления данных. "Представить в виде замыкания" это еще один способ сказать "скомпилировать", и так как макросы работают во время компиляции, они являются естественным средством для этого. После того, как макросы будут представлены, главы 23 и 24 покажут гораздо больше программ, основанных на используемой здесь стратегии.

[1] Эта версия предполагает, что сеть представляет собой дерево, которое должно быть в этом приложении.

7 7 Макросы

Макросы Lisp позволяют ввам определять операторы, которые реализуют преобразования. Определение макроса по суте является функцией, которая генерирует Lisp код, программа которая пишет программы Из этих небольших начинаний возникают большие возможности, а также неожиданые опасности. Главы 7-10 составляют учебное пособие о макросах. Эта глава объясняет, как работают макросы, дает примемы их написания и тестирования, и рассматривает проблемы использования макро стиля.

7.1 7-1 Как работает Макрос

Поскольку макросы могут вызываться и возвращать значения, они, как правило, связаны с функциями. Определения макросов иногда напоминают определения функций и неофициально люди называют то что на самом деле является макросом - "встраиваемой функцией." Но эта слишком далеко идущая аналогия может стать источником путаницы. Макросы работают по другому, нежели обычные функции, и знание как и чем макросы отличаются от функций, это ключ к их правильному ипользованию. Функция выдает результаы, а макрос выдает(производит) выражения, которые при вычислении, дают результаты.

Лучший способ начать - перейти прямо к примеру. Предположим, мы хотим написать макрос nil!, который устанавливает свой аргумент в значение nil. Мы хотим чтобы (nil! x) имел тот же эффект что и выражение (setq x nil). Мы сделаем это определив nil! как макрос, который превращает экземпляры первой формы в в экземпляры второй формы.

```
> (defmacro nil! (var)
      (list 'setq var nil))
NIL!
```

Перефразируя на Русский, это определение говорит Лиспу: "Всякий раз, когда ты видишь выражение вида (nil! var), преврати его в форму вида (setq var nil), прежде чем его вычислить."

Выражение сгенерированное макросом, будет вычислено вместо исходного вызова макроса. Вызов макроса это список, первым элементом которого является имя макроса. Что происходит, когда мы вводим вызов макроса (nil! x) на верхнем уровне? Лисп замечает что nil! это имя макроса, и:

1. строит выражение, указанное в приведенном выше определении, затем
2. вычисляет это выражение вместо исходного вызова макроса.

Шаг построения нового выражения называется расширением макроса(macroexpansion). Лисп смотрит на данное выше определение nil!, которое показывает как создать замену для вызова макроса. Определение nil! применяется как функция к выражениям приведенным в качестве аргументов в вызове макроса. Он(вызов) возвращает список из трех элементов: setq, выражение переданное в качестве аргумента макроса и nil. В нашем случае, аргумент переданный в nil! это x, и расшерение макроса равно (setq x nil).

После расширения макроса наступает второй шаг, вычисление. Lisp вычисляет расширение макроса - (setq x nil) как если бы вы ввели его, а не вызов макроса. Вычисление не всегда происходит сразу после расширения, как это происходит на верхнем

уровне. Вызов макроса, находящийся в определении функции будет расширен когда функция компилируется, но расширение - или объектный код, который получается из него - не будет вычисляться, до тех пор, пока функция не будет вызывана.

Многих трудностей связанных с макросами, с которыми вы может столкнуться, можно избежать, поддежкой четкого разделения между расширением макроса и вычислением. Когда пишут макросы, знают какие вычисления выполняются во время макрорасширения, и какие во время выполнения, эти два шага обычно воздейтсуют на объекты разных сортов. Шаг расширения макроса касается выражений, и шаг вычисления касается их значений.

Иногда расширение макроса может быть более сложным, чем в случае nil!. Расширение nil! было вызовом встроенной специальной формы, но иногда расширение макроса будет еще одним вызовом макроса, как русская матрешка, которая содержит в себе еще одну матрешку. В таких случаях расширение макроса просто продолжается пока оно не достигнет выражения, которое не является вызовом макроса. Процесс может занять сколько угодно шагов, пока расширение не закончиться.

Многие языки предлагают некоторые формы макросов, на макросы Лиспа являются единсвенными мощными. Когда файл Лиспа компилируется, парсер(разборщик синтаксических выражений) читает исходный код и отправляет его в компилятор. Вот гениальный штрих: вывод парсера состоит из списков объектов Лисп. С помощью макросов, мы можем манипулировать программой, в то время как она находиться в промежуточной форме, между парсером и компилятором. Если необходимо эти манипуляции могут быть очень обширными. Макрос, генерирующий расширение, обладает всеми возможностями Лисп. Действительно, макрос это действительно функция Лисп, та, которая возвращает выражения. Определение nil! содержит один вызов функции list, но другой макрос, может вызвать целую подпрограммму для генерации своего расширения.

Возможность изменить то, что видит компилятор, почти как возможность переписать этот код. Мы можем добавить любую конструкцию к языку, которую мы можем определить как преобразование в существующие конструкции.

7.2 7-2 Обратная кавычка

Обратная кавычка(Backquote) это специальная версия кавычки(цитирования), которую можно использовать для создания шаблонов выражений Лисп. Одним из наиболее распространенных применений обратной кавычки, является определение макросов.

Символьный знак обратная кавычка(backquote), ‘, назван так потому, что он напоминает обычную кавычку, ’, перевернутуня наоброт. Когда к выражению добавляется одна обратная кавычка, она ведет себя точно также как и обычная кавычка:

```
`(abc)is equal to '(abc).
```

Обратная кавычка становиться полезной только тогда, когда она появляется в сочетании с запятой(,) и запятой с "собакой" (,@). Если обратная кавычка создает шаблон, запятая создает слот внутри шаблона. Список с предшествующей обратной кавычкой эквивалентен вызову команды list с закоментированными элементами. То есть,

```
`(a b c) is equal to (list 'a 'b 'c).
```

В рамках обратной кавычки, знак запятой говорит Лиспу: "здесь надо отключить режим цитирования." Когда перед одним из элементов списка появляется запятая, она имеет эффект отмены его цитирования, который был бы к нему применен. Так

```
`(a ,b c ,d) is equal to (list 'a b 'c d).
```

Вместо символа b, его ЗНАЧЕНИЕ вставляется в результирующий список. Запятые работают не зависимо от того как глубоко они появляются во вложенном списке,

```
> (setq a 1 b 2 c 3)
3> `(a ,b c)
(A 2 C)
> `(a (,b c))
(A (2 C))
```

и они могут даже появляться после кавычки, или внутри квотированного подсписка:

```
> `(a b ,c (',(+ a b c)) (+ a b) 'c '((,a ,b)))
(A B 3 ('6) (+ A B) 'C '((1 2)))
```

Одна запятая противодействует эффекту одной обратной кавычки, поэтому запятые должны соответствовать обратным кавычкам. Скажем, что запятая окружена определенным оператором, если оператор ставиться перед запятой или перед выражением, которое её содержит. Например, в ‘(,a ,(b ‘,c))), последняя запятая окружена одной запятой и двумя обратными кавычками. Общее правило таково: запятая окруженная n запятыми должна быть окружена минимум n+1 обратной кавычкой. Очевидным следствием является то, что запятая не может появляться за пределами выражений окруженных обратной кавычкой. Обратные кавычки и запятые могут быть вложенными, пока подчиняются правилу указанному выше. Любое из следующих выражений сгенерирует ошибку, если его ввести на верхнем уровне:

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

Вложенные обратные кавычки, вероятно понадобятся только в макросах, определяющих макросы. И то и другое обсуждается в главе 16.

Обратная кавычка используется для создания списков.[1] Любой список, сгенерированный обратной кавычкой также может быть сгенерирован с использвоанием list и обычных кавычек. Преимущество обратных кавычек заключается лишь в том, что она облегчает чтение выражений, поскольку выражение которое цитирует обратная кавычка, напоминает выражение, которое она создает. В предыдущем разделе мы определили nil! как:

```
(defmacro nil! (var)
  (list 'setq var nil))
```

С помощью обратной кавычки тот же макрос может быть определен как:

```
(defmacro nil! (var)
  `(setq ,var nil))
```

которое в этом случае не так уж и отличается от предыдущего. Тем не менее, чем длинне определение макроса, тем более важно использовать обратную кавычку.

[1] Обратная кавычка также может использоваться для создания векторов, но это редко делается в определениях макросов.

Рисунок 7-1 содержит два возможных определения nif, макроса, который выполняет трехстороннее числовое выражение if.[2]

Первый аргумент должен вычисляться как число. Второй, третий или четверый аргументы вычисляются, в зависимости от того, был ли первый аргумент положительным, нулевым или отритцательным:

```
> (mapcar #'(lambda (x)
                    (nif x 'p 'z 'n))
                '(0 2-5 -8))
(ZPN)
```

с использованием обратных кавычек:

```
(defmacro nif (expr pos zero neg)
       `(case (truncate (signum ,expr))
            (1 ,pos)
            (0 ,zero)
            (-1 ,neg)))
```

без обратной кавычки:

```
(defmacro nif (expr pos zero neg)
       (list 'case
                (list 'truncate (list 'signum expr))
                (list 1 pos)
                (list 0 zero)
                (list -1 neg)))
```

Рисунок 7-1: Определение макроса с и без обратных кавычек.

Два определения на рисунке 7-1 определяют один и тот же макрос, но первое использует обратные кавычки, в то время как второе строит свое расширение путем явных вызовов команды list. Например, из первого определения легко уведить, что (nif x 'p 'z 'n) расширяется в

```
(case (truncate (signum x))
      (1 'p)
      (0 'z)
      (-1 'n))
```

потому что тело определения макроса выглядит так же, как генерируемое им расширение. Чтобы понять вторую версию, без обратных кавычек, вы должны проследить в своей голове создаваемое расширение.

Запятая с "собакой", ,@, это вариант запятой. Он ведет себя как запятая, с одним отличием: вместо того, чтобы просто вставить значение выражения, к которому оно прикрепляется, как это делает запятая, запятая с "собакой" вшивает этот результат. Вшивание можно рассматривать как вставку с удалением самого верхнего уровня скобок:

```
> (setq b '(1 2 3))
```

[2] Этот макрос определен немного коряво, чтобы избежать использования gensyms. Лучшее определение дано на странице 150.

```
(1 2 3)
> `(a ,b c)
(A (1 2 3) C)

> `(a ,@b c)
(A 1 2 3 C)
```

Запятя приводит к тому, что список (1 2 3) вставляется вместо b, в то время как запятая с "собакой" вызывает вставку туда элементов списка. Есть некоторые дополнительные ограничения на использование запятой с "собакой":

1. Для объединения аргументов, запятая с "собакой" должны быть заключены в некоторую последовательность. Ошибочно написать, что то вроде `` `,@b ``, потому, что не с чем склеивать значение b.
2. Объект, который должен быть "вшит", должен быть списком, если он не является последним элементом. Выражение `` `(a ,@1) `` будет вычислено в `(a . 1)`, но попытка вшить атом в середину списка, как в `` `(a ,@1 b) ``, вызовет ошибку.

Запятая с "собакой" имеет тенденцию использоваться в макросах, которые принимают неопределенное число аргументов и передают их функциям или макросам, которые также принимают неопределенное количество аргументов. Такая ситуация обычно возникает при реализации неявных блоков. Common Lisp имеет несколько операторов для группировки кода в блоки, включая block, tagbody и progn. Эти операторы редко появляются непосредственно в сиходном коде; они чаще скрыты, т.е скрыты макросами.

Неявный block встречается в любом встроенном макросе, который может иметь тело выражений. Например, и let, и cond предоставляют неявный progn. Простейшим встроенным макросом использующим это вероятно является when:

```
(when (eligible obj)
  (do-this)
  (do-that)
  obj)
```

Если (eligible obj) возвращает истину, остальные выражения будут вычислены, и выражение when в целом. вернет значение последнего выражения. В качестве примера использования запятой с "собакой", вот одно из возможных определений для when:

```
(defmacro our-when (test &body body)
  `(if ,test
            (progn
               ,@body)))
```

Это определение использует параметр &body (идентичный &rest за исключением его эффекта для красивой печати) чтобы принять произвольное количество аргументов, и запятую с "собакой" для "вшивания" их в выражение progn. При расширении макроса вызванного выше, три выражения тела появляются в пределах одного progn:

```
(if (eligible obj)
       (progn (do-this)
                (do-that)
```

```
                obj))
```

Большинство макросов для итераций "вшивают" свои аргументы аналогичным образом.

Эффект запятой с "собакой" может быть достигнут без обратных кавычек. Например, выражение `(a ,@b c) эквивалентно (cons 'a (append b (list 'c))). Запятая с "собакой" используется только для того, чтобы сделать выражения генерирующие выражения более читабельными.

Макро определения (обычно) генерируют списки. Хотя расширения макроса могут быть построены с использованием функции list, цитированные с помощью обратных кавычек шаблоны списков делают эту задачу значительно легче. Определение макросов с помощью defmacro и обратных кавычек, будет внешне напоминать функцию определенную с помощью defun. Пока вы не введены в заблуждение этим подобием, обратные кавычки делают опредление макроса и более легкими для написания и более легкими для чтения.

Обратные кавычки, так часто используют в определениях макросов, что люди иногда думают, что обратные кавычки это часть defmacro. Последнее, что нужно помнить о обратных кавычках, это то, что у нее есть собственная жизнь, отдельная от её роли в определениях макросов. Вы можете использовать обратную кавычку, где угдно, где необходимо строить последовательность:

```
(defun greet (name)
      `(hello ,name))
```

7.3 7-3 Определение Простых Макросов

В программировании, зачастую, лучший способ научиться - начать экспериментировать. Полное теоретическое понимание может прийти позже. Соответственно, в этом разделе представлен способ немедля начать писать макросы. Это работает только для узкого круга случаев, но там где это применимо, его можно применять довольно механически. (Если вы уже писали макросы ранее, вы пожете пропустить этот раздел.)

В качестве примера, мы рассмотрим как написать вариант встроенной функции Common Lisp - member. По умолчанию функция member использует eql для проверки на равенство. Если вы хотите проверять членство используя eq, вы должны прямо сказать:

```
(member x choices :test #'eq)
```

Если бы мы делем это много раз, мы могли бы написать вариант member который всегда использовал бы eq. В некоторых, более ранних диалектах Lisp быта такая функция, которая называлась memq:

```
(memq x choices)
```

Обычно мы определяем memq как встраиваемую функцию, но для примера мы перевоплотим ее как макрос.

```
call:                    (memq x choices)

expansion: (member x choices :test #'eq)
```

Рисунок 7-2: Диаграмма, используемая при написании memq.

Методика написания макросов: Начните с типичного вызова макроса, который вы хотите определить. Напишите его на листе бумаги, и ниже него запишите выражение, в которое он должен расширяться. Рисунок 7-2 показывает два таких выражения. Из вызова макроса, создайте список параметров для вашего макроса, сочинив некоторое имя параметра для каждого из аргументов. В нашем случае есть два аргумента, поэтому у нас будет два параметра, и назовем их obj и lst:

```
(defmacro memq (obj lst)
```

Теперь вернитесь к двум выражениям, которые вы записали ниже. Для каждого аргумента в вызове макроса, нарисуйте линию, соединяющую его с местом, где он появляется в расширении ниже. На рисунке 7-2 есть две параллельные линии. Чтобы написать тело макроса, обратите ваше внимание на расширение. Начните тело с обратной кавычки. Теперь, начнем чтение выражения расширения по одному выражению. Где бы вы ни нашли скобки, они не являются частью аргумента в вызове макроса, поместите их(скобки) в определение макроса, так чтобы за обратной кавычкой была левая скобка. Для каждого выражения в расширении

1. Если нет линии соединяющей его с вызовом макроса, запишите это выражение

2. Если есть соединение с одним из аргументов в вызове макроса, напишите далее символ, который находитсья в соответствующей позиции списка параметров макроса, поставьте перед ним запятую. ?@end enumerate

   Здесь нет связи с первым элементом, member, поэтому мы ипользуем само название member:

   ```
   (defmacro memq (obj lst)
     `(member
   ```

   Тем не менее, x имеет линию, ведущую к первому аргументу в исходном выражении, поэтому мы используем в теле макроса первый параметр, с запятой:

   ```
   (defmacro memq (obj lst)
     `(member ,obj
   ```

   Продолжая таким образом, завершаем определение макроса:

```
(while hungry
       (stare-intently)
       (meow)
       (rub-against-legs))

(do ()
         ((not hungry))
       (stare-intently)
       (meow)
       (rub-against-legs))
```

Рисунок 7-3: Диаграмма используемая в написании макроса while.

```
(defmacro memq (obj lst)
     `(member ,obj ,lst :test #'eq))
```

Пока что мы можем писать макросы, которые принимают фиксированное количество аргументов. Теперь предположим, что мы хотим написать макрос while, который принимает тестовое выражение и некоторое тело кода, и циклически проходит по коду(выполняет его), до тех пор пока тестовое выражение не возвратит истину. Рисунок 7-3 содержит пример цикла while описывающего поведение кошки.

Чтобы написать такой макрос, мы должны немного изменить нашу методику. Как и прежде, начнем с записи примера вызова макроса. Из него(вызова) построим список параметров макроса, но там где вы хотите принять неопределенное количество аргументов, завершите их параметром &rest или &body:

```
(defmacro while (test &body body)
```

Теперь напишите нужное расширение под вызовом макроса, и как и раньше, нарисуйте линии соединяющие аргументы в вызове макроса с их позицией в расширении. Тем не менее, когда у вас есть последовательность аргументов, которые будут собраны в один параметр &rest или &body, рассматривайте их как группу, рисуя одну линию для всей последовательности. Рисунок 7-3 показывает полученную диаграмму.

Чтобы написать тело определения макроса, продолжайте как и прежде идти вдоль расширения. Как и два предыдущих правила, нам нужно еще одно:

?@enumerate

3. 3. Если есть связь из ряда выражений в расширении к ряду аргументов в вызове макроса, запишите соответствующий параметр &rest или &body, поставив перед ним запятую с "собакой".

Таким образом, полученное определение макроса будет:

```
(defmacro while (test &body body)
  `(do ()
           ((not ,test))
       ,@body))
```

Чтобы построить макрос, который должен иметь body выражения, некоторый параметр должен действовать как воронка. Здесь несколько аргументов в вызове мак-

роса объединяются в body, а затем снова распадаются, когда body "вшивается" в расширение.

Подход описанный в этом разделе, позволяет нам написать простейшие макросы - те, которые просто перетасовывают свои параметры. Макросы могут много больше этого. Раздел 7-7 представит примеры, где расширения не могут быть представлены в виде простых списков с обратными кавычками, и генерирующие их программы. Макросы становяться программами сами по себе.

7.4 7-4 Проверка Расширения Макросов

Имея написанный макрос, как мы можем его протестировать? Макрос типа memq достаточно прост, так что достаточно просто взглянув на него сказать что он будет делать. При написании более сложных макросов, мы должны быть в состоянии проверить, правильно ли они расширяются.

На рисунке 7-4 показано определение макроса и два способа увидеть как он расширяется. Встроенная функция macroexpand принимает выражение и возвращает его расширенние макроса. Отправка вызова макроса в macroexpand показывает, как вызов макроса будет до конца развернут(расширен), перед его вычислением. Но полное расширение, это не всегда то что вы хотите, чтобы проверить макрос. Когда рассматриваемый макрос основывается на других макросах, они тоже будут расширены, поэтому полное расширение иногда может быть трудно прочитать.

По первому выражению, показанному на Рисунке 7-4, трудно сказать, действительно ли while расширяется как задумано, потому что встроенный макрос do так же расширяется. так же как и макрос prog который тоже расширяется. Что нам нужно, это способ увидеть результат только после одного шага расширения. Это цель встроенной функции macroexpand-1, показанной во втором примере; macroexpand-1 останавливается после всего лишь одного шага, даже если расширение все еще является вызовом макроса.

Когда мы хотим посмотреть на расширение вызова макроса, как бы нибыло неприятно, всегда нужно набирать

```
(pprint (macroexpand-1 '(or x y)))
```

Рисунок 7-5 определяет новый макрос, который вместо этого позволяет нам сказать:

```
(mac (or x y))
```

Обычно вы отлаживаете функции, вызывая их, а макросы - расширяя их. Но так как вызов макроса включает в себя два уровня вычислений, существует два

```
> (defmacro while (test &body body)
          `(do ()
              ((not ,test))
           ,@body))
WHILE

> (pprint (macroexpand '(while (able) (laugh))))

(BLOCK NIL
     (LET NIL
         (TAGBODY
          #:G61
          (IF (NOT (ABLE)) (RETURN NIL))
          (LAUGH)
          (GO #:G61))))
T> (pprint (macroexpand-1 '(while (able) (laugh))))
(DO NIL
          ((NOT (ABLE)))
     (LAUGH))
T
```

Рисунок 7-4: Макрос и две глубины его расширения.

```
(defmacro mac (expr)
     `(pprint (macroexpand-1 ',expr)))
```

Рисунок 7-5: Макрос для тестирования макрорасширения.

точки, где все может пойти не так. Если макрос работает не правильно, в большинстве случаев вы можете сказать, что что то не так, просто взглянув на расширение. Тем не менее, иногда, расширение будет выглядеть хорошо, и вы захотите вычислить его, чтобы увидеть, где возникают проблемы. Если расширение содержит свободные переменные, вы можете сначала установить несколько переменных. В некоторых системах вы можете скопировать расширение и вставить его на верхний уровень REPL, или выделить его и выбрать команду eval из меню. В худшем случае, вы можете установить переменную значением списка возвращаемого из macroexpand-1, и затем вызвать eval для него:

```
> (setq exp (macroexpand-1 '(memq 'a '(a b c))))
(MEMBER (QUOTE A) (QUOTE (A B C)) :TEST (FUNCTION EQ))
> (eval exp)
(ABC)
```

Наконец, расширение макросов это больше чем помощь в отладке, это еще один путь научиться писать макросы. Common Lisp имеет более сотни встроенных макросов, некоторые из них довольно сложные. Глядя на расширения этих макросов, вы часто будете в состоянии увидеть, как они были написаны.

7.5 7-5 Деструктуризация в Списке Параметров

Деструктуризация это обобщение вида присваивания assignment[3] выполняемого вызовами функций. Если вы определяете функцию от нескольких аргументов

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

тогда, когда функция вызывается

```
(foo 1 2 3)
```

параметрам функции присваиваются аргументы при вызове в соответствии с их положением: x значение 1, y значение 2, и z значение 3. Деструктуризация описывает ситуацию, когда такого рода позиционное присваивание выполняется для произвольных списковых структур, а также для плоских списков, таких как (x y z).

Common Lisp макрос destructuring-bind (новый в CLTL2) принимает образец(шаблон), аргумент вычисляющийся в список, и тело выражений, и вычисляет выражения с параметрами связывающимися по образцу с соответствующими элементами списка:

```
> (destructuring-bind (x (y) . z) '(a (b) c d)
         (list x y z))
(A B (C D))
```

Этот новый оператор и другие, подобные ему, составляют предмет главы 18.

Деструктуризация также возможна в списках параметров макроса. В Common Lisp defmacro позволяет спискам параметров быть произвольными списковыми структурами. Когда вызов макроса расширяется, компоненты вызова будут присвоены параметрам как если бы использовался destructuring-bind. Встроенный макрос dolist использует преимущества такой деструктуризации списка параметров. В вызове, подобном:

```
(dolist (x '(a b c))
      (print x))
```

функция расширения должна "вырвать" x и '(a b c) из списка, заданного как первый аргумент. Это может быть сделано неявным образом, передавая dolist соответствующий список параметров:[4]

```
(defmacro our-dolist ((var list &optional result) &body body)
      `(progn
           (mapc #'(lambda (,var) ,@body)
                    ,list)
           (let ((,var nil))
              ,result)))
```

В Common Lisp, макросы подобные dolist обычно заключют с список аргументы, не являющиеся частью тела(вычисляемых выражений). Поскольку он принимает не обязательный аргумент result, dolist в любом случае должен заключать свои первые

[3] Деструктуризация обычно наблюдается в операторах, которые создают привязки, а не выполняют присваивание. Тем не менее, концептуально, деструктуризация это способ присваивания значений, который будет также работать как длся существующих переменных, так и для новых. Т.е ничто не мешает написать нам деструктуризацию setq.

[4] Эта версия написана таким странным образом, чтобы избежать использования gensyms, который мы еще не ввели в употребление.

аргументы в отдельный список. Но даже если дополнительная списковая структура не нужна, она облегчает чтение вызовов dolist. Предположим, мы хотим определить макрос when-bind, такой как when за исключением, что он привязывает некоторую переменную к значению возвращаемому тестовым выражением. Этот макрос может быть реализован с помощью вложенного списка параметров:

```
(defmacro when-bind ((var expr) &body body)
      `(let ((,var ,expr))
           (when ,var
              ,@body)))
```

и вызывается следующим образом:

```
(when-bind (input (get-user-input))
      (process input))
```

вместо:

```
(let ((input (get-user-input)))
      (when input
          (process input)))
```

При экономном использовании, деструктуризация списка параметров може привести к более четкому коду. Как минимум, ее можно использовать в макросах, таких как when-bind и dolist, которые принимают два или более аргументов, сопровождаемых телом - последовательностью выражений.

```
(defmacro our-expander (name) `(get ,name 'expander))

(defmacro our-defmacro (name parms &body body)
      (let ((g (gensym)))
         `(progn
            (setf (our-expander ',name)
                    #'(lambda (,g)
                           (block ,name
                             (destructuring-bind ,parms (cdr ,g)
                               ,@body))))
            ',name)))

(defun our-macroexpand-1 (expr)
      (if (and (consp expr) (our-expander (car expr)))
           (funcall (our-expander (car expr)) expr)
           expr))
```

Рисунок 7-6: Эскиз defmacro.

7.6 7-6 Модель Макросов

Формальное описание что делают макросы, будет длинным и запутанным. Оптный программист все равно не хранит это описание в своей голове. Ему более удобно помнить, что делает defmacro представляя как он будет определен.

В Лиспе существует давняя традиция подобных объяснений. Руководство по программированию для Lisp 1.5, впервые опубликованное в 1962 году, дает для справки определение eval написанного на Лиспе. Поскольку defmacro сам по себе является макросом, мы можем дать ему тоже самое определение как на Рисунке 7-6. Это определение использует несколько методов, котоыре еще не были рассмотрены, поэтому некоторые читатели, если захотят, могут обратитсья к нему позже.

Определение на рисунке 7-6 дает довольно точное представление о том, что делает макрос, но, как и любой набросок это представление неполное. Наше определение не будет обрабатывать ключевое слово &whole должным образом. И что на самом деле defmacro сохранается в качестве макро-функции своего первого аргумента - это функция от двух аргументов: вызова макроса, и лексического окружения в котором происходит вызов. Тем не менее, эти функции(свойства) используются в основном в эзотерических макросах. Если вы будете работать в предположении, что макросы реализуются так как показано на рисунке 7-6, вы вряд ли ошибетесь. Каждый макрос определенный в этой книге будет работать, например.

Определение на рисунке 7-6 дает функцию расширения, которая является закоментированным кавычкой с решеткой('#) лямбда выражением. Это должно сделать ее замыканием: любые свободные символы в определении макроса должны ссылаться на переменные в окружении, где произошло определение макроса(вызов defmacro). Так что можно сказать следующее:

```
(let ((op 'setq))
      (defmacro our-setq (var val)
         (list op var val)))
```

Что касается CLTL2, это так. Но в CLTL1, расширители макросов определяются в нулевом лексическом окружении,[5] поэтому в некоторых старых реализациях, определение our-setq не будет работать.

7.7 7-7 Макросы как Программы

Определение макроса не обязательно должно быть списком заключенным в обратные кавычки. Макрос это функция, которая превращает один вид выражений в другой. Эта функция может вызывать функцию list для создания своего результата, но может также вызывать целую подпрограмму, состоящую из сотен строк кода.

Раздел 7-3 дал простой способ написания макросов. Используя эту методику, мы можем писать макросы, чьи расширения содержат те же подвыражения, и в вызове макроса. К сожалению, только самые простые макросы соответствуют этому условию. Как более сложный пример, рассмотрим встроенный макрос do. Невозможно написать do как макрос, который просто перетасовывает свои параметры. Расширение должно создавать сложные выражения, которые никогда не появляются в вызове макроса.

Более общий подход к написанию макросов - думать о виде выражении, которое вы хотите использовать(первая форма), расширенном в то что вы хотите получить(вторая форма), и затем написать программу, которая преобразует первую форму во вторую. Попробуйте расширить премер от руки, а затем посмотрите, что происходит,

[5] Для примера макроса, где это различие имеет значение, смотри примечание на стр. 393.

когда одна форма превращается в другую. Работая на примерах, вы сможете получить представление(идеи) о том, что потребуется для реализации вашего макроса.

На Рисунке 7-7 показан экземпляр do, и выражение в которое его следует расширить. Выполнение расширения в ручную - это хороший способ прояснить ваши идеи о том, как макрос должен работать. Например, это может быть не очевидно, пока не попробуешь написать расширение, что локальные переменные должны быть обновлены с использванием psetq.

Встроенный макрос psetq (называемый "параллельный(parallel) setq") ведет себя как setq, за исключением того, что все его (четные) аргументы будут вычислены до любого из выполняемых присвоений. Если обычный setq имеет более двух аргументов, то новое значение первого аргумента будет видно во время вычисления последующих(например четвертого):

```
(do ((w 3)
          (x 1 (1+ x))
          (y 2 (1+ y))
          (z))
         ((> x 10) (princ z) y)
       (princ x)
       (princ y))
```

должно быть расширено в нечто подобное:

```
(prog ((w 3) (x 1) (y 2) (z nil))
       foo
        (if (> x 10)
              (return (progn (princ z) y)))
        (princ x)
        (princ y)
        (psetq x (1+ x) y (1+ y))
        (go foo))
```

Рисунок 7-7: Желаемое расширение do.

```
> (let ((a 1))
        (setq a 2 b a)
        (list a b))
(2 2)
```

Здесь, поскольку a устанавливается первым, b получает новое значение, 2. Предполагается что psetq ведет себя как если бы все его аргументы присваиваются параллельно:

```
> (let ((a 1))
        (psetq a 2 b a)
        (list a b))
(2 1)
```

Таким образом, здесь b получает старое значение a. Макрос psetq предоставляется специально для поддержки макросов, таких как do, которые должны вычислять

некоторые свои аргументы параллельное. (Если бы мы использовали setq, мы бы в место этого определили do*.)

Глядя на расширение, также становитсья ясно, что мы не можем использовать foo как метку цикла(loop). Что если foo также используется как метка цикла(loop) в теле do? Глава 9 подробно рассматривает эту проблему; пока достаточно сказать, что вместо использования foo в макросе, расширение макроса должно использовать специальный анонимный символ возвращаемый функцией gensym.

```
(defmacro our-do (bindforms (test &rest result) &body body)
     (let ((label (gensym)))
       `(prog ,(make-initforms bindforms)
           ,label
           (if ,test
                (return (progn ,@result)))
           ,@body
           (psetq ,@(make-stepforms bindforms))
           (go ,label))))

(defun make-initforms (bindforms)
     (mapcar #'(lambda (b)
                     (if (consp b)
                         (list (car b) (cadr b))
                         (list b nil)))
               bindforms))

(defun make-stepforms (bindforms)
     (mapcan #'(lambda (b)
                     (if (and (consp b) (third b))
                         (list (car b) (third b))
                         nil))
               bindforms))
```

Рисунок 7-8: Реализация do.

Чтобы написать do, мы рассмотрим, что потребуется для преобразования первого выражения на Рисунке 7-7 во второе. Чтобы выполнить такое преобразование, на нужно сделать больше, чем просто получить и установить параметры макроса в правильные позиции какого-то цитированного обратной кавычкой списка. За начальным prog должен следовать список символов и их начальные привязки, которые должны быть извлечены из второго аргумента переданного do. Функция make-initforms на Рисунке 7-8 вернет тако список. Мы также должны построить список аргументов для psetq, но этот случай более сложный, потому что не все символы должны быть обновлены. На рисунке 7-8, make-stepforms возвращает аргументы для psetq. С этими двумя функциями, остальная чать определения становиться довольно простой.

Код на рисунке 7-8 отличается от того, как написано в официапльной реализации. Чтобы подчеркнуть вычисления, сделанные во врмея расширения, make-initforms и

make-stepforms былил разбиты на отдельные функции. В будущем, такой код обычно будет оставться в выражении defmacro.

С определением этого макроса, мы начинаем видеть, что могут делать макросы. Макрос имеет полный доступ к Lisp при построении расширения. Код используемый для парширения может сам по себе быть программой.

7.8 7-8 Стиль Макросов

Хороший стиль означает нечто другое для макросов. Стиль имеет значени, когда код либо читается людьми либо вычисляется Lisp. С макросами, оба эти действия происходят при немного необычных обстоятельствах.

Существуепт два различных вида кода, связанный с определением макроса: код расширителя, это код используемый макросом для генерации своего расширения, и код расширения, который появляется в самом расширении. Для каждого из этих типов кода принципы стиля разные. Для программы в целом, иметь хороший стиль - значит быть ясной и эффективной. Этот принцип изменяется в противоположных направлениях в этих двух типах кода макроса: код расширителя может отдать предпочтение ясности над эффективностью, а код расширения может отдать предпочтение эффективности над ясностью.

Эффективность важнее всего в скомпилированном коде, а в компилированном коде вызовы макросов уже расширены. Если код расширителя был эффективен, его компиляция прошла бы немного быстрее, но это не повлияет на то, насколько хорошо работает программа. Поскольку расширение вызовов макросов, составляет лишь небольшую часть из работы, выполняемой компилятором, макросы которые расширяются эффективно, обычно не оказывают большого значения даже на скорость компиляции. Вы можете безопасно написать код расширителя способом наиболее быстрым, как и первую версию программы. Если расширитель кода делает ненужную работу или много тратит, ну и что? Ваше время лучше потратить на улучшение других частей программы. Конечно, если в коде расширителя есть выбор между ясностью и скоростью, выбрать ясность предпочтительнее. Определения макросво обычно тяжелее читать, чем определения функций, потому что они содержат смесь выражений, вычисляемых в различное время. Если эта путаница может быть уменьшена за счет уменьшения эффективности кода расширителя, это будет хорошим выбором.

Например, предположим, что мы хотели версию and and как макрос. Поскольку (and a b c) эквивалентно (if a (if b c)), мы може написать and в терминах if как в первом определении на Рисунке 7-9. Согласно стандартам по которым мы судим об обычном коде, our-and написан плохо. Код расширителя рекурсивный, и в каждой рекурсии ищет длину последующих cdrs одного и того же списка. Если этот код будет вычисляться во время выполнения, было бы лучше определить этот макрос как в our-andb, который генерирует такое же расширение не тратя услий в пустую. Тем не менее, в качестве определения макроса our-and о очень хорош, если не лучший. Он может быть не эффективен по длительности вызова для каждой рекурсии, но его организация показывает более ясно, каким образом расширение зависит от числа коньюнктов(аргументов and).

```
(defmacro our-and (&rest args)
      (case (length args)
        (0 t)
        (1 (car args))
        (t `(if ,(car args)
                   (our-and ,@(cdr args))))))

(defmacro our-andb (&rest args)
      (if (null args)
           t(labels ((expander (rest)
                          (if (cdr rest)
                              `(if ,(car rest)
                                    ,(expander (cdr rest)))
                              (car rest))))
              (expander args))))
```

Рисунок 7-9: Два макроса, эквивалентных and.

Как всегда, здесь есть исключения. В Лиспе, различие между временем компиляции и временем выполнения является искуственным, поэтому любое правило, которое зависит от него, также искуственно. В некоторых программах время компиляции является временем выполнения. Если вы пишете программу основной целью которой является преобразование и для этого используете макросы, все меняется: код расширителя становитсья вашей программой и расширение её выводом. Конечно, при таких обстоятельствах код расширителя должен быть написан с учетом эффективности. Тем не менее, можно с уверенностью сказать, что большинство расширителей кода (a) влияют только на сокрость компиляции, и (b) не сильно на него и влияют, т.е ясность почти всегда должна быть на первом месте.

С кодом расширения все наоборот. Ясность менее важна для расширения макроса, потому что на нее редко смотрят, особенно другие люди. Запрещенные goto не полностью запрещенн в расширениях, и пренебрегаемый setq не совсем так уж пренебрегаем.

Сторонники структурного программирования не любили goto за то что он делал с исходным кодом(а он превращал его в "лапшу"). Это были не инструкции jump машинного языка, которые они считали вредными - до тех пор пока они были скрыты абстрактными конструкциями в исходном коде. Goto осуждены в Lisp именно потому, что их так легко спрятать: вместо них вы можете использовать do, и если у вас его нет, вы можете написать его. Конечно, если мы собираемся создавать новые абстракции поверх goto, goto будет существовать. Таким образом, это не обязательно плохой стиль использовать go в определении нового макроса, если его нельзя написать в терминах какого-либо существующего макроса.

Точно так же использование setq не одобряется, потому что трудно увидеть, где данная переменная получает свое значение. Однако, расширение макроса не будет читаться многими людьми, поэтому обычно использование setq для создания переменных приносит мало вреда в рамках расширения макроса. Если вы посмотрите на расширения некоторых из встроенных макросов, вы можете увидеть довольно много setq.

Несколько обстоятельств могут сделать ясность более предпочтительной в расширенном коде генерируемым макросом. Если вы пишете сложный макрос, вы можете читать расширение, по крайней мере пока, пока вы отлаживаете макрос. Кроме того, в простых макросах, только обратная кавычка разделяет расширяющий код от расширенного кода, поэтому если такие макросы генерируют не красивые расширения, его корявость будет слишком видна в вашем исходном коде. Тем не менее, даже когда ясность кода расширения становиться проблемой, эффективность все еще должна быть в приоритете. Эффективность важна в большинстве кода работающем во время выполнения. Две вещи делают это особенно важным для макро расширений: их повсеместность и невидимость.

Макросы часто используются для реализации утилит общего назначения, которые затем вызываются в программах везде. То что используется так часто, не может позволить себе быть неэффективным. То что похоже на безобидный маленький макрос, может после расширения всех обращений к нему составлять значительную часть вашей программы. Такой макрос должен получать больше внимания, чем казалось бы требует его длина. Особенно избегайте consing. Утилита которая требует ненужных усилий, может испортить производительность программы, которя в противном случае была бы эффективной.

Другая причина взглянуть на эффективность расширенного кода - это его невидмость. Если функция плохо реализована, это будет видно, каждый раз когда вы будет смотреть на ее определение. Но это не так с макросами. Из определения макроса, не очевидна не эффективность в расширенном коде, что является еще одной причиной смотреть на него.

7.9 7-9 Зависимость от макросов

Если вы переопределите функцию, другие функции, вызывающие ее, автоматически получат новую версию.[6] То же самое не всегда верно для макросов. Вызов макроса, который происходит в опеделении функции заменяется в ней его расширением, когда функция компилируется. Что если мы переопределим макрос после вызова компиляции функции? Поскольку не осталось никаких следов первоначального вызова макроса, расширени внутри функции не может быть обновлено. поведение функции будет отражать старое определние макоса:

```
> (defmacro mac (x) `(1+ ,x))
MAC
> (setq fn (compile nil '(lambda (y) (mac y))))
#<Compiled-Function BF7E7E>
> (defmacro mac (x) `(+ ,x 100))
MAC
> (funcall fn 1)
2
```

Подобные проблемы возникают, если код вызывающий некоторый макрос, скомпилирован раньше самого определения макроса. CLTL2 говорит что "определение

[6] За исключение встраиваемых функций, которые накладывают те же ограничения на переопределение, что и макросы.

макроса должно быть видно компилятором перед первым использованием макроса." Реализации по разному отвечают на нарушение этого правила.К счастью, легко избежать обоих типов проблемы. Есил вы придерживаетесь следующих двух принципов, вам не нужно беспокоиться о устаревших или несуществующих определениях макросов:

1. 1.Определяйте макросы перед функциями(или макросами) которые вызывают их.
2. 2. Когда макрос переопределен, также перекомпилируйте все функции(или макросы) которые вызывают его, напрямую или через другой макрос.

Было предложено поместить все макросы программы в отдельный файл, чтобы было проще убедиться, что определения макросов компилируются первыми. Т.е убрать их подальше. Разумно было бы ставить универсальные макросы, такие как while в отдельный файл, но утилиты общего назначения должны быть отделены от остальной части программы в любом случае, будь то функции или макросы.

Некоторые макросы написаны только для использования в одной определенной части программы и они должны быть определены вместе с кодом, который их использует. Пока определение каждого макроса появляется перед любыми обращениями к нему, ваши программы скомпилируются нормально. Сбор всех ваших макросов, просто потому что они макросы, не будет делать ничего, кроме как уложнять чтение вашего кода.

7.10 7-10 Макросы из функций

В этом разделе описывается, как преобразовать функии в макросы. Первый шаг в переводе функции в макрос это спросить себя, нужно ли вам это делать. Не можете ли вы с таким же успехом объявить функцию встраиваемой(inline) (стр. 26)?

Однако, есть несколько законных причин для рассмотрения того, как перевести функции в макросы. Когда вы начинаете писать макросы, иногда это помогает думать так, как если бы вы пишете функцию - подход, которые обычно дает не совсем правильные макросы, но которые по крайней мере, дают вам кое - что для начала работы. Другая причина взглянуть на связь между макросами и функциями, чтобы увидеть как они отличаются. Наконец, Lisp программисты иногда хотят преобразовать функции в макросы.

Сложность перевода функции в макрос зависит от ряда свойств функции. Самый простой класс для перевода - это функции, которые

1. 1. Имеют тело, состоящее из одного выражения
2. 2. Имеют список параметров, состоящий только из имен параметров.
3. 3. Не создают новых переменных (кроме параметров).
4. 4. Не являются рекурсивными (и не являются частью взаимно рекурсивной группы).

5.

5. Не имеют параметров которые встречаются в теле более одного раза.

6.

6. Не имеют параметра, значение которого используется другим параметром стоящим перед ним в списке параметров.

7.

7. Не содержат свободных переменных.

Одной из функций, которая соответствует всем этим критериям, является встроенная функция Common Lisp second, которая возвращает второй элемен списка. Она может быть определена:

```
(defun second (x) (cadr x))
```

Если определение функции удовлетворяет всем вышеуказанным условиям, вы можете легко преобразовать ее в эквивалентное определение макроса. Просто поставьте обратную кавычку перед телом и запятую перед каждым символом из списка параметров:

```
(defmacro second (x) `(cadr ,x))
```

Конечно, макрос нельзя вызывать при всех одинаковых обстоятельствах. Он не может быть доступен в качестве первого аргумента для apply или funcall, и его не следует вызывать в окружении, где вызываемые им функции имеют новые локальные привязки. Хотя, для обычных встраиваемых вызовов, макрос second должен выполнять тоже самое, что и функция second.

Техника немного меняетя когда тело имеет более одного выражения, потому что макрос должен расширяться в одно выражение, так что если условие 1 не выполняется, вы должны добавить команду progn. Функция noisy-second:

```
(defun noisy-second (x)
  (princ "Someone is taking a cadr!")
  (cadr x))

может быть продублирована следующим макросом:
(defmacro noisy-second (x)
  `(progn
        (princ "Someone is taking a cadr!")
        (cadr ,x)))
```

Когда функция не соответствует условию 2 потому что она имеет параметр &rest или &body, правила теже, за исключением того, что этот параметр, а не просто запятая перед ним, должен быть "вшит" в вызов команды list. Таким образом

```
(defun sum (&rest args)
  (apply #'+ args))
```

становиться

```
(defmacro sum (&rest args)
  `(apply #'+ (list ,@args)))
```

который в этом случае будет лучше переписать как:

```
(defmacro sum (&rest args)
```

```
  `(+ ,@args))
```

Когда условие 3 не выполняется - когда новые переменные создаются в теле функции - правило о вставке запятых должно быть изменено. Вместо вставки запятых перед всеми символами в списке параметров, мы ставим их до тех пор, пока переменные сслылаются на эти параметры. Например, в:

```
(defun foo (x y z)
  (list x (let ((x y))
                (list x z))))
```

ни один из последующих двух эземпляров x не ссылается на параметр x. Второй экзепляр не вычисляется вообще, и третий экземпляр ссылается на новую переменную установленную let. Так что только первый экземпляр x получит запятую:

```
(defmacro foo (x y z)
  `(list ,x (let ((x ,y))
                  (list x ,z))))
```

Функции иногда могут быть преобразованы в макросы когда условия 4, 5 и 6 не выполнены. Тем не менее, эти темы рассматриваются в последующих главах отдельно. Проблема рекурсии в макросах рассматривается в Разделе 10-4, и опасность множественных и неправильных(неупорядоченных) вычислений в разделе 10-1 и 10-2, соответственно.

Что касается условия 7, то можно моделировать замыкания с помощью макросов, используя технику похожую на ошибку(error) описанную на стр. 37. Но учитывая что это низкоуровевый хак, не созвучный с основным тоном этой книги, мы не будем вдаваться в подробности.

7.11 7-11 Макросы Символы

CLTL2 ввел новый тип макросов в Common Lisp, это макрос-символ(symbol-macro). В то время как обычный вызов макроса выглядит как вызов функции, "вызов" макроса-символа выглядит как символ.

Макрос-символ может быть определен только локально. Специальная форма symbol-macrolet внутри своего тела может заставить одиночный символ вести себя как выражение:

```
> (symbol-macrolet ((hi (progn (print "Howdy")
                                          1)))
        (+ hi 2))
"Howdy"
3
```

Тело symbol-macrolet будет вычисляться так, как будто каждый аргумент в позции hi будет заменен на (progn (print "Howdy") 1).

Концептуально, макросы символы похожи на макросы, которые не принимают никаких аргументов. Без аргументов, макросы становятся просто текстовыми сокращениями. Однако, это не говорит что макросы символы бесполезны. Они используются в главе 15 (стр 205) и главе 18 (стр 237), и в последнем случае они не заменимы.

8 8 Когда использовать Макросы

Как нам узнать, должна ли данная функция действительно быть функцией, а не макросом? В большинстве случаев существует четкое различие между необходимостью вызова макроса и когда его использовать не надо. По умолчанию мы должены использовать функции: т.е не использовать макрос там, где работу может сделать функция. Мы должны использовать макросы, только там, где они приносят определенное преимущество.

Когда макросы приносят преимущества? Это предмет этой главы. Обычно вопрос стоит не о преимуществе, а в необходимости. Большинство вещей которые мы делаем с помощью макросов, мы не могли бы сделать используя функции. В разделе 8-1 перечислены виды операторов, которые можно реализовать только как макрос. Тем не менее, есть небольшой (но интересный) класс пограничных случаев, в которых оператор может записан как функция или как макрос. Для этих ситуаций, в разделе 8-2 приведены аргументы за и против использования макроса. Наконец, рассмотрев, что могут делать макросы, мы обратимся в разделе 8-3 к связанному с ним вопросу: Какие вещи люди делают используя их?

8.1 8-1 Когда ничего другое не помогает

Это общий принцип хорошего дизайна: если вы находите похожий код в несокольких местах программы, вы должны написать подпрограмму и заменить аналогичные последовательности кода на вызовы подпрограммы. Когда мы применяем этот принцип к Lisp программам, мы должны решить, должна ли "подпрограмма" быть функцией или макросом.

В некоторых случаях легко выбрать написание макроса вместо функции, поскольку макрос может сделать то что нужно. Функцию, подобную 1+ можно записать как функцию или как макрос:

```
(defun 1+ (x) (+ 1 x))

(defmacro 1+ (x) `(+ 1 ,x))
```

Но while, из Раздела 7-3, можно определить только как макрос:

```
(defmacro while (test &body body)
  `(do ()
             ((not ,test))
           ,@body))
```

Нет способа продублировать поведение этого макроса с помощью функции. Определение while "вшивает" переданные ему в качестве тела выражения в тело цикла do, где они будут вычисляться только, если тестовое выражение возвращает nil. Но нет функции спобоной сделать это; при вызове функции, все ее аргументы вычисляются до того как функция будет вызвана.

Когда вам нужен макрос, что вам от него нужно? Макросы могут сделать только две вещи, которые не могут сделать функции: они могут контролировать (или предотвращать) вычисление своих аргументов, и они расширяются прямо в вызывающем контексте. Любому приложению, которому требуются макросы, в конце концов, требуется одно или оба этих свойства.

Неформальное объяснение, что "макросы не вычисляют своих аргументов" немного не правильно. Точнее было бы сказать, что макросы контролируют вычисление аргументов переданных в вызове макроса. В зависимости от того, где находиться аргумент в расширении макроса, он будет вычислен, один раз, много раз или вообще не будет вычисляться. Макросы используют это управление четремя основными способами:

1. 1. Преобразование. Макрос Common Lisp setf является одним из класса макросов, которые разбирают свои аргументы перед вычислением. Встроенной функции доступа необходимо определить обратное, целью которого установить, какой доступ производит функция. Обратное для car это rplaca, для cdr - rplacd, и так далее. С помощью setf мы можем использовать вызовы таких функций, как если бы они были переменными, значеня которых надо установить, как в (setf (car x) 'a), которая расширяется в (progn (rplaca x 'a) 'a). Чтобы выполнить этот трюк, setf должен заглянуть внутрь первого аргумента. Чтобы знать, что в приведенном выше случае требуется rplaca, setf должен увидеть, что первый аргумент - это выражение начинающееся с car. Таким образом setf, и любой другой оператор, которые преобразует свои аргументы, должен быть записан как макрос.
2. 2. Связывание. Лексические переменные должны появляться непосредственно в исходном коде. Например, первый аргумент для setq не вычисляется, поэтому все что построено на setq должно быть макросом, который расширяется в setq, а не функцией, которая его вызывает. Аналогично, для операторов типа let, аргументы которых должны выглядеть как параметры в лямбда выражении, для макросов типа do, которые расширяются в lets, и так далее. Любой новый оператор, который должен изменять лексические привязки своих аргументов должен быть написан в виде макроса.
3. 3. Условное вычисление. Все аргументы функции - вычисляются. В таких конструкциях, подобных when, когда мы хотим, чтобы некоторые аргументы вычислялись только при определенных условиях. Такая гибкость возможна только при использовании макросов.
4. 4. Многократное вычисление. Аргументы функции не просто вычисляются, они вычисляются только один раз. Нам необходим макрос для определения конструкции подобной do, где определенные аргументы должны вычисляться повторно.

   Есть также несколько способов получить пользу от встраиваемого расширения макросов. Важно подчеркнуть, что расширение появляются в лексическом контексте вызова макроса, поскольку два из трех вариантов использования макроса зависят от этого факта. Они являются:
5. 5.Использование окружения вызова. Макрос может генерировать расширение, содержащее переменные, которые связаны в контексте вызова макроса. Поведение следующего макроса:

   ```
   (defmacro foo (x)
     `(+ ,x y))
   ```

   зависит от привязки y в месте, где вызывается foo. Этот вид лексического проникновения обычно рассматривается как источник проблем, нежели как средство их решения. Обычно является плохим стилем написание подобных макросов. Идеал функционального программирования применим и к макросам: предпочтитель-

ный способ связи с макросом - через его параметры. Действительно, настолько редко требуется использовать окружение вызова, что в большинстве случаев это приводит к ошибкам. (См. главу 9.) Из всех макросов в этой книге, только макросы передачи продолжений(continuation-passing) (Глава 20) и некоторые части компилятора ATN (глава 23) используют среду вызова таким способом.

6. 6. Обертывание нового окружения. Макрос также может вызвать вычисление своих аргументов в новом лексическом окружении. Классическй пример это let, который может быть реализован как макрос использущий лямбду (стр. 144). В теле выражения, такого как (let ((y 2)) (+ x y)), y будет ссылаться на новую переменную.
7. 7. Отсутствие затрат на вызов функции. Третье последствие от встраивания расширения макроса состоит в том, что в скомпилированном коде нет никаких накладных расходов связанных с вызовом макроса. Во время выполнения вызов макроса уже заменен расширением. (То же самое, в принципе, верно для встраиваемых функций, объявленных inline.)

Примечательно, что случаи 5 и 6, когда непреднамеренно захватывается переменная, представляют собой проблему являющуюся худшей вещью, которой должен бояться автор макроса. Захват переменной обсуждается в главе 9.

Вместо седьмого способа использования макросов, было бы лучше сказать что есть шесть с половиной способов. В идельном мире, все компиляторы Common Lisp будут подчиняться объявлениям inline, и сберегать расходы на вызов функций будет задачей встраиваемых(inline) функций, а не макросов. Но идельный мир мы оставим читателю для рассмотрения в качестве упражнения.

8.2 8-2 Макрос или Функция?

В предыдущем разделе рассматривались простые случаи. Любой оператор, которому нужен доступу к его параметрам, прежде чем они будут вычислены, должен быть записан как макрос, потому что другого выбора нет. А как на счет операторов, которые могут быть записаны альтернативным способом(т.е как функции)? Рассмотрим для примера оператор avg, который возвращает среднее значение своих аргументов. Он может быть определен как функция

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

но есть хороший способ определить его как макрос,

```
(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

потому что в версия функции, влечт за собой использование ненужный вызов length, вызваемый каждый раз когда avg вызвается. Во время компиляции, вы можете не знать значения аргументов, но мы знаем, сколько их, поэтому вызов length можно сделать в этот момент(в момент компиляции). Вот несколько моментов, на которые следует обратить внимание, когда мы сталкиваемся с таким выбором:

ЗА

1. Вычисления во время компиляции. Вызов макроса включает в себя вычисления в два различных момента времени: Когда макрос расширяется и когда расширение

вычисляется. Все расширения макросов в Lisp программах выполняются когда программа компилируется, и каждая часть вычислений, которые могут быть сделаны во время компиляции, это часть вычислений не будет замедлять программу когда она работает. Если оператор может быть написан, так чтобы сделать некоторую свою работу на стадии расширении макроса, будет более эффективным сделать его макросом, поскольку какую бы работу не мог выполнить умнйл компилятор, функция должна будет выполняться во время выполнения. Глава 13 описывает макросы, такие как avg, которые выполняют часть своей работы во время фазы расширения.

2. Интеграция с Lisp. Иногда, использование макросов вместо функций делает программу более тесно интегрированной с Lisp. Вместо того чтобы писать программу для решения определенной проблемы, вы можете использовать макросы, чтобы преобразовать проблему в такую проблему, которую Lisp уже знает как решать. Это подход, когда он возможен, обычно делает программы меньше и более эффективными: меньше потому что Lisp делает часть вашей работы за вас, и более эффективными потому что производственная система Lisp имеет более высокую производительность чем пользовательские программы. Это преимущество проявляется на встроенных языках, которые описываются начиная с главы 19.
3. Экономия на вызовах функций. Вызов макроса расширяется прямо в код, в котором он находиться. Так что если вы напишите какой-то, часто используемый фрагмент когда в виде макроса, вы можете сэкономить на вызове функции, каждый раз когда он используется. В более ранних диалектах Lisp программисты пользовались этим свойством для экономии на вызове функций во время выполнения. В Common Lisp эту работу должны взять на себя функции объявленые встраиваемыми(inline). Объявляя функцию встраиваемой(inline), вы запрашиваете ее вставку в правильно скомпилированный код, вызвавший её, как это былобы с макросом. Тем не менее, здесь существует разрыв между теорией и практикой; CLTL2 (стр. 229) говорит, что "компилятор волен игнорировать эти объявления", что и делают некоторые компиляторы Common Lisp. Возможно все еще разумно использовать макросы для экономии на вызовах функций, если вы вынуждены использовать подобный компилятор.

   В некоторых случаях, объединяющих преимущества эффективности и тесной интеграции с помощью Lisp можно создать сильный аргумент для использования макросов. В запросе compiler Главы 19, объем вычислений, которые могут быть сдвинуты на время компиляции настолько велик, что оправдывает превращение всей программы в один гиганский макрос. Хотя это делается для скорости, это также приводит программу к более тесной интеграции с Lisp: в новой версии, проще использовать выражения работающие с арифметикой Lisp, например в запросах.

ПРОТИВ

4. Функции это данные, в то время как макросы больше похожи на инструкции для компилятора. Функция может быть передана как аргумент(например в apply), возвращена функцией или сохранена в структуре данных. Ничего из этого не возможно с макросами. В некоторых случаях вы можете получить то, что хотите, заключив вызов макроса в лябда-выражение. Например, это работает, если вы хотите использовать определение макроса в apply или funcall:

```
> (funcall #'(lambda (x y) (avg x y)) 1 3)
2
```

Однако это не удобно. Это не всегда работает, т.к. даже если макрос, как и avg, имеет параметр &rest, ему не всегда можно передать различное количество аргументов.

5. Ясность исходного кода. Определения макросов читать сложнее чем определения эквивалентных определений функций. Так что если написать нечто как макрос, делающий программу лишь незначительно лучше, тогда стоит подумать, на тем, что может быть лучше использовать вместо него функцию.
6. Ясность во время выполнения. Макросы иногда сложнее отлаживать, чем функции. Если вы получаете ошибку времени выполнения в коде, который содержит много вызовов макросов, код который вы видите в обратной трассе, может состоять из расширений вызовов всех этих макросов, и может иметь минимальное сходство с кодом, который вы изначально писали. И поскольку макросы исчезают при расширении, они не способствуют понимаемости происходящего во время выполнения. Обычно вы не можете использовать трассировку, чтобы увидеть как вызывается макрос. Если бы это вообще сработало, trace покажет вам вызов функции расширителя макроса, а не сам вызов макроса.
7. Рекурсия. Использование рекурсии в макросах не так просто как в функциях. Хотя функция расширения макроса может быть рекусривной, само расширение не может им быть. Раздел 10-4 посвящен теме рекусрии в макросах.

Все эти соображения должны быть учтены вместе при принятии решения, когда использовать макросы. Только опыт может сказать, что лучше. Тем не менее, примеры макросов,которые появляются в последующих главах, охватывают большинство ситуаций в которых макросы полезны. Если потенциальный макрос аналогичен приведенному здесь, тогда, вероятно, безопасно написать его написать как таковой.

Наконец, следует отметить, что ясность во время выполнения (пункт 6) редко является проблемой. Отладка кода, который использует многжество макросов, не будет такой уж сложной, как вы могли бы ожидать. Если бы определения макросов были длиной несколько сотен строк, неприятно было бы отлаживать их расширение во время выполнения. Но по крайней мере утилиты, имеют тенденцию быть написанными в небольших, доверенных слоях. Как правило их определения меньше 15 строк. Таким образом, даже если вы остлеживаете трассировку вызовов, такие макросы не сильно затуманивают ваш обзор.

8.3 8-3 Приложения для Макросов

Обдумав, что можно сделать с макросами, следующим вопросом будет: в каких приложениях мы можем их использовать? Наиболее близким к общему описанию использования макросов было бы сказать, что они используются в основном для синтаксических преобразований. Это не означает, что область применения макросов ограничена. Поскольку программы на Lisp состоят из списков[1], которые представляют собой структуры данных Lisp, "синтаксические преобразования" действительно могут иметь большое значение. Главы 19 - 24 представляют целые программы, цель

[1] Состоят из, исползуется в том смысле, что списки являются входными данными для компилятора, Функии больше не состоят из списков, как это было в более ранних диалектах

которых может быть описана как синтаксическое преобразование, и которые, по сути вся являются макросами.

Приложения макросов образуют континуум от небольших макросов общего назначения, вроде while, и до больших специальных макросов, определяемых в последующих главах. На одном конце утилиты, макросы напоминающие те, которые встроены в каждый Lisp. Они обычно маленькие, общие и написаны изолированно. Тем не менее, вы можете писать утилиты для определенных классов программ, а также когда у вас есть коллекция макросов для использования, скажем, в графических программах, так что они начинают выглядеть как язык программирования графики. На дальнем конце конитинуума, макросы, позволяющие писать на языке отличном от Lisp. Про макросы, используемые таким образом, можно сказать, что они реализуют встроенный язык.

Утилиты являются первым результатом стилья проектирования снизу-вверх. Даже когда программа слишком мала, чтобы быть построенной по слоям, она все равно может извлечь выгоду от дополнений к самым низким слоям, таким как сам Lisp. Утилита nil!, которая устанавливает свой аргумент в nil, не может быть определена иначе, чем как макрос:

```
(defmacro nil! (x)
   `(setf ,x nil))
```

Глядя на nil!, хочеться сказать, что она ничего не делает, а просто сохраняет ввод. И это верно, но все что делает любой макрос, это сохранение ввода. Если кто-то хочет подумайте об этом в этих терминах, что работа компилятора заключается в сохранении ввода на машинном языке. Значение утилит не следует недооценивать, поскольку их влияние является комулятивным: несколько слоев простых макросов могут создать различие между элегантной программой и непонятной программой.

Большинство утилит являются воплощением шаблонов. Когда вы заметили шаблон в вашем коде, рассмотрите возможность превратить его в утилиту. Шаблоны это просто то, в чем компьютеры хороши. Зачем вам следовать за ними(шаблонами), когда у вас может быть программа которая сделает это за вас? Предположим, при написании какой-то программы вы испсользуете во многих разных местах циклы do имеющие одинаковую общую форму:

```
(do ()
        ((not  condition ))
   .  body of code )
```

Когда вы обнаруживаете шаблон, повторящийся в вашем коде, этот шаблон часто имеет имя. Имя нашего шаблона это while. Если мы хотим представить его в виде новой утилиты, нам надо использовать макрос, потому что нам нужны условные и повторяющиеся вычисления. Если мы определим while используя определения со страницы 91:

```
(defmacro while (test &body body)
  `(do ()
           ((not ,test))
         ,@body))
```

тогда мы сможем заменить экземпляры шаблона на

```
(while  condition
```

```
.  body of code )
```

Это сделает код короче, а также объявит то что он делает более ясным голосом.

Возможность преобразовывать свои аргументы делает макросы полезными при написании интерфейсов. Соответствующий макрос позволить набирать более короткий текст, составлять более простые и короткие выражения. Хотя графические интерфейсы уменьшают необходимость в написании таких макросов для конечных пользователй, программисты используют этот тип макросов как обычно. Самый распространенный пример это defun, который делает привязку функций, похожей на определения в языках Паскаль или Си. Глава 2 упоминала, что следующие два выражения имеют примерно одинаковый эффект:

```
(defun foo (x) (* x 2))

(setf (symbol-function 'foo)
           #'(lambda (x) (* x 2)))
```

Таким образом defun может быть реализован как макрос который превращает первое выражение во второе. Мы могли бы представить его следующим образом:[2]

```
(defmacro our-defun (name parms &body body)
  `(progn
         (setf (symbol-function ',name)
                 #'(lambda ,parms (block ,name ,@body)))
         ',name))
```

Макросы подобные while и nil! могут быть описаны как утилиты общего назначения. Любая программа Lisp может их использовать. Но определенные домены могут иметь свои утилиты

```
(defun move-objs (objs dx dy)
      (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
        (dolist (o objs)
           (incf (obj-x o) dx)
           (incf (obj-y o) dy))
        (multiple-value-bind (xa ya xb yb) (bounds objs)
           (redraw (min x0 xa) (min y0 ya)
                     (max x1 xb) (max y1 yb)))))

(defun scale-objs (objs factor)
      (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
        (dolist (o objs)
           (setf (obj-dx o) (* (obj-dx o) factor)
                  (obj-dy o) (* (obj-dy o) factor)))
        (multiple-value-bind (xa ya xb yb) (bounds objs)
           (redraw (min x0 xa) (min y0 ya)
                     (max x1 xb) (max y1 yb)))))
```

Рисунок 8-1: Исходные move и scale.

[2] Для ясности, эта весрия игнорирует всю бухгалтерию, которую должен выполнять defun.

также. Нет оснований полагать, что базовый Lisp является единственным уровнем, на котором у вас есть для расширений язык программирования. Например, если вы пишете программу CAD, лучшие результаты иногда получаются при написании ее на двух уровнях: язык (или, если вы предпочитаете более скромный термин, инструментарий) для программ CAD, и в слое выше, ваше конкретное приложение.

Lisp стирает множество различий, которые другие языки воспринимают как должное. В другоих языках действительно существует концептуальное различие между временем компиляции и временем выполнения, программой и данными, языком и программой. В Lisp эти различия существуют только как разговорные соглашения. В нем нет разделительной линии, например, между языком и программой. Вы можете сами нарисовать разделительную линию соответствующую проблеме. Так что это не более чем вопрос терминологии, следует ли называть нижележащий слой кода инструментарием или языком. Одно преимущество рассмотрения его как язык, предполагает, что вы можете расширить этот язык, как вы делаете это с Lisp-ом, создавая утилиты.

Предположим, мы пишем интерактивную программу для 2D рисования. Для простоты, мы будем предполагать, что единственными объектами, обрабатываемыми программой, являются отрезки, представленные как начало x,y и вектор смещения dx, dy. Одна из вещей, которую должна делать такая программа это слайд группы объектов. Это цель функции move-objs на Рисунке 8-1. Для эффективности мы не хотим перерисовывать весь экран после каждой операции - только те части, которые изменились. Следовательно

```
(defmacro with-redraw ((var objs) &body body)
      (let ((gob (gensym))
              (x0 (gensym)) (y0 (gensym))
              (x1 (gensym)) (y1 (gensym)))
        `(let ((,gob ,objs))
            (multiple-value-bind (,x0 ,y0 ,x1 ,y1) (bounds ,gob)
               (dolist (,var ,gob) ,@body)
               (multiple-value-bind (xa ya xb yb) (bounds ,gob)
                     (redraw (min ,x0 xa) (min ,y0 ya)
                               (max ,x1 xb) (max ,y1 yb)))))))

(defun move-objs (objs dx dy)
      (with-redraw (o objs)
        (incf (obj-x o) dx)
        (incf (obj-y o) dy)))

(defun scale-objs (objs factor)
      (with-redraw (o objs)
        (setf (obj-dx o) (* (obj-dx o) factor)
                    (obj-dy o) (* (obj-dy o) factor))))
```

Рисунок 8-2: Преобразованные move и scale.

два вызова функции bounds, которая возвращает четыре координаты (min x, min y, max x, max y) представляют ограничивающий прямоугольник. Действующая часть функции move-objs зажата между двумя вызовами bound, которые ищуют ограничивающий прямоугольник до и после выполнения перемещения, и затем перерисовыем(redraw) весь измененный регион.

Функция scale-objs предназначена для изменения размера группы объектов. Поскольку ограничивающая область может увеличиваться или уменьшаться в зависимости от маштабирующего коэффициэнта, эта функция так же должна выполнять свою работу между двумя вызовами bounds. Если бы мы писали больше программ, мы бы увидели больше повторений этого шаблона: в функциях для поворота(rotate), переворота(flip), транспонирования(transpose), и так далее.

С помощью макроса, мы можем абстрагировать общий код, который имеют эти функции. Макрос with-redraw на Рисунке 8-2 предоставляет общий скелет функций представленных на Рисунке 8-1.[3]. В результате, эти функции можно теперь определить в четыре строки, каждую, как в конце рисунка 8-2. С этими двумя функциями новый макрос уже окупил себя по краткости. И насколько яснее становяться две функции, когда мы абстрагируемся от деталей перерисовки экрана, скрываем их макросом.

Один из способов посмотреть на with-redraw, рассматривать ее как конструкцию языка для написания интерактивных программ рисования. Поскольку мы разрабатываем больше таких макросов, они будут напоминать язык программирования, как по сути, так и по названию, и само наше приложение начнет показывать элегантность, которую можно ожидать от программы, написанную на языке, определенном для этих конкретных потребностей.

Другое основное использование макросов - реализация встроенных языков. Lisp является исключительно хорошим языком для написания языков программирования, потому что программы на Lisp могут быть выражены в виде списков, а Lisp имеет встроенный парсер(читатель/read) и компилятор (compile) для программ выраженных таким образом. Большую часть времени вам даже не надо вызывать компиляцию; Вы можете скомпилировать встроенный язык неявно, путем компиляции кода, который выполняет преобразования (стр. 25).

Встроенный язык, это язык написанный не столько поверх Lisp-а, сколько в сочетании с ним, так что его синтаксис представляет собой смесь Lisp-а и конструкций специфичных для нового языка. Наивный способ реализовать встроенный язык - это написать интерпретатор для него на Lisp-е. Лучший подход, когда это возможно, реализовать язык путем преобразований: преобразовывать каждое выражение в код Lisp, который интерпретатор должен был исполнить в порядке его вычисления. Это то место где используются макросы. Работа макросов состоит в том, чтобы точно преобразовать один тип выражений в другой, поэтому они являются естественным выбором при написании встроенных языков.

В целом, чем больше встроенный язык может быть реализован путем преобразований, тем лучше. С одной стороны, это уменьшает количество работы. Например,

[3] Определение этого макроса предвосхищает следующую главу использующу gensyms. Её назначение будет объяснено в ближайшее время.

если новый язык имеет арифметику, вам не нужно сталкиваться со всеми сложностями представления и манипулирования числовыми величинами. Если арифметические возможности Lisp достаточны для ваших целей, то вы можете просто преобразовать свои арифметические выражения в эквивалентные выражения Lisp, а остальное ставить Lisp-у.

Также использование преобразований обычно делает встроенные языки быстрее. Интерпретаторам присущи недостатки в отношении скорости. Например, когда код исполняется в цикле, интерпретатору, часто, приходиться повторять работу по преобразованию кода на каждой итерации, которая в скомпилированном коде могла быть выполнена только один раз. Поэтому, встроеный язык, который имеет свой собственный интерпретатор будет медленне, даже если интерпретатор сам скомпилирован. Но если выражения в новом языке превращаются в Lisp, полученный код может быть скомпилирован компилятором Lisp. Язык, реализованный таким образом, во время выполнения не страдает от накладных расходов на интерпретацию. Если не писать настоящий компилятор для вашего языка, макросы покажут лучшую скорость. Фактически, макросы которые преобразуют новый язык, можно рассматривать как компилятор для него - просто возлагающий большую часть работы на существующий компилятор Lisp.

Мы не будем рассматривать примеры встроенных языков, так как глаы 19-25 целиком посвящены этой теме. Глава 19 конкретно рассматривает разницу между интерпретацией и преобразованием встроенных языков и показывает один и тот же язык реализованный каждым из двух методов.

Одна книга по Common Lisp утверждает что область применения макросов ограничена, ссылаясь в качестве доказательства на тот факкт, что из операторов определенных в CLTL1, менее 10% были макросами. Это все равно, что сказать, что поскольку наш дом сделан из кирпича, наша мебель должна быть сделана из него тоже. Доля макросов в программе Common Lisp будет зависеть полностью от того, что она должна делать. Некоторые программы не содержат макаросов, а некоторые программы целиком могут быть макросами.

9 9 Захват Переменных

Макрос уязвимы для проблемы, называемой захватом переменных. Захват переменной происходит когда расширение макроса вызывает конфликт имен: когда какой то символ заканчивается ссылкой на переменную из другого контекста. Случайный захват переменной может вызвать трудно уловимые ошибки. Эта глава о том, как предвидеть и избежать их. Тем не менее, преднамеренный захват переменных является полезным методом программирования, и глава 14 полна макросов, которые полагаются на него.

9.1 9-1 Захват Аргумента Макроса

Макрос уязвимый к непреднамеренному захвату переменных, является макросом с ошибкой. Чтобы избежать написания такого макроса, мы должны точно знать, когда может произойти захват. Отдельные случаи захвата переменной можно отнести к одной из двух ситуаций: захват аргумента макроса и захват свободного символа. При захвате аргумента, символ передаваемый как аргумент в вызов макроса непреднамеренно ссылается на переменную, установленную расширением самого макроса. Рассмотрим следующее определение макроса for, который выполняет итерации для выражений body подобно циклу в for в языке Pascal:

```
(defmacro for ((var start stop) &body body)                    ; wrong
  `(do ((,var ,start (1+ ,var))
         (limit ,stop))
        ((> ,var limit))
      ,@body))
```

На первый взгляд этот макрос выглядит правильно. Кажеться он работает нормально:

```
> (for (x 1 5)
       (princ x))
12345
NIL
```

Действительно, ошибка настолько неуловима, что мы можем использовать эту версию макроса сотни раз, и он всегда будет работать отлично. Однако если мы вызовем этот макрос так:

```
(for (limit 1 5)
  (princ limit))
```

Мы ожидаем, что это выражение будет иметь тот же эффект, что и раньше. Но оно ничего не печатает; оно генерирует ошибку. Чтобы понять почему оно не работает, надо посмотреть на его расширение:

```
(do ((limit 1 (1+ limit))
        (limit 5))
       ((> limit limit))
  (princ limit))
```

Теперь очевидно, что идет не так. Здесь есть конфликт имен между локальным символом расширения макроса и символом переданным как аргумент в макрос. Рас-

ширение макроса захватывает limit. Этот символ возникает дважды в одном и том же do, что является ошибкой.

Ошибки, вызываемые захватом переменных редки, но то чего они не добирают в частоте, они берут злобностью. Этот захват был сравнительно мягким - здесь мы, по крайней мере, получили ошибку. Чаще всего, захватывающий переменную макрос будет просто выдвать не верные результаты без указания того, что пошло не так. В этом случае,

```
> (let ((limit 5))
        (for (i 1 10)
           (when (> i limit)
              (princ i))))
NIL
```

полученный код тихо ничего не делает.

9.2 9-2 Захват Свободного Символа

Менее часто, само определение макроса содержит символ, который непреднамеренно ссылается на привязку в окружении, где разворачивается макрос. Предположим, что какая то программа, вместо того чтобы печатать предупреждения пользователю по мере их возникновения, хочет сохранить предупреждения в списке, чтобы рассмотреть их позже. Один человек пишет макрос gripe, который принимает предупреждающее сообщение и добавляет его в глобальный список w:

```
(defvar w nil)

(defmacro gripe (warning)                                          ; wrong
  `(progn (setq w (nconc w (list ,warning)))
              nil))
```

Кто-то еще хочет написать функцию sample-ratio, возвращающую соотношени длин двух списков. Если любой из списков содержит менее двух элементов, функция должна возвращать nil вместо него, таже выдавая предупреждение, что она была вызвана для статистически незначимого случая. (Фактические предупреждения могут быть более информативными, но их содержание не имеет отношения к этому примеру.)

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
       (if (or (< vn 2) (< wn 2))
           (gripe "sample < 2")
           (/ vn wn))))
```

Если sample-ratio вызывается с w = (b), то он захочет выдать предупреждение, что один из его аргументов, имеет только один элемент, статистически не значим. Но когда вызов gripe будет расширен, он создаст код такой, как если бы sample-ratio было определено:

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
       (if (or (< vn 2) (< wn 2))
           (progn (setq w (nconc w (list "sample < 2")))
```

```
                    nil)
            (/ vn wn))))
```

Проблема здесь в том, что gripe используется в контексте, где w имеет свою собственную локальную привязку. Предупреждение, вместо сохранения в глобальном списке предупреждений, будет добавлено (nconced) в конец одного из параметров sample-ratio. Предупреждение не только будет потеряно, но и список (b), который вероятно используется в качестве данных в другом месте программы, будет добавлена посторонняя строка:

```
> (let ((lst '(b)))
       (sample-ratio nil lst)
       lst)
(B "sample < 2")
>w
NIL
```

9.3 9-3 Когда Происходит Захват

Множество авторов макросов просят посмотреть определение макроса и предугадать все возможные проблемы, возникающие из этих двух типов захвата. Захват переменной является хитрым вопросом, и требуется некоторый опыт, чтобы предвидеть все пути, которыми захватываемый символ может нанести вред программе. К счастью, вы можете обнаружить и устранять захватываемые символы в ваших определениях макросов без размышлений о том, как этот захват может испортить вашу программу. Этот раздел предоставляет простое правило для обнаружения захватываемых символов. Остальные разделы этой главы объясняют методы их устранения.

Правило для определения захватываемой переменной зависит от некоторого подчиненного понятия, которое должно быть определено в перувую очередь:

> Свобода: Символ s появляется в выражении свободным, когда он используется в качестве перемменной в этом выражении, но выражение не создает привязку для него.

В следующем выражении,

```
(let ((x y) (z 10))
  (list w x z))
```

w, x и z - все встречаются свободными в выражении list, которое не устанавливает привязок. Тем не менее, включающее выражение let устанавливает привязку для x и z, поэтому в целом в let только y и w являются свободными. Обратите внимание, что

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

второй экземпляр x является свободным - он не входит в сферу действия новой привязки устанавливаемой для x.

> Каркас: Каркас расширения макроса - это всё разложение за исключением всего что было частью аргументов в вызове макроса.

Если foo определено как:

```
(defmacro foo (x y)
```

```
  `(/ (+ ,x 1) ,y))
```

и вызывается как:

```
(foo (- 5 2) 6)
```

тогда этот вызов дает расширение макроса:

```
(/ (+ (- 5 2) 1) 6)
```

Каркас этого расширения представляет собой приведенное выше выражение с "отверстиями", куда были вставлены параметры x и y:

```
(/ (+    ____  1)  ____)
```

С определением этих двух концепций(свободы и каркаса), становиться возможно сформулировать правило для обнаружения захватываемых символов:

> Захватываемый: символ может быть захвачен в некотором разложении макроса, если (a) он появляется свободным в каркасе разложения макроса, или (b) он является связанным частью каркаса, в которой аргументы, передаваемые макросу, либо связываются, либо вычисляются.

Несколько примеров прояснят смысл этого правила. В простейшем случае:

```
(defmacro cap1 ()
  '(+ x 1))
```

x является захватываемым, поскольку он встречается свободным в каркасе макроса. Это то что вызвало ошибку в макросе gripe. В этом макросе:

```
(defmacro cap2 (var)
  `(let ((x ...)
             (,var ...))
        ...))
```

x является захватываемым, поскольку он связывается в выражении, где также связывается аргумент вызова макроса. (Это то, что пошло не так в макросе for.) Аналогично и для следующих двух макросов

```
(defmacro cap3 (var)
  `(let ((x ...))
        (let ((,var ...))
           ...)))

(defmacro cap4 (var)
  `(let ((,var ...))
        (let ((x ...))
           ...)))
```

в обоих x является захватываемым. Однако, если нет контекста, в котором привязка x и переменная переданная как аргумент будут обе видны, как в

```
(defmacro safe1 (var)
  `(progn (let ((x 1))
                   (print x))
                (let ((,var 1))
                   (print ,var))))
```

здесь x не будет захватываемой. Не все переменные, связанные в каркасе, находятся в опсности быть захваченными. Однако, если аргументы в вызове макроса вычисляются в установленной привязке в соответствии с каркасом,

```
(defmacro cap5 (&body body)
  `(let ((x ...))
        ,@body))
```

тогда связанные в ней переменные находятся под угрозой захвата: в cap5, x является захватываемой. Хотя в это случае,

```
(defmacro safe2 (expr)
  `(let ((x ,expr))
        (cons x 1)))
```

x не может быть захвачена, потому что, когда переданный аргумент expr вычисляется, новая привязка x не будет видна. Обратите внимание, что это только связывание каркасных переменных, о которых мы должны беспокоиться. В этом макросе

```
(defmacro safe3 (var &body body)
  `(let ((,var ...))
        ,@body))
```

ни один символ не подвергается риску случайного захвата (при условии, что пользователь ожидает, что первый аргумент будет связан).

Теперь давайте посмотрим на исходное определение for в свете нового правила для идентификации захватываемых символов:

```
(defmacro for ((var start stop) &body body)                    ; wrong
  `(do ((,var ,start (1+ ,var))
             (limit ,stop))
            ((> ,var limit))
          ,@body))
```

Теперь выясняется, что это определение уязвимо для захвата двумя способами: limit может быть передан в качестве первого аргумента для for, как в исходном примере:

```
(for (limit 1 5)
  (princ limit))
```

но столь же опасно, если limit встречается в теле цикла:

```
(let ((limit 0))
  (for (x 1 10)
       (incf limit x))
  limit)
```

Кто-либо используя for данным способом, будет ожидать что его собственная привязка для limit будет увеличиваться в цикле, и выражение в целом вернет 55; на самом деле, полько привязка limit, созданная каркасом расширения будет увеличиваться:

```
(do ((x 1 (1+ x))
         (limit 10))
        ((> x limit))
  (incf limit x))
```

и поскольку именно она контролирует итерации, цикл никогда не прекратиться.

Правила представленные в этом разделе, должны использоваться с оговоркой, что они предназначены только как ориентир. Они даже официально не установлены, не говоря уже о формальной корректности. Проблема захвата является неопределенной, так как зависит от ожиданий. Например, в выражении типа

```
(let ((x 1)) (list x))
```

мы не считаем ошибкой то, что когда вычисляется (list x), x будет ссылаться на новую переменную. Это то, что let должен делать. Правила обнаружения захвата также неточны. Вы можете написать макросы, которые пройдут эти тесты, и которые все равно еще будут уязвимы для непреднамеренного захвата. Например,

```
(defmacro pathological (&body body)                                    ; wrong
  (let* ((syms (remove-if (complement #'symbolp)
                                        (flatten body)))
               (var (nth (random (length syms))
                         syms)))
         `(let ((,var 99))
            ,@body)))
```

Когда вызывается этот макрос, выражения в теле будут вычисляться как если бы они были в progn - но одна случайная переменная в теле может иметь другое значение. Это явно захват, но он проходит все наши тесты, потому что переменная не встречается в каркасе. Однако, на практике, эти правила будут работать почти всегда: очень редко(если вообще) кто захочет написать макрос, как в примере выше.

Уявзвимый к захвату вариант:

```
(defmacro before (x y seq)
      `(let ((seq ,seq))
          (< (position ,x seq)
              (position ,y seq))))
```

Правильная версия:

```
(defmacro before (x y seq)
      `(let ((xval ,x) (yval ,y) (seq ,seq))
          (< (position xval seq)
              (position yval seq))))
```

Рисунок 9-1: Как избежать захвата с помощью let.

9.4 9-4 Как избежать захвата с помощью Лучших Имен

Первые два раздела делят отдельные случаи захвата переменной на два типа: захват аргумента, когда символ используемый в аргументе, перехватывается привязкой устанавливаемой каркасом макроса и захват свободного символа, когда свободный символ в разложении макроса захватывается в силу внешней привязки, в месте расширения макроса. Проблема в последнем случае обычно решается путем предоставления глобальным переменным отличительных имен. В Common Lisp, имена глобальным переменным обычно дают начинающимися и заканчивающимися звездочкой. Например, переменная определяющия текущий пакет называется *package*. (Такое имя может

быть произнесено "star-package-star", чтобы подчеркнуть, что это не обычная переменная.)

Так что на самом деле ответственноть за хранение предупреждений лежит на авторе макроса gripe, сохраняя предупреждения в переменной, которому надо было назвать переменную для хранения предупреждений, чем-то вроде *warnings*, а не просто w. Если автор sample-ratio использовал бы *warnings* как параметр, тогда бы он заслужил каждую ошибку, которую он получил, но он не может быть обвинен в том, что не подумал о безопасности использоваия вызова с параметром w.

9.5 9-5 Как избежать Захвата с помощью Предварительного Вычисления

Иногда захват аргументов можно вылечить, просто вычислив подвергаемые опасности аргументы за пределами любых создаваемых асширением макроса привязок. Простейшие случаи можно обработать, начав макрос с выражения let. Рисунок 9-1 содержит две версии макроса before, который принимает два объекта и последовательность и возвращает истину если первый объект встречается в последовательности раньше, чем второй.[1]

Первое определение не верно. Его начальный let гарантирует, что форма переданная как seq будет вычислена только один раз, но этого не достаточно, чтобы избежать следующих проблем:

```
> (before (progn (setq seq '(b a)) 'a)
                       'b
                       '(a b))
NIL
```

Это равносильно тому, чтобы спросить "Стоит ли a перед b в (a b)?" Если before верно, он вернул бы истину. Расширение макроса показывает, что на самом деле происходит: вычисление первого аргумента операции < переустанавливает список для поиска.

```
(let ((seq '(a b)))
        (< (position (progn (setq seq '(b a)) 'a)
                             seq)
           (position 'b seq)))
```

Чтобы избежать этой проблемы, достаточно вначале вычислить все аргументы в одном большом let. Таким образом, второе определение на рисунке 9-1 защищено от захвата.

К сожалению, метод использования let работает в узком диапазоне случаев: макросы где

1. все аргументы, подвергаются риску захвата, вычисляются только один раз, и
2. ни один из аргументов не должен вычисляться в области охвата привязок установленых каркасом макроса.

[1] Этот макрос используется только как пример. На самом деле это не долно быть реализовано как макрос, ни испольовать неэффективный алгоритм, который он показывает. Для праильного определения см. стр. 50.

Это правило исключает множество макросов. Предлагаемый макрос нарушает оба условия. Тем не менее, мы можем использовать вариант этой схемы, чтобы сделать макросы подобными защищенным от захвата: обернув его формы тела в лямбда выражение снаружи любых локально созданных привязок.

Некоторые макросы, в том числе используемые для итерации, дают расширения, где выражение появляющиеся в вызвое макроса будет вычисляться в рамках вновь созданных привязок. Например в определении for тело цикла должно вычисляться в do созданным макросом. Таким образом, переменные встречающиеся в теле цикла также уязвимы к захвату привязками устанавливаемыми в do. Мы можем защитить переменные в теле от такого захвата, обернув тело в замыкание, и внутри цикла, вместо вставки самих выражений, просто выполнить вызов замыкания.

Рисунок 9-2 показывает версию for которая использует этот метод. После замыкания

Уязвимая к захвату:

```
(defmacro for ((var start stop) &body body)
      `(do ((,var ,start (1+ ,var))
              (limit ,stop))
            ((> ,var limit))
          ,@body))
```

Правильная версия:

```
(defmacro for ((var start stop) &body body)
      `(do ((b #'(lambda (,var) ,@body))
              (count ,start (1+ count))
              (limit ,stop))
            ((> count limit))
          (funcall b count)))
```

Рисунок 9-2: Избежание захвата с помощью замыкания.

являющимся первым, что создает расширение for, свободные символы встречающиеся в теле body будут ссылаться на переменные из окружения вызова макроса. Теперь do взаимодействует с телом body через параметры замыкания. Все что нужно знать замыканию от do это номер текущей итерации, поэтому оно имеет только один параметр, символ указанный как переменная индекса в вызове макроса.

Техника обертывания выражений в лямбда функции не является универсальным средством. Вы можете использовать её для защиты кода тела, но замыкания не годяться когда, например, существует риск того, что одна и та же переменная будет дважды связана одним и тем же let или do (как в нашем исходном не работающем for). К счастью, в этом случае, переписав for и упаковав его тело в замыкание, мы также избавились от необходимости устанавливать привязки для аргумента var. Аргумент var старого for стал параметром замыкания и может быть заменен в do на символ count. Таким образом, новое определение for полностью невоспримиимчиво к захвату, как покажет тест в Разделе 9-3.

Недостаток использования замыканий состоит в том, что они могут быть менее эффективными. Мы можем ввести другой вызов функции. Потенциально хуже, если

компилятор не дает замыканию динамического пространства, пространство для него должно быть выделено в куче во время выполнения.

9.6 9-6 Как избежать захвата с помощью Gensyms

Есть один определенный способ избежать захвата аргументов макроса: заменить захватываемые символы с помощью gensyms. В исходной версии for, проблемы возникают когда два символа имеют одно и тоже имя. Если мы хотим избежать этого, т.е что бы каркас макроса содержал символ так же используемый в вызывающем коде, мы могли бы уповать на использование в определении макроса символов со странными именами:

```
(defmacro for ((var start stop) &body body)                    ; wrong
   `(do ((,var ,start (1+ ,var))
              (xsf2jsh ,stop))
             ((> ,var xsf2jsh))
           ,@body))
```

но это не решение проблемы. Это не устраняет ошибку, просто снижает вероятность ее проявления. И не так уж и менее вероятно, а все еще весьма возможно представить конфликты, возникающие во вложенных экземплярах одного и того же макроса.

Нам нужен какой-то способ убедиться, что символ уникален. Функция Common Lisp gensym сущетсвует только для этой цели. Она возвращает символ, вызывая gensym, который гарантированно не будет равен любому другому символу, набранному или составленному в программе.

Как Lisp может обещать это? В Common Lisp, каждый пакет содержит списко всех символов известных в этом пакете. (Для ознакомления с пакетами, см. стр. 381.) Символ, который находиться в списке называется интернированным(внедренным) в пакет. Каждый вызов gensym взвращает уникальный, интернированный символ. И так как каждый символ при чтении получает интернирование, никто не может напечатать что либо идентичное gensym. Таким образом, если вы начинаете выражение

(eq (gensym) ...

нет способа завершить его, чтобы оно вернуло истину.

Запросить gensym создать нам символ, все равно что, выбрать подход использующий символы со странными именами, но на один шаг более продвинутый - gensym даст нам символ, чьего имени нет даже в телефонной книге. Когда Lisp должен отобразить gensym,

```
> (gensym)
#:G47
```

то что он печатает, на самом деле просто Lisp эквивалент "John Doe", вымышленного произвольного имени, чьё имя не имеет значения. И чтобы быть уверенным, что это не какие то наши иллюзии, отображаемым символам gensym предшествует решетка с двоеточием, специальный макрос чтения, который существует только для того, чтобы вызывать ошибку, если мы когда-нибудь попытаемся прочитать gensym повторно.

Уязвимый для захвата:

```
(defmacro for ((var start stop) &body body)
      `(do ((,var ,start (1+ ,var))
              (limit ,stop))
             ((> ,var limit))
          ,@body))
```

Правильная версия:

```
(defmacro for ((var start stop) &body body)
      (let ((gstop (gensym)))
         `(do ((,var ,start (1+ ,var))
                 (,gstop ,stop))
               ((> ,var ,gstop))
            ,@body)))
```

Рисунок 9-3: Как избежать захвата с помощью gensym.

В CLTL2 Common Lisp, число в печатном представлении gensym происходит от *gensym-counter*, глобальной переменной всегда связанной с целым числом. Сбрасывая этот счетчик, мы можем заставить два gensyms печататься одинаково

```
> (setq x (gensym))
#:G48
> (setq *gensym-counter* 48 y (gensym))
#:G48
>(eq x y)
NIL
```

но они не будут идентичными.

Рисунок 9-3 содержит правильное определение for использующее gensyms. Там нет символа limit, который может совпасть с символами из форм переданных в макрос. Он был заменен символом сгенерированным gensym на месте. В каждом расширении макроса, вместо limit будет взят уникальный символ, созданый во время расширения.

Правильное определение for является сложным для получения с первой попытки. Законченный код, как и законченая теорема, часто проходят через много проб и ошибок. Так что не беспокойтесь , если вам придется написать несколько версий макроса. Начиная писать макрос подобный for, вы можете написать первую версию, не думая о захвате переменной, а затем вернуться и сделать gensyms для символов, которые можгут быть вовлечены в захваты.

9.7 9-7 Как избежать захвата с помощью пакетов(packages)

В некоторой степени можно избежать захвата, определяя макросы в своих собственных пакетах. Если вы создаете макрос в пакете и определяете for в нем, вы можете использовать определение данное первым

```
(defmacro for ((var start stop) &body body)
     `(do ((,var ,start (1+ ,var))
             (limit ,stop))
```

```
      ((> ,var limit))
    ,@body))
```

и безопасно вызывать его из любого другого пакета. Если вы вызываете for из другого пакет, скажем mycode, тогда даже если вы будете использовать limit в качестве первого аргуемента, это будет mycode::limit - символ отличный от символа macros::limit, который встречается в каркасе макроса.

Тем не менее, пакеты не дают общего решения проблемы захватов. Во-первых, макросы являются неотъемлемой частью некоторых программ, и было бы неудобно разделять их на отдельные пакеты. Во-вторых, этот подход не обеспечивает защиты от захвата другим кодом в макросах пакета.

9.8 9-8 Захват в Других Пространствах Имен

В предыдущих разделах говорилось о захвате, как если бы это была проблем свойственная исключительно переменным. Хотя большая часть проблем захвата это захват переменных, проблема может возникнуть и в других пространствах имен Common Lisp.

Функции так же могут быть локально связаны и привязки функций одинаково ответственны за непреднамеренный захват. Например:

```
> (defun fn (x) (+ x 1))
FN
> (defmacro mac (x) `(fn ,x))
MAC
> (mac 10)
11
> (labels ((fn (y) (- y 1)))
        (mac 10))
9
```

Как и предсказывает правило захвата, fn который встречается свободным в каркасе mac подвергается риску захвата. Когда fn локально пересвязывается, mac возвращает другое значение, чем ожидалось.

Что делать в этом случае? Когда символом подвергающимся риску захвата является встроенная функци или макрос, то разумнее ничего не делать! В CLTL2 (стр. 260) если имя чего-либо встроенного имеет локальную привязку функции или макроса, последствия не определены." Так что не имеет значения, что делал ваш марос, некто, кто перепривязывает встроенные функции, будет иметь проблемы не только с вашим макросом..

В противном случае, вы можете защитить имена функций от захвата аргументами макроса так же, как защищали бы имена переменных: используя gensyms как имя для любых функций, заданных локальными определениями в каркасе макроса. Избежать захвата свободных симолов, как в случае выше, немного сложнее. Способ защитить переменные от захвата свободных символов, дать им отчетливые глобальные имена: такие как *warnings* вместо w. Это решение не практично для функций, потому что не существует соглашения о различении имен глобальных функций - а большинство функций являются глобальными. Если вы обеспокоены тем. что макрос вызывается

в окружении, где функция может быть переопределена, вероятно лучшим решением, будет поместить ваш код в отдельный пакет.

Имена блоков также могут быть захвачены, как и теги используемые go и throw. Когда вашим макросам нужны такие символы, вы должны использовать gensyms, как в определении our-do на стр. 98.

Помните также , что операторы типа do неявно заключены в блок с именем nil. Таким образом, возврат(return) или (return-from nil) в пределах do возвращает из do, а не из содержащего его выражения:

```
> (block nil
        (list 'a
                (do ((x 1 (1+ x)))
                     (nil)
                   (if (> x 5)
                        (return-from nil x)
                        (princ x)))))
12345
(A 6)
```

Если бы do не создавалось в блоке с именем nil, этот пример вернул бы просто 6, а не (A 6).

Неявный блок(block) в do не является проблемой, потому что do и должен вести себя таким образом. Однако, вы должны понимать, что если вы пишите макросы, которые расширяются в do, они захватят имя блока nil. В макросе вроде for, return или return-from nil будет возвращать из выражения for, а не из вмещающего блока(block).

9.9 9-9 Зачем Беспокоиться?

Некоторые из предыдущих примеров довольно патологические. Глядя на них, некто может сказать "захват переменных маловероятен - зачем беспокоиться о нём?". Есть два способа ответить на этот вопрос. Один ответ с другим вопросом: зачем писать программы с небольшими ошибками, когда вы можете писать программы без ошибок?

Более линный ответ состоит в том, чтобы указать, что в реальных приложениях это опасно, предполагать что-нибудь о том, как будет использоваться ваш код. Любая программа на Lisp имеет то, что сейчас называется "открытой архитектурой." Если вы пишете код другим людям, они могут использовать его так, как вы никогда бы не ожидали. Вы должны беспокоитсья о том, что это просто люди. Программы тоже пишут программы. Может быть, что ни один человек не написал бы код так

```
(before (progn (setq seq '(b a)) 'a)
            'b
            '(a b))
```

но код, сгенерированный программами, часто выглядит подобным образом. Даже если отдельные макросы генерируют простые и разумные расширения, как только вы начнете вкладывать вызовы макросов, расширения могут стать большими программами, которые не похожи ни на что, что написал бы человек. При таких обстоятельствах стоит защищаться от случаев, какими бы надуманными они не были, когда они могут привести к неправильному расширению ваших макросов.

В конце концов, избежать захвата переменных не так уж и сложно. Это скоро станет вашей второй натурой. Классический Common Lisp defmacro похож на нож повара: элегантная идея, которая кажется опасной, но которой эксперты уверенно пользуются.

10 10 Другие Ошибки в Макросах

Написание макросов требует особой осторожности. Функция изолирована в своем собственном лексическом мире, но макрос. поскольку он расширяется в вызывающем его коде, может преподнести пользователю неприятный сюрприз, если он не будет написан очень тщательно. Глава 9 обяснила захват переменных, самый большой подобный сюрприз. В этой главе рассматриваются больше проблем, чтобы избежать их при определении макросов.

10.1 10-1 Число Вычислений

Несколько неправильных версий for появились в предыдущей главе. Рисунок 10-1 показывает еще два, в сопровождении правильной версии для сравнения.

Хотя он не уязвим к захвату, второй for содержит ошибку. Он будет создавать расширение в котором, форма переданная в качестве условия остановки(stop) будет вычисляться на каждой итерации. В лучшем случае, этот вид макроса не эффективен, многократно вычислять то, что можно вычислить один раз. Если stop имеет сторонние эффекты, макрос на самом деле может дать неправильные результаты. Например, этот цикл никогда не прекратиться, потоу что цель отступает(отодвигается) на каждой итерации:

```
> (let ((x 2))
     (for (i 1 (incf x))
        (princ i)))
12345678910111213...
```

При написании макросов подобных for нужно помнить, что аргументы макроса это формы, а не значения. В зависимости от ого, где они появляются в расширении, они могут вычислятья более одного раза.

Правильная версия:

```
(defmacro for ((var start stop) &body body)
      (let ((gstop (gensym)))
        `(do ((,var ,start (1+ ,var))
                (,gstop ,stop))
              ((> ,var ,gstop))
            ,@body)))
```

Подлежит многократному вычислению:

```
(defmacro for ((var start stop) &body body)
      `(do ((,var ,start (1+ ,var)))
           ((> ,var ,stop))
         ,@body))
```

Неверный порядок вычилений:

```
(defmacro for ((var start stop) &body body)
      (let ((gstop (gensym)))
        `(do ((,gstop ,stop)
                (,var ,start (1+ ,var)))
              ((> ,var ,gstop))
            ,@body)))
```

Рисунок 10-1: Управление вычислением аргументов

В этом случае решение состоит в том, чтобы привязать переменную к значению возвращающему формой stop, и обращаться к этой переменной во время цикла.

Если они явно не предназначены для итерации, макросы должны гарантировать, что выражения вычисляются ровно столько раз, сколько они появляются в вызове макроса. Сущствуют очевидные случаи, когда это правило не применяется: Common Lisp или будет менее полезным (он станет Pascal или) если все его рагументы будут всегда вычисляться. Но в таких случаях пользователь точно знает сколько вычислений ожидать. Это не так для второй версии for: у пользователя нет причин предполагать, что форма stop вычисляетя более одного раза и на самом деле нет причин, по которым это должно происходить. Макрос написанный подобно второй версии for вероятнее всего написан так по ошибке.

Непреднамеренные множественные вычисления являются особенно сложной проблемой для макросов построенных на setf. Common Lisp предоставляет несколько утилит для упрощения написания таких макросов. Эта проблема и её решение обсуждаются в Главе 12.

10.2 10-2 Порядок вычислений

Порядок в котором вычисления выражений, хотя и не так важен, как количество их вычислений, но иногда может стать проблемой. В Common Lisp при вызове функции, аргументы вычисляются слева на право:

```
> (setq x 10)
10
> (+ (setq x 3) x)
6
```

и хорошей практикой, будет, делать то же самое для макросов. Макросы обычно должны обеспечивать вычисление выражений в том же порядке, в котором они появляются в вызове макроса.

На Рисунке 10-1, третья версия for также содержит хитрую ошибку. Параметр stop будет вычисляться до параметра start, даже если они отображаются в обратном порядке при вызове макроса:

```
> (let ((x 1))
        (for (i x (setq x 13))
         (princ i)))
13
NIL
```

Этот макрос дает приводящее в замешательство ощущение о возвращении во времени. Вычисление формы stop влияет на значение, возвращаемое формой start, даеж если форма start появляется в текстовом виде - первой.

Правильная весрия for гарантирует, что его аргументы будут вычисляться в том порядке, в котором они появляются:

```
> (let ((x 1))
        (for (i x (setq x 13))
         (princ i)))
12345678910111213
NIL
```

Теперь установка x в форме stop не повлияет на значение возвращаемое предыдущими аргументами.

Хотя предыдущий пример является надуманным, есть случаи, когда подобного рода проблемы действительно происходили, и такую ошибку чрезвычайно сложно найти. Возможно мало кто напишет код, в котором вычисление одного аргумента макроса влияет на значение возвращаемое другим, но люди могут случайно сделать то, что никогда бы не сделали нарочно. Утилита не должна маскировать ошибки, если используется по назначению. Если кто нибудь написал код, как в предыдущих примерах, вероятно сделал он это по ошибке, но правильная версия for упростит обнаружение ошибки.

10.3 10-3 Не функциональный код Расширителя

Lisp ожидает, что код, который генерирует расширение макросов, будет чисто функциональным, в смысле описанным в Главе 3. Расширяющий код должен зависеть только от форм передаваемых ему в качестве аргументов, и не должен оказвать влияние на мир, кроме как путем возвращения значений.

Начиная с CLTL2 (стр. 685), можно с уверенностью предположить, что вызовы макросов в скомпилированном коде не подлежит повторному расширению во время выполнения. В противном случае, Common Lisp не дает никаких гарантий о том, когда и как часто будет расширен вызов макроса. Считается ошибкой для расширения макроса варьироваться в зависимости от того или друго. Например, предположим, мы хотели посчитать, сколько раз используется какой либо макрос. Мы не можем просто выполнить поиск по исходным файлам, потому что макрос может быть вызван в коде,

который генерируется самой программой. Поэтому мы моглибы захотеть определить макрос следующим образом:

```
(defmacro nil! (x)                                                    ; wrong
     (incf *nil!s*)
     `(setf ,x nil))
```

С этим определением, глобальный *nil!s* будет увеличиваться при каждом вызвове расширения макроса nil!. Тем не менее, мы ошибаемся, если ожидаем, что значение этой переменной скажат нам, как часто вызвался nil!. Данный вызов может быть и частым, если он расширен более одного раза. Например, препроцессору, который выполняет преобразования в вашем исходном коде, возможно, придется расширить вызовы макросов в выражения, прежде чем он сможет определить, стоит ли его преобразовывать.

Основное правило, код расширителя не должен зависеть ни от чего. кроме его аргументов. Так что любому макросу, например, который строит свое расширение из строк, следует быть осторожным, чтобы не предполагать, что пакет от которого зависит расширение макроса, будет загружен во время расширения. Это краткий, но довольно патологический пример,

```
(defmacro string-call (opstring &rest args)                           ; wrong
     `(,(intern opstring) ,@args))
```

определяет макрос, который принимает печатное имя оператора и расширеят вызов к нему:

```
> (defun our+ (x y) (+ x y))
OUR+
> (string-call "OUR+" 2 3)
5
```

Вызов intern принимает строку и возвращает соответствующий символ. Тем не менее, если мы опускаем необязательный аргумент package(пакет), он делает это в текущем пакете. Таким образом, расширение будет зависеть от текущего пакета во время создания расширения, и если our+ не виден в этом пакете, расширение будет с вызовом неизвестной функции.

Миллер и Бенсон в книге "Lisp Style and Design"(Лисп Стиль и Дизайн) упоминают один особенно некрасивый прмер проблем, возникающих из-за побочных эффектов в коде расширителя. В Common Lisp, начиная с CLTL2 (стр. 78), списки связанные с параметром &rest не являются гарантированно новыми, свеже сформированными. Они могут разделять структуру со списками находящимися в другим местах программы. Следовательно, вы не должны разрушающе изменять параметр &rest, потому что вы не знаете, что еще вы измените при этом.

Эта возможность влияет как на функции, так и на макросы. С функциями, проблемы возникают при использовании apply. В правильной реализаци Common Lisp может произойти следующее. Предположим, мы определили функцию et-al, которая возвращает список аргументов с добавлением et al в конец:

```
(defun et-al (&rest args)
  (nconc args (list 'et 'al)))
```

Если бы мы вызвавали эту функцию обычным образом, она бы работала нормально:

```
> (et-al 'smith 'jones)
```

```
(SMITH JONES ET AL)
```

Однако, если мы вызываем ее через apply, она могла бы изменить существующие структуры данных:

```
> (setq greats '(leonardo michelangelo))
(LEONARDO MICHELANGELO)
> (apply #'et-al greats)
(LEONARDO MICHELANGELO ET AL)
> greats
(LEONARDO MICHELANGELO ET AL)
```

По крайней мере, правильная реализация Common Lisp могла бы сделать это, хотя, кажется пока, так не делает никто.

Для макросов, опасность больше. Макрос который изменил параметр &rest тем самым может изменить вызов макроса. То есть, вы можете случайно создать самопереписывающую программу. Опасность этого также более реальна - эта на самом деле происходит при использовании существующих реализаций лиспа. Если мы определим макрос, которые соединяет(nconcs) нечто с его аргументом &rest[1]

```
(defmacro echo (&rest args)
  `',(nconc args (list 'amen)))
```

и затем определим функцию, которая вызывает его:

```
(defun foo () (echo x))
```

В одном широко используемом Common Lisp, произойдет следующее:

```
> (foo)
(X AMEN AMEN)
> (foo)
(X AMEN AMEN AMEN)
```

foo не только возвращает неправильный результат, он каждый раз возвращает отличный от предыдущего результат, потому что после каждого расширения макроса изменяется определение foo.

Этот пример также иллюстрирует высказанную ранее мысль о множественных расширениях данного вызова макроса. В этой конкретной реализации, первый вызов foo возвращает списки с двумя amens. По какой то причине эта реализация расширила вызов макроса когда foo был определен, и также по одному разу в каждом из последующих его вызовов.

Было бы более безопасно определить echo как:

```
(defmacro echo (&rest args)
  `'(,@args amen))
```

потому что, запятая с "собакой"(comma-at) эквивалентны дополнению, а не nconc. После переопределения этого макроса, функцию foo тоже нужно будет переопределить, даже если она не скомпилирована, потому что предыдущая версия echo вызвала его перезапись.

В макросах этой опасности подвержены не только параметр &rest. Любой аргумент макроса, который является списком, должен быть оставлен в покое. Если мы

[1] ‘’,(foo) это эквивалент ‘(quote ,(foo)).

определим макрос, который изменяет один из своих аргументов, функцию которая его вызывает,

```
(defmacro crazy (expr) (nconc expr (list t)))

(defun foo () (crazy (list)))
```

тогда исходный код вызывающей функции может быть изменен, как это происходит в одной реализации лиспа. В первый раз мы вызваем ее:

```
> (foo)
(T T)
```

Это происходит как в скомпилированном, так и в интерпретиуемом коде.

Заключение: не пытайтесь избежать конструирования списков путем деструктивного изменения параметра являющегося списковой структурой. Получающиеся программы будут не переносимы, если вообще будут работать. Если вам необходмо избежать конструирования списка(cons) в функции которая принимает переменное количество аргументов,

одним из решений будет использование макроса, и таким образом, смещения конструирования списка(consing) вперед, с времени выполнения, на время работы компилятора. Для такого применения макросов, смотри Главу 13.

Также следует избегать разрушающих операций над выражениями возвращаемых расширителем макроса, если эти выражения содержат заквотированные списки. Это не является ограничением для макросов как таковых, но является примером принципа, изложенного в разделе 3-3.

10.4 10-4 Рекурсия

Иногда естественно определять функцию рекурсивно. Имеется несколько по сути рекурсивных функций функций подобных такой:

```
(defun our-length (x)
  (if (null x)
          0(1+ (our-length (cdr x)))))
```

Это определение, кажется более естественным(хотя вероятно более медленным), чем его итерационный эквивалент:

```
(defun our-length (x)
  (do ((len 0 (1+ len))
           (y x (cdr y)))
          ((null y) len)))
```

Функция, которая не является ни рекурсивной, ни частью некоторого взаимнорекурсивного набора функций, может быть преобразована в макрос, с помощью простого метода, описанного в разделе 7-10. Однако, простая вставка кавычек и запятых не будет работать с рекурсивной функцией. Давайте возьмем встроенный nth в качестве примера. (Для простоты, наша версия nth не будет проверять ошибки.) На рисуноке 10-2 показана ошибочная попытка определить nth как макрос. Внешне, nthb по видимому эквивалентно ntha, но программа, содержащая вызов nthb не будет компилироваться, поскольку расширение вызова никогда не прекратиться.

В целом, для макросов нормально содержать ссылки на другие макросы, если расширение шаблона где то заканчивается. Проблема с nthb в том, что каждое расширение содержит ссылку на сам nthb. Функциональная версия ntha завершается, потому что она получает значение n, которое уменьшается при каждом рекурсивном вызове. Но расширитель макроса имеет доступ только к формам, но не к их значениям. Когда компилятор пытается разширить макрос, скажем (nthb x y), первое расширение даст

```
(if (= x 0)
        (car y)
        (nthb (- x 1) (cdr y)))
```

Это будет работать:

```
(defun ntha (n lst)
      (if (= n 0)
           (car lst)
           (ntha (- n 1) (cdr lst))))
```

Это не компилируется:

```
(defmacro nthb (n lst)
      `(if (= ,n 0)
             (car ,lst)
             (nthb (- ,n 1) (cdr ,lst))))
```

Рисунок 10-2: Ошибочная аналогия с рекурсивной функцией.

который в свою очередь расшириться в:

```
(if (= x 0)
        (car y)
        (if (= (- x 1) 0)
             (car (cdr y))
             (nthb (- (- x 1) 1) (cdr (cdr y)))))
```

и так далее в бесконечный цикл. Это нормально для макроса расширении иметь вызов самого себя, если такое поведение рано или поздно заканчивается.

Опасная вещь в рекурсивных макросах, таких как nthb, в том, что они обычно хорошо работают под интерпретатором. Затем, когда у вас наконец-то рабочая программа и вы пытаетесь ее скомпилировать, она даже не компилируется. Хотя только это, но обычно это указывает на то, что проблема связана с рекурсивным макросом; компилятор просто входит в бесконенчный цикл и оставляет вас в раздумьях, о том, что что-то пошло не так.

В этом случае, ntha является хвостовой(хорошей) рекурсией. Функция с хвостовой рекурсией может быть легко превращена в итерационный эквивалент, а затем использована в качестве модели для макроса. Макрос типа nthb может быть написан

```
(defmacro nthc (n lst)
  `(do ((n2 ,n (1- n2))
            (lst2 ,lst (cdr lst2)))
           ((= n2 0) (car lst2))))
```

поэтому, в принципе, возможно продублировать рекурсивную функцию с помощью макроса. Однако, преобразование более сложных рекурсивных функций, может быть затруднено или даже невозможно.

```
(defmacro nthd (n lst)
   `(nth-fn ,n ,lst))

(defun nth-fn (n lst)
   (if (= n 0)
        (car lst)
        (nth-fn (- n 1) (cdr lst))))

(defmacro nthe (n lst)
   `(labels ((nth-fn (n lst)
                     (if (= n 0)
                        (car lst)
                        (nth-fn (- n 1) (cdr lst)))))
       (nth-fn ,n ,lst)))
```

Рисунок 10-3: Два способа решения проблем.

В зависимости от того, для чего вам нужен макрос, вы можете найти достаточным для использование сочетания макроса и функции. Рисунок 10-3 показывает два способа создать то, что кажеться рекурсивным макросом. Первая стратегия, воплощенная в nthd, просто заставить расширяться макрос в вызов рекурсивной функции. Например, если вам нужен макрос для того, чтобы избавить пользвателей от необходимости квотировать аргументы, тогда такого подхода должно хватить.

Если вам нужен макрос, потому что вы хотите вставить всё его расширение в лексическое окружение, где происходит вызов макроса, тогда вы, скорее всего, захотите последовать примеру определяющему nthe. Встроенная метка специальной формы (Раздел 2-7) создает локальное определение функции. В то время как, каждое расширение nthc будет вызывать глобально определенную функцию nth-fn, каждое расширение nthe будет иметь свою собственную версию такой функции внутри себя.

Хотя вы не можете перевести рекурсивную функцию непосредственно в макрос, вы можете написать макрос, расширение которого генерируется рекурсивно. Функция расширения макроса это обычная функция Lisp и, конечно, она может быть рекурсивной. Например, если бы мы определяли версию встроенного or, мы хотели бы использовать рекурсивную функцию расширяния.

На рисунке 10-4 показаны два способа определить рекурсивную функцию расширения для or. Макрос or вызывает рекурсивную функцию or-expand для генерации своего расширения. Этот макрос будет работать, равно как и эквивалентный orb. Хотя orb рекурсивный, он рекурсивен по аргументам макроса (которые доступны во время расширения макроса), а не по их значениям (которые не доступны). Может показаться, что расширение будет содержать ссылку на само себя orb, но вызов orb генерируемый одним

```
(defmacro ora (&rest args)
      (or-expand args))

(defun or-expand (args)
      (if (null args)
              nil
         (let ((sym (gensym)))
               `(let ((,sym ,(car args)))
                     (if ,sym
                        ,sym
                        ,(or-expand (cdr args)))))))

(defmacro orb (&rest args)
      (if (null args)
         nil
         (let ((sym (gensym)))
               `(let ((,sym ,(car args)))
                     (if ,sym
                        ,sym
                        (orb ,@(cdr args)))))))
```

Рисунок 10-4: Функции рекусрсивного расширения.

шагом расширения макроса, будет заменен на let в следующем шаге, приводя окончательное расширение, в ничто иное как стек вложенных let; (orb x y) расширяется в код эквивалентный:

```
(let ((g2 x))
  (if g2
         g2
         (let ((g3 y))
           (if g3 g3 nil))))
```

На самом деле ora и orb эквивалентны, и какой стиль использовать - это вопрос личного предпочтения.

11 11 Классические Макросы

В этой главе показано, как определить наиболее часто используемые типы макросов. Они деляться на три категории- с большим количеством пересечений. Первой группой являются макросы которые создают контекст. Любой оператор, который вычисляет свои аргументы в новом контексте, вероятно должен быть определен как макрос. Первые два раздела описывыают два основных типа контекста и показывают, как определять макросы для каждого из них.

В следующих трех разделах описаны макросы для условных и повторных вычислений. Оператор, аргументы которого должны вычисляться менее одного раза, или более чем один раз, также должен быть определен как макрос. Там нет четкого различия межд операторами для условных и повторных вычислений: некоторые примеры в этой главе делают оба вычисления(а также связывание). Последний раздел объясняет другое сходство между условными и повторяющимися вычислениями: в некоторых случаях можно выполнять оба типа вычислений используя функции.

11.1 11-1 Создание контекста

Контекст здесь имеет два смысла. Одним из видов контекста является лексическое окружение(среда). Специальная форма let создает новое лексическое окружение; выражения в теле let будут вычисляться в окружении, которое может содержать новые переменные. Если x установлен в a на верхнем уровне, то

```
(let ((x 'b)) (list x))
```

выражение вернет все же (b), поскольку вызов list будет сделан в окружении, содержащем новый x, чьим значением является b.

```
(defmacro our-let (binds &body body)
      `((lambda ,(mapcar #'(lambda (x)
                                    (if (consp x) (car x) x))
                                binds)
          ,@body)
        ,@(mapcar #'(lambda (x)
                           (if (consp x) (cadr x) nil))
                  binds)))
```

Рисунок 11-1: Реализация макроса let.

Оператор, который должен иметь тело выражений, обычно должен быть определен как макрос. За исключением случаев, таких как prog1 и progn, цель такого оператор, обычно будет вызвать вычисление тела в каком то новом контексте. Марос должен будет обернуть вокруг тела код, создающий контекст, даже если контекст не включает новые лексические переменные.

Рисунок 11-1 показывает, как let может быть определен через макрос использующий lambda. our-let расширяется в применение функции -

```
(our-let ((x 1) (y 2))
  (+ x y))
```

expands into

```
((lambda (x y) (+ x y)) 1 2)
```

Рисунок 11-2 содержит три новых макроса, которые устанавливают лексическое окружение. Раздел 7-5 использовал when-bind в качестве примера деструктуризации списка параметров, поэтому этот макрос уже был описан на странице 94. Более общий when-bind* принимает список пар вида (symbol expression) - той же формы, что и первый аргумент let. Если какое либо выражение возвращает nil, всё выражение when-bind* вернет nil. В противном случае его тело будет вычисляться, как будто каждый символ связан с помощью let*:

```
> (when-bind* ((x (find-if #'consp '(a (1 2) b)))
                        (y (find-if #'oddp x)))
        (+ y 10))
11
```

Наконец, макрос with-gensyms сам по себе используется для написания макросов. Много определений макросов начинаются с создания символов с помощью gensyms, иногда их чило очень больше. Макрос with-redraw (стр. 115) должен был создать пять:

```
(defmacro when-bind ((var expr) &body body)
   `(let ((,var ,expr))
         (when ,var
           ,@body)))

(defmacro when-bind* (binds &body body)
   (if (null binds)
          `(progn ,@body)
          `(let (,(car binds))
                (if ,(caar binds)
                   (when-bind* ,(cdr binds) ,@body)))))

(defmacro with-gensyms (syms &body body)
   `(let ,(mapcar #'(lambda (s)
                              `(,s (gensym)))
                          syms)
         ,@body))
```

Рисунок 11-2: Макросы которые связывают переменные.

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
          (x0 (gensym)) (y0 (gensym))
          (x1 (gensym)) (y1 (gensym)))
        ...))
```

Такие определения гораздо проще с with-gensyms, который связывет целый список переменных со символами генерируемыми gensym. С новым макросом мы могли бы просто записать:

```
(defmacro with-redraw ((var objs) &body body)
  (with-gensyms (gob x0 y0 x1 y1)
        ...))
```

Этот новый макрос будет использоваться в оставшихся главах.

Если мы хотим связать некоторые переменные, а затем от некоторых условий, вычислить одно из набора выражений, мы просто используем условное выражение в let:

```
(let ((sun-place 'park) (rain-place 'library))
  (if (sunny)
          (visit sun-place)
          (visit rain-place)))
```

```
(defmacro condlet (clauses &body body)
      (let ((bodfn (gensym))
            (vars (mapcar #'(lambda (v) (cons v (gensym)))
                          (remove-duplicates
                            (mapcar #'car
                                    (mappend #'cdr clauses))))))
        `(labels ((,bodfn ,(mapcar #'car vars)
                          ,@body))
           (cond ,@(mapcar #'(lambda (cl)
                               (condlet-clause vars cl bodfn))
                           clauses)))))

(defun condlet-clause (vars cl bodfn)
      `(,(car cl) (let ,(mapcar #'cdr vars)
                    (let ,(condlet-binds vars cl)
                      (,bodfn ,@(mapcar #'cdr vars))))))

(defun condlet-binds (vars cl)
      (mapcar #'(lambda (bindform)
                  (if (consp bindform)
                      (cons (cdr (assoc (car bindform) vars))
                            (cdr bindform))))
              (cdr cl)))
```

Рисунок 11-3: Комбинация cond и let.

К сожалению, нет удобной идиомы длля противоположной ситуации, когда мы хотим всегда вычислить один и тот же код, но где привязки длжны отличаться в зависимости от некоторых условий.

Рисунок 11-3 содержит макрос предназначеный для таких ситуаций. Назаченное ему имя, предполагает, что condlet ведет себя как потомок cond и let. Он требует

в качестве аргументов список связывающих предложений, за которым следует тело кода. Каждое из связвающих предложений защищено тестовым выражением; тело кода будет вычислено с привязками указанными предложением, чье тестовое выражене первым вернет истинное значение. Переменные, которые встречаются в одних предложениях и их нет в других будут связанаы с nil, если в успешном предложении не указаны привязки для них:

```
> (condlet (((= 1 2) (x (princ 'a)) (y (princ 'b)))
            ((= 1 1) (y (princ 'c)) (x (princ 'd)))
            (t       (x (princ 'e)) (z (princ 'f))))
        (list x y z))
CD
(D C NIL)
```

Определение condlet можно понять как обобщение определения our-let. Последнее превращает свое телол в функцию, которая применяется к результатам вычислений инициирующих значений форм. condlet расширяется в код, которые определяет локальную функцию используя labels; в нем предложение cond определяет, какой набором форм инициализирующих значения будет выполнен и передан в функцию.

Обратите внимание, что расширитель использжует mappend вместо mapcan для извлечения имен переменных из связывающих предложений. Это сделано потому, что mapcan является деструктивной функций, а как указано в Разделе 10-3, опасно изменять структуру списка параметров.

11.2 11-2 Макрос with-

Существует еще один вид контекста по мимо лексического окружения. В более широком смысле, контекст - это состояние мира, в том числе значений специальных переменных, содержимое структур данных и состояние снаружи Lisp. Операторы, которые создают этот вид контекста, также должны быть определены как макросы, если только их тело кода не должно быть упаковано в замыкание.

Имена макросов создающих контекст часто начинаются с префикса with-. Большинство обычно используемых макросов этого типа, вероятно with-open-file. Его тело вычисляется в контексте вновь открытого файла, привязванного к определяемой пользователем переменной:

```
(with-open-file (s "dump" :direction :output)
  (princ 99 s))
```

После вычисления этого выражения файл "dump" автоматически закроется, и его содержимое будет состоять из двух знаков "99".

Этот оператор должен быть явно определен как макрос, поскольку он связывает переменную s. Тем не менее, операторы, которые вызывают вычисление форм в новом контексте, в любом случае должны быть определены как макросы. Макрос ignore-errors новый в CLTL2, вызывает вычисление своих аргументов как будто они выполняются в progn. Если в некоторый момент возникает ошибка, вся форма ignore-errors просто возвращет nil. (это было бы полезно, например при чтении набираемых пользователем символьных знаков). Хотя ignore-errors не создает переменных, он все равно должен быть определен как макрос, потому что его аргументы вычисляются в новом контексте.

Обычно макросы, которые создают контекст, расширяются в блок кода; дополнительные выражения могут быть помещены перед телом, после него или по обоим его сторонам. Если код появляется после тела, его цель может состоять в том, чтобы оставить систему в устойчивом состоянии - убрать, чтото уже не нужное. Например, with-open-file должен закрыть открытый им файл. В таких ситуациях обычно макрос создающий контекст, расширяется в unwind-protect.

Цель unwind-protect состоит в том, чтобы гарантировать, что определенные выражения вычислятся, даже если выполнение будет прервано. Он принимает один или более аргументов, которые вычисляются по порядку. Если все пойдет гладко, он вернет значение первого аргумента, как prog1. Разница в том, что остальные аргументы будут вычислены даже если ошибка или исключение прервет вычисление первого аргумента.

```
> (setq x 'a)
A> (unwind-protect
        (progn (princ "What error?")
                 (error "This error."))
        (setq x 'b))
What error?
>>Error: This error.
```

Форма unwind-protect в целом выдает ошибку. Однако, после возвращения на верхний уровень, мы замечаем, что второй аргумент все же был вычислен:

```
>x
B
```

Поскольку with-open-file расширяется в форму unwind-protect, открываемый файл обычно будет закрыт, даже если во время вычисления его тела произойдет ошибка.

Макросы создающие контекст в основном пишуться для конкретных приложений. Так например, предположим , что мы пишем программу, которая работает с несколькими базам данных. Одновременно программа общается только с одной базой данных, указанной в переменной *db*. Перед использованием базы данных мы должны заблокировать её, чтобы никто другой не смог использовать ее одновременно с нами. Когда мы закончим, мы должны снять блокировку. Если мы хотим получить значение запроса q в базе данных db, мы можем сказать что то вроде:

```
(let ((temp *db*))
     (setq *db* db)
     (lock *db*)
     (prog1 (eval-query q)
              (release *db*)
              (setq *db* temp)))
```

С помощью макрса мы можем скрыть всю эту бухгалтерию. Рисунок 11-4 определяет макрос, который позволит нам иметь дело с базами данных н а более высоком уровне абстракции. С помощью with-db, мы бы просто сказали:

Чистый макрос:

```
(defmacro with-db (db &body body)
   (let ((temp (gensym)))
       `(let ((,temp *db*))
         (unwind-protect
            (progn
                 (setq *db* ,db)
                 (lock *db*)
                 ,@body)
            (progn
                 (release *db*)
                 (setq *db* ,temp))))))
```

Комбинация макроса и функции:

```
(defmacro with-db (db &body body)
   (let ((gbod (gensym)))
       `(let ((,gbod #'(lambda () ,@body)))
         (declare (dynamic-extent ,gbod))
         (with-db-fn *db* ,db ,gbod))))

(defun with-db-fn (old-db new-db body)
   (unwind-protect
       (progn
        (setq *db* new-db)
        (lock *db*)
        (funcall body))
       (progn
        (release *db*)
        (setq *db* old-db))))
```

Рисунок 11-4: Типичный макрос with-.

```
(with-db db
  (eval-query q))
```

Вызов with-db также безопаснее, поскольку он разворачивается в unwind-protect вместо простого prog1.

Два определения with-db на рисунке 11-4 илюстрируют два возможных пути написания этого макроса. Первый чистый макрос, второй комбинация макроса и функции. Второй подход становиться более практичным, так как

```
(defmacro if3 (test t-case nil-case ?-case)
      `(case ,test
          ((nil) ,nil-case)
          (?       ,?-case)
          (t       ,t-case)))

(defmacro nif (expr pos zero neg)
      (let ((g (gensym)))
        `(let ((,g ,expr))
            (cond ((plusp ,g) ,pos)
                      ((zerop ,g) ,zero)
                      (t ,neg)))))
```

Рисунок 11-5: Макрос для условных вычислений.

у искомого макроса with- возрастает сложность.

В CLTL2 Common Lisp, объявление dynamic-extent позволяет замыканию содержащему тело более эффективно его размещать (в реализации CLTL1 оно будет игнорироваться). Нам нужно это замкание только на время вызова with-db-fn, и объявление говорит о том же, позволяя компилятору выделить для него место в стеке. Это пространство будет автоматически восстановлено(возвращено) при выходе из выражения let, вместо того, чтобы быть восстановленым позже с помощью сборщика мусора.

11.3 11-3 Условное Вычисление

Иногда мы хотим, чтобы аргумент в вызове макроса вычслялся только при выполнении определенного условия. Это находиться за пределами возможностей функции, которые всегда вычисляют свои аргументы. Встроенные операторы, такие как if, and, и cond защищают некоторые из своих аргументов от вычислений, если другие аргументы не возвращают определенных значений. Для примера, в этом выражении

```
(if t'phew
        (/ x 0))
```

третий аргумент вызвал бы ошибку деления на ноль, если бы был вычислен. Но поскольку только первые два аргумента будут когда либо вычислены, этот if в целом будет всегда безопасно возвращать phew.

Мы можем создавать новые операторы такого рода, написав макросы, которые расширяются в вызовы существующих макросов. Два макроса на рисунке 11-5 являются двумя из многих возможных варинтов if. Определение if3 показывает, как мы можем опредилить условие для трех значной логики. Вместо того. чтобы рассматривать nil как ложь и все остальное как истину, этот макрос рассматривате три категории истин: истина(true), ложь(false) и неопределенность(uncertain), представленную как ?. Это может быть использовано в следующем описании пятилетнего ребенка:

```
(while (not sick)
  (if3 (cake-permitted)
         (eat-cake)
         (throw 'tantrum nil)
         (plead-insistently)))
```

```
(пока (не болен)
  (if3 (торт-разрешен)
       (ем-торт)
       (выброшу 'истерику nil)
       (настойчиво-умоляю)))
```

Новое условие расширяется в case. (ключ nil должен быть заключен в список, поскольку в одиночку ключ nil будет неоднознаным.) Только один из последних трех аргументов будет вычислен, в зависимости от значения первого аргумента.

Имя nif означает "числовой(numeric) if." Другая реализация этого макроса представлена на странице 86. Он принимает числовое выражение в качестве первого аргумента и в зависимости от его знака вычисляется один из трех оставшихся аргументов.

```
> (mapcar #'(lambda (x)
                  (nif x 'p 'z 'n))
              '(0 1 -1))
(ZPN)
```

Рисунок 11-6 содержит еще несколько макросов, которые используют условные вычисления. Макрос in должен эффективно проверять членство в наборе. Когда вы хотите проверить, является ли объект одним из множества альтернатив, вы можете выразить запрос как дизъюнкцию:

```
(let ((x (foo)))
  (or (eql x (bar)) (eql x (baz))))
```

или вы можете выразить это в терминах членства в множестве:

```
(member (foo) (list (bar) (baz)))
```

Последнее более абстрактнро, но менее эффективно. Выражение member привлекает неэффективность из двух источников. Это создание списка(conse), поскольку он должен собрать альтернативы в список для функции поиска члена(member). И все альтернативные формы в списке должны быть вычислены, хотя некоторые значения, возможно, никогда нам не понадобяться. Если значение (foo) равно значению (bar), тогда нет необходимости вычслять значение (baz). Безотносительно его концептуальных преимуществ, использовать member далеко не лучший способ. Мы можем получить ту же абстракцию более эффективно с помощью макроса: сочетающего абстракуию member с эффективностью or. Эквиваленое выражение in

```
(defmacro in (obj &rest choices)
      (let ((insym (gensym)))
       `(let ((,insym ,obj))
          (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                        choices)))))

(defmacro inq (obj &rest args)
      `(in ,obj ,@(mapcar #'(lambda (a)
                              `',a)
                          args)))

(defmacro in-if (fn &rest choices)
      (let ((fnsym (gensym)))
       `(let ((,fnsym ,fn))
          (or ,@(mapcar #'(lambda (c)
                            `(funcall ,fnsym ,c))
                        choices)))))

(defmacro >case (expr &rest clauses)
      (let ((g (gensym)))
       `(let ((,g ,expr))
          (cond ,@(mapcar #'(lambda (cl) (>casex g cl))
                          clauses)))))

(defun >casex (g cl)
      (let ((key (car cl)) (rest (cdr cl)))
       (cond ((consp key) `((in ,g ,@key) ,@rest))
             ((inq key t otherwise) `(t ,@rest))
             (t (error "bad >case clause")))))
```

Рисунок 11-6: Макросы для условных вычислений.

```
(in (foo) (bar) (baz))
```

имеет ту же форму, что и выражение member, но расширяется в

```
(let ((#:g25 (foo)))
  (or (eql #:g25 (bar))
      (eql #:g25 (baz))))
```

Как это часто бывает, когда сталкиваешся с выбором между ясностью выражения и эффективностью, мы получаем неразрешимую дилему, написав макрос мы получаем первое и второе, одновременно.

Произноситься как "в очереди(in queue)," inq это квотированный вариант in, так же как setq использует set. Выражение

```
(inq operator + - *)
```

расширяется в

```
(in operator '+ '- '*)
```

Как и member по умолчанию, in и inq используют eql для проверки на равенство. Когда вы захотите использовать какую то другую проверку test-or или любую другую

функцию от одного аргумента - вы можете использовать более общий in-if. Как in соответствует member, так in-if соответствует some. Выражение

```
(member x (list a b) :test #'equal)
```

может быть продублировано, как

```
(in-if #'(lambda (y) (equal x y)) a b)
```

и

```
(some #'oddp (list a b))
```

становиться

```
(in-if #'oddp a b)
```

Используя комбинацию cond и in, мы можем определить полезный варинат case. Макрос Common Lisp case предполагает, что его ключи становятся константами. Иногда нам может потребоваться поведение выражения case, но с ключами, которые вычисляются. Для таких ситуаций мы определяем >case, такой же как case за исключением того, что ключи которые охраняют каждое предложение, вычисляются перед сравнением. (Знак > в названии предназначен для предложения обозначения стрелки, используемое для представления вычислений.) Поскольку >case использует in, он вычисляет не больше ключей, чем нужно.

Поскольку ключи могут быть выражениями Lisp, невозможно определить является ли (x y) вызовом или списком из двух ключей. Чтобы избежать двусмысленности, ключи (кроме t и других) должны всегда указваться в списке, даже если есть только один из них. В выражениях case, nil не может выступать в качестве начала(car) предложения из за двусмысленности. В выражении >case, nil больше не является двусмысленным, как и начало(car) предложения, но он означает, что остальная часть предложения никогда не будет вычислена.

Для ясности, код который генерирует расширение для каждого предложения >case определяется как отдельная функция, >casex. Обратите внимание, чтот >casex сама использует inq.

```
(defmacro while (test &body body)
      `(do ()
           ((not ,test))
         ,@body))

(defmacro till (test &body body)
      `(do ()
           (,test)
         ,@body))

(defmacro for ((var start stop) &body body)
      (let ((gstop (gensym)))
        `(do ((,var ,start (1+ ,var))
              (,gstop ,stop))
             ((> ,var ,gstop))
           ,@body)))
```

Рисунок 11-7: Простые макросы итерации(повторения).

11.4 11-4 Итерации(Повторения)

Иногда проблема с функциями возникает не в том, что их аргументы всегда вычисляются, а в том что они вычисляются только один раз. Поскольку каждый аргумент функции будет вычислен ровно один раз, если мы хотим определить оператор который получает тело выражений и повторяет их вычисление, нам нужно определить его как макрос.

Простейшим примером будет макрос, который последовательно вычисляет свои аргументы forever:

```
(defmacro forever (&body body)
  `(do ()
       (nil)
     ,@body))
```

Это именно то, что делает встроенный макрос цикла loops, если вы не даете ему ключевых слов цикла. Может показаться, что у такого цикла нет будущего (или слишком много будущего) т.к он представляет бесконечный цикл. Но в сочетании с block и return-from, этот вид макроса становиться наиболее естественным способом выразить циклы, где завершение цикла, по своей природе, это чрезвычайная ситуация.

Некоторые из простейших макросов для итерации показаны на рисунке 11-7. Мы уже видели while (на стр. 91), чье тело будет вычисляться пока тестовое выражение возвращает истину. Обратное ему till, которое делает тоже самое пока тестовое выражение возвращает ложь. Наконец for, также показанное раньше (стр. 129), выполняет итерации для диапазона чисел.

По определению эти макросы расширяются в do, мы делаем доступным использование go и return в их телах. Как do наследует эти права от block и tagbody, так и while, till и for наследует их от do. Как объяснено на странице 131, метка nil неявного block обернутого вокруг do будет захвачена определенными на Рисунке 11-7. Это скорее особенность, чем ошибка, но об этом следует явно упомянуть.

Макросы необходимы когда нам нужно определить более мощную конструкцию для итерации. Рисунок 11-8 содержит два обобщения dolist; оба вычисляют свои тела с набором(tuple) переменных, связанных с последовательными подпоследовательностями списка(list). Например, если заданы два параметра do-tuples/o будет итерировать по парам:

```
> (do-tuples/o (x y) '(a b c d)
         (princ (list x y)))
(A B)(B C)(C D)
NIL
```

С теми же аргументами, do-tuples/c сделает тоже самое, а затем завернёт на начало списка, для получения недостающих аргументов:

```
> (do-tuples/c (x y) '(abcd)
         (princ (list x y)))
(A B)(B C)(C D)(D A)
NIL
```

Оба макроса возвращают nil, если в теле нет явного возрвата return.

Этот вид итерации часто необходим в программах, которые имеют дело с некоторыми представлениями пути. Суфиксы /o и /c предполагают две версии перемещения по открытым и замкнутым путям, соответственно. Например, если точки это список точек и (drawline x y) рисует линию между x и y, а затем рисует путь от первой точки к последней мы напишем.

```
(do-tuples/o (x y) points (drawline x y))
```

тогда как, если точки это список вершин многоугольника, мы рисуем его периметр написав

```
(do-tuples/c (x y) points (drawline x y))
```

Список аргументов, указанных в качестве первого аргумента, может быть любой длины, и итерация буде продолжаться кортежами этой же длины. Если задан только один параметр, оба

```
(defmacro do-tuples/o (parms source &body body)
      (if parms
         (let ((src (gensym)))
           `(prog ((,src ,source))
               (mapc #'(lambda ,parms ,@body)
                       ,@(map0-n #'(lambda (n)
                                     `(nthcdr ,n ,src))
                                 (1- (length parms))))))))

(defmacro do-tuples/c (parms source &body body)
      (if parms
         (with-gensyms (src rest bodfn)
           (let ((len (length parms)))
             `(let ((,src ,source))
                   (when (nthcdr ,(1- len) ,src)
                     (labels ((,bodfn ,parms ,@body))
                       (do ((,rest ,src (cdr ,rest)))
                            ((not (nthcdr ,(1- len) ,rest))
                             ,@(mapcar #'(lambda (args)
                                           `(,bodfn ,@args))
                                       (dt-args len rest src))
                             nil)
                         (,bodfn ,@(map1-n #'(lambda (n)
                                               `(nth ,(1- n)
                                                     ,rest))
                                           len)))))))))

(defun dt-args (len rest src)
      (map0-n #'(lambda (m)
                  (map1-n #'(lambda (n)
                              (let ((x (+ m n)))
                                (if (>= x len)
                                    `(nth ,(- x len) ,src)
                                    `(nth ,(1- x) ,rest))))
                          len))
              (- len 2)))
```

Рисунок 11-8: Макросы для итерации на подпоследовательностях.

```
(do-tuples/c (x y z) '(a b c d)
      (princ (list x y z)))
```

расширяется в:

```
(let ((#:g2 '(a b c d)))
      (when (nthcdr 2 #:g2)
        (labels ((#:g4 (x y z)
                       (princ (list x y z))))
          (do ((#:g3 #:g2 (cdr #:g3)))
              ((not (nthcdr 2 #:g3))
                 (#:g4 (nth 0 #:g3)
                         (nth 1 #:g3)
                         (nth 0 #:g2))
                 (#:g4 (nth 1 #:g3)
                         (nth 0 #:g2)
                         (nth 1 #:g2))
                 nil)
            (#:g4 (nth 0 #:g3)
                    (nth 1 #:g3)
                    (nth 2 #:g3))))))
```

Рисунок 11-9: Expansion of a call to do-tuples/c.

вырождаются в dolist:

```
> (do-tuples/o (x) '(a b c) (princ x))
ABC
NIL
> (do-tuples/c (x) '(a b c) (princ x))
ABC
NIL
```

Определение do-tuples/c является более сложным, чем определение do-tuples/o, поскольку он должен заворачивать по достижении конца списка. Если есть n параметров do-tuples/c должен выполнить еще n-1 итераций перед возвратом:

```
> (do-tuples/c (x y z) '(abcd)
       (princ (list x y z)))
(A B C)(B C D)(C D A)(D A B)
NIL

> (do-tuples/c (wxyz)'(abcd)
          (princ (list w x y z)))
   (A B C D)(B C D A)(C D A B)(D A B C)
   NIL
```

Расширение предшествующего вызова do-tuples/c показано на рисунке 11-9. Сложная часть для генерации - это последовательность вызовов представляющих переход на начало списка. Эти вызовы (в данном случае два из них) генерируются dt-args.

11.5 11-5 Итерация с Множественными Значениями

Встроенные макросы do существуют дольше, чем возврат множественных значений. К счастью do может развиваться в соответствии с новой ситуацией, поскольку эволюция Lisp находиться в руках программиста. Рисунок 11-10 содержит версию do* приспособленную для работы с множественными значениями. С mvdo*, каждое начальное предложение может связать больше чем одну переменную:

```
> (mvdo* ((x 1 (1+ x))
                 ((y z) (values 0 0) (values z x)))
               ((> x 5) (list x y z))
        (princ (list x y z)))
(1 0 0)(2 0 2)(3 2 3)(4 3 4)(5 4 5)
(656)
```

Этот вид итерации полезен, например, в интерактивных графических программах, которые часто имеют дело с несколькими величинами, таким как координаты и регионы(области).

Предположим, что мы хотим написать простую интерактивную игру, в которой объект, должен избежать зажатия между двумя преследующими его объектами. Если два преследователя оба ударят вас одновременно, вы проиграете; если они столкнутья друг с другом первыми, вы выграете. На рисунке 11-11 показано, как можно написать основной цикл этой игры, используя mvdo*.

Также возможно написать mvdo, которая связывает свои локальные переменные параллельно:

```
> (mvdo ((x 1 (1+ x))
                ((y z) (values 0 0) (values z x)))
              ((> x 5) (list x y z))
        (princ (list x y z)))
(1 0 0)(2 0 1)(3 1 2)(4 2 3)(5 3 4)
(645)
```

Необходимый для psetq в определении do был описан на странице 96. Чтобы определить mvdo нам необходима версия psetq поддерживающая множественные значения. Поскольку Common Lisp такой не имеет, мы дожны сами ее написать, как на рисунке 11-12. Новый макрос работает следующим образом:

```
(defmacro mvdo* (parm-cl test-cl &body body)
   (mvdo-gen parm-cl parm-cl test-cl body))

(defun mvdo-gen (binds rebinds test body)
   (if (null binds)
          (let ((label (gensym)))
            `(prog nil
               ,label
               (if ,(car test)
                    (return (progn ,@(cdr test))))
               ,@body
               ,@(mvdo-rebind-gen rebinds)
               (go ,label)))
          (let ((rec (mvdo-gen (cdr binds) rebinds test body)))
            (let ((var/s (caar binds)) (expr (cadar binds)))
              (if (atom var/s)
                   `(let ((,var/s ,expr)) ,rec)
                   `(multiple-value-bind ,var/s ,expr ,rec))))))

(defun mvdo-rebind-gen (rebinds)
   (cond ((null rebinds) nil)
            ((< (length (car rebinds)) 3)
             (mvdo-rebind-gen (cdr rebinds)))
            (t(cons (list (if (atom (caar rebinds))
                              'setq
                              'multiple-value-setq)
                          (caar rebinds)
                          (third (car rebinds)))
                    (mvdo-rebind-gen (cdr rebinds))))))
```

Рисунок 11-10: Связывающая множественные значения версия do*.

```
> (let ((w 0) (x 1) (y 2) (z 3))
        (mvpsetq (w x) (values 'a 'b) (y z) (values w x))
        (list wxyz))
(AB01)
```

Определение mvpsetq опирается на три служебные функции: mklist (стр 45), group (стр 47), и shuffle, определенную здесь, которая смешивает два списка:

```
(mvdo* (((px py) (pos player)              (move player mx my))
            ((x1 y1) (pos obj1)            (move obj1 (- px x1)
                                                      (- py y1)))
            ((x2 y2) (pos obj2)            (move obj2 (- px x2)
                                                      (- py y2)))
            ((mx my) (mouse-vector) (mouse-vector))
            (win       nil                 (touch obj1 obj2))
            (lose      nil                 (and (touch obj1 player)
                                                (touch obj2 player))))
           ((or win lose) (if win 'win 'lose))
      (clear)
      (draw obj1)
      (draw obj2)
      (draw player))
```

(pos obj) возвращает два значения x,y представляющих позицию объекта obj. Первоначально три объекта имеют случайное положение.

(move obj dx dy) перемещает объект obj в зависимости от его типа и вектора dx,dy . Возвращает два значения x,y указывающих на новыую позицию.

(mouse-vector) возвращает два значения dx,dy указывающих на текущее перемещение мыши.

(touch obj1 obj2) возвращает истину если obj1 и obj2 соприкасаются.

(clear) очищает игровой регион.

(draw obj) рисует obj в его текущей позиции.

Рисунок 11-11: Игра в мяч.

```
> (shuffle '(a b c) '(1 2 3 4))
(A1B2C34)
```

С помощью mvpsetq, мы можем определить mvdo как показано на Рисунке 11-13. Подобно condlet, этот макрос использует mappend вместо mapcar чтобы избежать изменения исходного вызова макроса. Идиома mappend-mklist сглаживает у дерева один уровень:

```
> (mappend #'mklist '((a b c) d (e (f g) h) ((i)) j))
(ABCDE(FG)H(I)J)
```

```
(defmacro mvpsetq (&rest args)
   (let* ((pairs (group args 2))
          (syms (mapcar #'(lambda (p)
                            (mapcar #'(lambda (x) (gensym))
                                    (mklist (car p))))
                        pairs)))
      (labels ((rec (ps ss)
                 (if (null ps)
                     `(setq
                        ,@(mapcan #'(lambda (p s)
                                      (shuffle (mklist (car p))
                                               s))
                                  pairs syms))
                     (let ((body (rec (cdr ps) (cdr ss))))
                       (let ((var/s (caar ps))
                             (expr (cadar ps)))
                         (if (consp var/s)
                             `(multiple-value-bind ,(car ss)
                                                   ,expr
                                ,body)
                             `(let ((,@(car ss) ,expr))
                                ,body)))))))
       (rec pairs syms))))

(defun shuffle (x y)
   (cond ((null x) y)
         ((null y) x)
         (t (list* (car x) (car y)
                   (shuffle (cdr x) (cdr y))))))
```

Рисунок 11-12: Версия работющего с Множественным значением psetq.

Чтобы понять этот довольно большой макрос. на рисунке 11-14 приведен пример его расширения.

11.6 11-6 Потребность в Макросах

Макрос это не единственный способ защитить аргументы от вычисления. Другой, заключается в том чтобы обернуть их в замыкание. Условные и повторяющиеся вычисления похожи, потому что ни одна проблема по своей сути не требует макросов. Например, мы могли бы написать версию

```
(defmacro mvdo (binds (test &rest result) &body body)
      (let ((label (gensym))
           (temps (mapcar #'(lambda (b)
                                    (if (listp (car b))
                                         (mapcar #'(lambda (x)
                                                        (gensym))
                                                   (car b))
                                         (gensym)))
                                binds)))
        `(let ,(mappend #'mklist temps)
           (mvpsetq ,@(mapcan #'(lambda (b var)
                                       (list var (cadr b)))
                                  binds
                                  temps))
           (prog ,(mapcar #'(lambda (b var) (list b var))
                          (mappend #'mklist (mapcar #'car binds))
                          (mappend #'mklist temps))
                 ,label
                 (if ,test
                     (return (progn ,@result)))
                 ,@body
                 (mvpsetq ,@(mapcan #'(lambda (b)
                                           (if (third b)
                                                 (list (car b)
                                                       (third b))))
                                     binds))
                 (go ,label)))))
```

Рисунок 11-13: Версия do связывающая множественные значения.

if как функцию:

```
(defun fnif (test then &optional else)
  (if test
         (funcall then)
         (if else (funcall else))))
```

Мы бы защитили аргументы then и else, выражая их как замыкания, так вместо

```
(if (rich) (go-sailing) (rob-bank))
```

```
(mvdo ((x 1 (1+ x))
           ((y z) (values 0 0) (values z x)))
          ((> x 5) (list x y z))
      (princ (list x y z)))
```

Расширяется в:

```
(let (#:g2 #:g3 #:g4)
    (mvpsetq #:g2 1
                 (#:g3 #:g4) (values 0 0))
    (prog ((x #:g2) (y #:g3) (z #:g4))
         #:g1
         (if (> x 5)
               (return (progn (list x y z))))
         (princ (list x y z))
         (mvpsetq x (1+ x)
                      (y z) (values z x))
         (go #:g1)))
```

Рисунок 11-14: Расширение вызова mvdo.

мы бы сказали

```
(fnif (rich)
        #'(lambda () (go-sailing))
        #'(lambda () (rob-bank)))
```

Если все, что нам нужно, это условные вычисления, макросы не являются абсолютно необходмыми. Они просто делают программы яснее. Тем не менее, макросы необходмы, когда мы хотим разобрать аргументы формы, или связывать переменные передаваемые в качестве аргументов.

Тоже самое отностся к макросам для итерации. Хотя макросы предлагают естественный способ определить итерационную конструкцию, за которой может следовать тело выражений, можно выполнить итерацию с использованием функций, если тело цикла само упаковано в функцию.[1] Например, встроенная функция mapc, это функциональный аналог dolist. Выражение

```
(dolist (b bananas)
   (peel b)
   (eat b))
```

имеет те же побочные эффекты, что и

```
(mapc #'(lambda (b)
                (peel b)
                (eat b))
          bananas)
```

(хотя первый возвращает nil, а последующий возвращает список бананов). Мы могли бы аналогично реализовать forever как функцию,

```
(defun forever (fn)
```

[1] Возможно написать итерационную функцию, которая не нуждается в аргументах, обернутых в функцию. Мы могли бы написать функцию, которая вызывала бы eval для выражений переданных ей в качестве аргументов. Для объяснения того, почему обычно плохо вызывать eval, смотри стр. 278.

```
(do ()
        (nil)
      (funcall fn)))
```

если бы мы были готовы передать это как замыкание вместо тела выражений.

Тем не менее, итерационные конструкции обычно хотят делать больше, чем просто итерацию, как это forever делает: они обычно хотят сделать комбинацию связвания и итерации. С использованием функций, перспективы связывания ограничены. Если вы хотите связать переменные для последовательности элементов списков, вы можете использовать одну из функций отображения(mapping). Но если требования усложняются, вам придется писать макрос.

12 12 Обобщенные переменные

В главет 8 упоминалось, что одним из преимуществ макросов является их способность преобразовывать свои аргументы. Одним из макросов такого рода является setf. Эта глава рассматривает смысл(подтекст) setf, и затем показывает некотоыре примеры макросов, которые могут быть построены на подобном принципе.

Написание правильных макросов на setf удивительно сложно. Чтобы вникнуть в тему, в первом разделе приведен простой пример, который немного неверен. В следующем разделе объясняется, что не так с этим макросом, и показывается как исправить это. В третьем и четвертом разделах представлены примеры утилит, основанных на setf, и в последнем разделе объясняется как определить ваши собственные инверсии setf.

12.1 12-1 Концепция(Идея)

Встроенный макрос setf является обобщением setq. Первый аргумент для setf может быть вызовом, представляющим собой запрос на определение значения переменной в некоторой структуре данных(например в списке), а не простой переменной:

```
> (setq lst '(a b c))
(ABC)
> (setf (car lst) 480)
480
> lst
(480 B C)
```

В общем (setf x y) можно понимать как высказывание "смотри чтобы этот x имел значение равное y." Как макрос setf может заглянуть внутрь своих аргументов, чтобы увидеть, что нужно сделать, чтобы сделать это утверждение правдой. Если первый аргумент (после расширения макроса) является символом, setf просто расширяется в setq. Но если аргумент является запросом, setf расширяется в соответствующее утверждение. Поскольку второй аргумент это константа, предыдущий пример может расшириться до:

```
(progn (rplaca lst 480) 480)
```

Это преобразование из запроса в утверждение называется инверсией. Все самые часто используемые функции доступа Common Lisp имеют предопределенные инверсии, включаяя car, cdr, nth, aref, get, gethash, и функции доступа, созданные defstruct. (Полный список приведен в CLTL2, стр. 125.)

Выражение, которое может служить первым аргументом для setf называется обобщенной переменной. Обобщенные переменные оказались мощной абстракцией. Вызов макроса напоминает обобщенную переменную тем, что любой вызов макроса, который расширяется в обратимую ссылку, сам будет обратимым.

Когда мы также пишем наши собственные макросы поверх setf, эта комбинация приводит к заметному улучшению ясности программы. Одним из макросов, которые мы можем определить поверх setf является toggle(переключатель),[1]

```
(defmacro toggle (obj)                                          ; wrong
```

[1] Это определение не является правильным, почему объясняется в следующем разделе.

```
  `(setf ,obj (not ,obj)))
```

который переключает (логическое) значение обобщенной переменной:

```
> (let ((lst '(a b c)))
       (toggle (car lst))
       lst)
(NIL B C)
```

Теперь рассмотрим следующий пример приложения. Предположим, кто-то - писатель мыльных опер, энергичный занятый человек, или партийный чиновник - хочет вести базу данных из всех отношений между жителями небольшого города. В числе требуемых таблиц, необходима та, в которой записываются друзья людей:

```
(defvar *friends* (make-hash-table))
```

Записи в этой хеш-таблице сами являются хеш-таблицами, в которых имена потенциальных друзей отображаются в t или nil:

```
(setf (gethash 'mary *friends*) (make-hash-table))
```

Чтобы записать John другом Mary, мы могли бы сказать:

```
(setf (gethash 'john (gethash 'mary *friends*)) t)
```

Город разделен между двумя фракциями. Как обычно делают фракции, каждая говорит "кто не с нами, тот против нас", поэтому все в городе вынуждены присоединиться к той или иной стороне. Таким образом, когда кто то переходит на другую сторону, все его друзья становяться врагами, а все его враги становяться друзьями.

Чтобы переключить(toggle), что x является другом y используя только встроенные операторы, мы можем сказать:

```
(setf (gethash x (gethash y *friends*))
           (not (gethash x (gethash y *friends*))))
```

это довольно сложное выражение, хотя гораздо проще, чем без использования setf. Если мы определим макрос доступа к базе данных следующим образом:

```
(defmacro friend-of (p q)
  `(gethash ,p (gethash ,q *friends*)))
```

тогда с этим макросом и toggle, мы были бы лучше подготовлены к изменениям в базе данных. Предыдущее обновление могло бы быть выражено просто как:

```
(toggle (friend-of x y))
```

Обобщенные переменные похожи на здоровую пищу, которая имеет приятный вкус. Они дают программам виртуозную модульность, и все же элегантно красивы. Если вы предоставляете доступ к вашим структурам данных с помощью макросов или обратимых функций, другие модули могут использовать setf для изменения ваших струкутр данных, без необходимости знать подробности их представления.

12.2 12-2 Проблема Многократнрых Вычислений

Предыдущий раздел предупреждал, что наше первоначальное определение toggle было неверным:

```
(defmacro toggle (obj)                                          ; wrong
  `(setf ,obj (not ,obj)))
```

Он подвержен проблеме, описанной в разделе 10-1, множественное вычисление. Проблема возникает когда его аргумент имеет побочные эффекты. Наример, если lst это список объектов мы пишем:

```
(toggle (nth (incf i) lst))
```

тогда мы ожидаем переключения(toggling) (i+1)-го элемента. Однако, с текущим определением toggle этот вызов расширяется в:

```
(setf (nth (incf i) lst)
          (not (nth (incf i) lst)))
```

Здесь i увеличивается дважды, и устанавливается (i+1)й элемент противоположностью от (i+2)го элемента. Так что в этом примере

```
> (let ((lst '(t nil t))
              (i -1))
        (toggle (nth (incf i) lst))
        lst)
(T NIL T)
```

вызов toggle, кажеться, не имеет никакого эффекта.

Не достаточно просто взять выражение, передаваемое в качестве аргумента toggle и вставить его как первый аргумент для setf. Мы можем заглянуть внутрь выражения и посмотреть, что оно делает: если оно содержит подчиненные формы, мы должны разбить их на части и вычислить их отдельно, в случае если они имеют побочные эффекты. В общем, это сложное дело.

Чтобы сделать это проще, Common Lisp предоставлялет макрос, который автоматически определяет ограниченный класс макросов для setf. Этот марос называется define-modify-macro и он принимает три аргумента: имя макроса, его дополнительные параметры(после обобщенной переменной),и имя функции[2] которая дает новое значение для обобщенной переменной.

Используя define-modify-macro, мы можем определить toggle следующим образом:

```
(define-modify-macro toggle () not)
```

Перефразируя, тут говориться "чтобы вычислить выражение формы (toggle place), найдите место, указанное в place, и если в нем храниться значение val, замените его на значение (not val)." Здесь новый макрос используется в том же самом примере:

```
> (let ((lst '(t nil t))
              (i -1))
        (toggle (nth (incf i) lst))
        lst)
(NIL NIL T)
```

Эта версия дает правильный результат, но его можно сделать более общим. Поскольку setf и setq могут принимать произвольное количество аргументов, то же можно делать и toggle. Мы можем добавить эту возможность, определив другой макрос поверх modify-macro, как на рисунке 12-1.

[2] Имя функции в общем смысле: или 1+ или (lambda (x) (+ x 1)).

```
(defmacro allf (val &rest args)
       (with-gensyms (gval)
        `(let ((,gval ,val))
           (setf ,@(mapcan #'(lambda (a) (list a gval))
                           args)))))

(defmacro nilf (&rest args) `(allf nil ,@args))

(defmacro tf (&rest args) `(allf t ,@args))

(defmacro toggle (&rest args)
       `(progn
         ,@(mapcar #'(lambda (a) `(toggle2 ,a))
                   args)))

(define-modify-macro toggle2 () not)
```

Рисунок 12-1: Макросы которые оперируют обобщенными переменными.

12.3 12-3 Новые Утилиты

В этом разделе приведены некоторые примеры новых утилит, которые работают с обобщенными переменными. Они должны быть макросами, чтобы передать свои аргументы без изменений в setf.

На рисуноке 12-1 показаны четыре новых макроса построенных на основе setf. Первый allf, предназначен для устновки нескольких обобщенных переменных в одно и тоже значение. На нем построены nilf и tf, которые устанавливают свои аргументы в nil и t, соответственно. Эти макросы просты, но они имеют значение.

Как и setq, setf может принимать несколько аргументов - перемежающиеся переменные и значения:

```
(setf x1y2)
```

Как и эти новые утилиты, но вы можете пропустить половину этих аргументов. Если вы хотите, инициализировать несколько перемеых в nil, вместо

```
(setf x nil y nil z nil)
```

вы можете просто сказать

```
(nilf x y z)
```

```
(define-modify-macro concf (obj) nconc)

(define-modify-macro conc1f (obj)
      (lambda (place obj)
         (nconc place (list obj))))

(define-modify-macro concnew (obj &rest args)
      (lambda (place obj &rest args)
         (unless (apply #'member obj place args)
           (nconc place (list obj)))))
```

Рисунок 12-2: Списковые операции с обобщенными перменными.

Последний макрос, toggle, описанный в предыдущем разделе: он похож на nilf, но придает каждому из своих аргументов противоположное логическое значение.

Эти четыре макроса иллюстрируют важный момент о операторах присваивания. Даже если мы намереваемся использовать оператор только для обычных переменных, стоит написать его расширяющимся в setf, а не в setq. Если первый аргумент является символом setf будет расширяться в setq в любом случае. Так мы сможем получить обобщение в setf без дополнительных затрат, и очень редко необходимо использвать setq в расширениях макроса.

Рисунок 12-2 содержит три макроса для деструктивного изменения концов списков. В разделе 3-1 упоминается, что не безопасно полагаться на побочные эффекты

```
(nconc x y)
```

и что вместо этого, нужно писать

```
(setq x (nconc x y))
```

Эта идиома воплощена в concf. Более специализированные conc1f и concnew похожи на push и pushnew для другого конца списка: conc1f добавляет один элемент в конец списка, и concnew делает тоже самое, но только если элемент уже не является членом списка.

В разделе 2-2 упоминается, что имя функции может быть лямбда-выражением, а также символом. Таким образом, можно передать целое лямбда-выражение в качестве третьего аргумента для define-modify-macro, как в определении conc1f. С помощью conc1 со страницы 45, этот макрос мог быть написан так:

```
(define-modify-macro conc1f (obj) conc1)
```

Макросы на Рисунке 12-2 должны использоваться с одной оговоркой. Если вы планируете построить список путем добавления элементов в конец, может быть предпочтительнее использовать push, и затем развернуть с помощью nreverse список. Дешевле сделать что то с началом списка, чем с его концом, потому что прежде чем чтото делать в конце списка, вначале вы должны до него добраться. Вероятно, для поощрения эффективного программирования в Common Lisp есть много операторов для первого и лишь немного для второго.

12.4 12-4 Более Сложные Утилиты

Не все макросы в setf могут быть определены с помощью define-modify-macro. Например предположим, что мы хотим определить макрос f для применения деструктивной

функции к обобщенной переменной. Встроенный макрос incf это сокращенная аббревиатура для setf для +. Вместо

```
(setf x (+ x y))
```

мы просто скажем

```
(incf x y)
```

Новый f должен быть обобщением этой идеи: в то время как incf расширяется в вызов + f рассширяется в вызов оператора указанного в качестве первого аргумента. Например, в определении scale-objs на странице 115, нам пришлось написать

```
(setf (obj-dx o) (* (obj-dx o) factor))
```

С f эта запись станет:

```
(_f * (obj-dx o) factor)
```

Неправильным способом написать f будет:

```
(defmacro _f (op place &rest args)                                    ; wrong
  `(setf ,place (,op ,place ,@args)))
```

К сожалению, мы не можем определить правильный f с помощьюw define-modify-macro, поскольку оператор, который будет применен к обобщенной переменной, задан в качестве аргумента.

Более сложные макросы, подобные этому, должны быть написаны в ручную. Чтобы такие макросы было легче писать, Common Lisp предоставляет функци get-setf-method, которая принимает обобщенную переменную и возвращает всю информацию, необходимую для получения или установки её значения. Мы увидим, как использовать эту информацию генерируя в ручную расширение для:

```
(incf (aref a (incf i)))
```

Когда мы вызываем get-setf-method для обобщенной переменной, мы получаем пять значений предназначенных для использования в качестве ингредиентов в расширении макроса:

```
> (get-setf-method '(aref a (incf i)))
(#:G4 #:G5)
(A (INCF I))
(#:G6)
(SYSTEM:SET-AREF #:G6 #:G4 #:G5)
(AREF #:G4 #:G5)
```

Первые два значения это списки временных переменных и значений, которые должны им присвоены. Итак мы можем начать расширение с:

```
(let* ((#:g4 a)
       (#:g5 (incf i)))
  ...)
```

Эти привязки должны быть созданы в let*, потому что в общем случае значения форм могут ссылаться на более ранние переменные. Третье[3] и пятое значения являются еще одними временными переменными и формой, которая будет возвращать

[3] Третье значение в настоящее время является списком из одного элемента. Оно возвращается в виде списка, чтобы предоставить (таким образом это еще далеко не всё) возможность для хранения нескольких значений в обобщенных переменных.

исходное значение обобщенной переменной. Поскольку мы хотим добавить 1 к этому значению, мы заключаем последнюю в вызов 1+:

```
(let* ((#:g4 a)
            (#:g5 (incf i))
            (#:g6 (1+ (aref #:g4 #:g5))))
   ...)
```

Наконец, четвертое значение, возвращаемое get-setf-method это присваивание, которое мы должны сделать в рамках новых привязок:

```
(let* ((#:g4 a)
            (#:g5 (incf i))
            (#:g6 (1+ (aref #:g4 #:g5))))
   (system:set-aref #:g6 #:g4 #:g5))
```

Чаще всего эта форма будет ссылаться на внутренние функции, которые не являются частью Common Lisp. Обычно setf прячет наличие этих функций, но они должны гдето существовать. Все они зависят от реализации, поэтому переносимый код должен использовать формы возвращаемые get-setf-method, а не ссылаться на такие функции, например как system:set-aref, на прямую.

Теперь для реализации _f мы пишем макрос, которые делает почти то же, что делали мы, когда расширяли incf в ручную. Разница лишь в том, что вместо оборачивания последней формы в let* в вызове 1+, мы оборачиваем ее выражение из аргументов передаваемых _f. Определение f показано на Риснуке 12-3.

```
(defmacro _f (op place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
       `(let* (,@(mapcar #'list vars forms)
               (,(car var) (,op ,access ,@args)))
          ,set)))

(defmacro pull (obj place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
       (let ((g (gensym)))
        `(let* ((,g ,obj)
                ,@(mapcar #'list vars forms)
                (,(car var) (delete ,g ,access ,@args)))
           ,set))))

(defmacro pull-if (test place &rest args)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
       (let ((g (gensym)))
        `(let* ((,g ,test)
                ,@(mapcar #'list vars forms)
                (,(car var) (delete-if ,g ,access ,@args)))
           ,set))))

(defmacro popn (n place)
  (multiple-value-bind (vars forms var set access)
                       (get-setf-method place)
       (with-gensyms (gn glst)
        `(let* ((,gn ,n)
                ,@(mapcar #'list vars forms)
                (,glst ,access)
                (,(car var) (nthcdr ,gn ,glst)))
           (prog1 (subseq ,glst 0 ,gn)
                  ,set)))))
```

Рисунок 12-3: Более сложный макрос setf.

Эта утилита довольно полезна. Теперь когда у нас она есть, мы легко можем, например, заменить любую именованную функцию с запомнинаием(memoized) (Разел 5-3) эквивалентом.[4] для запоминания(memoize) foo мы бы сказали:

```
(_f memoize (symbol-function 'foo))
```

Наличие _f также облегчает определение других макросов в setf. Нампример, теперь мы можем определить conc1f (Рисунок 12-2) как:

```
(defmacro conc1f (lst obj)
```

[4] Встроенные функции не долны запоминаться таким образом. Common Lisp запрещает переопределение встроенных функций.

```
   `(_f nconc ,lst (list ,obj)))
```

Рисунок 12-3 содержит другие полезные макросы для setf. Следующий, pull, предназначен в качестве дополнения к встроенному pushnew. Эта пара более различающая аргументы подобна паре push и pop; pushnew помещает новый элемент в список, если он не являетися уже членом списка, и pull разрушающе удалят выделенный элемент из списка. Параметр &rest в определении pull делает pull способным принимать все те же параметры ключевый слова как и в delete:

```
> (setq x '(1 2 (a b) 3))
(12(AB)3)
> (pull 2 x)
(1 (A B) 3)
> (pull '(a b) x :test #'equal)
(1 3)
>x
(1 3)
```

Вы могли бы думать об этом макросе как если был определен как:

```
(defmacro pull (obj seq &rest args)                                    ; wrong
   `(setf ,seq (delete ,obj ,seq ,@args)))
```

хотя, если это действительно было бы определено таким образом, оно было бы связано с проблемами как по порядку, так и по количеству вычилений. Мы могли бы определить версию pull как простой modify-macro:

```
(define-modify-macro pull (obj &rest args)
   (lambda (seq obj &rest args)
        (apply #'delete obj seq args)))
```

но так как modify-macros должен принимать обобщенную переменную, в качестве первого аргумента, мы должны были бы передать первые два аргумента в обратном порядке, что будет менее интуитивно понятно.

Более общий pull-if принимает начальный аргумент функции, и расширяется в delete-if вместо delete:

```
> (let ((lst '(1 23456)))
        (pull-if #'oddp lst)
        lst)
(246)
```

Эти два макроса иллюстрируют еще один общий момент. Если основная функция принимает необязательные аргументы, то и макрос должен основываться на них. Оба, и pull и pull-if передают необзательные аргументы своим функциям удаления.

Последний макрос на рисунке 12-3, popn, является обобщением pop. Вместо простого выталкивания по одному элементу из списка, он выталкивает и возвращает подпоследовательность произвольной длины:

```
> (setq x '(a b c d e f))
(ABCDEF)
> (popn 3 x)
(ABC)
>x
```

```
(DEF)
```

Рисунок 12-4 содержит макрос, который сортирует свои аргументы. Если x и y являются переменными и мы хотим убедиться, что x не является меньшим из двух значений, мы можем написать:

```
(if (> y x) (rotatef x y))
```

Но если мы хотим сделать это для трех и более переменных, требуемый код растет очень быстро. Вместо, того, чтобы писать его в ручную, мы можем сделать так, чтобы sortf писал его за нас. Этот макрос принимает оператор сравнения пюст любое количество обобщенных переменных, и меняет их значения до тех пор, пока они не будут в порядке определяемом оператором. В простейшем случае аргументы могут быть обычными переменными:

```
> (setq x1y2z3)
3> (sortf > x y z)
3> (list x y z)
(321)
```

```
(defmacro sortf (op &rest places)
      (let* ((meths (mapcar #'(lambda (p)
                                (multiple-value-list
                                  (get-setf-method p)))
                            places))
             (temps (apply #'append (mapcar #'third meths))))
        `(let* ,(mapcar #'list
                        (mapcan #'(lambda (m)
                                    (append (first m)
                                            (third m)))
                                meths)
                        (mapcan #'(lambda (m)
                                    (append (second m)
                                            (list (fifth m))))
                                meths))
           ,@(mapcon #'(lambda (rest)
                         (mapcar
                           #'(lambda (arg)
                               `(unless (,op ,(car rest) ,arg)
                                  (rotatef ,(car rest) ,arg)))
                           (cdr rest)))
                     temps)
           ,@(mapcar #'fourth meths))))
```

Рисунок 12-4: Макрос который сортирует свои аргументы.

В общем, это могут быть любые обратимые выражения. Предположим, что cake является обратимой функцией и которая возвращает чей либо кусочек пирога, a bigger функция сравнения определенная на кусочках пирога. Если мы хотим применить

правило, что торт moe не меньше, чем торт larry, который не меньше чем торт curly, мы запишем:

```
(sortf bigger (cake 'moe) (cake 'larry) (cake 'curly))
```

Определение sortf в общих чертах аналогично определению _f. Оно начинается с let* в котором связываются временные переменные возвращаемые get-setf-method и связваются начальные значения обобщенных переменных. Ядром sortf является центральное выражение mapcon, которое генерирует код сортирующий эти временные переменные. Код, сгенерированный этой частью макроса, растет экспотенциально вместе с числом аргументов. После сортировки, обобщенные переменные переприсваиваются с помощью

```
(sortf > x (aref ar (incf i)) (car lst))
```

расширяется (в одной из возможных реализаций) в:

```
(let* ((#:g1 x)
           (#:g4 ar)
           (#:g3 (incf i))
           (#:g2 (aref #:g4 #:g3))
           (#:g6 lst)
           (#:g5 (car #:g6)))
   (unless (> #:g1 #:g2)
       (rotatef #:g1 #:g2))
   (unless (> #:g1 #:g5)
       (rotatef #:g1 #:g5))
   (unless (> #:g2 #:g5)
       (rotatef #:g2 #:g5))
   (setq x #:g1)
   (system:set-aref #:g2 #:g4 #:g3)
   (system:set-car #:g6 #:g5))
```

Рисунок 12-5: Расширение вызова sortf.

форм, возвращаемых get-setf-method. Используемый алгоритм представляет собой сортировку пузыкрьком O(n2), но этот макрос не предназначен для вызова с большим количеством аргументов.

На Рисунке 12-5 показано расширение вызова sortf. В начальноml let*, аргументы и их подформы тщательно вычисляются в порядке слева на право. Затем появляются три выражения, которые сравнивают, и возможно заменяют значения временных переменных: первый сравнивается со вторым, затем первый с третьим, затем второй с третьим. Наконец, обобщенные переменные переприсваиваются в порядке с лева на право. Хотя проблема возникает редко, аргументы макроса должны присваиваться слева на право, а также вычисляться в том же порядке.

Операторы типа _f и sortf имеют определенное сходство с функциями, которые принимают функциональные аргументы. Следует понимать, что они имеют воплне очевидное различие. Функция подобная find-if получает функцию и вызывает её; а макрос подобный _f получает имя, и делает его началом(car) выражения. Оба и _f и

sortf могут быть написаны , чтобы принимать функциональные аргументы. Например, _f может быть написан:

```
(defmacro _f (op place &rest args)
     (let ((g (gensym)))
        (multiple-value-bind (vars forms var set access)
                                    (get-setf-method place)
        `(let* ((,g ,op)
                  ,@(mapcar #'list vars forms)
                  (,(car var) (funcall ,g ,access ,@args)))
             ,set))))
```

и вызываться (f #'+ x 1). Но оригинальная версия _f может сделать, что угодно, и так как она имеет дело с именами, она также может принять имя макроса или специальной формы. Например, кроме +, вы можете вызвать nif (стр. 150):

```
> (let ((x 2))
        (_f nif x 'p 'z 'n)
        x)
P
```

12.5 12-5 Определение Инверсий

Раздел 12-1 объяснил, что любой вызов макроса, который расширяется в обратимую ссылку, сам по себе, будет обратимым. Вам не нужно определять операторы как макросы, только для того чтобы сделать их обратимыми. Используя defsetf вы можете указать Lisp, как инвертировать любую функцию или вызов макроса.

Этот макрос можно использовать двумя способами. В простейшем случае его аргументы это два символа:

```
(defsetf symbol-value set)
```

В более сложной форме, вызов defsetf подобен вызову defmacro, с дополнительным параметром для обновления значения формы. Например, определим возможную инверсию формы для car:

```
(defsetf car (lst) (new-car)
     `(progn (rplaca ,lst ,new-car)
               ,new-car))
```

Между defmacro и defsetf есть одно очень важное отличие: последнее автоматически создает идентификаторы gensyms для своих аргументов. С определением данным выше,

```
(setf (car x) y) расшириться до:
(let* ((#:g2 x)
               (#:g1 y))
     (progn (rplaca #:g2 #:g1)
               #:g1))
```

```
(defvar *cache* (make-hash-table))

(defun retrieve (key)
   (multiple-value-bind (x y) (gethash key *cache*)
       (if y(values x y)
           (cdr (assoc key *world*)))))

(defsetf retrieve (key) (val)
   `(setf (gethash ,key *cache*) ,val))
```

Рисунок 12-6: Асиментричная инверсия.

Таким образом, мы можем писать расширители defsetf не беспокоясь о захвате переменных, или числе и порядке вычислений.

В CLTL2 Common Lisp можно задавать инверсии setf непосредственно используя defun, поэтому предыдущий пример также может быть записан, как:

```
(defun (setf car) (new-car lst)
  (rplaca lst new-car)
  new-car)
```

Обновленное значение должно быть первым параметром в такой функции. Она также, по соглашению, должна возвращать это значение как значение функции.

До сих пор примеры показывали, что обобщенные переменные предположительно ссылаются на место в структуре данных. Злодей уносит свою заложницу в темницу, а спасающий герой поднимает ее обратно; они оба следуют одному и тому же пути, но в разных направлениях. Не удивительно, если у людей есть представление, что setf должен работать таким образом, поскольку все предопределенные инверсии, кажеться, имеют эту форму; действительно, место - это условное название параметра, который должен быть перевернут(инвертирован).

В принципе, setf является более общим: форма доступа и её инверсия не должны работать на той же структуре данных. Предположим, что в каком то, приложении мы хотим кешировать обновления базы данных. Это может быть необходимо, например, если бы реальные обновления на лету делать не эффективно, или если все обновления должны быть сверены для согласованности перед их совершением(принятием).

Предположим, что *world* является фактической базой данных. Для простоты, мы сделаем её ассоциативным списком, элементы которого имеют форму (key . val). Рисунок 12-6 показывает функцию поиска названую retrieve. Если *world* это

```
((a . 2) (b . 16) (c . 50) (d . 20) (f . 12))
```

тогда

```
> (retrieve 'c)
50
```

В отличии от вызова car, вызов retrieve не ссылается к конкретному месту структуры данных. Возвращаемое значение может придти от одного из двух мест. И инверсия retrieve, также определенная на рисунке 12-6, относиться только к одному из них:

```
> (setf (retrieve 'n) 77)
77
> (retrieve 'n)
```

```
77
T
```

Этот поиск возвращает вторым значением t, указывающим, что ответ был найден в кеше.

Как и сами макросы, обобщенные переменные являются абстракцией замечательной мощности. Здесь, вероятно, еще многое предстоит открыть. Конечно, индивидуальные пользователи, вероятно, обнаружат способы, которыми использование обобщенных переменных может привести к более элегантным и более мощным программам. Но также возможно использовать инверсии setf новыми способами, или открыть другие классы, аналогичных полезных преобразований(трансформаций).

13 13 Вычисления во время компиляции

В предыдущих главах описано несколько типов операторов, которые должны быть реализованы макросами. Они описывают класс проблем, которые должны решаться функциями, но где макросы более эффективны. Раздел 8-2 перечисляет плюсы и минусы использования макросов в данной ситуации. Среди плюсов был пункт "вычисление во время компиляции." Определив оператор как макрос, вы можете иногда заставить выполнить часть своей работы, во время его расширения. Эта глава рассматривает макросы, которые используют эту возможность.

13.1 13-1 Новые Утилиты

В разделе 8-2 упоминается возможность при использовании макросов перенести вычисления на время компиляции программы. Там, в качестве примера, у нас был макрос avg, который возвращал среднее значение своих аргументов:

```
> (avg pi 4 5)
4.047...
```

На рисунке 13-1 показано определение avg, сначала как функции и затем как макроса. Когда он определяется как макрос, вызов length может быть сделан во время компиляции. В версии макроса, мы также избегаем затрат на разбор параметров &rest во время выполнения. Во многих реализациях, avg будет скорее всего написан как макрос.

Тип экономии, который достигается за счет знания количества аргументов во время расширения макроса может быть объединен, с типом, который мы получаем из in (стр. 152), где удалось избежать даже вычисления некоторых аргументов. Рисунок 13-2 содержит две версии most-of, который возвращает истину если большинство его аргументов истина:

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

Рисунок 13-1: Перемещение вычислений при поиске среднего значения.

```
(defun most-of (&rest args)
  (let ((all 0)
        (hits 0))
    (dolist (a args)
      (incf all)
      (if a (incf hits)))
    (> hits (/ all 2))))

(defmacro most-of (&rest args)
  (let ((need (floor (/ (length args) 2)))
        (hits (gensym)))
    `(let ((,hits 0))
       (or ,@(mapcar #'(lambda (a)
                         `(and ,a (> (incf ,hits) ,need)))
                     args)))))
```

Рисунок 13-2: Перемещение и избегание вычислений.

```
> (most-of t t t nil)
T
```

Весрсия макроса расширяется в код, подобный in, вычисляет только столько аргументов, сколько надо. Например, (most-of (a) (b) (c)) расширяется в эквивалент:

```
(let ((count 0))
     (or (and (a) (> (incf count) 1))
         (and (b) (> (incf count) 1))
         (and (c) (> (incf count) 1))))
```

В лучшем случае, вычисляется чуть больше половины аргументов.

```
(defun nthmost (n lst)
  (nth n (sort (copy-list lst) #'>)))

(defmacro nthmost (n lst)
  (if (and (integerp n) (< n 20))
        (with-gensyms (glst gi)
           (let ((syms (map0-n #'(lambda (x) (gensym)) n)))
               `(let ((,glst ,lst))
                  (unless (< (length ,glst) ,(1+ n))
                    ,@(gen-start glst syms)
                    (dolist (,gi ,glst)
                      ,(nthmost-gen gi syms t))
                    ,(car (last syms))))))
        `(nth ,n (sort (copy-list ,lst) #'>))))

(defun gen-start (glst syms)
  (reverse
       (maplist #'(lambda (syms)
                    (let ((var (gensym)))
                      `(let ((,var (pop ,glst)))
                          ,(nthmost-gen var (reverse syms)))))
                  (reverse syms))))

(defun nthmost-gen (var vars &optional long?)
  (if (null vars)
       nil
       (let ((else (nthmost-gen var (cdr vars) long?)))
          (if (and (not long?) (null else))
               `(setq ,(car vars) ,var)
               `(if (> ,var ,(car vars))
                    (setq ,@(mapcan #'list
                                    (reverse vars)
                                    (cdr (reverse vars)))
                          ,(car vars) ,var)
                    ,else)))))
```

Рисунок 13-3: Использование аргументов, известных во время компиляциии.

Макрос также может быть в состоянии перенести вычисления на время компиляции, есил значения конкретных аргументов известны. На рисунке 13-3 приведен пример такого макроса. Функциян nthmost принимает чисмло n и список чисел и возвращает n-й по величине из них; как и другие функции работающие с последовательноостями индекс начинается с нуля:

```
> (nthmost 2 '(2 6 1 5 3 4))
4
```

Функциональная версия написана очень просто. Она сортирует список и вызывает nth для результата. Поскольку сортировка деструктивна, nthmost копирует список перед его сортировкой. Такое написание nthmost не эффективно в двум причинам:

оно выполняет создание списка и оно сортирует весь список аргументов, хотя все, что нас волнует это старшие n элементов.

Если мы знаем n во время компиляции, мы можем по другому подойти к проблеме. Оставшаяся часть кода на Рисунке 13-3 определяет версию nthmost в виде макроса. Первая вещь, которую делает этот макрос, это посмотреть на свой первый аргумент. Если первый аргумент не является буквальным числом(т.е записанным не переменной, а цифрами), макрос расширяется в тот же код, что приведен выше. Если первый аргумент это число, мы можем пойти по другому пути. Скажем, если вы хотели найти третье по величине самое большое печенье на тарелке, вы могли бы это сделать посмотрев на каждое печенье по очереди, всегда держа в своих руках три самых больших печенины найденных до сих пор. Когда бы вы осмотрели все печенье, самое маленькое печенье в ваших руках - это то, что вы ищите. Если n это небольшая константа, не пропорциональная количеству печенин, то этот метод даст вам необходимую печенину с меньшими усилиями, чем та которая требует предварительной сортировки всех печенин.

Это стратегия, которой можно придерживатсья когда n известно во время расширения макроса. После своего расширения, макрос создает n переменных, а затем вызовав nthmost-gen генерирует код который должен вычисляться при просмотре каждой печенины. Рисунок 13-4 показывает пример расширения макроса. Макрос nthmost ведет себя так же, как оригинальная функция, за исключением того, что он не может быть передан в качестве аргумента для применения(apply). Обоснование для применения макроса это обычная эффективность: версия макроса не выполняет конструирования списка, и если n небольшая константа, выполняет меньше сравнений.

Чтобы иметь эффективные программы, нужно потрудиться чтобы написать такоие огромные макросы? В этом случае, вероятно нет. Две версии nthmost приведены как пример общего принципа: когда некоторые аргументы известны во время компиляции, вы можете использовать марос для генерации более эффективного кода. Будете вы пользоваться этой возможностью или нет, будет зависеть от того, сколько вы выиграете, и сколько еще усилий потербуется, чтобы написать эффективную версию макроса. Версия nthmost реализованная в виде макроса, длинная и сложная, ее стоит писать только в крайних случаях. Однако информация, известная во время компиляции, всегда является фактором, который стоит учитывать, даже если вы решите не использовать её в своих интересах.

```
      (nthmost 2 nums)
```
expands into:
```
      (let ((#:g7 nums))
         (unless (< (length #:g7) 3)
             (let ((#:g6 (pop #:g7)))
              (setq #:g1 #:g6))
             (let ((#:g5 (pop #:g7)))
              (if (> #:g5 #:g1)
                   (setq #:g2 #:g1 #:g1 #:g5)
                   (setq #:g2 #:g5)))
             (let ((#:g4 (pop #:g7)))
              (if (> #:g4 #:g1)
                   (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g4)
                   (if (> #:g4 #:g2)
                        (setq #:g3 #:g2 #:g2 #:g4)
                        (setq #:g3 #:g4))))
             (dolist (#:g8 #:g7)
              (if (> #:g8 #:g1)
                   (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g8)
                   (if (> #:g8 #:g2)
                        (setq #:g3 #:g2 #:g2 #:g8)
                        (if (> #:g8 #:g3)
                               (setq #:g3 #:g8)
                               nil))))
             #:g3))
```
Рисунок 13-4: Расширение nthmost.

13.2 13-2 Example Кривые Безье(Bezier)

Как и макрос with- (Раздел 11-2), макрос для вычислений во время компиляции вероятнее всего, будет написан для конкретного приложения, чем как утилита общего назначения. Что может знать утилита общего назначения во время компиляции? Количество переданных ей аргументов , и возможно некоторые из их значений. Если мы хотим использовать другие ограничения, они, вероятно, должны быть индивидуальными программами.

В качестве примера в этом разделе показано, как макросы могут ускорить генерацию кривых Безье. Кривые должны генерироваться быстро, если ими управляют в интерактивном режиме. Оказывается, что если количество сегментов в кривой известно заранее, большая часть вычислений может быть выполнена во время компиляции. Написав наш генератор кривых curve-generator в виде макроса, мы можем вшивать предварительно вычисленные значения прямо в код. Это должно быть даже более быстрым, чем обычная оптимизация с их хранением в массиве.

Кривая Безье определяется в терминах четырех точек - двух конечных точек и двух котнрольных(управляющих) точек. Когда мы работаем в двух измерениях, эти точки определяют параметрическоие уравнения для координат x и y точек на кривой.

Если две конечные точки это (x0, y0) и (x3, y3) и две управляющие точки это (x1, y1) и (x2, y2), то уравнения определяющие точки на кривой будут:

```
                                  3                              3
x =(x3 - 3*x2 + 3*x1 - x0)*u  + (3*x2 - 6*x1 + 3*x0)*u + (3*x1 - 3*x0)*u + x

                                  3                          2
y =(y3 - 3*y2 + 3*y1 - y0)*u + (3*y2 - 6*y1 + 3*y0)*u  + (3*y1 - 3*y0)*u + y
```

Если мы вычислим эти уравнения для n значений u между 0 и 1, мы получим n точек на кривой. Например, если мы хотим нарисовать кривую как 20 сегментов, то мы бы вычислили уравнения для u = .05, .1,..., .95. Нет небходимости вычислять их для u равных 0 или 1, поскольку если u = 0 они дадут просто первую конечную точку (x 0, y0), и если u = 1 они дадут вторую конечную точку (x3, y3).

Очевидная оптимизация состоит в том, чтобы сделать n фиксированным, предварительно расчитать степень u и сохранить их в массиве размером (n-1) × 3. Определив генератор кривой(curve-generator) как макрос, мы можем сделать еще лучше! Если n изветсно во время расширения, программа может просто расшириться до n команд рисующих линии. Предварительно вычисленные степени u, вместо того чтобы храниться в массиве, могут быть вставлены в виде буквальных значений прямо в код расширения макроса.

На рисунке 13-5 приведен макрос curve-generating, который реализует эту стратегию. Вместо того, чтобы рисовать линии сразу, он создает массив сгенерированных точек. Когда кривая перемещается в интерактивном режиме, каждый экземпляр должен быть нарисован дважды: один раз чтобы показать её, и снова стереть её прежде чем нарисавать ее ещё. Тем временем точки должны где то храниться.

С n = 20, genbez расширяется до 21 setf. Так как степени u вшиваются непосредственно в код, мы экономим на затратах по их поиску во время выполнения, и на стоимости вычисления их при запуске. Подобно степеням u, индексы массива представляются в виде констант в расширении, поэтому проверка границ для (setf aref) может быть выполнена так же во время компиляции.

13.3 13-3 Применения

Последующие главы содержат несколько других макросов, которые используют информацию, доступную во время компиляции. Хорошим примером является if-match (стр. 242). Сопоставитель-шаблонов сравнивающий две последовательности, возможно содержащие переменные, чтобы увидеть, есть ли какой-то способ присвоения значениий переменным, который сделает эти две последовательности равными.

```
(defconstant *segs* 20)
(defconstant *du*       (/ 1-0 *segs*))
(defconstant *pts* (make-array (list (1+ *segs*) 2)))

(defmacro genbez (x0 y0 x1 y1 x2 y2 x3 y3)
  (with-gensyms (gx0 gx1 gy0 gy1 gx3 gy3)
       `(let ((,gx0 ,x0) (,gy0 ,y0)
              (,gx1 ,x1) (,gy1 ,y1)
              (,gx3 ,x3) (,gy3 ,y3))
          (let ((cx (* (- ,gx1 ,gx0) 3))
                (cy (* (- ,gy1 ,gy0) 3))
                (px (* (- ,x2 ,gx1) 3))
                (py (* (- ,y2 ,gy1) 3)))
           (let ((bx (- px cx))
                 (by (- py cy))
                 (ax (- ,gx3 px ,gx0))
                 (ay (- ,gy3 py ,gy0)))
             (setf (aref *pts* 0 0) ,gx0
                   (aref *pts* 0 1) ,gy0)
             ,@(map1-n #'(lambda (n)
                           (let* ((u (* n *du*))
                                  (u^2 (* u u))
                                  (u^3 (expt u 3)))
                             `(setf (aref *pts* ,n 0)
                                    (+ (* ax ,u^3)
                                       (* bx ,u^2)
                                       (* cx ,u)
                                       ,gx0)
                                    (aref *pts* ,n 1)
                                    (+ (* ay ,u^3)
                                       (* by ,u^2)
                                       (* cy ,u)
                                       ,gy0))))
                       (1- *segs*))
             (setf (aref *pts* *segs* 0) ,gx3
                   (aref *pts* *segs* 1) ,gy3))))))
```

Рисунок 13-5: Макрос для генерации кривых Безье.

Конструкция if-match показывает, что если одна из последовательностей известна во время компиляции и только она содержит переменные, тогда сопоставление может быть сделано более эффективно. Весто сравнения двух последовательностей во время выполнения и создания списков для хранения привязок переменных, установленных в процессе сравнения, мы можем макросом сгенерировать код для выполнения точных сравнений, продиктованных известной последовательностью и можем хранить все привязки в реальных переменных Lisp-а.

Встраиваемые языки описываемые в Главах 19-24 также, по большей части, используют информацию, доступную во время компиляциии. Так как встроенный язык

является своего рода компилятором, это вполне естественно, что он должен использовать такую информацию. Как правило, чем сложнее макрос, тем больше ограничений он накладывает на аргументы, и тем выше ваши шансы использовать эти ограничения для генерации эффективного кода.

14 14 Анафорические(ссылающиеся или отсылающие на/к ранее сказанное/му) Макросы

Глава 9 рассматривала захват переменных исключительно как проблему, как нечто, что происходит не преднамеренно, и что может повлиять на программы в худшую сторону. Эта глава покажет, что захват переменных также можно использовать конструктивно. Есть некоторые полезные макросы, которые нельзя было бы написать без этого.

В программах на Lisp нередко хочеться проверить, является ли возвращаемое выражением значение не nil, и если да, что-то сделать с этим значением. Если выражение дорого вычислять, тогда обычно нужно сделать, что то вроед этого:

```
(let ((result (big-long-calculation)))
  (if result
        (foo result)))
```

Но не было бы проще, если бы мы могли просто сказать, как мы это сделали бы на английском:

```
(if (big-long-calculation)
     (foo it))
```

Используя преимущества захвата переменных, мы можем написать версию if, которая работае подобным способом.

14.1 14-1 Варианты Анафоризмов

В естественном языке, анафор - это выражение, которое ссылается к уже сказанному. Наиболее распространённым анафором в Английском языке вероятно является "it,"(это, оно, он, оно) как в "Возьми гаеечный ключ, и положи it(его) на стол." Анафоры очень удобны в повседневном языке - представьте, как вы бы обходидлись без них, но они не очень много используются в языках программирования. По большей части, это хорошо. Анафорные выражения часто являются по настоящему двусмысленными, а современные языки программирования не предназначены для устранения двусмысленностей.

Тем не менее, можно ввести ограниченную форму анофоры в программы на Lisp не вызывая двусмысленности. Оказывается, анафор очень похож на захват символа. Мы можем использовать анафору в программах, назначая определенные символы, используемые в качестве местоимений, а затем намеренно писать макросы захватывающие эти символы.

В новой версии if, символ it является тем, который мы хотим захватить. Анафорный if, коротко называемый aif, определяется следующим образом:

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
        (if it ,then-form ,else-form)))
```

и используется как в предыдущем примере:

```
(aif (big-long-calculation)
        (foo it))
```

Когда вы используете aif, символl it является связываемым слева с результатом возвращаемым тестовым предложением. В вызове макроса, it кажется свободным, но на самом деле выражение (foo it) будет вставлено при расширении aif в контекст, в котором символ it является связанным:

```
(let ((it (big-long-calculation)))
  (if it (foo it) nil))
```

Таким образом, символ, который выглядит свободным в исходном коде, является связанным в расширении макроса. Все анафорические макросы в этой главе используют вариации одной и той же техники.

Рисунок 14-1 содержит анафорические варианты нескольких операторов Common Lisp. После aif идет awhen, очевидный анафорический вариант when(когда):

```
(awhen (big-long-calculation)
  (foo it)
  (bar it))
```

И aif и awhen часто полезны, но awhile вероятно уникальный среди анафорических макросов в том, что он является более полезным чем его кузен while (определенный на странице 91). Макросы типа while и awhile обычно используются в ситуациях где программе необходимо опрашивать какой то внешний источник. И когда вы опрашиваете источник, если вы не ждете, что он просто изменит свое состояние, вам обычно хочется что-то сделать с возвращенным из него объектом:

```
(defmacro aif (test-form then-form &optional else-form)
       `(let ((it ,test-form))
            (if it ,then-form ,else-form)))

(defmacro awhen (test-form &body body)
       `(aif ,test-form
                (progn ,@body)))

(defmacro awhile (expr &body body)
       `(do ((it ,expr ,expr))
              ((not it))
            ,@body))

(defmacro aand (&rest args)
       (cond ((null args) t)
                ((null (cdr args)) (car args))
                (t `(aif ,(car args) (aand ,@(cdr args))))))

(defmacro acond (&rest clauses)
       (if (null clauses)
             nil
             (let ((cl1 (car clauses))
                      (sym (gensym)))
                `(let ((,sym ,(car cl1)))
                     (if ,sym
                           (let ((it ,sym)) ,@(cdr cl1))
                           (acond ,@(cdr clauses)))))))
```

Рисунок 14-1: Анафорные варианты операторов Common Lisp.

```
(awhile (poll *fridge*)
   (eat it))
```

Определение aand немного сложнее чем предыдущие. Оно представляет анафорическую верисию and; во время вычисления каждого из его аргументов, it будет привязан к значению возвращенному предыдущим аргументом.[1] На практике, aand, как правило используется в программах, которые делают условные запросы, как в:

```
(aand (owner x) (address it) (town it))
```

который возвращает город(town) (если он есть), затем адресс(address) (если он есть) владельца (если он есть) x. Без aand, это выражение должно было бы написано как:

```
(let ((own (owner x)))
  (if own
          (let ((adr (address own)))
             (if adr (town adr)))))
```

[1] Хотя кто-то склонен думать одновременно об and и or, было бы бессмысленно писать анафорническую версию or. Аргумент в выражении or вычисляется только, если предыдущий аргумент вычисляется в nil, так что анафору будет не на что ссылаться в aor.

Определение aand показыает, что расширение будет варьироваться в зависимости от количества аргументов в вызове макроса. Если аргументов нет, то aand как и обычный and, должен просто вернуть t. В противном случае расширение генерируется рекурсивно, каждый шаг дает один слой в цепочке вложенных aif:

```
(aif  first argument
          expansion for rest of arguments )
```

Расширение aand должно прекратиться, когда останется один аргумент, вместо работы до nil подобно большинству рекурсивных функций. Если бы рекурсия продолжалась до тетх пор, пока не осталось бы никаких аргументов, расширение всегда будет иметь форму:

```
(aif  c1
         ...(aif  cn t)...)
```

Такое выражение всегда будет возвращать t или nil, а приведенный выше пример не будет работать как задумано.

Раздел 10-4 предупредил, что если макрос всегда дает расширение, содержащее вызов самого себя, расширение никогда не закончиться. Хотя рекурсивный aand является безопасным, потому что в базовом случаее его расширение не обращается к aand.

Последний пример, acond, предназначен для тех случаев, когда в остатке предложения cond хочется использовать значение возвращаемое тестомым выражением. (Эта ситуация возникает так часто, что некоторые реализации Scheme предоставляют способ использовать значение возвращаемое тестовым выражением в предложении cond.)

В расширениии предложения acond, результат тестового выражения будет вначале храниться в переменной gensym, чтобы символ it мог быть связан только в оставшейся части предложения. Когда макросы создают привязки, они должны всегда делать это в самой узкой области видимости. Вот, если бы мы обошлись без

```
(defmacro alambda (parms &body body)
      `(labels ((self ,parms ,@body))
        #'self))

(defmacro ablock (tag &rest args)
      `(block ,tag
        ,(funcall (alambda (args)
                           (case (length args)
                             (0 nil)
                             (1 (car args))
                             (t `(let ((it ,(car args)))
                                   ,(self (cdr args))))))
                  args)))
```

Рисунок 14-2: Еще варианты анафорных макросов.

gensym и вместо этого сразу бы свзяали it с результатом тестового выражения, как в:

```
(defmacro acond (&rest clauses)                                    ; wrong
```

```
(if (null clauses)
       nil
       (let ((cl1 (car clauses)))
         `(let ((it ,(car cl1)))
              (if it
                   (progn ,@(cdr cl1))
                   (acond ,@(cdr clauses)))))))
```

тогда привязку it также будет иметь в своем окружении последующее тестовое выражение. Рисунок 14-2 содержит несколько более сложных анафорических вариантов. Макрос alambda используется для создания само рекурсивной функции. Когда кто то хочет буквально ссылаться на рекурсивную функцию? Мы можем сослаться на неименованную функцию используя решетку с квотированием (#') для лямбда-выражения:

```
#'(lambda (x) (* x 2))
```

Но как объяснялось во второй главе, вы не можете просто выразить рекурсивную функцию с помощью простого лямбда выражения. В место этого вы должны определить локальное определение именованной функции используя labels. Следующая функция (воспроизводим со страницы 22)

```
(defun count-instances (obj lists)
  (labels ((instances-in (list)
                  (if list
                       (+ (if (eq (car list) obj) 1 0)
                             (instances-in (cdr list)))
                       0)))
        (mapcar #'instances-in lists)))
```

получает объект и список, и возвращает список числа вхождений объекта в каждый элемент списка:

```
> (count-instances 'a '((a b c) (darpa)(dar)(aa)))
(1212)
```

С анафорой мы можем сделать то, что представляет собой буквальную рекурсивную функцию. Макрос alambda использует labels для создания функции, и может использоваться для выражения, например, рекурсивной функции факториала:

```
(alambda (x) (if (= x 0) 1 (* x (self (1- x)))))
```

Используя alambda мы можем определить эквивалентную версию count-instances следующим образом:

```
(defun count-instances (obj lists)
  (mapcar (alambda (list)
                  (if list
                       (+ (if (eq (car list) obj) 1 0)
                            (self (cdr list)))
                       0))
                lists))
```

В отличии от других макросов на Рисунке 14-1 и 14-2, которые все захватывают символ it, alambda захватывает символ self. Экземпляр alambda расширяется до выражения labels в котором self связывается с определяемой функцией. Помимо того

что оно меньше, выражение alambda выглядит как привычные лямбда выражения, создавая код, который легче читать, при их использовании.

Новый макрос используется в определении ablock, анафорической версии встроенной специальной формы block. В block, аргументы вычисляются с лева на право. Тоже самое происходит и в ablock, но внутри каждого выражения переменная it будет связана со значением предыдущего выражения.

Этот макрос следует использовать по своему усмотрению. Хотя иногда, ablock будет превращаеть, то что могло быть хорошей функциональной программой, в императивнрую форму. Следущий, к сожалениею, характерно некрасивый пример:

```
> (ablock north-pole
        (princ "ho ")
        (princ it)
        (princ it)
        (return-from north-pole))
ho ho ho
NIL
```

Всякий раз, когда макрос, который делает преднамеренный захват переменной, экспортируется в другой пакет, необходимо также экспортировать захватываемый символ. Например, куда бы не экспортировался aif, it также должен экспортироваться. В противном случае it, которое появлялется в определении макроса, будет отличаться от символа it используемого при вызове макроса.

14.2 14-2 Неудачи

В Common Lisp символ nil выполняет три разных задачи. Он в первую очередь является пустым списком, так что

```
> (cdr '(a))
NIL
```

Как и пустой список, nil используется для представления ложности, как в

```
>(=10)
NIL
```

И наконец, функции возвращающие nil указывают на неудачу(сбой). Например, работа встроенного find-if должна возвращать первый элемент списка, который удовлетворяет некоторому критерию(тесту). Если такой элемент не найден, find-if вернет nil:

```
> (find-if #'oddp '(2 4 6))
NIL
```

К сожалению, мы не можем отличить этот случай от случая, когда find-if завершается успехом, но успех состоит в нахождении nil:

```
> (find-if #'null '(2 nil 6))
NIL
```

На практике, это не вызывает особых проблем, если использовать nil для представления ложности и пустого списка. На самом деле это может быть довольно удобно. Тем не менее, огорчает, иметь nil для представления неудачи, потому что это означает, что результат возвращаемый функциями типа find-if может быть не однозначный.

Возникает проблема различения неудачи и найденного значения nil возвращаемого любой функцией которая осуществляет поиск. Common Lisp предлагает не менее трех решений этой проблемы. Наиболее распространенный подход, перед возвратом множественных значений, возвращать списковую структуру. Нет проблем в различении неудачи с использованием assoc, например; в случае успеха должна возвращаться пара в качестве результата:

```
> (setq synonyms '((yes . t) (no . nil)))
((YES . T) (NO))
> (assoc 'no synonyms)
(NO)
```

Следуя этому подходу, если мы беспокоимся о неоднозначности при использовании find-if, мы будем использовать member-if, который вместо простого возврата элемента, удовлетворяющего выражению test, возвращает весь (остаток списка) cdr, который начинается с него:

```
> (member-if #'null '(2 nil 6))
(NIL 6)
```

С момента появления возвратат множественных значений, было принято другое решение этой проблемы: использовать одно значени для данных и второе для указания на успех или неудачу поиска. Таким образом работает встроенный gethash. Он всегда возвращает два значения, второе указывает, было ли что найдено, или нет:

```
> (setf edible                                    (make-hash-table)
              (gethash 'olive-oil edible) t
              (gethash 'motor-oil edible) nil)
NIL
> (gethash 'motor-oil edible)
NIL
T
```

Так что, если вам необходимо различать все три возможных случая, вы можете использовать идиому, такую как следующее выражение:

```
(defun edible? (x)
     (multiple-value-bind (val found?) (gethash x edible)
        (if found?
              (if val 'yes 'no)
              'maybe)))
```

тем самым отличая ложь от неудачи:

```
> (mapcar #'edible? '(motor-oil olive-oil iguana))
(NO YES MAYBE)
```

Common Lisp поддерживает еще один способ обозначить неудачу: Функция доступа принимает как аргумент специальный объект, предположительно сгенерированный gensym, который возвращается в случае неудачи. Этот подход используется с get, который получает необязательный аргумент, указывающий, что вернуть, если указанное свойство не найдено:

```
> (get 'life 'meaning (gensym))
#:G618
```

Подход, где возможен возврат множественного значения, используемый gethash, является самым ясным. Мы не хотим передавать дополнительные аргументы каждой функции доступа, как мы делаем с get. И между двумя другими альтернативами, использование множественных значений является более общим; find-if может быть написан, чтобы возвращать два значения, но gethash так написать нельзя, без выполнения создания списка(consing), чтобы вернуть устраняющий неоднозначность списковую структуру. Таким образом, при написании новых функций для поиска, или других задач, где возможна неудача, обычно лучше следовать модели поведения используемой в gethash.

Идиома, которую можно найти в edible? это просто разновидность учета, который хорошо скрыт макросом. Для функций доступа таких как gethash нам нужна новая версия условного оператора aif, которая вместо связывания и проверки того же значения, возвращаемого функцией доступа, связывает первый возвращенный элемент(val), но также проверяет второе значение. Новая версия aif, называемая aif2, показана на рисунке 14-3. Используя её мы могли бы написать edible? как:

```
(defun edible? (x)
  (aif2 (gethash x edible)
          (if it 'yes 'no)
          'maybe))
```

Рисунок 14-3 также содержит аналогично измененные версии awhen, awhile, и acond. Для примера использования acond2, см. определение соответствующее стр. 239. Используя этот макрос мы можем выразить cond в виде функции, которая в противном случае была бы намного длиннее и менее симетричной.

Встроенный read указывает на неудачу так же как get. Ему требуется необязательный аргумент, говорящий, следует ли генерировать ошибку в случае eof, и если нет, какаое значени вернуть. Рисунок 14-4 содержит альтернативную версию read, которая использует второе возвращаемое значение для сообщения о неудаче: read2 возвращает два значения, входное выражение и флак, который равен nil при eof. Он вызывает read с символом сгенерированным gensym, который должен быть возвращен в случае eof, но чтобы избавиться от хлопот по построению gensym для каждого вызова read2, функция определяется как замыкание со скрытой копией символа из gensym сгенерированного во время компиляции.

Рисунок 14-4 также содержит удобный макрос для пепебора выражений в файле, написанный с использованием awhile2 и read2. Используя do-file мы могли бы, например, написать версию load как:

```
(defun our-load (filename)
  (do-file filename (eval it)))
```

```
(defmacro aif2 (test &optional then else)
      (let ((win (gensym)))
         `(multiple-value-bind (it ,win) ,test
             (if (or it ,win) ,then ,else))))

(defmacro awhen2 (test &body body)
      `(aif2 ,test
                  (progn ,@body)))

(defmacro awhile2 (test &body body)
      (let ((flag (gensym)))
         `(let ((,flag t))
             (while ,flag
                  (aif2 ,test
                        (progn ,@body)
                        (setq ,flag nil))))))

(defmacro acond2 (&rest clauses)
      (if (null clauses)
           nil
           (let ((cl1 (car clauses))
                    (val (gensym))
                    (win (gensym)))
              `(multiple-value-bind (,val ,win) ,(car cl1)
                   (if (or ,val ,win)
                        (let ((it ,val)) ,@(cdr cl1))
                        (acond2 ,@(cdr clauses)))))))
```

Рисунок 14-3: Анафорические макросы использующие множественные значения.

14.3 14-3 Ссылочная Прозрачность

Иногда говорят, что анафорические макросы нарушают ссылочную прозрачность, которую Gelernter и Jagannathan определяют следующим образом:

> Язык ссылочно прозрачен, если (a) какждое подвыражение может быть заменено любым другим, равным по ему по значению и (b) все вхождения выражения в данном контексте приводят к одному и тому же значению.

Обратите внимание, что стандарт применяется к языкам, а не к программам. Нет языков с присваиванием являющихся ссылочно прозрачными. Первый и последний x в этом выражении

```
(let ((g (gensym)))
   (defun read2 (&optional (str *standard-input*))
       (let ((val (read str nil g)))
         (unless (equal val g) (values val t)))))

(defmacro do-file (filename &body body)
   (let ((str (gensym)))
       `(with-open-file (,str ,filename)
              (awhile2 (read2 ,str)
               ,@body))))
```

Рисунок 14-4: Файловые утилиты.

```
(list x(setq x (not x))
        x)
```

дают разные значения, поскольку вмешивается setq. Правда, это ужасный код. Тот факт, что это возможно, означает что Lisp не является ссылочно прозрачным.

Norvig(Норвиг) отмечает, что было бы удобно переопределить if как:

```
(defmacro if (test then &optional else)
  `(let ((that ,test))
        (if that ,then ,else)))
```

но отклоняет этот макрос на том основании, что он нарушает ссылочную прозрачность.

Однако проблема здесь заключается в переопределении встроенных операторов, а не в использовании анафоры. Предложение (b) приведенного выше определения требует, чтобы выражение всегда возвращало одно и тоже значение "в данном контексте." Это не проблема if, внутри этого let выражения,

```
(let ((that 'which))
  ...)
```

символ that обозначает новыю переменную, потому что let объявлено чтобы создать новый контекст.

Проблема с макросом в том, что он переопределяет if, который не предполагает созданиеп нового контекста. Эта проблема исчезнет, если мы дадим анафорическим макросам отличные имена. (Начиная с CLTL2, это переопределение так или иначе является незаконным.) На протяжении всей части определения aif устанавливается новый контекст в котором it является новой переменной, такой макрос не нарушает ссылочную прозрачность.

Теперь, aif действительно нарушает другое соглашение, которое не имеет ничего общего со ссылочной прозрачностью: всегда указывать вновь устанавливаемые переменные в исходном коде. Выражение let выше ясно указывает на то, что that хочет сослаться на новую переменную. Можно утверждать, что привязка it в aif не достаточно ясна(видна). Однако это не очень сильный аргумент: aif создает только одну переменную, и создание этой переменной является единственной причиной его использования.

Common Lisp и сам не рассматривает это соглашение как неприкосновенное. Привязка функции CLOS call-next-method зависит от контекста точно таким же образом,

как привязка символа it делается в теле aif. (Для намека о том, как будет реализован call-next-method, смотрите макрос defmeth на странице 358.) В любом случае, такие соглашения введены только для достижения цели: облегчения чтения программ. И анафоры делают программы более легко читаемыми, также как они облегчают чтение в Английском языке.

15 15 Макосы возвращающие Функции

В главе 5 показано, как писать функции, котоыре возвращают другие функции. Макрос упрощает задачу объединения операторов. Эта глава покажет, как использовать макросы для построения абстракций, эквивалентных тем, которые определены в главе 5, но яснее и более эффективнее.

15.1 15-1 Построение Функций

Если f и g являются функциями, тогда f ∘ g(x)=f (g(x)). Раздел 5-4 показал, как реализовать оператор ∘ как функцию Lisp называемую compose:

```
> (funcall (compose #'list #'1+) 2)
(3)
```

В этом разделе, мы рассмотрим способы определения лучших компоновщиков(объединителей) функцийс помощью макросов. На рисунке 15-1 содержиться общий построитель функций называемый fn, который создает составные функции из их описаний. Его аргументом должно быть выражение вида (operator . arguments). Оператор(operator) может быть именем функции или макроса или compose, который рассматривается отдельно. Аргументами могут быть имена фунций или макросов от одного аргумента, или выражения которые могут быть аргументами для fn. Например,

```
(fn (and integerp oddp))
```

дает функцию, эквивалентную

```
#'(lambda (x) (and (integerp x) (oddp x)))
```

```
(defmacro fn (expr) `#',(rbuild expr))

(defun rbuild (expr)
      (if (or (atom expr) (eq (car expr) 'lambda))
           expr
           (if (eq (car expr) 'compose)
                (build-compose (cdr expr))
                (build-call (car expr) (cdr expr)))))

(defun build-call (op fns)
      (let ((g (gensym)))
        `(lambda (,g)
            (,op ,@(mapcar #'(lambda (f)
                                    `(,(rbuild f) ,g))
                                fns)))))

(defun build-compose (fns)
      (let ((g (gensym)))
        `(lambda (,g)
            ,(labels ((rec (fns)
                           (if fns
                                `(,(rbuild (car fns))
                                   ,(rec (cdr fns)))
                                g)))
                (rec fns)))))
```

Рисунок 15-1: Макрос общий построитель функций.

Если мы используем compose в качестве оператора, мы получаем функцию представляющую композицию аргументов, но без явных funcall, которые были бы необходимы, когда композиция(compose) была определена как функция. Например,

```
(fn (compose list 1+ truncate))
```

расширяется в:

```
#'(lambda (#:g1) (list (1+ (truncate #:g1))))
```

которая позволяет встроенную компиляцию простых функций, таких как list и 1+. Макрос fn берет имена операторов в общем смысле; лямбда-выражения также разрешено использовать, как в

```
(fn (compose (lambda (x) (+ x 3)) truncate))
```

которое расширяется в

```
#'(lambda (#:g2) ((lambda (x) (+ x 3)) (truncate #:g2)))
```

Здесь функция выраженная как лямбда-выражение, обязательно будет скомпилирована встроенной в вызывающий контекст, в то время как квотированное с решеткой('#) лямбда-выражение, переданное в качестве аргумента функции compose должна будет вызвана, посредством funcall.

В разделе 5-4 показано, как определить еще три построителя функций: fif, fint, и fun. Теперь они включены в общий макрос fn. Использование and как оператора дает пересечение операторов, заданных в качестве аргументов:

```
> (mapcar (fn (and integerp oddp))
              '(c 3 p 0))
(NIL T NIL NIL)
```

в то время как or дает их объединение:

```
> (mapcar (fn (or integerp symbolp))
              '(c 3 p 0-2))
(T T T NIL)
```

и if возвращает функцию, тело которой является условным:

```
> (map1-n (fn (if oddp 1+ identity)) 6)
(224466)
```

Однако, мы можем использовать другие функции Lisp, кроме этих трех:

```
> (mapcar (fn (list 1- identity 1+))
              '(1 2 3))
((012)(123)(234))
```

и аргументы в выражении fn сами могут быть выражениями:

```
> (remove-if (fn (or (and integerp oddp)
                                    (and consp cdr)))
                  '(1 (a b) c (d) 2 3-4 (e f g)))
(C (D) 2 3-4)
```

Заставляя fn рассматривать compose как особый случай его более мощным. Если вы вложите аргументы в fn, вы получите функциональную композицию. Например,

```
(fn (list (1+ truncate)))
```

расширяется в:

```
#'(lambda (#:g1)
          (list ((lambda (#:g2) (1+ (truncate #:g2))) #:g1)))
```

которая ведет себя как

```
(compose #'list #'1+ #'truncate)
```

Макрос fn рассматривает compose как особый случай только для упрощения чтения таких вызовов.

15.2 15-2 Рекурсия по Cdrs(по окончанию списка)

В разделах 5-5 и 5-6 показано, как писать функции, которые создают рекурсивные функции. Следующие два раздела показывают как анафорические макросы могут обеспечить ясный интерфейс к функциям, которые мы там определили.

В разделе 5-5 показано, как определить построитель рекурсеров для плоских списков с именем lrec. С lrec мы можем выразить вызов:

```
(defun our-every (fn lst)
     (if (null lst)
           t(and (funcall fn (car lst))
                 (our-every fn (cdr lst)))))
```

например для oddp как:

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f)))
```

```
            t)
```

Здесь макросы могут облегчить жизнь. Сколько мы должны сказать, чтобы выразить рекурсивную функцию? Если мы можем обратиться анафорически к текущему началу(car) списка (как it) и рекурсивно вызвать (как rec), мы должны быть в состоянии обойтись, чем-то вроде:

```
(alrec (and (oddp it) rec) t)
```

Рисунок 15-2 содержит определение макроса, который позволяет нам сказать это.

```
> (funcall (alrec (and (oddp it) rec) t)
                '(1 3 5))
T
```

```
(defmacro alrec (rec &optional base)
   "cltl2 version"
   (let ((gfn (gensym)))
       `(lrec #'(lambda (it ,gfn)
                   (symbol-macrolet ((rec (funcall ,gfn)))
                      ,rec))
              ,base)))

(defmacro alrec (rec &optional base)
   "cltl1 version"
   (let ((gfn (gensym)))
       `(lrec #'(lambda (it ,gfn)
                   (labels ((rec () (funcall ,gfn)))
                      ,rec))
              ,base)))

(defmacro on-cdrs (rec base &rest lsts)
   `(funcall (alrec ,rec #'(lambda () ,base)) ,@lsts))
```

Рисунок 15-2: Макросы для рекурсии по Списку.

Новый макрос работает путем преобразования выражения переданного как второй аргумент в функцию для передачи в lrec. Поскольку второй аргумент может ссылаться анафорически на it или rec, в расширении макроса тело функции должно находиться в области привязок, установленных для этих символов.

На рисунке 15-2 фактически есть две разные версии alrec. Версии используемой в предыдущих примерах требуется символьные макросы (Раздел 7-11). Только последние версии Common Lisp имеют символьные макросы, поэтому Рисунок 15-2 также содержит чуть менее удобную версию alrec, в которой rec определена как локальная функция. Цена этого неудобства том, что как функция, rec должна быть заключена в скобки:

```
(alrec (and (oddp it) (rec)) t)
```

Оригинальная версия предпочтительнее в реализациях Common Lisp, которые предоставляют symbol-macrolet.

Common Lisp, с его отдельным пространство имепн для функций делает неудобным использование этих построителей рекурсий для определения именованных функций:

```
(setf (symbol-function 'our-length)
       (alrec (1+ rec) 0))
```

```
(defun our-copy-list (lst)
      (on-cdrs (cons it rec) nil lst))

(defun our-remove-duplicates (lst)
      (on-cdrs (adjoin it rec) nil lst))

(defun our-find-if (fn lst)
      (on-cdrs (if (funcall fn it) it rec) nil lst))

(defun our-some (fn lst)
      (on-cdrs (or (funcall fn it) rec) nil lst))
```

Рисунок 15-3: Функции Common Lisp определенные с помощью on-cdrs.

Последний макрос на Рисунке 15-2 предназначен для того, чтобы сделать это еще более абстрактным. С помощью on-cdrs мы могли бы вместо этого, сказать:

```
(defun our-length (lst)
  (on-cdrs (1+ rec) 0 lst))

(defun our-every (fn lst)
  (on-cdrs (and (funcall fn it) rec) t lst))
```

На Рисунке 15-3 показаны некоторые существующие функции Common Lisp определенные с помощью нового макроса. Выраженные с помощью on-cdrs, эти функции упрощены к самой общей форме, и мы замечаем сходство между ними, которые иначе не были бы очевидны.

Рисунок 15-4 содержит некотоыре утилиты, которые можно легко определить с помощью on-cdrs. Первые три unions, intersections, и differences реализуют объединение, пересечение и дополнение множеств, соответственно. Common Lisp имеет встроенные функции для этих операций, но они могут принимать только два списка одновременно. Таким образом, если мы хотим найти объединение трех списков, мы должны сказать:

```
> (union '(a b) (union '(b c) '(c d)))
(A B C D)
```

Новое unions ведет себя также как union, но принимают произвольное количество аргументов, так что мы можем сказать:

```
> (unions '(a b) '(b c) '(c d))
(D C A B)
```

```
(defun unions (&rest sets)
   (on-cdrs (union it rec) (car sets) (cdr sets)))

(defun intersections (&rest sets)
   (unless (some #'null sets)
        (on-cdrs (intersection it rec) (car sets) (cdr sets))))

(defun differences (set &rest outs)
   (on-cdrs (set-difference rec it) set outs))

(defun maxmin (args)
   (when args
        (on-cdrs (multiple-value-bind (mx mn) rec
                     (values (max mx it) (min mn it)))
                 (values (car args) (car args))
                 (cdr args))))
```

Рисунок 15-4: Новые утилиты определенные с помощью on-cdrs.

Подобно union, unions не сохраняет поряедок элементов в начальных списках.

Тоже самое можно сказать в отношении между Common Lisp intersection и более общим intersections. В определении этой функции, для эффективности добавлена проверка на пустой аргумент; это приводит к более короткой схеме вычислений если одно из множеств является пустым.

Common Lisp также имеет функцию называемую set-difference, которая принимает два списка и возвращает элементы первого, которых нет во втором:

```
> (set-difference '(a b c d) '(a c))
(D B)
```

Наша новая версия обрабатывает несколько аргументов также как и остальные. Нарпимер, (differences x y z) эквивалента (set-difference x (unions y z)), хотя без создания списка которое влечет за собой последнее выражение.

```
> (differences '(a b c d e) '(a f) '(d))
(BCE)
```

Эти операторы множеств предназначены только для примера. Там нет реальной необходимости в них, поскольку они представляют собой вырожденный случай рекурсии, уже обработанный встроенной функцией reduce. Например, вместо

```
(unions ...)
```

с таким же успехом можно сказать

```
((lambda (&rest args) (reduce #'union args)) ...)
```

Однако в общем случае, on-cdrs является более мощным, чем reduce.

Поскольку rec ссылается на вызвов вместо значения, мы можем использовать on-cdrs для создания функций, которые возвращают множественные значения. Последняя функция на рисунке 15-4, maxmin, использует эту возможность, чтобы найти максимальный и минимальный элементы при единственном обходе списка:

```
> (maxmin '(3 4 2 8 5 1 6 7))
```

```
8
1
```

Также было бы возможно испольовать on-cdrs в некотором коде, который появляется в последующих главах. Например, compile-cmds (стр. 310)

```
(defun compile-cmds (cmds)
      (if (null cmds)
           'regs
           `(,@(car cmds) ,(compile-cmds (cdr cmds)))))
```

можно было бы определить просто:

```
(defun compile-cmds (cmds)
      (on-cdrs `(,@it ,rec) 'regs cmds))
```

15.3 15-3 Рекурсия на Поддеревьях

То что макросы делали для рекурсии на списках, они также могут делать для рекурсии на деревьях. В этом разделе, мы используем макросы для определения более ясных интерфейсов для рекурсеров по деревьям определенным в Разделе 5-6.

В Разделе 5-6 мы определили два создателя рекурсии по деревьям, ttrav, который всегда проходит все дерево, и trec, который является более сложным, но позволяет контролировать, когда нужно остановить рекурсию. Используя эти функции мы можем выразить our-copy-tree

```
(defun our-copy-tree (tree)
      (if (atom tree)
           tree
           (cons (our-copy-tree (car tree))
                    (if (cdr tree) (our-copy-tree (cdr tree))))))
```

как

```
(ttrav #'cons)
```

и вызвов rfind-if

```
(defun rfind-if (fn tree)
  (if (atom tree)
        (and (funcall fn tree) tree)
        (or (rfind-if fn (car tree))
             (and (cdr tree) (rfind-if fn (cdr tree))))))
```

например для oddp как:

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
       #'(lambda (tree) (and (oddp tree) tree)))
```

Анафорические макросы могут улучшить интерфейс к trec, как это было сделано для lrec в предыдущем разделе. Макрос, достаточный для общего случая, должен быть в состоянии анафорически ссылаться на три вещи: текущее дерево, которое мы назовем it, рекурсию вниз по левому поддереву, которую мы назовем left, и на рекурсию вниз по правому поддереву, которую мы назовем right. Установив эти соглашения, мы будем в состоянии выразить предыдущие функции в терминах нового макроса следующим образом:

```
(atrec (cons left right))
```

```
(atrec (or left right) (and (oddp it) it))
```

Рисунок 15-5 содержит определение этого макроса.

В весрисях Lisp которые не имеют symbol-macrolet, мы можем определить atrec используя второе определение на Рисунке 15-5. Эта версия определяет left и right как локальные функции, таким образом our-copy-tree может быть выражено как:

```
(atrec (cons (left) (right)))
```

Для удобства, мы также определим макрос on-trees, который аналогичен on-cdrs из предыдущего раздела. На Рисунке 15-6 показаны четыре функции из раздела 5-6 определенные с помощью on-trees.

Как отмечено в Главе 5, функции созданные генератором рекурсеров, определенные в этой главе не будут иметь хвостовой рекурсии. Использование on-cdrs или on-trees для определения функций не обязательно даст наибольшую эффективную реализацию. Лежащие в основе trec и lrec, эти макросы в основном предназначены для использования в прототипах и в части программы, где эффективность не имеет первостепенного значения. Тем не менеее, основная идея этой главы и главы 5 заключается в том, что можно написать генераторы функций и предоставить ясный интерфейс в виде макросов для них. Эта же техника может быть одинаково хорошо использована для создания генераторов функций, которые создают действительно эффективный код.

```
(defmacro atrec (rec &optional (base 'it))
      "cltl2 version"
      (let ((lfn (gensym)) (rfn (gensym)))
       `(trec #'(lambda (it ,lfn ,rfn)
                  (symbol-macrolet ((left (funcall ,lfn))
                                    (right (funcall ,rfn)))
                    ,rec))
              #'(lambda (it) ,base))))

(defmacro atrec (rec &optional (base 'it))
      "cltl1 version"
      (let ((lfn (gensym)) (rfn (gensym)))
       `(trec #'(lambda (it ,lfn ,rfn)
                  (labels ((left () (funcall ,lfn))
                           (right () (funcall ,rfn)))
                    ,rec))
              #'(lambda (it) ,base))))

(defmacro on-trees (rec base &rest trees)
      `(funcall (atrec ,rec ,base) ,@trees))
```

Рисунок 15-5: Макросы для рекурсии на деревьях.

```
(defun our-copy-tree (tree)
      (on-trees (cons left right) it tree))

(defun count-leaves (tree)
      (on-trees (+ left (or right 1)) 1 tree))

(defun flatten (tree)
      (on-trees (nconc left right) (mklist it) tree))

(defun rfind-if (fn tree)
      (on-trees (or left right)
                (and (funcall fn it) it)
                tree))
```

Рисунок 15-6: Функции определенные с использованием on-trees.

```
(defconstant unforced (gensym))

(defstruct delay forced closure)

(defmacro delay (expr)
   (let ((self (gensym)))
        `(let ((,self (make-delay :forced unforced)))
           (setf (delay-closure ,self)
                    #'(lambda ()
                         (setf (delay-forced ,self) ,expr)))
           ,self)))

(defun force (x)
   (if (delay-p x)
          (if (eq (delay-forced x) unforced)
                 (funcall (delay-closure x))
                 (delay-forced x))
          x))
```

Рисунок 15-7: Реализация force и delay.

15.4 15-4 Ленивые вычисления

Ленивые вычисления означают выполнение вычислений выражения только тогда, когда вам нужно его значение. Один из способов использования ленивого(отложенного) вычисления является создание объекта известного как задержка(delay). delay это обещание выдать значение выражения, если оно необходимо в более позднее время. Между тем, так как promise(обещание) является объектом Lisp, оно может служить многим целям представляя его значение. И когда значение выражения понадобиться, delay может вернуть его Scheme имеет встроенную поддержку для задержек(delays). Оператор Scheme force и delay можно реализовать в Common Lisp как показано на Ри-

сунке 15-7. delay представлена как структура из двух частей. Первое поле указывает, была ли delay уже вычислена, и если это так, содержит его значение. Второе поле содержит замыкание, которое надо вызвать, чтобы найти значение, которое представляет delay. Макрос delay принимает выражение, и возвращает delay представляющую его значение:

```
> (let ((x 2))
        (setq d (delay (1+ x))))
#S(DELAY ...)
```

Вызвать замыкание внутри delay значит форсировать задержку(force the delay). Функция force берет любой объект: для обычных объектов это функция identity(ничего не делает, а возвращает сам объект),но для задержек(delays) она требует значение, которое эта delay представляет.

```
> (force 'a)
A> (force d)
3
```

Мы используем force всякий раз, когда имеем дело с объектами, которые могут быть задержаны(быть delay). Например, если мы сортируем список, который может содержать задержки(delay), мы должны сказть:

```
(sort lst #'(lambda (x y) (> (force x) (force y))))
```

Немного не удобно использовать задержки(delay) в этой незащищенной форме. В реальном приложении они могут быть скрыты под другим слоем абстракции.

16 16 Макросы определяющие Макросы.

Шаблоны в коде часто указывают на не обходимость в новых абстракциях. Это правило выполняется так же, как для кода, так и для макросов. Когда несколько макросов имеют определения подобной формы, мы можем написать макрос для их создания(генерации). В этой главе представлены три примера макросов создающих макро-определения: одно для определения абревиатур, одно для определения макросов доступа, и третье для определения анафорических макросов, похожих на описанные в Разделе 14-1.

16.1 16-1 Сокращения(Аббревиатуры)

Самое простои использование макросов - это сокращения. Некотоыре операторы Common Lisp имеют довльно длинные имена. Высокий рейтинг среди них (хотя ни в коем случе не самое длинное) у destructuring-bind, в котором 18 знаков. Следствие из принципа Стила(Steele's principle) (стр. 43) заключается в том, что часто используемые операторы должны иметь более короткие имена. ("Мы думаем, что добавление является дешеовой операцией, отчасти потому, что мы можем записать его одним знаком: '+'.") Встроенный макрос destructuring-bind вводит новый уровень абстракции, но фактический выигрыш в краткости нивелируется его длинным именем:

```
(let ((a (car x)) (b (cdr x))) ...)

(destructuring-bind (a . b) x ...)
```

Программу, также как и текст, легче всего читать, если она содержит не более 70 знаков в строке. Мы с самого начала становимся в невыгодное положение, когда длина отдельных имен составляет четверть этой величины.

```
(defmacro abbrev (short long)
      `(defmacro ,short (&rest args)
          `(,',long ,@args)))

(defmacro abbrevs (&rest names)
      `(progn
          ,@(mapcar #'(lambda (pair)
                        `(abbrev ,@pair))
                    (group names 2))))
```

Рисунок 16-1: Автоматическое определение сокращений.

К счастью, в таком языке как Lisp, вам не нужно жить со всеми этими решениями разработчиков. Имея определение

```
(defmacro dbind (&rest args)
  `(destructuring-bind ,@args))
```

вам больше никогда не будет нужно снова использовать длинное имя. Аналогично для multiple-value-bind, которое длиннее и чаще используется.

```
(defmacro mvbind (&rest args)
```

```
  `(multiple-value-bind ,@args))
```

Обратите внимание, насколько близки определения dbind и mvbind. В самом деле, этой формулы &rest и запятая с собакой(,@) будет достаточно, для того чтобы определить макрос сокращения(аббривиатуру) для любой функции,[1], или специальной формы. Зачем делать много определений по модели mvbind, когда у нас есть макросы, которые мы можем использовать?

Чтобы определить макрос определяющий макрос, нам часто нужны вложенные обратные кавычки. Вложенные обратные кавычки, как известно, трудно понять. В конце концов с общим случаем мы разберемся, но не стоит ожидать, что мы взглянув на произвольное выражение в обратных кавычках скажем, что оно выдает. Это не ошибка в Lisp, что это невозможно, и не ошибка в обозначениях, на которые нельзя просто посмотреть, подобно, сложному интегралу, и сказать, какое значение он имеет. Трудность заключается в самой проблеме, а не в обозначениях.

Однако, как и при поиске интеграла, мы можем разбить анализ обратных кавычек на маленькие шаги, каждый из которых можно легко проследить. Предположим, мы хотим написать макрос abbrev, который позволит нам определить mvbind просто сказав

```
(abbrev mvbind multiple-value-bind)
```

Рисунок 16-1 содержит определение этого макроса. Как мы его получили? Определение такого макроса может быть получено из примера расширения. Одним из его расширений является:

```
(defmacro mvbind (&rest args)
       `(multiple-value-bind ,@args))
```

Понять происхождение будет проще, если мы вынем multiple-value-bind из обратных кавычек, поскольку мы знаем что оно будет аргументом для возможного макроса. Это дает нам эквивалентное определение

```
(defmacro mvbind (&rest args)
       (let ((name 'multiple-value-bind))
        `(,name ,@args)))
```

Теперь мы берем это выражение и превращаем его в шаблон. Мы добавляем к нему обратную кавычку и заменяем выражения, которые будут изменяться, на переменные.

```
`(defmacro ,short (&rest args)
        (let ((name ',long))
         `(,name ,@args)))
```

Последний шаг состоит в том, чтобы упростить это выражение путем замены(подстановки) вместо name поставить ',long внутри выражения с обратной кавычкой:

```
`(defmacro ,short (&rest args)
        `(,',long ,@args))
```

который и даст нам тело макроса определенного на Рисунке 16-1.

Рисунок 16-1 также содержит abbrevs, для случаев, когда мы хотим определить несколько сокращений за один раз.

```
(abbrevs dbind destructuring-bind
```

[1] Хотя аббревиатуру нельзя передать в apply или funcall.

```
                mvbind multiple-value-bind
                mvsetq multiple-value-setq)
```

Пользователь abbrevs не должен вставлять дополнительные скобки, потому что abbrevs вызывает group (стр. 47) для группировки аргументов по два. Это вообще хорошая вещь для макросов - избавить пользователей от логически ненужного ввода скобок и group будет полезна для большинства таких макросов.

```
(defmacro propmacro (propname)
      `(defmacro ,propname (obj)
          `(get ,obj ',',propname)))

(defmacro propmacros (&rest props)
      `(progn
          ,@(mapcar #'(lambda (p) `(propmacro ,p))
                    props)))
```

Рисунок 16-2: Автоматическое определение макросов доступа.

16.2 16-2 Свойства

Lisp предлагает множество способов связать свойства с объектами. Если рассматриваемый объект может быть представлен как символ, один из самых удобных(хотя и наименее эффективных) способов, заключается в использовании списка свойств символа. Чтобы описать тот факт, что объект o имеет свойство p, значение которого равно v, мы модифицируем список свойств o:

```
(setf (get o p) v)
```

Итак, чтобы сказать что ball1 имеет красный(red/значение) цвет(color/свойство), мы говорим:

```
(setf (get 'ball1 'color) 'red)
```

Если мы будем часто ссылаться на некоторые свойства объектов, мы можем определить макрос, чтобы получать их:

```
(defmacro color (obj)
   `(get ,obj 'color))
```

и затем использовать color вместо get:

```
> (color 'ball1)
RED
```

Поскольку вызовы макросов прозрачны для setf (см. Главу 12) мы также можем сказать:

```
> (setf (color 'ball1) 'green)
GREEN
```

Такие макросы имеют преимущество в том, что скрывают конкретный способ, которым программа представляет цвет(color) объекта. Списки свойств медленные: дальнейшие версии программы могут, для увеличения скорости, представлять свойства, такие как цвет(color) как поле в структуре, или как запись в хеш-таблице. Когда для

получения данных используется фасад макроса, такого как color, становиться легким, даже в сравнительно зрелой программе, изменять код самого низкого уровня. Если программа переключается с использования списков свойств, на использование структур, ничто не выходит за рамки необходимости изменения макросов доступа; никакой код который видит только фасад, не должен знать о перестройке, происходящей за ним.

Для свойства weight мы можем определить макрос, аналогичный написанному для color:

```
(defmacro weight (obj)
  `(get ,obj 'weight))
```

Подобно сокращениям в предыдущем разделе, определения color и weight почти одинаковы. Здесь propmacro (Рисунок 16-2) может играть ту же роль, что и abbrev выше.

Макрос определяющий макросы может быть разработан тем же процессом, что и любой другой макрос: смотрим на вызов макроса, затем на его предполагаемое расширение, затем выясняем как превратить первое во второе. Мы хотим

```
(propmacro color)
```

расширить в

```
(defmacro color (obj)
  `(get ,obj 'color))
```

Хотя это выражение само по себе является определением макроса defmacro, мы все же можем создать его из шаблона, заключив его в обратные кавычки и поместив перед именами параметров запятую вместо экземпляра color. Как и в предыдущем разделе, мы начинаем с преобразования его таким образом, чтобы вынести все экземпляры color из под обратных кавычек:

```
(defmacro color (obj)
  (let ((p 'color))
       `(get ,obj ',p)))
```

Теперь мы идем вперед и делаем шаблон,

```
`(defmacro ,propname (obj)
    (let ((p ',propname))
         `(get ,obj ',p)))
```

который упрощается до

```
`(defmacro ,propname (obj)
       `(get ,obj ',',propname))
```

Для случаев, когда вся группа имен свойств должна быть определена как макросы, есть propmacros (Рисунок 16-2), который расширяется в ряд индивидуальных вызовов propmacro. Подобно abbrevs, этот скромный кусочек кода на самом деле является макросом определяющим макросы определяющие макросы(macro-defining-macro-defining).

Хотя в этом разделе рассматриваются списки свойств, техника описанная здесь является общей. Мы могли бы использовать её для определения макросов доступа к данным, хранящимся в любой форме.

16.3 16-3 Анафорные Макросы

Раздел 14-1 дал определения нескольких анафорических макросов. когда вы используете макрос, подобные aif или aand, при вычислении некоторых аргументов символ it будет привязан к значению возвращаемому одним из предыдущих выражений. Так что вместо

```
(let ((res (complicated-query)))
  (if res
           (foo res)))
```

вы можете использовать просто

```
(aif (complicated-query)
         (foo it))
```

и вместо

```
(let ((o (owner x)))
  (and o (let ((a (address o)))
                  (and a (city a)))))
```

просто

```
(aand (owner x) (address it) (city it))
```

В разделе 14-1 представлены семь анафорических макросов: aif, awhen, awhile, acond, alambda, ablock, и aand. Эти семь ни в коем случае не являются единственными полезными анафорическими макросами этого типа. На самом деле, мы можем определить анафорический вариант любой функции или макроса Common Lisp. Многие из этих макросов будут похожи на mapcon: редко используемыми, но незаменимыми при необходимости.

Например, мы можем определить a+ так, чтобы и с aand, it всегда был связан со значением, возвращаемым предыдущим аргументом. Следующая функция рассчитывает стоимость обедов в Массачусетсе:

```
(defmacro a+ (&rest args)
   (a+expand args nil))

(defun a+expand (args syms)
   (if args
        (let ((sym (gensym)))
           `(let* ((,sym ,(car args))
                        (it ,sym))
               ,(a+expand (cdr args)
                                (append syms (list sym)))))
        `(+ ,@syms)))

(defmacro alist (&rest args)
   (alist-expand args nil))

(defun alist-expand (args syms)
   (if args
        (let ((sym (gensym)))
           `(let* ((,sym ,(car args))
                        (it ,sym))
               ,(alist-expand (cdr args)
                                (append syms (list sym)))))
        `(list ,@syms)))
```

Рисунок 16-3: Определение a+ и alist.

```
(defun mass-cost (menu-price)
  (a+ menu-price (* it .05) (* it 3)))
```

Налог на еду в Массачусетсе составляет 5%, и жители часто расчитывают утроенный налог. По этой формуле, общая стоимость жареных морепродуктов от Dolphin Seafood будет:

```
> (mass-cost 7-95)
9-54
```

но это включает в себя салат и печный картофель.

Макрос a+, определенный на Рисунке 16-3, основан на рекурсивной функции a+expand, генерирующей его расширение. Общая стратегия a+expand состоит в том, чтобы остаток списка(cdr) передавался как аргумент в вызов макроса, генерируя последовательность вложенных выражений let; каждый let оставляет it связанным с другим аргументов, но также связывает определенный gensym

```
(defmacro defanaph (name &optional calls)
       (let ((calls (or calls (pop-symbol name))))
        `(defmacro ,name (&rest args)
            (anaphex args (list ',calls)))))

(defun anaphex (args expr)
      (if args
           (let ((sym (gensym)))
              `(let* ((,sym ,(car args))
                        (it ,sym))
                   ,(anaphex (cdr args)
                             (append expr (list sym)))))
           expr))

(defun pop-symbol (sym)
      (intern (subseq (symbol-name sym) 1)))
```

Рисунок 16-4: Автоматическое определение анафорических макросов.

с каждым аргументом. Функция расширения накапливает список этих gensyms, и когда он достигает конца списка аргументов, она возвращает выражение + с gensyms в качестве аргументов. Итак, выражение

```
(a+ menu-price (* it .05) (* it 3))
```

дает расширение макроса:

```
(let* ((#:g2 menu-price) (it #:g2))
  (let* ((#:g3 (* it 0-05)) (it #:g3))
        (let* ((#:g4 (* it 3)) (it #:g4))
           (+ #:g2 #:g3 #:g4))))
```

Рисунок 16-3 также содержит аналогичное определение alist:

```
> (alist 1 (+ 2 it) (+ 2 it))
(1 3 5)
```

Еща раз, определения a+ и alist почти идентичны. Если мы хотим, определить больше макросов подобных этим, они также будут в основном дублировать их код. Зачем? Разве у нас нет программ для их создания? Макрос defanaph на Рисунке 16-4 будет это делать. С defanaph, определение a+ и alist также просто как

```
(defanaph a+)
(defanaph alist)
```

Расширения a+ и alist определенные таким образом, будут идентичны расширениям сделанным с помощью кода на Рисунке 16-3. Макрос defanaph создающий макро определение создаст анафорический вариант чего-либо, чьи аргументы вычисляются в соответствии с обычным правилом вычисления для функций. То есть, defanaph будет работать на чем угодно, чьи аргументы все вычисляются и вычисляются с лева на право. Так что вы не можете использовать эту версию defanaph для определения aif или awhile, но вы можете использовать ее для определения анафорного варианта любой функции.

Как a+ вызывает a+expand для создания своего расширения, defanaph определяет макрос, который вызовет anaphex что бы сделать тоже самое. Общий расширитель

anaphex отличается от a+expand только получением как аргумента имени функции, которое должно появиться в конце расширения. Фактически, a+ теперь можно определить:

```
(defmacro a+ (&rest args)
  (anaphex args '(+)))
```

Ни для anaphex, ни для a+expand нет необходимости в определении их как отдельных функций: anaphex мог быть определен с помощью labels или alambda внутри defanaph. Генераторы расширения здесь разбиты на отдельные функции только лишь для ясности.

По умолчанию, defanaph определяет, что вызывать при расширении откинув первую букву (предположительно a(анафор)) из своего первого аргумента. (Эту операцию выполняет pop-symbol.) Если пользователь предпочитает указать альтернативное имя, он может быть передан в качестве необязательного аргумента. Хотя defanaph может может создавать анафорические варианты всех функций и некоторых макросов, он накладывает некоторые досадные ограничения:

1. Он работает только для операторов, все аргументы которых вычисляются.
2. В разложении макроса it всегда связывается с последовательными аргументами. А в некоторых случаях - например в awhen - мы хотим, чтобы it оставалось связанным со значением первого аргумента.
3. Он не будет работать для такого макроса как setf, который ожидает обобщенную переменную в качестве первого аргумента.

Давайте рассмотрим, как снять некоторые из этих ограничений. Часть первой проблемы может быть решена путем решения второй. Чтобы создать расширение для макроса подобного aif нам нужна модифицированная версяи anaphex, которая заменяет только первый аргумент в вызове макроса:

```
(defun anaphex2 (op args)
  `(let ((it ,(car args)))
        (,op it ,@(cdr args))))
```

Эта не рекурсивная версия anaphex не должна гарантировать, что разложение макроса будет связывать it с результатами вычисления последовательных аргументов макроса, чтобы он мог генерировать расширение, которое не обязательно вычисляет все аргументы указванные в вызове макроса. Только первый аргумент должен быть вычислен, чтобы связать it с его значением. Так aif может быть определен как:

```
(defmacro aif (&rest args)
       (anaphex2 'if args))
```

Это определение будет отличаться от оригинала на странице 191 только в том месте где оно будет выдавать жалобу, если в aif будет переадано неправильное количество аргументов; для корректного вызова макроса, эти два определения создают идентичные расширения.

Третья проблема, которая заключается в том, что defanaph не будет работать с обобщенными переменными, может быть решена с помощью использования в расширении функции _f (стр. 173). Операторы, такие как setf, могут быть обработаны вариантом anaphex2 определяемым следующим образом:

```
(defun anaphex3 (op args)
```

```
`(_f (lambda (it) (,op it ,@(cdr args))) ,(car args)))
```

Этот расширитель предполагает, что вызов макроса будет иметь один или несколько аргументов, первый из которых будет обобщенной переменной. Используя его мы можем определить asetf таким образом:

```
(defmacro asetf (&rest args)
       (anaphex3 'setf args))
```

На Рисунке 16-5 показаны все три функции расширители, объединенные под управлением одного макроса, нового defanaph. Пользователь сообщает тип требуемого расширения макроса с помощью необязательного параметра, ключевого слова, котоый указывает правило вычисления, которое будет использоваться аргументов в вызове макроса. Если этот параметр является:

:all (по умолчанию) расширение макроса будет идти по модели alist. Все аргументы в вызове макроса будут вычислены, причем it всегда будет связан со значением предыдущего аргумента.

:first расширение макроса будет происходит по модели aif. Только первый аргумент будет обязательно вычислен и it будет связано с его значением.

:place разложение будет идти по модели asetf. Первый аргумент будет рассматриваться как обобщенная переменная и it будет связан с её первоначальным значением.

Используя новый defanaph, некоторые из предыдущих примеров будут определены следующим образом:

```
(defmacro defanaph (name &optional &key calls (rule :all))
   (let* ((opname (or calls (pop-symbol name)))
          (body (case rule
                      (:all       `(anaphex1 args '(,opname)))
                      (:first `(anaphex2 ',opname args))
                      (:place `(anaphex3 ',opname args)))))
      `(defmacro ,name (&rest args)
         ,body)))

(defun anaphex1 (args call)
   (if args
       (let ((sym (gensym)))
          `(let* ((,sym ,(car args))
                  (it ,sym))
             ,(anaphex1 (cdr args)
                        (append call (list sym)))))
       call))

(defun anaphex2 (op args)
   `(let ((it ,(car args))) (,op it ,@(cdr args))))

(defun anaphex3 (op args)
   `(_f (lambda (it) (,op it ,@(cdr args))) ,(car args)))
```

Рисунок 16-5: Более общий defanaph.

```
(defanaph alist)
(defanaph aif :rule :first)
(defanaph asetf :rule :place)
```

Одним из преимуществ asetf явтяется то, что он позволяет определить большой класс макросов на обобщенных переменных, не беспокоясь о множественном вычислении. Например, мы можем определить incf как:

```
(defmacro incf (place &optional (val 1))
  `(asetf ,place (+ it ,val)))
```

и описать pull (стр. 173) как:

```
(defmacro pull (obj place &rest args)
  `(asetf ,place (delete ,obj it ,@args)))
```

17 17 Макросы Чтения

Различают три больших момента в жизни Lisp выражения: время чтения, время компиляции и время выполнения. Функции управлют во время выполнения. Макросы дают нам возможность выполнять преобразования в программах во время компиляции. В этой главе обсуждаются макросы чтения, которые выполняют свою работу во время чтения.

17.1 17-1 Макросы Знаки

В соответствии с общей философией Lisp, у вас есть большой контроль над читателем (reader). Его поведение управляется свойствами и переменными, которые все могут быть изменены на лету. Читатель(reader) может быть запрограммирован на нескольких уровнях. Самый простой способ изменить его поведение - определить новые Макросы-Знаки.

Макрос-Знак это знак(литерал, буква), который требует особого отношения от читателя (reader) Lisp. Например, строчная буква a обычно обрабатывается так же как строчная буква b, но левая скобка "(" - это нечто совсем иное: она говорит Lisp-у начать чтение списка. Каждый такой знак имеет функцию связанную с ним, которая сообщает Lisp читателю (reader), что делать когда встречается этот знак. Вы можете изменить функцию связанную с существующим Макросом-Знаком или определить новые Макросы-Знаки самостоятельно.

Встроенная функция set-macro-character обеспечивает один из способов определения Макросов Чтения. Она требует знак и функцию, а заем при чтении встречая указанный знак возвращает результат вызова этой функции.

Один из старейших макросов чтения в Lisp является ', цитата/кавычка. Вы можете обойтись и без кавычки - ' , всегда записывая (quote a) вместо 'a, но это будет очень утомительно, и сделает ваш код сложнее для чтения. Макрос чтения кавычка, делает возможным использование 'a как аббревиатуру для (quote a). Мы могли бы определить его как на Рисунке 17-1.

```
(set-macro-character #\'
      #'(lambda (stream char)
          (list 'quote (read stream t nil t))))
```

Рисунок 17-1: Возможное определение мароса знака - '.

Когда читатель(read) встречается с экземпляром ' в обычном контексте (т.е не в "a'b" или |a'b|), он вернет результат вызова этой функции в текущий поток и знак. (Функция игнорирует этот второй параметр, который всегда будет знаком кавычки.) Таким образом, когда читатель(read) видит 'a, он вернет (quote a).

Последние три аргумента для read, соответственно, управляют должен ли конец файла вызывать ошибку, какое значение вернуть в противном случае, и признак того что вызов read происходит внутри другого вызова read. Почти во всех макросах чтения, второй и четвертый аргументы должны быть t, и поэтому третий аргумент не имеет значения.

Макросы чтения и обычные макросы, оба типа являются функциями нижнего уровня. И как функции, которые генерируют расширения макросов, функци связанные с Макросами-Знаками не должны иметь побочных эффектов, кроме как в потоке(stream) из которого они производят чтение. Common Lisp явно не дает никаких гарантий относительно того, когда и как часто будет вызываться функция связанная с макросом чтения (См. CLTL2, стр. 543.)

Макросы и макросы чтения смотрят на нашу программу на разных этапах. Макросы получают считанную программу, когда она уже была проанализирована и разобрана в Lisp обекты читателем(reader), и макросы чтения работают с программой, когда она еще является текстом. Тем не менее, вызывая read для этого текста, макрос чтения может, если захочет, получить проанализированные объекты Lisp. Таким образом, макросы чтения, по крайней мере так же мощны, как и обычные макросы.

В действительности, макросы чтения являются даже более мощными, по крайней мере, в двух направлениях. Макрос чтения влияет на все, что читает Lisp, в то время как обычный макрос расширяется только в код. И так как макаросы чтения обычно вызывают read рекурсивно, выражения подобные

```
''a
```

превращаются в

```
(quote (quote a))
```

тогда как, если бы мы попытались определить сокращение для кавычки используя обычный макрос,

```
(defmacro q (obj)
  `(quote ,obj))
```

```
(set-dispatch-macro-character #\# #\?
  #'(lambda (stream char1 char2)
      `#'(lambda (&rest ,(gensym))
           ,(read stream t nil t))))
```

Рисунок 17-2: Макросы чтения для постоянных функций.

он будет работать изолированно,

```
> (eq 'a (q a))
T
```

но не когда вложен. Например,

```
(q (q a))
```

будет расширяться в

```
(quote (q a))
```

17.2 17-2 Диспетчерские Макросы-Знаки

Решетка с кавычкой #', подобно другим маросам чтения начинающимся с решетки #, является примером подвида, называемым диспетчерскими макросами чтения. Они появляются как два знака, первый из которых называется диспетчерским знаком. Цель таких макросов чтнения простая, чтобы можно было максимально использовать

набор ASCII знаков; один знак позволяет иметь столько макросов чтения, сколько всего одиночных знаков.

Вы можете (с помощью make-dispatch-macro-character) определить свой собственный диспетчерский макрос-знак, но так как # уже определен как один из них, вы можете использовать его. Некоторые комбинации начинающиеся с # явно зарезервированы для использования вами; другие доступны, в том смысле, что они еще не имеют предопределенного значения в Common Lisp. Полный список приведен в CLTL2, стр. 531.

Новые комбинации диспетчерских макро знаков могут быть определены путем вызова функции set-dispatch-macro-character, похожей на set-macro-character за исключением того, что она принимает два знаковых аргумента. Одной из зарезервированных комбинаций для программиста является #?. На Рисунке 17-2 показано, как определить эту комбинацию как макрос чтения для постоянных функций.Теперь #?2 будет читаться как функция, которая принимает любое количество аргументов и возвращает 2. Например:

```
> (mapcar #?2 '(a b c))
(2 2 2)
```

```
(set-macro-character #\] (get-macro-character #\)))

(set-dispatch-macro-character #\# #\[
 #'(lambda (stream char1 char2)
      (let ((accum nil)
               (pair (read-delimited-list #\] stream t)))
         (do ((i (ceiling (car pair)) (1+ i)))
              ((> i (floor (cadr pair)))
                (list 'quote (nreverse accum)))
           (push i accum)))))
```

Рисунок 17-3: Макрос чтения определяющий разделители.

Этот пример создает новый оператор выглядящий довольно бессмысленным, но в программах которые используют много функциональных аргументов, постоянные функции часто необходимы. По факту, некоторые диалекты предоставляют встроенную функцию, которая вызывается всегда для их определения.

Обратите внимание, что вполне нормально использовать макросы-Знаки в определении этого макроса-знака: как и в любом Lisp выражении, они исчезают, когда определение будет прочитано. Так же хорошо использовать Макросы-Знаки после #?. Определение #? вызывает read, поэтому макросы-знаки, подобные ' и #' ведут себя как обычно:

```
> (eq (funcall #?'a) 'a)
T> (eq (funcall #?#'oddp) (symbol-function 'oddp))
T
```

17.3 17-3 Разделители

После простых макросов-знаков, наиболее часто определяемыми макросами-знаками являются разелители списка. Другой комбинацией зарезервированной для пользователя, является #[. На рисунке 17-3 приведен пример, как этот символ может быть определен как более сложный вид левой скобки. Он определяет выражения вида #[x y] для читателя(read) как список всех целых чисел заключенных между x и y, включительно:

```
> #[2 7]
(2 3 4 5 6 7)
```

Единственно новая вещь в этом макросе чтения это вызов read-delimited-list, встроенной функции предусмотренной, как раз для таких случаев. Её первый аргуентом является знак, рассматриваемый как конец списка. Чтобы] был распознан как разделитель, он должен быть первым в этой роли, отсюда и предварительный вызов set-macro-character.

```
(defmacro defdelim (left right parms &body body)
  `(ddfn ,left ,right #'(lambda ,parms ,@body)))

(let ((rpar (get-macro-character #\) )))
  (defun ddfn (left right fn)
    (set-macro-character right rpar)
    (set-dispatch-macro-character #\# left
       #'(lambda (stream char1 char2)
            (apply fn
                     (read-delimited-list right stream t))))))
```

Рисунок 17-4: Макрос для определения макросов чтения разделителей.

Большинство потенциальных определений макросов чтения разделителей будут дублировать большую часть кода на рисунке 17-3. Макрос можно поместить в более абстрактный интерфейс всего этого механизма. На Рисунке 17-4 показано, как мы можем определить утилитуs для определенения макросов чтения разделителя. Макрос defdelim принимает два знака, списко параметров и тело кода. Список параметров и тело кода неявно определяют функцию. Вызов defdelim определяет первый символ как диспетчерский макрос чтения, который выполняет чтение до второго указанного знака, а затем возвращает результат применения функции к тому, что он прочитал. Между прочим, тело функции на рисунке 17-3 также требует утилиты: которую мы уже определили, фактически: mapa-b, со страницы 54. Используя defdelim и mapa-b, макрос чтения определенный на Рисунке 17-3 может быть записан:

```
(defdelim #\[ #\] (x y)
     (list 'quote (mapa-b #'identity (ceiling x) (floor y))))
```

Другим полезным применением макроса чтения разделителей может быть функциональная композиция. Раздел 5-4 определяет оператор для функциональной композиции:

```
> (let ((f1 (compose #'list #'1+))
              (f2 #'(lambda (x) (list (1+ x)))))
```

```
        (equal (funcall f1 7) (funcall f2 7)))
T
```

Когда мы составляем встроенные функции, такие как list и 1+, нет причин ждать времени выполнения для того, чтобы выполнить вызов compose. Раздел 5-7 предлагал, аллтернативу; добавив макрос чтения решетка с точкой #. к выражению compose,

```
#.(compose #'list #'1+)
 (defdelim #\{ #\} (&rest args)
       `(fn (compose ,@args)))
```

Рисунок 17-5: Макрос чтения для функциональной композиции.

мы могли бы заставить его выполниться во время чтения.

Здесь мы показываем подробно, но более ясное решение. Макрос чтения на Рисунке 17-5 определяет выражение вида #{f 1 f 2 ...fn} как чтение композиции функций f 1, f 2,...,fn. Таким образом:

```
> (funcall #{list 1+} 7)
(8)
```

Он работает путем генерации вызова функций до fn (стр. 202), которая создает функцию во время компиляции.

17.4 17-4 Когда Что Происходит

В заключении, может быть полезно разобраться в возможной путанице. Если макросы чтения вызываются до обычных макросов, как получается, что эти макросы могут расширяться в выражения, которые содержат макросы чтения? Например, макрос:

```
(defmacro quotable ()
  '(list 'able))
```

генерирует расширение с кавычкой в нем. Или нет? На самом деле происходит следующее, обе кавычки в определении этого макроса раскрываются когда выражение defmacro читается, первращаясь в

```
(defmacro quotable ()
  (quote (list (quote able))))
```

Обычно нет ничего плохого в том, что разложения макроса могут содержать макросы чтения, потому что определение макроса чтения не (или не должно) измениться, между временем чтения и временем компиляции.

18 18 Деконструкция

Деконструкция это обобщение присваивания. Операторы setq и setf выполняют присванивание для отдельных переменных. Деконструкция сочетает в себе присванивание с доступом: вместо того, чтобы давать единственную переменную в качестве первого аргумента, мы дает образец переменных, каждому из которых присваивается значение, встречающееся в соответствующей позиции в некоторой структуре.

18.1 18-1 Деконструкция Списков.

Начиная с CLTL2, Common Lisp включает новый макроса называемый destructuring-bind. Этот макроса был кратко представлен в главе 7. Здесь мы рассмотрим его более подробно. Предположим, что lst является списком из трех элементов, и мы хотим связать x с первым элементом, y со вторым, и z с третьим. В незрелом CLTL1 Common Lisp, мы бы должны были сказать:

```
(let ((x (first lst))
        (y (second lst))
        (z (third lst)))
  ...)
```

С новым макросом весто этого мы можем сказать:

```
(destructuring-bind (x y z) lst
  ...)
```

что не только короче, но и понятнее. Читатели понимают визуальные подсказки быстрее, чем текстовые. В последнем примере нам сразу показывают связь между x, y и z; а в первом случае, мы должны делать вывод об этом.

Если такой простой случай проясняется с помощью деконструкции, представьте себе улучшение ясности более сложных. Первый аргумент в destructuring-bind может быть произвольно сложным деревом. Представьте

```
(destructuring-bind ((first last) (month day year) . notes)
                            birthday
  ...)
```

будет записано с использованием let и функций доступа к списку. Что поднимает еще один аргумент: деконструкция облегчает написание программ, а также облегчает их чтение.

Деконструкция существовала и в CLTL1 Common Lisp. Если шаблоны в примерах выше выглядят знакомо, это потому, что они имеют ту же форму, что и списки параметров макросов. Фактически, destructuring-bind это код, используемый для разбора списка аргументов макроса, теперь предоставляемый отдельно. Вы можете положить что нибудь в шабон, который вы бы представили в виде списка параметров макроса, за одним незначительным исключением(ключевого слова &environment).

Установление массовых привязок является привлекательной идеей. Следующие разделы описывают несколько вариаций на эту тему.

18.2 18-2 Другие Структуры

Нет причин ограничивать деконструкцию списками. Любой сложный объект является кандидатом для этого. В этом разделе показано, как писать макросы подобные destructuring-bind для других видов объектов.

Следующим естественным шагом является обработка в целом последовательностей. Рисунок 18-1 содержит макрос называемый dbind, который напоминает destructuring-bind, но обрабатывающий любой вид последовательностей. Второй аргумент может быть списком, вектором, или любой их комбинацией:

```
> (dbind (a b c) #(1 2 3)
        (list a b c))
(123)
> (dbind (a (b c) d) '( 1 #(2 3) 4)
        (list abcd))
(1234)
> (dbind (a (b . c) &rest d) '(1 "fribble" 2 3 4)
        (list abcd))
(1 #\f "ribble" (2 3 4))
```

```
(defmacro dbind (pat seq &body body)
      (let ((gseq (gensym)))
       `(let ((,gseq ,seq))
          ,(dbind-ex (destruc pat gseq #'atom) body))))

(defun destruc (pat seq &optional (atom? #'atom) (n 0))
      (if (null pat)
         nil
         (let ((rest (cond ((funcall atom? pat) pat)
                           ((eq (car pat) '&rest) (cadr pat))
                           ((eq (car pat) '&body) (cadr pat))
                           (t nil))))
          (if rest
                `((,rest (subseq ,seq ,n)))
                (let ((p (car pat))
                        (rec (destruc (cdr pat) seq atom? (1+ n))))
                 (if (funcall atom? p)
                        (cons `(,p (elt ,seq ,n))
                              rec)
                        (let ((var (gensym)))
                          (cons (cons `(,var (elt ,seq ,n))
                                      (destruc p var atom?))
                                rec))))))))

(defun dbind-ex (binds body)
      (if (null binds)
         `(progn ,@body)
         `(let ,(mapcar #'(lambda (b)
                           (if (consp (car b))
                                 (car b)
                                 b))
                        binds)
           ,(dbind-ex (mapcan #'(lambda (b)
                                  (if (consp (car b))
                                       (cdr b)))
                              binds)
                      body))))
```

Рисунок 18-1: Операции деконструкции обобщенной последовательности.

Макрос чтения #(для представления векторов, и #\ для представления символьных знаков. Поскольку "abc" = #(#\a #\b #\c), первым элементом "fribble" является знак #\f. Для простоты, dbind поддерживает только ключевые слова &rest и &body keywords.

По сравнению с большинством макросов, dbind большой. Стоит поучиться реализации данного макроса, не только для того чтобы понять как он работает, но и потому что он воплощает главный урок о Lisp программировании. Как уже упоминалось в разделе 3-4, программы Lisp могут быть написаны так, чтобы их легко было прове-

рять. В большинстве кода, мы должны сбалансировать это желание с необходимостью достижения высокой скорости. К счастью, как объяснил Раздел 7-8, скорость не так важна в коде расширителя. При написании кода, который генерирует расширение макроса, мы можем облегчить себе жизнь. Расширение dbind создает две функции, destruc и dbind-ex. Возможно, их обе объединить в одну функцию, которая сделала бы все за один проход. Но зачем? А две раздельные функции, легче тестировать. Зачем обменивать это преимущество на скорость, которая нам не нужна?

Первая функция destruc, обходит шаблон и связывает каждую переменную с расположением соответствующего объекта во время выполнения:

```
> (destruc '(a b c) 'seq #'atom)
((A (ELT SEQ 0)) (B (ELT SEQ 1)) (C (ELT SEQ 2)))
```

Необязательный третий аргумент - это предикат используемый для отделения структуры шаблона от содержимого шаблона.

Чтобы сделать доступ более эффективным, к каждой подпоследовательности будет привязана новая (gensym) переменная:

```
> (destruc '(a (b . c) &rest d) 'seq)
((A (ELT SEQ 0))
 ((#:G2 (ELT SEQ 1)) (B (ELT #:G2 0)) (C (SUBSEQ #:G2 1)))
 (D (SUBSEQ SEQ 2)))
```

Вывод destruc передается в dbind-ex, который генерирует большую часть расширения макроса. Он переводит дерево созданное destruc в ряд вложенных let:

```
> (dbind-ex (destruc '(a (b . c) &rest d) 'seq) '(body))
(LET ((A (ELT SEQ 0))
          (#:G4 (ELT SEQ 1))
          (D (SUBSEQ SEQ 2)))
  (LET ((B (ELT #:G4 0))
             (C (SUBSEQ #:G4 1)))
       (PROGN BODY)))
```

```
(defmacro with-matrix (pats ar &body body)
      (let ((gar (gensym)))
        `(let ((,gar ,ar))
            (let ,(let ((row -1))
                       (mapcan
                          #'(lambda (pat)
                               (incf row)
                               (setq col -1)
                               (mapcar #'(lambda (p)
                                           `(,p (aref ,gar
                                                      ,row
                                                      ,(incf col))))
                                       pat))
                          pats))
               ,@body))))

(defmacro with-array (pat ar &body body)
      (let ((gar (gensym)))
        `(let ((,gar ,ar))
            (let ,(mapcar #'(lambda (p)
                              `(,(car p) (aref ,gar ,@(cdr p))))
                          pat)
               ,@body))))
```

Рисунок 18-2: Деконструкция массивов.

Обратите внимание, что dbind, как и destructuring-bind, предполагает, что он найдет всю структуру списка, которую он просматирвает. Оставшиеся переменные не просто связываются с nil, а с использованием multiple-value-bind. Если последовательность, заданная во время выполнения, не имеет всех ожидаемых элементов, операторы деконструкции выдают ошибку:

```
> (dbind (a b c) (list 1 2))
>>Error: 2 is not a valid index for the sequence (1 2)
```

Какие еще объекты имеют внутреннюю структуру? Обычно есть массивы, которые отличаются от векторов, наличием более чем одной размерности. Если мы определеим макрос деконструкции для массивов, как мы представим шаблон? Для двумерных массивов, все еще будет практичным использовать список. Рисунок 18-2 содержит макрос with-matrix для деконструкции двумерных массивов.

```
(defmacro with-struct ((name . fields) struct &body body)
       (let ((gs (gensym)))
          `(let ((,gs ,struct))
              (let ,(mapcar #'(lambda (f)
                                `(,f (,(symb name f) ,gs)))
                            fields)
                ,@body))))
```

Рисунок 18-3: Деконструкция структур.

```
> (setq ar (make-array '(3 3)))
#<Simple-Array T (3 3) C2D39E>
> (for (r 0 2)
          (for (c 0 2)
             (setf (aref ar r c) (+ (* r 10) c))))
NIL
> (with-matrix ((a b c)
                          (def)
                          (g h i)) ar
          (list abcdefghi))
(012101112202122)
```

Для больших массивов или массивов размероностью 3 и выше, нам надо использовать другой подход. Мы вряд ли захотим связывать переменные с каждым элементом большого массива. Будет более практичным сделать шаблон с разреженным представлением массива, содержащим переменные только для нескольких элементов, а также координаты для их их идентификации. Второй макрос на Рисунке 18-2 создан по этому принципу. Здесь мы используем его, для того чтобы получить диагональ предыдущего массива:

```
> (with-array ((a 0 0) (d 1 1) (i 2 2)) ar
          (values a d i))
01122
```

С этим новым макросом мы начали отходить от шаблонов, элементы которых должны появляться в фиксированном порядке. Мы можем создать подобный макрос для привязки переменных к полякм в структурах созданных с помощью defstruct. Такой макрос определен на Рисунке 18-3. Первым аргументов в шаблоне будет префик, связанный со стуктурой, а остальные - имена полей. Для построения вызовов доступа этот макрос использует symb (стр. 58).

```
> (defstruct visitor name title firm)
VISITOR
> (setq theo (make-visitor :name "Theodebert"
                                       :title 'king
                                       :firm 'franks))
#S(VISITOR NAME "Theodebert" TITLE KING FIRM FRANKS)
> (with-struct (visitor- name firm title) theo
         (list name firm title))
("Theodebert" FRANKS KING)
```

18.3 18-3 Ссылки

CLOS приносит с собой макрос для деконструкции экземпляров. Предположим, что tree (дерево) это клас с тремя слотами: порода(species), возраст(age), и высота(height), и my-tree это экземпляр tree. В

```
(with-slots (species age height) my-tree
  ...)
```

мы можем ссылаться на слоты my-tree, как если бы они были обычными переменными. В пределах тела with-slots, символ height ссылается на слот height. Он не просто связан со значением хранящимся в нем, но он ссылается на этот слот, так что если мы напишем:

```
(setq height 72)
```

тогда слоту height в my-tree будет присвоено значение 72. Этот макрос работает как определение height как макрос-символ (Раздел 7-11) который расширяется до ссылки на слот. На самом деле, он был должен поддерживать макросы, такие как with-slots, symbol-macrolet был добавлен в Common Lisp.

Является или нет на самом деле with-slots макросом деконструкции, он играет ту же прагматичну роль, что и destructuring-bind. Так как обычная деконструкция это вызов по значению, это новый вид - вызов по имени. Как бы мы его не называли, он выглядит очень полезным. Какие еще макросы мы можем определить по этому принципу?

Мы можем создать вызов по имени для любого макроса деконструкции, расширяя его до symbol-macrolet, а не до let. На рисунке 18-4 показана версия dbind модифицированная для поведения, аналогичного with-slots. Мы можем использовать with-places также как dbind:

```
> (with-places (a b c) #(1 2 3)
        (list a b c))
(1 2 3)
```

```
(defmacro with-places (pat seq &body body)
   (let ((gseq (gensym)))
        `(let ((,gseq ,seq))
              ,(wplac-ex (destruc pat gseq #'atom) body))))

(defun wplac-ex (binds body)
   (if (null binds)
         `(progn ,@body)
         `(symbol-macrolet ,(mapcar #'(lambda (b)
                                        (if (consp (car b))
                                            (car b)
                                            b))
                                    binds)
                           ,(wplac-ex (mapcan #'(lambda (b)
                                                  (if (consp (car b))
                                                      (cdr b)))
                                              binds)
                                      body))))
```

Рисунок 18-4: Ссылочная деконструкция последовательностей.

Но новый макрос также дает нам возможность присваивать позиции в последовательностях с помощью setf, как мы это делали с помощью with-slots:

```
> (let ((lst '(1 (2 3) 4)))
        (with-places (a (b . c) d) lst
          (setf a 'uno)
          (setf c '(tre)))
        lst)
(UNO (2 TRE) 4)
```

Как и в случае с with-slots, переменные теперь ссылаются на соответствующие места в структуре. Однако есть одно важное отличие: вы должны использовать setf, а не setq, чтобы установить эти псевдо-переменные. Макрос with-slots должен вызывать code-walker (стр 273), чтобы преобразовать setq в setf внутри своего тела. Здесь, написание code-walker потребует много кода для небольшого уточнения.

Если with-places является более общим чем dbind, почему бы просто не использовать его постоянно? В то время как dbind связывает переменную со значением, with-places связывает ее с набором инструкций для поиска значения. Каждая ссылка требует поиска. Где dbind будет связвать символ с со значением (elt x 2), with-places сделает символ с символом макросом, который расширяется в (elt x 2). Таким образом, если с вычисляется n раз в теле, это повлечет за собой n вызвов elt. Если вы на самом деле не хотите присваивать(setf) переменные созданные деконструкцией, dbind будет работать быстрее.

Определение with-places немного отличается от определения dbind (Рисунок 18-1). Bn wplac-ex (ранее dbind-ex) let стала symbol-macrolet. С помощью аналогичных изменений, мы можем сделать версию вызова по имени для любого обычного деконструирующего макроса.

18.4 18-4 Сопоставления

Как деконструкция является обобщением присваивания, сопоставление с образцом является обобщением деконструкции. Термин "сопоставление с образцом" имеет много смыслов. В данном конетексте, он означает сравнение двух структур, возможно содержащих переменные, чтобы выяснить, существует ли какой либо способо присвоения значений переменным, который сделает их равным. Например, если ?x и ?y являются переменными, то два списка

```
(p ?x ?y  c ?x)
(p a   b  c  a)
```

соответсвуют, когда ?x = a и ?y = b. И списки

```
(p ?x b ?y a)
(p ?y b c a)
```

соответствуют, когда ?x = ?y = c.

Предположим, что программа работает путем обмена сообщениями с внешним источником. Чтобы ответить на сообщение, программе надо сообщить, какой тип имеет это сообщение, а также извлечь конкретное содержание. С оператором сопоставления мы можем объединить эти два шага.

Чтобы написать такой оператор, мы должны придумать способ различения переменных. Мы не можем просто сказать, что все символы являются переменными, потому что мы хотим, чтобы символы появлялись как аргументы в образцах. Здесь мы скажем, что переменная образца это символ начинающийся со знака вопроса. Если это станет не удобным, это соглашение можно изменить, просто определив предикат var?.

Рисунок 18-5 содержит функцию сопоставления с образцом, аналогичную той, которая приводилась во введении в Lisp. Мы даем match два списка, и если они могут быть сопоставлены, мы получаем назад список показывающий как это сделать:

```
> (match '(pabca)'(p?x?yc?x))
((?Y . B) (?X . A))
T
```

```
(defun match (x y &optional binds)
  (acond2
   ((or (eql x y) (eql x '_) (eql y '_)) (values binds t))
   ((binding x binds) (match it y binds))
   ((binding y binds) (match x it binds))
   ((varsym? x) (values (cons (cons x y) binds) t))
   ((varsym? y) (values (cons (cons y x) binds) t))
   ((and (consp x) (consp y) (match (car x) (car y) binds))
     (match (cdr x) (cdr y) it))
   (t (values nil nil))))

(defun varsym? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (labels ((recbind (x binds)
             (aif (assoc x binds)
                  (or (recbind (cdr it) binds)
                      it))))
    (let ((b (recbind x binds)))
      (values (cdr b) b))))
```

Рисунок 18-5: Функции Сопоставления.

```
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T> (match '(a b c) '(a a a))
NIL
NIL
```

Когда match сравнивает свои аргументы элемент за элементом, она создает присваивания значений переменным, называемые привязками, в в параметре binds. Если сопоставление проходит успешно, match возвращает сгенерированные привязки, в противном случае возвращается nil. Поскольку не все успешные совпадения генерируют какие-либо привязки, match как и gethash, возвращает второе значение, чтобы указать, был ли вызов match успешными или неудачным:

```
> (match '(p ?x) '(p ?x))
NIL
T
```

```
(defmacro if-match (pat seq then &optional else)
  `(aif2 (match ',pat ,seq)
         (let ,(mapcar #'(lambda (v)
                           `(,v (binding ',v it)))
                       (vars-in then #'atom))
           ,then)
         ,else))

(defun vars-in (expr &optional (atom? #'atom))
  (if (funcall atom? expr)
      (if (var? expr) (list expr))
      (union (vars-in (car expr) atom?)
             (vars-in (cdr expr) atom?))))

(defun var? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))
```

Рисунок 18-6: Медленные операторы сопоставления.

Когда match возвращает nil и t, как указано выше, это указывает на успешное совпадение, которое не дало никаких привязок.

Как и в Prolog, match рассматривает (подчеркивание) как произвольный знак. Он может соответствовать чему угодно, и не влияет на привязки:

```
> (match '(a ?x b) '(_ 1 _))
((?X . 1))
T
```

Учитывая match, легко написать версию dbind для сопоставления с образцом. На Рисунке 18-6 содержиться макрос с именем if-match. Как и dbind, его первые два аргуемнта это образец и последовательность, и он устанавливает привязки сравнивая образец с последовательностью. Однако, вместо тело у него есть еще два аргумента: предложение then вычисляемое с новыми привязками, если совпадение найдено; и предложение else вычисляемое в случае сбоя сопоставления. Вот простая функция, которая использует if-match:

```
(defun abab (seq)
     (if-match (?x ?y ?x ?y) seq
          (values ?x ?y)
          nil))
```

Если match выполняется успешно, оно устанавливает значения для ?x и ?y, которые будут возвращены:

```
> (abab '(hi ho hi ho))
HI
HO
```

Функция vars-in возвращает все переменные образца в выражении. Он вызывает var? чтобы проверить, является что-либо переменной. На данный момент var? является идентичным varsym? (Рисунок 18-5), который используется для обнаружения переменных в списках привязок. У нас есть две разные функции на случай если мы захотим использовать разные представления для двух типов переменых.

Определеное на Рисунке 18-6 if-match является коротки, но не очень эффективным. Оно делает слишком много работы во время выполнения. Мы проходим по двум последовательностям во время выполнения, хотя первая уже известна во время копиляции. Что еще хуже, в процессе сопоставления, мы создаем списки для хранения привязок переменных. Если мы воспользуемся информацией известной во время коомпиляции, мы сможем написать версию if-match, которая не будет выполнять не нужных сравнений и не имеет недостатков использования cons.

Если одна из последовательностей известна во время компиляции, и это та которая содержит переменные, тогда мы можем действовать по другому. В вызове match любой аргуемнт может содержать переменные. Ограничивая переменные только первым аргументом if-match, мы позволяем указать во время компиляции, какие переменные будут учавствовать в сопоставлении. Тогда вместо создания списков привязок переменных, мы могли бы хранить значения переменных в самих переменных.

Новая версия if-match показана на рисунках 18-7 и 18-8. Когда мы можем предсказать, какой код будет вычисляться во время выполнения, мы можем просто сгенерировать его во время компиляции. Здесь, вместо расширения в вызов match, мы генерируе код который выполняет только правильныеп сравнения.

Если мы собираемся использовать переменную ?x, чтобы содержать привязку ?x, как нам представить переменную, для которой совпадение еще не было установлено match? Здесь мы укажем, что переменная образца не связана, связывая её с gensym. Поэтому if-match начинается с генерации генерации кода, который будет связывать все переменные образца с gensyms. В этом случае, вместо того чтобы расширятсья в with-gensyms, безопасно создавать gensyms один раз во время компиляции и вставлять их непросредственно в расширение.

Остальная часть расширения генерируется с помощью pat-match. Этот макрос принимает те же аргументы что и if-match; единственное отличие состоит в том, что он не устанавливает новых привязок для переменных образца. В некоторых ситуациях это является преимуществом, и глава 19 будет использовать pat-match в качестве самостоятельного оператора.

В новом операторе сопоставления различие между содержимым и структурой образца будет определяться функцией simple?. Если мы хотим иметь возможность использовать квотированные литералы в образце, коду деконструкции (и vars-in) нужно запретить входить внутрь списков, первым элементом которых является quote. С новым оперетором сопоставления, мы можем использовать списки в качестве элементов образца, просто заключая их в символ квотирования.

```
(defmacro if-match (pat seq then &optional else)
      `(let ,(mapcar #'(lambda (v) `(,v ',(gensym)))
                     (vars-in pat #'simple?))
         (pat-match ,pat ,seq ,then ,else)))

(defmacro pat-match (pat seq then else)
      (if (simple? pat)
           (match1 `((,pat ,seq)) then else)
           (with-gensyms (gseq gelse)
             `(labels ((,gelse () ,else))
                    ,(gen-match (cons (list gseq seq)
                                      (destruc pat gseq #'simple?))
                                then
                                `(,gelse))))))

(defun simple? (x) (or (atom x) (eq (car x) 'quote)))

(defun gen-match (refs then else)
      (if (null refs)
           then
           (let ((then (gen-match (cdr refs) then else)))
             (if (simple? (caar refs))
                    (match1 refs then else)
                    (gen-match (car refs) then else)))))
```

Рисунок 18-7: Опертор быстрого сопоставления.

Как и dbind, pat-match вызывает destruc чтобы получить список вызвов, которые будут испольовать часть его аргументов во время выполнения. Этот список передается в gen-match, который рекурсивно генерирует код сопоставления для вложенных образцов, а затем в match1, который генерирует код сопоставления для каждого листа дерева образца.

Большая часть кода, который появится в расширении if-match, происходит из match1, что показано на рисунке 18-8. Эта функция рассматривает четыре случая. Если аргумент образца является gensym, то это одна из невидимых переменных, созданных destruc для хранения подсписков и все, что нам нужно сделать во время выполнения, это проверить, что он имеет правильную длину. Если элемент образца является произвольным знаком (_), генерировать код не нужно. Если элемент образца является переменной, match1 генерирует код, чтобы сопоставить его снова, или установить его соответствие с соответствующей частью последовательности, заданной во время выполнения. В проитвном случае элемент образца считается буквенным значением и match1 генерирует код для сравнения его с соответствующей частью последовательности.

```
(defun match1 (refs then else)
   (dbind ((pat expr) . rest) refs
        (cond ((gensym? pat)
                `(let ((,pat ,expr))
                     (if (and (typep ,pat 'sequence)
                              ,(length-test pat rest))
                         ,then
                         ,else)))
              ((eq pat '_) then)
              ((var? pat)
               (let ((ge (gensym)))
                   `(let ((,ge ,expr))
                      (if (or (gensym? ,pat) (equal ,pat ,ge))
                            (let ((,pat ,ge)) ,then)
                            ,else))))
              (t `(if (equal ,pat ,expr) ,then ,else)))))

(defun gensym? (s)
   (and (symbolp s) (not (symbol-package s))))

(defun length-test (pat rest)
   (let ((fin (caadar (last rest))))
        (if (or (consp fin) (eq fin 'elt))
              `(= (length ,pat) ,(length rest))
              `(> (length ,pat) ,(- (length rest) 2)))))
```

Рисунок 18-8: Быстрый оператор сопоставления (продолжение).

Давайте посмотрим на примеры того, как генерируются некоторые части расширения. Предположим, мы начали с

```
(if-match (?x 'a) seq
        (print ?x)
        nil)
```

Образец будет передан в destruc, с некоторыме gensym (назовем его g для разборчивости) в представленную последовательность:

```
(destruc '(?x 'a) 'g #'simple?)
```

получаем:

```
((?x (elt g 0)) ((quote a) (elt g 1)))
```

В начало этого списка мы подставляем (g seq):

```
((g seq) (?x (elt g 0)) ((quote a) (elt g 1)))
```

и отправляем его в gen-match. Как и в наивной реализации length (стр 22), gen-match сначала рекурсивно проходит до конца списка, а затем строит свое возвращаемое значение на обратном пути. Когда у него заканчиваются элементы, gen-match возвращает его аргумент then, который будет ?x. На обратном пути рекурсии это возвращаемое значение будет передано в качестве аргумент в match1.Теперь мы можем вызвать как:

```
(match1 '(((quote a) (elt g 1))) '(print ?x) ' else function )
```

получая:

```
(if (equal (quote a) (elt g 1))
        (print ?x)
         else function )
```

Это в свою очередь, станет аргументом then для друго вызова match1, значение которого станет then аргументом для последнего вызова match1. Полное расширение этого if-match показано на рисунке 18-9.

В этом расширении gensyms используются двумя совершенно не связанными способами. Переменные, используемые для хранения частей дерева во время выполнения имеют сгенерированные gensym имена, чтобы избежать захвата. И переменные ?x изначально связвается с gensym, чтобы указать, что ей не было присвоено значение путем сопоставления.

В новом if-match, элементы образца теперь вычисляются, а не заключются неявно в кавычки. Это означает, что Lisp переменные могут использоваться в образцах, а также в выражениях в кавычках:

```
> (let ((n 3))
        (if-match (?x n 'n '(a b)) '(13n(ab))
           ?x))
1
```

Два новых улучшения появляютс потому, что новая версися вызывает destruc (Рисунок 18-1). Теперь образец может содержать ключевые слова &rest или &body (match не беспокоит их). И поскольку destruc использует опрераторы общей последовательности elt и subseq, новый if-match будет работать для любого типа последовательностей. Если abab определен с новой версией, он может быть использован также для векторов и строк:

```
(if-match (?x 'a) seq
     (print ?x))
```

расширяется в:

```
(let ((?x '#:g1))
  (labels ((#:g3 nil nil))
     (let ((#:g2 seq))
       (if (and (typep #:g2 'sequence)
                   (= (length #:g2) 2))
            (let ((#:g5 (elt #:g2 0)))
                 (if (or (gensym? x) (equal ?x #:g5))
                    (let ((?x #:g5))
                        (if (equal 'a (elt #:g2 1))
                             (print ?x)
                             (#:g3)))
                    (#:g3)))
            (#:g3)))))
```

Рисунок 18-9: Расширение if-match.

```
> (abab "abab")
```

```
#\a
#\b
> (abab #(1 2 1 2))
12
```

На самом деле, образцы могут быть сложными, как образцы для dbind:

```
> (if-match (?x (1 . ?y) . ?x) '((a b) #(1 2 3) a b)
            (values ?x ?y))
(A B)
#(2 3)
```

Обратите внимание, что во втором возвращаемом значении, отображаются элементы вектора. Чтобы векторы печатались таким образом, установите *print-array* в t.

В этой главе мы начали переходить грань в новый вид программирования. Мы начали с простых макросв для деконструкции. И в финальной версии из if-match у нас получилось что то больше похожее на собственный язык. Остальные главы описывают целый класс программ, которые работают на той же философии.

19 19 Компилятор Запросов

Некоторые из макросов определенных в предыдущей главе, были большими. Чтобы сгенерировать рассширение if-match потребовался весь код на Рисунках 18-7 и 18-8, плюс destruc из Рисунка 18-1. Макросы такого размера естественным образом ведут к нашей последней теме - встроенным языкам. Если небольшие макросы являются расширениями Lisp, большие макросы определяют внутриязыковые подъязыки, возможно со своим собственным синтаксисом или структурой управления. Мы увидели начало этого в if-match, который имел свое собственное отдельное представление для переменных.

Язык, реализованный внутри Lisp называется встроенным языком. Как и "полезность(utility)", этот термин не является четко определенным; if-match вероятно, все еще считается утилитой, но приближается к границе.

Встроенный язык не похож на язык, реализованный традиционным транслятором или интерпретатором. Он реализуется в рамках существующего языка, обычно путем преобразования. Не должно былть никакого барьера между базовым языком и расширением: должна быть возможность, свободного их смешивания. Для разработчкика это может означать огромную экономию усилий. Вы можете встраивать только то, что вам нужно, а в остальном использовать базовый язык.

Преобразование в Lisp, предполагает макросы. В некоторой степени вы можете реализовать встроенный языки с препроцессорами. Но препроцессоры обычно работают только с текстом, в то время как макросы используют уникальное свойство Lisp: между читателем(reader) и компилятором, ваша программа Lisp представляется в виде списков объектов Lisp. Преобразования, сделанные на этом этапе, могут быть намного умнее.

Наиболее известным примером встроенного языка является CLOS - Common Lisp Object System. Если вы хотите создать объектно-ориентированную версию обычного языка, вам придется написать новый компилятор. В Lisp это не так. Настройка компилятора заставит CLOS работать быстрее, но в принципе компилятор менять вообще не нужно. Все это можно написать и на Lisp.

В остальных главах приведены примеры встроенных языков. В этой главе описывается, как встроить в Lisp программу отвечающую на запросы к базе данных. (В этой программе вы заметите семейное сходствов с if-match.) Первые разделы описывают, как написать систему, которая интерпретирует запросы. Затем эта программа реализуется как компилятор запросов - по сути, как один большой макрос - что делает её более эффективной и лучше интегрированной с Lisp.

19.1 19-1 База Данных

Для наших нынешних целей формат базы данных не имеет большого значения. Здесь для удобства мы будем хранить информацию в списках. Например, мы представим тот факт, что Joshua Reynolds был английским художником, который жил с 1723 по 1792:

```
(painter reynolds joshua english)
(dates reynolds 1723 1792)
```

Не существует канонического(общепризнанного) способа сведения информации в списки. Мы также могли бы использовать один большой список:

```
(painter reynolds joshua 1723 1792 english)
```

Пользователь должен решить, как организовать записи в базе данных. Единственным ограничением является то, что записи(факты) будут проиндексированы в соответствии с их первым элементом (передикатом). В этих рамки подойдет любая совместимая форма, хотя некоторые формы могут выполнять запросы быстрее, чем другие.

Любой системе баз данных требуются как минимум две операции: одна для изменения базы данных и одна для её проверки. Код показанный на Рисунке 19-1 передоставляет эти операции в базовой форме. База данных представлена в виде хеш-таблицы, заполненной списками фактов, хешированных в соответствии с их пердикатом(первым элементом).

Хотя функции базы данных, определенные на Рисунке 19-1 поддерживают несколько баз данных, все они по умолчанию работают с *default-db*. Как и в случае с пакетами в Common Lisp, программы, которым не требуется несколько баз данных, даже не должны упоминать их. В этой главе во всех примерах будет использоваться *default-db*.

Мы инициализируем систему, вызывая clear-db, который очищает текущую базу данных. Мы можем искать факты с заданным предикатом с помощью db-query, и вставлять новые факты в базу данных с помощью db-push. Как объяснено в Разделе 12-1, макрос который расширяется в обратимую ссылку, сам по себе будет обратимым. Так как db-query определен таким же образом, мы можем просто вставить новые факты в db-query в виде его предикатов. В Common Lisp, записи хеш-таблицы инициализируются как nil

```
(defun make-db (&optional (size 100))
  (make-hash-table :size size))

(defvar *default-db* (make-db))

(defun clear-db (&optional (db *default-db*))
  (clrhash db))

(defmacro db-query (key &optional (db '*default-db*))
  `(gethash ,key ,db))

(defun db-push (key val &optional (db *default-db*))
  (push val (db-query key db)))

(defmacro fact (pred &rest args)
  `(progn (db-push ',pred ',args)
          ',args))
```

Рисунок 19-1: Базовые функции работы с базой данных.

если не указано иное, поэтому любой ключ изначально имеет пустой список, связанный с ним. Наконец, макрос fact добавляет новый факт в базу данных.

```
> (fact painter reynolds joshua english)
(REYNOLDS JOSHUA ENGLISH)
> (fact painter canale antonio venetian)
(CANALE ANTONIO VENETIAN)
> (db-query 'painter)
((CANALE ANTONIO VENETIAN)
     (REYNOLDS JOSHUA ENGLISH))
T
```

Значение t возвращаемое как второе значение db-query появляется потому, что db-query расширяется в gethash, который возвращает в качестве второго значения флаг, позволяющий различать, не найде на ли запись, или найдена запись, значение которой равно nil.

19.2 19-2 Запросы Сопоставление с Образцом(Pattern-Matching Queries)

Вызов db-query не очень гибкий способ просмотра содержимого базы данных. Обычно пользователь хочет задать вопросы, которые зависят не только от первого элемента факта. Язык запросов - это язык для выражения

```
query         : ( symbol  argument *)
              : (not  query )
              : (and  query *)
              : (or  query *)
argument  : ? symbol
              :  symbol
              :  number
```

Рисунок 19-2: Синтаксис запросов.

более сложных запросов. В типичном языке запросов пользователь может запросить все значения, которые удовлетворяют некоторой комбинации ограничений, например, фамилии всех художников родившихся в 1697.

Наша программа предоставляет декларативный язык запросов. В декларативном языке запросов пользователь указывает ограничения, которым должны удовлетворять ответы и предоставляет системе возможность выяснить, как их сгенерировать. Этот способ выражения запросов близок к форме, которую люди используют в повседневной беседе. С нашей программой мы можем выразить пример запроса, запросив все x, то есть факты формы (painter x ...), и факты формы (dates x 1697 ...). Мы можем сослаться на всех художников(painters) родившихся в 1697 году записав:

```
(and (painter ?x ?y ?z)
       (dates ?x 1697 ?w))
```

Помимо принятия простых запросов, состоящих из предиката и некоторых аргументов, наша программа сможет отвечать на произвольно сложные запросы, объеди-

ненные логическими операторами, такими как and и or. Синтаксис языка запросов показан на рисунке 19-2.

Поскольку факты индексируются по их предикатам, переменные не могут появлятья в позиции предиката. Если вы готовы отказаться от преимуществ индексирования, вы можете обойти это ограничение, всегда используя один и тот же предикат и делая первый аргумент предикатом де-факто.

Как и в большинстве подобных систем, в этой программе скептическое представление об истинности: некоторые факты известны, а все остальное ложно. Оператор not завершается успехом, если рассматриваемый факт отсутствует в базе данных. В некоторой степени вы могли бы представить явную ложь методом Мира Уэйна:

```
(edible motor-oil not)
```

Однако оператор not не будет относиться к этим фактам иначе, чем другие.

В языках программирования существует принципиальное различие между интерпретируемыми и скомпилированными программами. В этой главе мы рассмотрим тот же вопрос в отношении запросов. Интеретатор запросов принимает запрос и использует его для генерации ответов из базы данных. Компилятор запросов принимаетт запрос и генерирует программу, которая при запуске дает тот же результат. В следующих разделах описывается интерпретатор запросов, а затем компилятор запросов.

19.3 19-3 Интерпретатор Запросов

Для реализации декларативного языка запросов мы будем использовать утилиты сопоставления с образцом, определенные в разделе 18-4. Функции, показанные на рисунке 19-3 интерпретируют запросы представленные в форме показанной на рисунке 19-2. Центральной функцией в этом коде является interpret-query, которая рекурсивно работает проходя через структуру сложного запроса, генерируя в процессе привязки. Вычисление сложных запросов происходит слева на право, как и в самом Common Lisp.

Когда рекурсия опускается вниз к образцам для фактов, interpret-query вызывает lookup. Здесь происходит сопоставление с образцом. Функция lookup принимает образец состоящий из предиката и списка аргументов, и возвращает список всех привязок, которые делают соответствующим образец некоторым фактам в базе данных. Он получает все записи базы данных для предиката и вызывает match(сопоставление) (стр 239), чтобы сравнить каждую из них с образцом. Каждое успешное сравнение возвращает список привязок, а lookup, в свою очередь,i возвращает список всех этих списков.

```
> (lookup 'painter '(?x ?y english))
(((?Y . JOSHUA) (?X . REYNOLDS)))
```

Эти результаты затем фильтруются или объединяются в зависимости от окружающих логических операторов. Окончательный результат возвращается в виде списка наборов привязок. Учитывая утверждения показанные на Рисунке 19-4, вот пример интерпретации запроса указанного ранее в этой главе:

```
> (interpret-query '(and (painter ?x ?y ?z)
                         (dates ?x 1697 ?w)))
(((?W . 1768) (?Z . VENETIAN) (?Y . ANTONIO) (?X . CANALE))
```

```
((?W . 1772) (?Z . ENGLISH) (?Y . WILLIAM) (?X . HOGARTH)))
```

Как правило, запросы могут быть объединены и вложены без ограничений. В некоторых случаях существуют тонкие ограничения на синтаксис запросов, но лучше всего с ними разобраться посмотрев некотоыре примеры использования этого кода.

Макрос with-answer обеспечивает понятный способ использования интерпретатора запросов в программах Lisp. В качестве первого аргумента он принимает любой правильно сформированный запрос; остальные аргументы обрабатываются как тело кода. Макрос with-answer расширяется в

```
(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
       `(dolist (,binds (interpret-query ',query))
             (let ,(mapcar #'(lambda (v)
                               `(,v (binding ',v ,binds)))
                           (vars-in query #'atom))
               ,@body))))

(defun interpret-query (expr &optional binds)
  (case (car expr)
        (and (interpret-and (reverse (cdr expr)) binds))
        (or      (interpret-or (cdr expr) binds))
        (not (interpret-not (cadr expr) binds))
        (t       (lookup (car expr) (cdr expr) binds))))

(defun interpret-and (clauses binds)
  (if (null clauses)
       (list binds)
       (mapcan #'(lambda (b)
                   (interpret-query (car clauses) b))
                (interpret-and (cdr clauses) binds))))

(defun interpret-or (clauses binds)
  (mapcan #'(lambda (c)
                (interpret-query c binds))
              clauses))

(defun interpret-not (clause binds)
  (if (interpret-query clause binds)
       nil
       (list binds)))

(defun lookup (pred args &optional binds)
  (mapcan #'(lambda (x)
                (aif2 (match x args binds) (list it)))
              (db-query pred)))
```

Рисунок 19-3: Интерпретатор Запросов.

```
(clear-db)
(fact painter hogarth william english)
(fact painter canale antonio venetian)
(fact painter reynolds joshua english)
(fact dates hogarth 1697 1772)
(fact dates canale 1697 1768)
(fact dates reynolds 1723 1792)
```

Рисунок 19-4: Утверждение фактов для примера.

код, который собирает все наборы привязок, сгенерированных запросом, затем итерирует тело выражений с переменными связанными в запросе, которые определены в каждом наборе привязок. Переменные, которые появляются в with-answer могут (обычно) использоваться внутри его тела. Когда запрос успешен, но не содержит переменных with-answer исполняет тело кода только один раз.

С базой данных, определенной на рисунке 19-4, на риснуке 19-5 показаны некоторые примеры запросов, сопровождаемые переводами на английский язык. Поскольку сопоставление с образцом выполняется с помощью match, можно использовать подчеркивание как безразличное поле в образце.

Чтобы эти примеры были короткими, код внутри тел запросов не делает ничего, кроме печати нескольких результатов. В общем, тело with-answer может состоять из любых выражений Lisp.

19.4 19-4 Ограничения на связывание

Существуют некоторые ограничения на то, какие переменные будут связаны запросом. Например, почему запрос

```
(not (painter ?x ?y ?z))
```

должен назначать какие либо привязки ?x и ?y вообще? Существует бесконечное количество комбинаций ?x и ?y, которые не являются именем какого либо художника(painter). Таким образом, мы добавляем следующее ограничение: оператор not будет отфильтровывать привязки, котоыре уже созданы, как в

```
(and (painter ?x ?y ?z) (not (dates ?x 1772 ?d)))
```

но вы не можете ожидать, что он будет генерировать привязки сам по себе. Мы должны создать наборы привязок, при поиске художников(painters), прежде чем мы сможем отобрать тех, кто не родился в 1772. Если бы мы поместили предложения в обратном порядке:

```
(and (not (dates ?x 1772 ?d)) (painter ?x ?y ?z))                    ; wrong
```

Первое имя(имя) и национальность каждого художника называемого Hogarth.

```
> (with-answer (painter hogarth ?x ?y)
        (princ (list ?x ?y)))
(WILLIAM ENGLISH)
NIL
```

Последнее имя(фамилия) каждого художника родившегося в 1697. (Наш оригинальный пример.)

```
> (with-answer (and (painter ?x _ _)
                            (dates ?x 1697 _))
        (princ (list ?x)))
(CANALE)(HOGARTH)
NIL
```

Последнее имя(фамилия) и год рождения любого кто умер в 1772 или 1792.

```
> (with-answer (or (dates ?x ?y 1772)
                           (dates ?x ?y 1792))
        (princ (list ?x ?y)))
(HOGARTH 1697)(REYNOLDS 1723)
NIL
```

Последнее имя(фамилия) каждого английского художника, который родился не в тот же год, что и Venetian one.

```
> (with-answer (and (painter ?x _ english)
                            (dates ?x ?b _)
                            (not (and (painter ?x2 _ venetian)
                                      (dates ?x2 ?b _))))
        (princ ?x))
REYNOLDS
NIL
```

Рисунок 19-5: Использование интерпретатора запросов.

тогда мы получим nil в качестве результата, если в 1772 году рождались художники. Даже в первом примере, мы не должны ожидать, что сможем использовать значение ?d в теле выражения with-answer.

Кроме того, выражение вида (or q1 ... qn) гарантировано генерирует реальные привязки только для переменных, которые встречаются во всех qi. Если with-answer содержал запрос

```
(or (painter ?x ?y ?z) (dates ?x ?b ?d))
```

можно ожидать использования привязки ?x, потому что не зависимо от того, какой из подзапросов будет успешным, он сгенерирует привязку для ?x. Но ни ?y, ни ?b не гарантировано получение привязки из запроса, хотя тот или другой её получат. Переменные образца не связанные с запросом, будут равны nil для этой итерации.

19.5 19-5 Компилятор Запросов

Код на рисунке 19-3 делает то что мы хотим, но неэффективно. Он анализирует структуру запроса во время выполнения, хотя она известна уже во время компиляции. И он

содержит списки для хранения привязок переменных, когда мы можем использовать переменные для хранения своих собственных значений. Обе эти проблемы могут быть решены путем определения with-answer другим способом.

Рисунок 19-6 оеределяет новую версию with-answer. Новая версия продолжает тенденцию, которая началась с avg (стр. 182), и продолжилась в if-match (стр. 242): она выполняет во время компиляции большую часть работы, которую старая версия выполняла во время выполнения. Код на рисунке 19-6 имеет внешнее сходство с кодом на рисунке 19-3, но ни одна из этих функций не вызывается во время выполнения. Вместо того, чтобы генерировать привязки, они генерируют код, который становиться частью расширения with-answer. Во время выполнения этот код будет генерировать все привязки, которые удовлетворяют запросу в соответствии с текущим состоянием базы данных.

По сути, эта программа является одним большим макросом. На рисунке 19-7 показано расширение макроса with-answer. Большая часть работы выполняется с помощью pat-match (стр. 242), который сам является макросом. Теперь единственными новыми функциями, необходимыми во время выполнения, являются функции базы данных, показанные на Рисунке 19-1.

Когда with-answer вызвается на верхнем уровне, компиляция запросов имеет мало преимуществ. Код, представляющий запрос, генерируется, выполняется, а затем отбрасывается. Но когда в программе на Lisp появляется выражение with-answer, код, представляющий запрос, становиться частью расширения этого макроса. Поэтому, когда содержащая его программа компилируется, код для всех запросов будет скомпилирован и встроен в процесс.

Хотя основным преимуществом нового подхода является скорость, он также позволяет лучше интегрировать выражения with-answer в код, в котором они появляются. Здесь показаны два конкретных улучшения. Во-первых, теперь вычисляются аргументы в запросе, поэтому мы можем сказать:

```
> (setq my-favorite-year 1723)
1723
> (with-answer (dates ?x my-favorite-year ?d)
        (format t "~A was born in my favorite year.~%" ?x))
REYNOLDS was born in my favorite year.
NIL
```

```
(defmacro with-answer (query &body body)
  `(with-gensyms ,(vars-in query #'simple?)
        ,(compile-query query `(progn ,@body))))

(defun compile-query (q body)
  (case (car q)
        (and (compile-and (cdr q) body))
        (or     (compile-or (cdr q) body))
        (not (compile-not (cadr q) body))
        (lisp `(if ,(cadr q) ,body))
        (t      (compile-simple q body))))

(defun compile-simple (q body)
  (let ((fact (gensym)))
        `(dolist (,fact (db-query ',(car q)))
              (pat-match ,(cdr q) ,fact ,body nil))))

(defun compile-and (clauses body)
  (if (null clauses)
        body
        (compile-query (car clauses)
                               (compile-and (cdr clauses) body))))

(defun compile-or (clauses body)
  (if (null clauses)
        nil
        (let ((gbod (gensym))
                 (vars (vars-in body #'simple?)))
              `(labels ((,gbod ,vars ,body))
                 ,@(mapcar #'(lambda (cl)
                                    (compile-query cl `(,gbod ,@vars)))
                               clauses)))))

(defun compile-not (q body)
  (let ((tag (gensym)))
        `(if (block ,tag
                 ,(compile-query q `(return-from ,tag nil))
                 t)
              ,body)))
```

Рисунок 19-6: Компилятор Запросов.

```
(with-answer (painter ?x ?y ?z)
      (format t "~A ~A is a painter.~%" ?y ?x))

расширяется интерпретатором запросов в:

(dolist (#:g1 (interpret-query '(painter ?x ?y ?z)))
      (let ((?x (binding '?x #:g1))
            (?y (binding '?y #:g1))
            (?z (binding '?z #:g1)))
        (format t "~A ~A is a painter.~%" ?y ?x)))

и компилятором запросов в:

(with-gensyms (?x ?y ?z)
      (dolist (#:g1 (db-query 'painter))
        (pat-match (?x ?y ?z) #:g1
               (progn
                  (format t "~A ~A is a painter.~%" ?y ?x))
               nil)))
```

Рисунок 19-7: Два расширения одного и того же запроса.

Это можно было бы сделать и в интерпретаторе запросов, но только за счет явного вызова eval. И даже тогда не было бы возможности ссылаться на лексические переменные в аргументах запроса.

Посокльку аргументы в запросах теперь вычисляются, любой буквенный аргумент (например english), который не вычисляет сам себя, теперь должен заключаться в кавычки. (См. Рисунок 19-8.)

Второе преимущество нового подхода состоит в том, что теперь гораздо проще включать нормальные выражения Lisp в запросы. Компилятор запросов добавляет оператор lisp, за которым может следовать любое выражение Lisp. Как и оператор not, он не может генерировать привязки сам по себе, но отсеивает привязки, для которых выражение возвращает nil. Оператор lisp полезен для получения встроенных предикатов, таких как >:

```
> (with-answer (and (dates ?x ?b ?d)
                            (lisp (> (- ?d ?b) 70)))
        (format t "~A lived over 70 years.~%" ?x))
CANALE lived over 70 years.
HOGARTH lived over 70 years.
NIL
```

Хорошо реализованный встроенный язык может иметь цельный интерфейс с базовым языком с обеих сторон.

The first name and nationality of every painter called Hogarth.

```
> (with-answer (painter 'hogarth ?x ?y)
       (princ (list ?x ?y)))
(WILLIAM ENGLISH)
NIL
```

The last name of every English painter not born in the same year as a Venetian painter.

```
> (with-answer (and (painter ?x _ 'english)
                           (dates ?x ?b _)
                           (not (and (painter ?x2 _ 'venetian)
                                     (dates ?x2 ?b _))))
       (princ ?x))
REYNOLDS
NIL
```

The last name and year of death of every painter who died between 1770 and 1800 exclusive.

```
> (with-answer (and (painter ?x _ _)
                           (dates ?x _ ?d)
                           (lisp (< 1770 ?d 1800)))
       (princ (list ?x ?d)))
(REYNOLDS 1792)(HOGARTH 1772)
NIL
```

Рисунок 19-8: Использование компилятора запросов.

Помимо этих двух дополнений - вычисления аргументов и нового оператора lisp - язык запросов, поддерживаемый компилятором запросов, идентичен языку, поддерживаемому интерпретатором. На рисунке 19-8 показаны примеры результатов, полученных кодом сгенерированным компилятором запросов с базой данных определенной на рисунке 19-4.

В разделе 17-2 приведены две причины, по которым лучше скомпилировать выражение, чем передаавать его в виде списка, для выполнения. Первая из них, быстрота и возможность вычислять выражение в окружающем лексическом контексте. Преимущества составления запросов в точности аналогичны. Работа, которая раньше выполнялась во время выполнения, теперь выполняется во время компиляции. И поскольку запросы компилируются как часть окружающего Lisp кода, они могут использовать преимущества окружающего лексического контекста.

20 20 Продолжения

Продолжение - это программа, замороженная в действии: единый/отдельный функциональный объект содержащий состояние вычисления. Когда объект вычисляется, сохраненное вычисление перезапускается с того места, где оно было остановлено. При решении определенных типов проблем может быть полезно сохранить состояние программы и перезапустить её позже. Например в многопроцессной обработке продолжение представляет собой приостановленный процесс. В недетерминированных программах поиска продолжение может представлять собой узел в дереве поиска.

Продолжения могут быть трудными для понимания. Эта глава подходит к теме в два этапа. В первой части главы рассматривается использование продолжений в Scheme, в которую встроена их поддержка. После объяснения поведения продолжений, во второй части показано, как использовать макросы для создания продолжений в программах Common Lisp. Главы 21-24 используют макросы определенные здесь.

20.1 20-1 Продолжения Scheme

Одним из основных отличий Scheme от Common Lisp является явная поддержка продлжений. В этом разделе показано, как работают продолжения в Scheme. (На Рисунке 20-1 перечислены некоторые другие различия между Scheme и Common Lisp.)

Продолжение это функция, представляющая будущее вычислений. Всякий раз, когда выражение вычисляется, что-то(другое выражение) ждет значения, которое оно вернет.

1. Scheme не делает различий между тем, что Common Lisp называет символ-значение и символ-функция символа. В Scheme, переменная имеет единственное значение, которое может быть либо функцией или каким-либо другим видом объекта. Таким образом, в Scheme нет необходимости использовать решётку с кавычкой(#') или funcall. Common Lisp:

   ```
   (let ((f #'(lambda (x) (1+ x))))
     (funcall f 2))
   ```

   будет на Scheme:

   ```
   (let ((f (lambda (x) (1+ x))))
     (f 2))
   ```

2. Поскольку у Scheme есть только одно пространство имен, ей не нужны отдельные операторы (например defun и setq) для присваивания значений в случае функции или переменной. Вместо этого она имеет определение, которое примерно эквивалентно defvar, и set! - которое заменяет setq. Глобальные переменные должны быть созданы с помощью define, прежде чем они могут быть установлены с помощью set!.

3. В Scheme, именованные функции обычно определяются с помощью define, который заменяет defun и defvar. Common Lisp:

   ```
   (defun foo (x) (1+ x))
   ```

   Scheme имеет два возможных варианта:

   ```
   (define foo (lambda (x) (1+ x)))
   (define (foo x) (1+ x))
   ```

4. В Common Lisp, аргументы функции вычисляются слева на право. В Scheme, порядок вычисления намеренно не указан. (И разработчики радуются удивлению тех, кто про это забывает.)

5. Вместо t и nil, в Scheme есть #t и #f. Пустой списокt, (), в некоторых реализациях имеет значение истина, а в других ложь.

6. Предложение по умолчанию в выражениях cond и case имеет ключ else в Scheme, а не t как в Common Lisp.

7. Несколько встроенных операторов имеют разные имена: consp это pair?, null это null?, mapcar это (почти) map, и так далее. Обычно это должно быть очевидно из контекста.

Рисунок 20-1: Несколько отличий между Scheme и Common Lisp.

Например, в

```
(/ (- x 1) 2)
```

когда (- x 1) вычисляется, внешнее выражение / ожидает его значения, а что-то еще ожидает значения выражения / и так далее, и так далее, вплоть до верхнего уровня- где ожидает print(в цикле REPL).

Мы можем думать о продолжении в любой момент времени как о фунции одного аргумента. Если предыдущее выражение было введено на верхнем уровне, то при вычислении подвыражения (- x 1), продолжение будет:

```
(lambda (val) (/ val 2))
```

То есть, оставшаяся часть вычисления может быть продублирована путем вызова этой функции для возвращаемого значения. Если вместо выражения встречающегося в следующем контексте

```
(define (f1 w)
  (let ((y (f2 w)))
        (if (integer? y) (list 'a y) 'b)))

(define (f2 x)
  (/ (- x 1) 2))
```

и f1 был вызван из верхнего уровня, тогда когда (- x 1) был вычислен, продолжение будет эквивалентно

```
(lambda (val)
  (let ((y (/ val 2)))
        (if (integer? y) (list 'a y) 'b)))
```

В Scheme, продолжения яляются объектами первого класса, как и функции. Вы можете запросить у Scheme текущее продолжение, и она создаст вам функцию от одного аргумента, представляющую будущие вычисления. Вы можете хранить этот объект так долго, как захотите, и при вызове он перезапустит вычисление, которое имело место при его создании.

Продолжение можно понимать как обобщение замыканий. Замыкание это функция плюс указатели на лексические переменные, видимые во время его создания. Продолжение является функцией плсю указатель на весь стек, ожидающий ее в момент создания. Когда продолжение вычисляется, оно возвращает значение, используя собственную копию стека, игнорируя текущий стек. Если продолжение создано в момент T1 и вычисляется в момент T2, оно будет вычисляться с помощью стека, ожидающего в момент T1.

Программы Scheme имеют доступ к текущему продолжению через встроенный оператор call-with-current-continuation (call/cc для краткости). Когда программа вызывает call/cc для функции с одним аргументом:

```
(call-with-current-continuation
  (lambda (cc)
        ...))
```

функции будет передана другая функция, представляющая текущее продолжение. Сохраняя где-то значение cc, мы сохраняем состояние вычисления в точке call/cc.

В этом примере, мы добавляем список, последним элементом которого является значение, возвращаемое выражением call/cc:

```
> (define frozen)
FROZEN
> (append '(the call/cc returned)
                (list (call-with-current-continuation
                        (lambda (cc)
                          (set! frozen cc)
                          'a))))
(THE CALL/CC RETURNED A)
```

call/cc возвращает символ a, но сначала сохраняет продолжение в глобальной переменной frozen.

Вызов frozen перезапустит старое вычисление в точке call/cc. Какое бы значение мы не передали frozen оно будет возвращено как значение call/cc:

```
> (frozen 'again)
(THE CALL/CC RETURNED AGAIN)
```

Продолжения не исчерпываются(заканчиваютс) вычислением. Их можно вызывать повторно, как и любой другой функциональный объект:

```
> (frozen 'thrice)
(THE CALL/CC RETURNED THRICE)
```

Когда мы вызываем продолжение в каком-то другом вычислении, мы более ясно видим, что значит вернуть обратно старый стек:

```
> (+ 1 (frozen 'safely))
(THE CALL/CC RETURNED SAFELY)
```

Здесь, ожидающий "+" игнорируется, когда вызывается frozen. Последний возвращает стек, который находился в состоянии ожидания в момент его создания: через list, затем append, и на верхний уровень, к печати результата. Если бы значение frozen венулось, как при обычном вызове функции, вышепреведенная операция привела бы к возникновению ошибки, когда + попробовал бы сложить 1 со списком.

Продолжения не получают уникальной копии стека. Они могут разделять переменные с другими продолжениями или с текущими вычислениями. В этом примере два продолжения используют один и тот же стек:

```
> (define froz1)
FROZ1
> (define froz2)
FROZ2
> (let ((x 0))
        (call-with-current-continuation
              (lambda (cc)
               (set! froz1 cc)
               (set! froz2 cc)))
        (set! x (1+ x))
        x)
1
```

поэтому вызов любого из них будет возвращать последовательные целые числа:

```
> (froz2 ())
2> (froz1 ())
3
```

Так как значение выражения call/cc будет отброшено, не имеет значения какой аргумент мы дадим froz1 и froz2.

Теперь, когда мы можем сохранять состояние вычислений, что нам с ними делать? Главы 21-24 посвящены приложениям, в которых используются продолжения. Здесь мы рассмотрим простой пример, который хорошо передает вкус программирования с

сохраненными состояниями: у нас есть набор деревьев и мы хотим генерировать списки, содержащие один элемент из каждого дерева, пока мы не получим комбинацию, удовлетворяющую некоторому условию.

Деревья могут быть представлены в виде вложенных списков. На странице 70 описан способ представленя одного вида деревьев в виде списка. Здесь мы используем другой, который позволяет внутренним узлам иметь (атомарные) значения и любое количество дочерних элементов. В этом представлении внутренний узел становится списком, его начало(car) содержит значение, хранящееся в узле(node) и его хвост(cdr) содержит представления дочерних узлов. Например, два дерева, показанные на рисунке 20-2 могут быть представлены как:

```
(define t1 '(a (b (d h)) (c e (f i) g)))
(define t2 '(1 (2 (3 6 7) 4 5)))
```

На Рисунке 20-3 показаны функции, которые выполняют обход в глубину на таких деревьях. В реальной программе мы хотели бы что-то сделать с узлами, когда мы встречаемся с ними. Здесь мы просто печатаем их. Функция dft, приведенная для сравнения, выполняет обычный обход в глубину:

```
> (dft t1)
ABDHCEFIG()
```

Рисунок 20-2: Два Дерева.

Функция dft-node следует тому же пути через дерево, но обрабатывает узлы по одному за раз. Когда dft-node достигает узла, она следует за началом(car) узла, и заталкивает в *saved* продолжение для исследования хвоста узла(cdr).

```
> (dft-node t1)
A
```

Вызов перезапуска(restart) продолжает обход, выдавая последнее сохраненное продолжение и вызывая его.

```
> (restart)
B
```

В конечном итоге не останется ни одного сохраненного состояния, условие которое давало сигнал перезапуска продолжений вернет done:

```
...> (restart)
G> (restart)
DONE
```

Наконец, функция dft2 аккуратно упаковывает то, что мы только что сделали в ручную:

```
> (dft2 t1)
ABDHCEFIG()
```

```
(define (dft tree)
      (cond ((null? tree) ())
              ((not (pair? tree)) (write tree))
              (else (dft (car tree))
                       (dft (cdr tree)))))

(define *saved* ())

(define (dft-node tree)
      (cond ((null? tree) (restart))
              ((not (pair? tree)) tree)
              (else (call-with-current-continuation
                          (lambda (cc)
                            (set! *saved*
                                    (cons (lambda ()
                                                (cc (dft-node (cdr tree))))
                                          *saved*))
                            (dft-node (car tree)))))))

(define (restart)
      (if (null? *saved*)
           'done
           (let ((cont (car *saved*)))
              (set! *saved* (cdr *saved*))
              (cont))))

(define (dft2 tree)
      (set! *saved* ())
      (let ((node (dft-node tree)))
        (cond ((eq? node 'done) ())
                (else (write node)
                          (restart)))))
```

Рисунок 20-3: Проход по Дереву используя продолжения.

Обратите внимание, что в определении dft2 нет явной рекурсии или итерации: печатаются последовательные узлы, потому что продолжения вызываемые restart всегда возвращаются через одно и тоже условие cond в dft-node.

Такая программа работает как рудник. Она копает начальную шахту вызывая dft-node. Пока возвращаемое значение не будет done, код следующий за вызовом dft-node будет вызывать restart, который снова возвращает управление обратно в стек. Этот процесс продолжается, пока возвращаемое значение не сигнализирует, что рудник пуст. Вместо печати этого значения, dft2 возвращает #f. Поиск с продолжениями представляет собой новый способ мышления о программах: поместите правильный код в стек и получите результат, многократно возвращаясь к нему.

Если мы хотим проходить только одно дерево за раз, как в dft2, то нет смысла использовать эту технику. Преимущество dft-node в том, что у нас может быть несколько экземпляров одновременно. Предположим, у нас есть два дерева, и мы

хотим сгенерировать, в порядке поиска в глубину, перекрестное произведение их элементов.

```
> (set! *saved* ())
()
> (let ((node1 (dft-node t1)))
        (if (eq? node1 'done)
             'done
             (list node1 (dft-node t2))))
(A 1)
> (restart)
(A 2)
...> (restart)
(B 1)
```

...

Используя обыные методы, нам пришлось бы предпринять явные шаги, чтобы сохранять наше положение на двух деревьях. С продолжениями состояние двух текущих обходов поддерживается автоматически. В таком простом случае, как этот, сохранить наше место на дереве не так уж и сложно. Деревья являются постоянными структурами данных, поэтому, по крайней мере, у нас есть какой-то способ получить "наше место" на дереве. Самое замечательное в продолжениях заключается в том, что они могут так же легко, сохранить наше местоположение в середине любого вычисления, даже если с ним не связаны постоянные структуры данных. Вычислениея не должны даже иметь конечного числа состояний, если мы только хотим перезапустить конечное число из них.

Как будет показано в главе 24, оба эти соображения оказываются важными для реализации Пролога. В программах на Прологе, "деревья поиска" не являются реальными струкурами данных, но подразумемвают это, когда программа генерирует результаты. И часто, деревья бесконечны, и в этом случае мы не можем надеяться обыскать одно, прежде чем искать в следующем; у нас нет выбора, кроме как сохранить наше место, так или иначе.

20.2 20-2 Макросы Передачи Продолжений

Common Lisp не предоставляет call/cc, но с небольшими дополнительными усилиями мы можем сделать тоже самое, что и в Scheme. В этом разделе показано, как использовать макросы для построения продолжений в программан на Common Lisp. Продолжения Scheme дают нам две вещи:

1. Привязки всех переменных, во то время когда было создано продолжение.
2. Состояние вычислений - что должно было произойти с тех пор.

В лексически ограниченном Lisp, замыкания дают перове из них. Оказывается, что мы можем использовать замыкания и для поддержки второго, сохраняя состояние вычислений, так же в привязках переменных.

Макросы, показанные на рисунке 20-4, позволяют выполнять вызовы функций, сохраняя продолжения. Эти макросы заменяют встроенные формы Common Lisp для определения функций, их вызова и возврата значений.

Функции, которые хотят использовать продолжения (или вызывают функции, которые это делают), должны быть определены с =defun вместо defun. Синтаксис =defun такой же, как и у defun, но его эффект несколько отличается. Вместо определения просто функции, = defun определяет функцию и макрос, который расширяется до ее вызова. (Макрос должен быть определен первым в случае, если функция вызывает себя сама.) Функция будет иметь тело, которое было передано в =defun, но будет иметь дополнительный параметр *cont*, состоящий из списка своих параметров. В расширении макроса эта функция получит *cont* вместе с другими аргументами. Так

```
(=defun add1 (x) (=values (1+ x)))

расширяет макрос в
(progn (defmacro add1 (x)
                `(=add1 *cont* ,x))
            (defun =add1 (*cont* x)
               (=values (1+ x))))
```

Когда мы вызываем add1, мы фактически вызываем не функцию, а макрос. Макрос расширяется до вызова функции,[1], Но с одним дополнительным параметром: *cont*. Таким образом, текущее значение *cont* всегда неявно передается при вызове оператора, определенного с помощью =defun.

Для чего нужен *cont*? Он будет связан с текущим продолжением. Определение =values показывает, как будет использоваться это продолжение. Любая функция, определенная с помощью =defun, должна возвращать с помощью =values или вызывать какую-то другую функцию, которая делает это.

[1] Функциям, созданным =defun, преднамеренно присваиваются интернированные имена, чтобы их можно было отследить. Если бы в трассировке не было необходимости, было бы безопаснее использовать gensym имена.

```
(setq *cont* #'identity)

(defmacro =lambda (parms &body body)
   `#'(lambda (*cont* ,@parms) ,@body))

(defmacro =defun (name parms &body body)
   (let ((f (intern (concatenate 'string
                                          "=" (symbol-name name)))))
       `(progn
            (defmacro ,name ,parms
               `(,',f *cont* ,,@parms))
            (defun ,f (*cont* ,@parms) ,@body))))

(defmacro =bind (parms expr &body body)
   `(let ((*cont* #'(lambda ,parms ,@body))) ,expr))

(defmacro =values (&rest retvals)
   `(funcall *cont* ,@retvals))

(defmacro =funcall (fn &rest args)
   `(funcall ,fn *cont* ,@args))

(defmacro =apply (fn &rest args)
   `(apply ,fn *cont* ,@args))
```

Рисунок 20-4: Макросы передающие продолжения.

Синтаксис =values такой же, как и у той же формы Common Lisp. Он может возвращать несколько значений, если существует =bind с тем же числом аргументов, ожидающий их, но не может возвращать несколько значений на верхний уровень.

Параметр *cont* сообщает функции, определенной =defun, что делать с возвращаемым значением. Когда =values макро расширяется, он будет захватывать *cont* и использовать его для имитации возврата из функции. Выражение

```
> (=values (1+ n))
```

расширяется в

```
(funcall *cont* (1+ n))
```

На верхнем уровне значение *cont* является тождеством, которое просто возвращает то, что ему передано. Когда мы вызываем (add1 2) с верхнего уровня, вызов получает макрорасширение эквивалентное

```
(funcall #'(lambda (*cont* n) (=values (1+ n))) *cont* 2)
```

Ссылка *cont* в этом случае получит глобальную привязку. Таким образом, выражение =values будет макрорасширяться в эквивалент:

```
(funcall #'identity (1+ n))
```

который просто добавляет 1 к n и возвращает результат.

В таких функциях, как add1, мы проходим через все эти проблемы только для того, чтобы имитировать, что делают, в любом случае, вызов функции Lisp и возврат:

```
> (=defun bar (x)
        (=values (list 'a (add1 x))))
BAR
> (bar 5)
(A 6)
```

Дело в том, что теперь мы взяли вызов функции и ее возврат под свой собственный контроль и можем сделать другие вещи, если захотим.

Именно манипулируя *cont*, мы получим эффект продолжений. Хотя *cont* является глобальным значением, оно редко будет использоваться: *cont* почти всегда будет параметром, захватываемым =values и макросами, определенными =defun. Например, в теле add1 *cont* является параметром, а не глобальной переменной. Это различие важно, потому что эти макросы не работали бы, если бы *cont* не была локальной переменной. Вот почему *cont* получает свое начальное значение в setq вместо defvar: последний также объявит его специальным.

Третий макрос на рис. 20-4, =bind, предназначен для использования так же, как и multiple-value-bind(множественное связывание). Он принимает список параметров, выражение и тело кода: параметры привязываются к значениям, возвращаемым выражением, и тело кода вычисляется с помощью этих привязок. Этот макрос должен использоваться всякий раз, когда дополнительные выражения должны вычисляться после вызова функции, определенной с помощью =defun.

```
> (=defun message ()
        (=values 'hello 'there))
MESSAGE

> (=defun baz ()
        (=bind (m n) (message)
          (=values (list m n))))
BAZ
> (baz)
(HELLO THERE)
```

Обратите внимание, что расширение =bind создает новую переменную с именем *cont*. Тело baz макроса расширяется в:

```
(let ((*cont* #'(lambda (m n)
                          (=values (list m n)))))
  (message))
```

который в свою очередь становится:

```
(let ((*cont* #'(lambda (m n)
                          (funcall *cont* (list m n)))))
  (=message *cont*))
```

Новое значение *cont* является телом выражения =bind, поэтому, когда message «возвращается» с помощью функции *cont*, результатом будет вычисление тела кода. Однако (и это ключевой момент) в теле =bind:

```
#'(lambda (m n)
        (funcall *cont* (list m n)))
```

cont, который был передан в качестве аргумента =baz, все еще виден, поэтому, когда тело кода в свою очередь вычисляет =values, оно будет иметь возможность вернуться к исходной вызывающей функции. Замыкания связаны между собой: каждая привязка *cont* является замыканием, содержащим предыдущую привязку *cont*, образуя цепочку, которая ведет обратно к глобальному значению.

Мы можем увидеть тот же эффект, но в меньшем масштабе, здесь:

```
> (let ((f #'identity))
        (let ((g #'(lambda (x) (funcall f (list 'a x)))))
          #'(lambda (x) (funcall g (list 'b x)))))
#<Interpreted-Function BF6326>
> (funcall * 2)
(A (B 2))
```

В этом примере создается функция, которая является замыканием, содержащим ссылку на g, которая сама является замыканием, содержащим ссылку на f. Подобные цепочки замыканий были построены компилятором сетей на стр. 80.

1. Список параметров функции, определенной с помощью =defun, должен состоять исключительно из имен параметров.
2. Функции, использующие продолжения или вызывающие другие функции, которые это делают, должны быть определены с помощью =lambda или =defun.
3. Такие функции должны завершаться, либо путем возврата значений с использованием =values, или вызовом другой функции, которая подчиняется этому ограничению.
4. Если в сегменте кода встречается выражение =bind, =values, =apply, или =funcall, это должен быть хвостовой вызов. Любой код, который будет вычисляться после =bind, должен быть помещен в его тело. Поэтому, если мы хотим иметь несколько привязок =binds следующих друг за другом, они должны быть вложенными:

```
(=defun foo (x)
       (=bind (y) (bar x)
         (format t "Ho ")
         (=bind (z) (baz x)
            (format t "Hum.")
            (=values x y z))))
```

Рисунок 20-5: Ограничения накладываемые на макросы передающие продолжения.

Остальные макросы, =apply и =funcall, предназначены для использования с функциями, определенными =lambda. Обратите внимние, что "функции" определенные с помощью =defun, поскольку на самом деле они являются макросами, не могут быть заданы в качестве аргументов для apply или funcall. Обход этой проблемы аналогичен уловке упомянутой на стр. 110. Она заключается в том, чтобы упаковать вызов в другую =lambda:

```
> (=defun add1 (x)
        (=values (1+ x)))
```

```
ADD1
> (let ((fn (=lambda (n) (add1 n))))
        (=bind (y) (=funcall fn 9)
           (format nil "9 + 1 = ~A" y)))
"9+1=10"
```

На рисунке 20-5 обобщены все ограничения, накладываемые на макросы передающие продолжения. Функции, которые ни сохраняют продолжения, ни вызывают другие функции, которые это делают, не должны использовать эти специальные макросы. Например, исключаются встроенные функции, такие как list.

Рисунок 20-6 содержит код из Рисунка 20-3, переведенный со Scheme на Common Lisp, и использующий макросы передающие продолжения вместо продолжений Scheme.

```
(defun dft (tree)
      (cond ((null tree) nil)
            ((atom tree) (princ tree))
            (t (dft (car tree))
               (dft (cdr tree)))))

(setq *saved* nil)

(=defun dft-node (tree)
      (cond ((null tree) (restart))
            ((atom tree) (=values tree))
            (t (push #'(lambda () (dft-node (cdr tree)))
                     *saved*)
               (dft-node (car tree)))))

(=defun restart ()
      (if *saved*
         (funcall (pop *saved*))
         (=values 'done)))

(=defun dft2 (tree)
      (setq *saved* nil)
      (=bind (node) (dft-node tree)
       (cond ((eq node 'done) (=values nil))
             (t (princ node)
                (restart)))))
```

Рисунок 20-6: Обход дерева с использованием макросов передающих продолжения.

С тем же примером дерева, dft2 работает также как и раньше:

```
> (setq t1 '(a (b (d h)) (c e (f i) g))
          t2 '(1 (2 (3 6 7) 4 5)))
(1 (2 (3 6 7) 4 5))
> (dft2 t1)
```

```
ABDHCEFIG
NIL
```

Сохранение состояний нескольких обходов также работает как в Scheme, хотя пример становитсья немного длиннее:

```
> (=bind (node1) (dft-node t1)
         (if (eq node1 'done)
              'done
              (=bind (node2) (dft-node t2)
                (list node1 node2))))
(A 1)
> (restart)
(A 2)
...> (restart)
(B 1)
...
```

Связывая вместе в цепочку лексические замыкания, программы Common Lisp могут создавать свои собственные продолжения. К счастью, замыкания связываются вместе внутри макросов представленных на Рисунке 20-4, и пользователь может получить эффект использования продолжений, не задумываясь о его происхождении.

Все главы 21-24 так или иначе опираются на продолжения. Эти главы покажут, что продолжения это абстракция необычной силы. Они могут быть не слишком быстрыми, особенно когда они реализованы поверх языка в виде макросов, но абстракции, которые мы можем строить на их основе, делают создание некоторых программ намного быстрее, и ускорение этой работы также очень важно.

20.3 20-3 Code-Walkers(Путешественник по Коду) и преобразование к CPS(стилю передачи продолжений)

Макросы описанные в предыдущем разделе, представляют собой компромисс. Они дают нам силу продолжений, но только если мы напишем наши программы опредленным образом. Правило 4 на Рисунке 20-5 означает, что мы должны написать

```
(=bind (x) (fn y)
       (list 'a x))
```

скорее чем

```
(list 'a                                                    ; wrong
           (=bind (x) (fn y) x))
```

Настоящий call/cc не накладывает таких ограничений на программиста. call/cc может получить продолжение в любой точке программы любой формы. Мы могли бы реализовать оператор имеющий всю мощь call/cc, но это потребовало бы намного больше работы. В этом разделе описывается как это можно сделать.

Программа на Lisp может быть преобразована в форму называемую "стиль передачи продолжений"(continuation-passing style). Программы прошедшие полное преобразование CPS, невозможно читать, но можно понять дух этого процесса, взглянув на

код, который был частично преобразован. Следующая функция для реверсирования списков:

```
(defun rev (x)
  (if (null x)
          nil
          (append (rev (cdr x)) (list (car x)))))
```

имеет эквивалентную версию написанную в стиле передачи продолжений:

```
(defun rev2 (x)
  (revc x #'identity))

(defun revc (x k)
  (if (null x)
          (funcall k nil)
          (revc (cdr x)
                  #'(lambda (w)
                          (funcall k (append w (list (car x))))))))
```

При использовании стиля передачи продолжений, функции получают дополнительный параметр (здесь k), значением которого будет продолжение. Продожение является замыканием, представляющим то, что должно быть сделано с текущим значением функции. В первой рекурсси продожением является identity; т.е то что должно быть сделано, это то, что функция дожна просто вернуть значение своего аргумента. На второй рекурсии продолжение будет эквивалентно:

```
#'(lambda (w)
        (identity (append w (list (car x)))))
```

что говорит о том, что нужно сделать, это добавить начало(car) списка к текущему значению и вернуть результат.

Как только вы сможете выполнить преобразование CPS, становиться легко написать call/cc. В программе, которая подверглась преобразованию CPS, всегда присутствуют все текущие продожения, и call/cc может быть реализован как простой макрос, который вызывается с некоторой функцией в качестве его аргумента.

Для преобразования кода в CPS нам понадобиться программа-обходчик(code-walker), которая обходит дерево, представляющее исходный код программы. Написание code-walker для Common Lisp это серьёзная задача. Чтобы быть полезным, code-walker должен делать больше, чем просто обходить выражения. Он также должен знать много о том, что означают эти выражения. Например, code-walker не может просто мыслить в терминах символов. Символ может представлять собой, помимо прочего, функцию, переменную, имя блока или тег для перехода. code-walker должен использовать контекст, чтобы отличать одно от другого, и действовать соответственно.

Поскольку написание code-walker выходит за рамки этой книги, макросы описанные в этой главе, являются наиболее практичной альтернативой. Макросы в этой главе разделяют работу по созданию продолжений вместе с пользователем. Если пользователь пишет программы в чем-то достаточно близком к CPS, макросы будут делать все остальное. Вот чем на самом деле является правило 4: если все, что следует за выражением =bind, находиться внутри его тела, затем между значением *cont* и

кодом в теле =bind, программа имеет достаточно инофрмации для создания текущего продолжения.

Макрос =bind специально написан, чтобы сделать этот стиль программирования наиболее естественным. На практике ограничения, накладываемые макросами передающими продолжения, терпимы.

21 21 Множество Процессов(многопроцессность/Многозадачность)

В предыдущей главе показано, как продолжения позволяют работающей программе получить свое собственное состояние и сохранить его для последующего перезапуска. В этой главе рассматривается модель вычислений, в которой компьютер запускает не одну отдельную программу, а набор независимых процессов. Концепция процесса(задачи) тесно соответствует с нашей конецепцией состояния программы. Написав дополнительный слой макросов поверх тех, что были описаны в предыдущей главе, мы можем встроить многозадачность в программы Common Lisp.

21.1 21-1 Абстракция Процесса/Задачи

Много Процессность/Задачность - это удобный способ выражения программ, которые должны выполнять несколько действий одновременно. Традиционный процессор выполняет одну инструкцию за раз. Сказать что несколько процессов выполняются одновременно, это не значит, что они каким-то образом преодолевают это аппаратное ограничение: это означает, что они позволяют нам мыслить на уровне абстракции, на котором нам не нужно точно указывать, что делает компьютер в некоторый момент времени. Точно также как виртуальная память позволяет нам действовать так, как если бы у компьютера было больше памяти, чем на самом деле, понятие процесса/задачи позволяет нам действовать так, как если бы компьютер мог запускать более одной программы одновременно.

Изучение процессов традиционно относиться к области операционных систем. Но полезность процессов как абстракции не ограничивается операционными системами. Они одинаково полезны и в других приложениях реального времени, и в симуляциях.

Большая часть работы, проделываемой по управлению несколькими процессами, была посвящена предотвращению определенных типов проблем. Тупик(Deadlock) - это одна классическая проблема для нескольких процессов: два процесса ждут, пока другой что-то сделает, подобно двум людям, каждый из которых отказывается переступить порог прежде чем другой это сделает. Другая проблема - это запрос, который ловит систему в несогласованном состоянии, скажем, запрос баланса, который поступает, когда система переводит средства с одного счета на другой. В этой главе рассматривается только сама абстракция процесса/задачи; код, представленный здесь, может быть использован для тестирования алгоритмов предотвращаения тупиковых или несогласованных состояний, но сам по себе он не обеспечивает никакой защиты от этих проблем.

Реализация в этой главе следует правилу, неявному во всех программах этой книги: как можно меньше нарушать Lisp. По сути, программа должна быть как можно больше похожа на модификацию языка, а не на отдельное приложение написанное на нем. Создание программ согласованных с Lisp делает их более надежными, как машины, чьи части хорошо сочетаются друг с другом. Это также экономит усилия; иногда вы можете заставить Lisp сделать для вас удивительное количество работы.

Цель этой главы - создать язык, который поддерживает несколько процессов. Наша стратегия состоит в том, чтобы превратить Lisp в такой язык, добавив несколько новых операторов. Основные элементы нашего языка будут следующими:

Функции будут определяться с помощью макросов =defun или =lambda из предыдущей главы.

Процессы будут создадваться из вызовов функций. Нет ограничений на количество активных процессов или количество процессов, созданных из какой-либо одной функции. Каждый процесс будет иметь приоритет, изначально заданный в качестве аргумента при его создании.

Выражения ожидания(Wait) могут возникать внутри функций. Выражение ожидания будет принимать переменную, тестовое выражение и тело кода. Если процесс встречает команду ожидать(wait), процесс будет приостановлен в этой точке, до тех пор пока тестовое выражение не вернет истину. Как только процесс перезапустится, будет вычислено тело кода с переменной, связанной со значением тестового выражения. Тестовые выражения, обычно, не должны иметь побочных эффектов, потому что нет никаких гарантий относительно того, когда и как часто они будут вычисляться.

Планирование(Scheduling) будет сделано по приоритету. Из всех процессов, которые могут быть перезапущены, система будет запускать процесс с наивысшим приоритетом.

Процесс по умолчанию будет запущен, если никакой другой процесс запустить нельзя. Это будет процесс REPL (read-eval-print loop).

Создание и удаление большинства объектов будет возможно на лету. Из запущенных процессов можно будет определять новые функции, а также создавать и уничтожать процессы.

```
(defstruct proc pri state wait)

(proclaim '(special *procs* *proc*))

(defvar *halt* (gensym))

(defvar *default-proc*
            (make-proc :state #'(lambda (x)
                                  (format t "~%>> ")
                                  (princ (eval (read)))
                                  (pick-process))))

(defmacro fork (expr pri)
   `(prog1 ',expr
            (push (make-proc
                        :state #'(lambda (,(gensym))
                                   ,expr
                                   (pick-process))
                        :pri      ,pri)
                  *procs*)))

(defmacro program (name args &body body)
   `(=defun ,name ,args
       (setq *procs* nil)
       ,@body
       (catch *halt* (loop (pick-process)))))
```

Рисунок 21-1: Структура процесса и его создание.

Продолжения позволяют сохранять состояние программы Lisp. Возможность хранить несколько состояний одновременно не так уж далеко стоит от реализации многопроцессности. Начиная с макросов, определенных в предыдущей главе, нам нужно менее 60 строк кода для реализации многопроцессности.

21.2 21-2 Реализация

Рисунки 21-1 и 21-2 содержат весь код, необходимый для поддержки многопроцессности. Рисунок 21-1 содержит код для основных структур данных, процесса по умолчанию, инициализации и создания процессов. Процессы или procs, имеют следующую структуру:

pri является приоритетом процесса, который должен быть положительным числом.

state является продолжением, представляющим состояние приостановленного процесса. Процесс перезапускается путем вызова как функции(funcalling) этого состояния.

wait - это обычно функция, которая должна возвращать истину для перезапуска процесса, но вначале wait для вновь созданного процесса равен nil. Процесс с wait равным nil всегда можно перезапустить.

Программа использует три глобальные переменные: *procs*, список приостановленных процессов; *proc*, текущий запущеный процесс; и *default-proc*, процесс по умолчанию.

Процесс по умолчанию выполняется только тогда, когда нет другого готового к выполнению процесса. Он имитирует верхний уровень Lisp. В этом цикле, пользователь может остановить программу, или ввести выражение, позволяющее перезапустить приостановленные процессы. Обратите внимание, что процесс по умолчанию явно вызывает eval. Эта одна из немногих ситуаций, в которых это допустимо. Обычно не рекомендуется вызывать eval во время выполнения по двум причинам:

1. Это не эффективно: eval работает с необработанным(сырым) списком, и он должен либо скомпилировать его на месте, либо вычислить в интерпретаторе. В любом случае это медленнее, чем предварительно скомпилировать код и потом просто вызвать его.
2. Это менее мощно, потому что выражение вычисляется без лексического контекста. Среди прочего, это означает, что вы не можете ссылаться на обычные переменные, видимые снаружи вычисляемого выражения.

Обычно, вызывать eval это все равно что покупать что-то в сувенирном магазине в аэропорту. Дождавшись последнего момента, вы должны заплатить высокую цену за ограниченный выбор второсортных товаров.

Случаи подобные этому, являются редкими случаями, когда ни один из двух предыдущих аргументов не применим. Мы не могли составить выражение заранее. Мы только сейчас читаем их; их ранее не было. Аналогично, выражение не может ссылаться на окружающие лексические переменные, потому что выражения, набранные на верхнем уровне, находятся в нулевой лексической среде. Фактически, определение этой функции отражает ее английское описание: она читает и вычисляет, то что вводит пользователь.

Макрос fork создает экземпляр процесса вызывая функцию. Функции определяются обычно с помощью =defun:

```
(=defun foo (x)
  (format t "Foo was called with ~A.~%" x)
  (=values (1+ x)))
```

Теперь когда мы вызваем fork с вызовом функции и номером приоритета:

```
(fork (foo 2) 25)
```

новый процесс помещается в *procs*. Новый процесс имеет приоритет равный 25, а процедура wait равна nil, поскольку она еще не была запущена, а процедура state состоит из вызова foo с аргументом 2.

Макрос program позволяет нам создать группу процессов и запустить их вместе. Определение:

```
(program two-foos (a b)
  (fork (foo a) 99)
  (fork (foo b) 99))
```

макрорасширяется в два выражени fork, зажатых между кодом который очищает приостановленные процессы, и другим кодом, который повторясь выбирает процесс для запуска. Вне этого цикла, макрос устанавливает тег, в который можно добавить

элемент управления для завершения программы. Как gensym, этот тег не будет конфликтовать с тегами, установленными пользовательским кодом. Группа процессов, определенных как program, не возвращает никакого конкретного значения и предназначена только для вызоыва с верхнего уровня.

После того, как процессы созданы, код планирования процссов вступает в управление над ними. Этот код показан на Рисунке 21-2. Функция pick-process выбирает и запускает процесс с наивысшим приоритетом, который может быть перезапущен. Выбор этого процесса является задачей most-urgent-process. Приостановленный процесс может быть запущен, если у него нет функции ожидания, или его функция ожидания wait возвращает истину. Среди приемлимых процессов выбирается тот, который имеет наивысший приоритет. Выигравший процесс и значение возвращаемое его функцией wait (если она есть) возвращаются в pick-process. Всегда будет какой-нибудь выигрышный процесс, потому что процесс по умолчанию всегда хочет запуститься.

Оставшаяся часть кода на рисунке 21-2 определяет операторы используемые для переключения управления между процессами. Стандартное выражение wait, как оно используется в функции pedestrian на Рисунке 21-3. В этом примере, процесс ждет, пока что-то не появиться в списке *open-doors*, а затем печатает сообщение:

```
> (ped)
>> (push 'door2 *open-doors*)
Entering DOOR2
>> (halt)
NIL
```

Ожидание(wait) по духу аналогично =bind (стр 267), и имеет те же ограничения, что и последняя ее вычисление. Все, что мы хотим, чтобы произошло после wait, должно быть помещено в ее тело. Таким образом, если мы хотим, чтобы процесс ожидал несколько раз, выражения ожидания должны быть вложенными. Утверждая факты предназначенные друг для друга, процессы могут сотрудничать в достижении некоторой цели, как показано на рисунке 21-4.

```
(defun pick-process ()
      (multiple-value-bind (p val) (most-urgent-process)
       (setq *proc* p
                 *procs* (delete p *procs*))
       (funcall (proc-state p) val)))

(defun most-urgent-process ()
      (let ((proc1 *default-proc*) (max -1) (val1 t))
       (dolist (p *procs*)
         (let ((pri (proc-pri p)))
            (if (> pri max)
                  (let ((val (or (not (proc-wait p))
                                 (funcall (proc-wait p)))))
                    (when val
                      (setq proc1 p
                            max     pri
                            val1 val))))))
       (values proc1 val1)))

(defun arbitrator (test cont)
      (setf (proc-state *proc*) cont
            (proc-wait *proc*) test)
      (push *proc* *procs*)
      (pick-process))

(defmacro wait (parm test &body body)
      `(arbitrator #'(lambda () ,test)
                    #'(lambda (,parm) ,@body)))

(defmacro yield (&body body)
      `(arbitrator nil #'(lambda (,(gensym)) ,@body)))

(defun setpri (n) (setf (proc-pri *proc*) n))

(defun halt (&optional val) (throw *halt* val))

(defun kill (&optional obj &rest args)
      (if obj
         (setq *procs* (apply #'delete obj *procs* args))
         (pick-process)))
```

Рисунок 21-2: Планирование Процессов.

```
(defvar *open-doors* nil)

(=defun pedestrian ()
     (wait d (car *open-doors*)
       (format t "Entering ~A~%" d)))

(program ped ()
     (fork (pedestrian) 1))
```

Рисунок 21-3: Один процесс с одним wait.

Процессы, созданные из visitor(посетитель) и host(хоста), если им представлена одна и та же дверь(door), обменяются управлением через сообщения на blackboard(черной доске):

```
> (ballet)
Approach DOOR2. Open DOOR2. Enter DOOR2. Close DOOR2.
Approach DOOR1. Open DOOR1. Enter DOOR1. Close DOOR1.
>>
```

Существует еще один, более простой тип выражения ожидания(wait): yield, единственная цель которого - дать возможноть другим процессам с более высоким приоритетом, работать. Процесс может захотеть выполнить yield после выполнения выражения setpri, которое сбрасывает приоритет текущего процесса. Как и в случае wait, любой код, который будет выполнен после выхода, должен быть помещен в его тело.

Программа на Рисунке 21-5 иллюстрирует, как два оператора работают вместе. Изначально, у варваров(barbarians) было две цели: заватить(capture) Рим(Rome) и разграбить(plunder) его. Зхват города имеет (немного) более высокий приоритет и поэтому будет выполняться первым. Однако после захвата города приоритет процесса захвата уменьшается до 1. Затем происходит голосование, и стартует процесс разрграбления(plunder), как процесс с наивысшим приоритетом.

```
> (barbarians)
Liberating ROME.
Nationalizing ROME.
Refinancing ROME.
Rebuilding ROME.
>>
```

Только после того, как варвары(barbarians) разграбили римские дворцы и освободили за выкуп патрициев, процесс захвата возобновляется, и варвары начинают укреплять свои позиции.

В основе выражений wait лежит более общий арбитр(arbitrator). Эта функция сохраняет текущий процесс, а затем вызывает процесс выбора(pick-process) для того чтобы запустить какой-либо процесс

```
(defvar *bboard* nil)

(defun claim          (&rest f) (push f *bboard*))

(defun unclaim (&rest f) (pull f *bboard* :test #'equal))

(defun check          (&rest f) (find f *bboard* :test #'equal))

(=defun visitor (door)
      (format t "Approach ~A. " door)
      (claim 'knock door)
      (wait d (check 'open door)
        (format t "Enter ~A. " door)
        (unclaim 'knock door)
        (claim 'inside door)))

(=defun host (door)
      (wait k (check 'knock door)
        (format t "Open ~A. " door)
        (claim 'open door)
        (wait g (check 'inside door)
           (format t "Close ~A.~%" door)
           (unclaim 'open door))))

(program ballet ()
      (fork (visitor 'door1) 1)
      (fork (host 'door1) 1)
      (fork (visitor 'door2) 1)
      (fork (host 'door2) 1))
```

Рисунок 21-4: Синхронизация с черной доской.

(возможно тот же самый) запускает снова. Ему передаются два аргумента: тестовая функция и продолжение. Первый будет сохранен как процедура ожидания wait процесса, который будет приостановлен, и будет вызвана позже , чтобы определить, можно ли его перезапустить. Последний станет состоянием процесса proc-state, и его вызов перезапустит приостановленный процесс.

Макросы wait и yield создают продолжение этой функции, просто заключив ее в тело в лямбда выражение. Например,

```
(wait d (car *bboard*) (=values d))
```

```
(=defun capture (city)
   (take city)
   (setpri 1)
   (yield
       (fortify city)))

(=defun plunder (city)
   (loot city)
   (ransom city))

(defun take (c)           (format t "Liberating ~A.~%" c))
(defun fortify (c) (format t "Rebuilding ~A.~%" c))
(defun loot (c)           (format t "Nationalizing ~A.~%" c))
(defun ransom (c) (format t "Refinancing ~A.~%" c))

(program barbarians ()
   (fork (capture 'rome) 100)
   (fork (plunder 'rome) 98))
```

Рисунок 21-5: Влияние изменения приоритетов.

расширяется в:

```
(arbitrator #'(lambda () (car *bboard*))
                #'(lambda (d) (=values d)))
```

Если код подчиняется ограниченям, перечисленным на Рисунке 20-5, созданное замыкание тела ожидания(wait) сохранит все текущие продолжения. Вместе с его расширением =values второй аргумент становиться:

```
#'(lambda (d) (funcall *cont* d))
```

Поскольку замыкание содержит ссылку на *cont*, приостановленный процесс с этой функцией ожидания будет иметь указатель того, куда он направлялся во время приостановки.

Оператор остановки(halt) останавливает всю программу, возвращая управление обратно тегу установленному расширением program. Он принимает необязательный аргумент, который будет возвращаен в качестве значения program. Поскольку процесс по умолчанию всегда готов к запуску, единственный способ завершить программу явный вызов halt. Неважно, какой код следует за halt, поскольку он никогда не будет вычисляться.

Отдельные процессы можно убивать вызвав kill. Если нет аргументов, этот оператор убивает текущий процесс. В этолм случае, kill напоминает выражение wait, которое игнорирует сохранение текущего процесса. Если заданы аргументы для kill, они становяться аргументами для удаления(delete) в списке процессов. В текущем коде, мало что можно сказать о выражении kill, поскольку процессы не имеют множества свойств, на которые можно ссылаться. Однако более сложная система будет ассоциировать больше информации с процессами - отметки времени, владельцы и так далее. Процесс по умолчанию не может быть уничтожен, поскольку он не сохраняется в списке *procs*.

21.3 21-3 Меньше чем быстрый прототип

Процессы, смоделированные с продожениями, не будут столь же эффективны. как процессы реальной операционной системы. Какая польза от программ, подобных той, что приведена в этой главе?

Такие программы полезны так же, как эскизы(наброски). В исследовательском программировании или быстром прототипировании, программа является не самоцелью, а средством выражения(реализации) свой идей. Во многих других областях, то что служит этой цели называется эскизом. Архитектор, в принципе, мог бы спроектировать целое здание в своей голове. Тем не менее, большинство архитекторов, кажется думают лучше с карандашами в руках: дизайн здания обычно разрабатывается в виде серии предварительных набросков.

Быстрое прототипирование - это набросок программы. Как и первые наброски архитектора, программные прототипы, как правило, рисуются несколькими быстрыми штрихами. Соображения стоимости и эффективности вначале игнорируются до разработки идеи в полной мере. Результатом, на этом этапе может стать невыстраиваемое здание или безнадежно неэффективная часть программы. Но эскизы все равно ценны,потому что

1. Они излагают информацию кратко.
2. Они предлагают возможность поэксперементировать.

Программа описанная в этой главе, как и в последующих главах, представляет собой эскиз. Он передает очертания много процессной обработки в нескольких широких штихах. И хотя он будет достаточно эффективным для использования в производственном программном обеспечении, он может быть весьма полезен для экспериментов с другими аспектами много процессности, такими как алгоритмы планирования.

В главах 22-24 представлены другие применения продолжений. Ни одно из них не является достаточно эффективным для использования в производстве программного обеспечения. Поскольку Lisp и быстрое проттипрование развивались вместе, Lisp включает в себя множество функций, специально предназначенных для проттипирования: неэффективных но удобных функций, таких как списки свойств(property lists), параметры ключевые слова, и в этом отношении списки. Продолжения, вероятно, относяться к этой категории. Они сохраняют больше состояния чем требуется программе. Так что наша реализация Prolog, например, является хорошим способом понимания языка, но не эффективным способом его реализации.

Эта книга больше посвящена тем типам абстракций, которе можно построить на Lisp, чем вопросам эффективности. Тем не менее, важно понимать, что Lisp это язык для написания производственных программ, а также язык для написания прототипов. Если Lisp и имеет репутацию медленного языка, то это во многом потому, что что многие программисты останавливаются на прототипе. Это просто писать на Lisp быстрые программы. К сожалению, еще легче писать на нем медленные. Начальная версия программы Lisp похожа на бриллиант: маленькая, чистая и очень дорогая. Может быть великим искушением оставить ее в этом состоянии.

В других языках, как только вы преуспеете в трудной задаче, заставить вашу программу работать, она уже может быть приемлемо эффективной. Если вы выложите пол плиткой размером с ноготь вашего большого пальца, ваши отходы будут

небольшими. Кто-то привыкший к разработке программного обеспечения по этому принципу, может столкнуться с трудностями при преодолении идеи, что когда программа работает, она закончена. "В Lisp вы можете писать программы в кратчайшие сроки," - может подумать он - "но они медленные.". На самом деле, это не так. Вы можете получить быстрые программы, но вы должны работать для этого. В этом отношении, использование Lisp похоже жизни в богатой стране, а не в бедной: может показаться прискорбным, что нужно работать чтобы остаться худым, но конечно это лучше, чем работать чтобы остаться в живых, и конечно быть худым, как само собо разумеющееся.

В менее абстрактных языках вы работаете над функциональностью. В Lisp вы работаете над скоростью. К счастью, работать над скоростью проще: у большинства программ есть только несколько критических разделов, в которых скорость имеет значение.

22 22 Недетерминированность

Языки программирования помогают не утонуть в деталях. Язык Лисп хорош тем, что сам управляется со многими из них, предоставляя программисту выжать максимум из ограниченной способности удерживать сложное. Эта глава посвящена тому, как с помощью макросов можно обращаться с ещё одним важным классом подробностей: подробностей преобразования недетерминированного алгоритма в детерминированный.

В главе пять частей. Первая объясняет смысл недетерминированности. Вторая описывает реализацию недетерминированного /выбора/ и /неудачи/ на Scheme с использованием продолжений. В третьей представлена версия на Common Lisp на основе передающих продолжения макросов из 20-ой главы. Четвёртая показывает, как понять оператор отсечения (cut) вне зависимости от Пролога. Последняя предлагает уточнения исходных недетерминированных операторов.

Оператор недетерминированного выбора используется в дальнейшем при написании ATN-компилятора в 23-ей главе и встроенного Пролога в 24-ой.

22.1 22-1 Общая идея

Недетерминированный алгоритм — алгоритм на основе сверхъестественного предвидения. Зачем говорить о них, не располагая сверхъестественными компьютерами? Потому что недетерминированный алгоритм можно имитировать детерминированным. Это особенно просто в чисто функциональных программах (не имеющих побочных эффектов). В них его можно реализовать с помощью поиска с отступлением.

Эта глава посвящена имитации неопределённости в функциональных программах. Располагая таким имитатором, мы рассчитываем справляться с проблемами, разрешимыми на действительно недетерминированной машине. Зачастую программа с сверхъестественными озарениями пишется проще обычной, так что этот имитатор иметь в наличии хорошо.

Данный раздел очерчивает класс возможностей, предоставляемых нам неопределённостью; следующий демонстрирует их полезность в некоторых программах. Примеры написаны на Scheme. (О некоторых различиях между Scheme и Common Lisp сказано в начале 20-ой главы.)

Недетерминированный алгоритм отличается от детерминированного использованием специальных операторов /выбора/ и /неудачи/. /Выбор/ принимает ограниченное множество и возвращает один элемент. Чтобы объяснить принцип работы /выбора/, нужно ввести понятие вычислительного /будущего/.

Здесь /выбор/ представлен функцией =choose=, принимающей список и возвращающей его элемент. Для каждого элемента есть набор будущих, принимаемых вычислением при условии выбора этого элемента. В следующем примере

```
(let ((x (choose '(1 2 3))))
  (if (odd? x)
      (+ x 1)
      x))
```

к моменту достижения =choose= у вычисления три возможных будущих:

1. Если =choose= возвращает 1, вычисление по ветви \то" вернёт 2.

2. Если =choose= возвращает 2, вычисление по ветви \иначе" вернёт 2.
3. Если =choose= возвращает 3, вычисление по ветви \то" вернёт 4.

В данном случае мы знаем точное будущее сразу, как только =choose= вернёт какое-либо значение. Вообще же каждый выбор связан с набором возможных будущих, потому что в некоторых из них могут быть дополнительные выборы. К примеру

```
(let ((x (choose '(2 3))))
  (if (odd? x)
        (choose '(a b))
        x))
```

после первого выбора имеется два набора будущих:

1. Если =choose= возвращает 2, вычисление по ветви \иначе" вернёт 2.
2. Если =choose= возвращает 3, вычисление по ветви \то" разделится на два возможных будущих, одно из которых возвращает =a=, другое — =b=.

У первого набора одно будущее, у второго — два; всего — три.

Здесь важно, что каждый из альтернативных выборов связан со своим набором возможных будущих. Какой из них будет возвращён? Можно предположить, что /выбор/ работает следующим образом:

1. Он вернёт лишь тот набор будущих, в котором хотя бы одно не заканчивается /неудачей/.
2. /Выбор/ из нуля альтернатив эквивалентен /неудаче/.

Так, к примеру, в:

```
(let ((x (choose '(1 2))))
  (if (odd? x)
        (fail)
        x))
```

каждый возможный выбор имеет по одному будущему. Поскольку выбор 1 содержит вызов =fail=, может быть выбрано лишь 2. Поэтому выражение в целом детерминированно, всегда возвращая 2.

Однако, следующее выражение недетерминированно:

```
(let ((x (choose '(1 2))))
  (if (odd? x)
        (let ((y (choose '(a b))))
              (if (eq? y 'a)
                  (fail)
                  y))
        x))
```

После первого выбора, у =1= два возможных будущих и у =2= одно. У первого, однако, будущее детерминированно, поскольку выбор =a= вызвал бы =fail=. Поэтому выражение в целом может вернуть либо =b=, либо =2=.

Наконец, однозначно следующее выражение:

```
(let ((x (choose '(1 2))))
  (if (odd? x)
```

```
        (choose '())
        x))
```

потому что выбор =1= означает последующий выбор без единого варианта. Так что этот пример эквивалентен пред-предпоследнему.

Возможно, это ещё не стало очевидным, но мы обрели абстракцию изумительной силы. В недетерминированных алгоритмах можно сказать \выбери элемент так, чтобы ничего в дальнейшем не привело к неудаче". Например, вот полностью корректный алгоритм для установления, есть ли у кого-либо предок по имени Игорь:

```
Function Ig(n)
  if name(n) = `Igor'
        then return n
  else if parents(n)
        then return Ig(choose(parents(n)))
  else fail
```

Оператор /неудачи/ используется, чтобы повлиять на значение, возвращаемое /выбором/. Если встречается /неудача/, /выбор/ сработал неправильно. Но он по определению выбирает правильно. Поэтому всё, что нам нужно, чтобы предотвратить вычисление определённой ветви - поместить где-либо в ней /неудачу/. Так, рекурсивно проходя поколения предков, функция Иг на каждом шаге выбирает ветвь, ведущую к Игорю, угадывая, по отцовской или материнской линии идти.

Это как если бы программа могла указать /выбору/ взять один из альтернативных элементов, использовать возвращённое значение сколько понадобится, и ретроспективно решить, используя /неудачу/ как запрет, что /выбору/ нужно было взять. И вуаля, оказывается, что /выбор/ его и взял. Именно в этом смысле говорят, что /выбор/ обладает предвидением.

На деле, конечно, /выбор/ не сверхъестественен. Всякая его реализация имитирует нужное угадывание отступлением от ошибок, подобно мыши в лабиринте. Но всё это отступление - скрываемо. Располагая лишь какими-либо /выбором/ и /неудачей/, уже можно писать алгоритмы подобно вышеприведённому, как если бы действительно возможно было угадать, по пути какого из предков следовать. Используя /выбор/, можно получить алгоритм поиска в проблемной области, написав лишь алгоритм её обхода.

```
(define (descent n1 n2)
      (if (eq? n1 n2)
            (list n2)
            (let ((p (try-paths (kids n1) n2)))
                  (if p (cons n1 p) #f))))

(define (try-paths ns n2)
      (if (null? ns)
            #f
            (or (descent (car ns) n2)
                  (try-paths (cdr ns) n2))))
```

Рисунок 22-1: Детерминированный поиск по дереву.

```
(define (descent n1 n2)
      (cond ((eq? n1 n2) (list n2))
                  ((null? (kids n1)) (fail))
                  (else (cons n1 (descent (choose (kids n1)) n2)))))
```

Рисунок 22-2: Недетерминированный поиск по дереву.

22.2 22-2 Поиск

Многие классические проблемы можно описать как проблемы поиска, и недетерминированность часто оказывается для них полезной абстракцией. Допустим, =nodes= содержит список вершин в дереве, а функция =(kids n)= возвращает наследников вершины =n=, либо =#f= в их отсутствие. Мы хотим определить функцию =(descent n1 n2)=, возвращающую список вершин на каком либо пути между =n1= и её наследником =n2=. Рисунок 22.1 представляет детерминированный вариант этой функции.

Недетерминированность позволяет программисту не заботится о способе поиска пути. Можно просто сказать /выбору/ найти вершину =n= такую, чтобы от неё до цели был путь. Этот вариант =descent=, изображённый на рисунке 22.2, проще.

Версия показанная на Рисунке 22-2 не занимается явным поиском вершины на правильном пути. Он написан в предположении, что =choose= выбирает желаемую =n=. Привыкший лишь к детерминированным программам может и не заметить, что =choose= словно /угадывает/, какая =n= войдёт в удачное вычисление.

```
(define (two-numbers)
    (list (choose '(012345))
             (choose '(012345))))

(define (parlor-trick sum)
    (let ((nums (two-numbers)))
       (if (= (apply + nums) sum)
             `(the sum of ,@nums)
             (fail))))
```

Рисунок 22-3: Выбор в подпрограмме.

through the computation which follows without failing.

Возможно, ещё убедительнее возможности /выбора/ продемонстрирует угадывание при вызове функций. На рисунке 22.3 пара функций угадывает два числа, суммирующихся к заданному. Первая, =two-numbers=, недетерминированно выбирает два числа и возвращает их в виде списка. Вторая, =parlor-trick=, обращается за ним к первой. Отметим, что =two-numbers= не знает о заданном числе

Если два угаданных /выбором/ числа не образуют требуемой суммы, вычисление не удаётся. Можно считать, что =choose= избегает неудачных вычислительных путей, если есть хоть один удачный. Предположительно, при задании числа в правильном диапазоне, =choose= угадывает верно; так и происходит:[fn:: Поскольку порядок вычисления аргументов в Scheme (в отличие от Common Lisp, в котором он слева направо), этот вызов может вернуть и =(THE SUM OF 5 2)=.]

```
> (parlor-trick 7)
(THE SUM OF 2 5)
```

В случае простого поиска, встроенная функция =find-if= из Common Lisp сработает не хуже. Где же преимущество недетерминированного выбора? Почему не пройти просто в цикле по списку альтернатив в поиске желаемого элемента? Ключевое отличие /выбора/ от обыкновенной итерации в том, что его область действия по отношению к /неудаче/ не ограничена. Недетерминированный /выбор/ смотрит сколь угодно далеко в будущее; если в будущем случится что-либо, аннулирующее прошлый /выбор/, можно считать, что он и не совершался. Как было показано на примере =parlor-trick=, оператор неудачи работает даже после возврата из функции, содержащей /выбор/.

Такие же неудачи случаются и при поиске в Прологе. Недетерминированность в нём полезна, поскольку одна из характерных особенностей этого языка — возможность получать ответы на запросы по одному за раз. Не возвращая все удовлетворяющие ответы сразу, Пролог справляется с рекурсивными правилами, которые иначе бы выдавали бесконечное множество ответов.

Вашим первым впечатлением от =descent=, возможно, как и от сортировки слиянием, был вопрос: где же выполняется работа? Как и при сортировке слиянием, она происходит неявно, но всё же происходит. В разделе 22.3 описана реализация /выбора/, превращающая все вышеприведённые примеры в рабочие программы.

Эти примеры иллюстрируют значение недетерминированности как абстракции. Лучшие абстракции программирования сокращают не только код, но и мысль. В теории автоматов некоторые доказательства затруднительно даже понять без

обращения к недетерминированности. Язык, допускающий недетерминированность, вероятно, предоставляет программистам сравнимое преимущество.

22.3 22-3 Реализация на Scheme

Этот раздел объясняет, как имитировать недетерминированность с помощью продолжений. Рисунок 22.4 содержит реализацию /выбора/ и /неудачи/ на Scheme, задействующую отступления. Ищущая с отступлением программа должна как-либо сохранять достаточно информации для следования по иным альтернативам, если избранная заканчивается неудачей. Эта информация хранится в виде продолжений в глобальном списке =*paths*=.

Функция =choose= принимает список альтернатив =choices=. Если он пуст, вызывается =fail=, возвращающая вычисление обратно к последнему /выбору/. Если он имеет вид =(first . rest)=, =choose= добавляет в =*paths*= продолжение, в котором =choose= вызывается с =rest=, и возвращает =first=.

Функция =fail= проще, она всего лишь забирает продолжение из =*paths*= и вызывает его. Если сохранённых путей больше нет, она возвращает символ =. Однако недостаточно просто вернуть его, иначе он станет результатом последнего вызова =choose=. Нужно вернуть его прямо на верхний уровень. Мы достигаем этого, связывая =cc= с продолжением, в котором определена =fail= — предположительно, на верхнем уровне. Вызывая =cc=, =fail= возвращает прямо туда.

Реализация на рисунке 22.4 использует =*paths*= в качестве стека, всегда возвращаясь обратно к последнему моменту выбора. Эта стратегия, называемая /хронологическим отступлением/, осуществляет поиск проблемной области в глубину. И слово «недетерминированность» часто ассоциируют только с реализацией, ищущей в глубину. Так - и в

```
(define *paths* ())
(define failsym '@)

(define (choose choices)
   (if (null? choices)
       (fail)
       (call-with-current-continuation
          (lambda (cc)
             (set! *paths*
                   (cons (lambda ()
                               (cc (choose (cdr choices))))
                         *paths*))
             (car choices)))))

(define fail)

(call-with-current-continuation
   (lambda (cc)
       (set! fail
             (lambda ()
               (if (null? *paths*)
                     (cc failsym)
                     (let ((p1 (car *paths*)))
                       (set! *paths* (cdr *paths*))
                       (p1)))))))
```

Рисунок 22-4: Scheme реализация choose и fail.

классической статье Флойда о недетерминированных алгоритмах, и недетерминированных парсерах, и в Прологе. Однако, нужно отметить, что реализация на рисунке 22.4 - не единственная возможная, и даже не корректная. В принципе, /выбор/ должен уметь возвращать объекты, удовлетворяющие любой вычислимой спецификации, тогда как наш вариант =choose= и =fail= может никогда не завершиться, если граф содержит циклы.

С другой стороны, на практике недетерминированность часто означает именно поиск в глубину, эквивалентный нашему, оставляя на пользователе обязанность избегать циклов в области поиска. Однако заинтересованный читатель найдёт реализацию настоящих /выбора/ и /неудачи/ в последнем разделе этой главы.

22.4 22-4 Реализация на Common Lisp

В этом разделе говорится о том, как написать /выбор/ и /неудачу/ на Common Lisp. Предыдущий раздел показал лёгкость имитации недетерминированности на Scheme с использованием =call/cc=; ведь продолжения - прямое воплощение нашей теоретической идеи вычислительного будущего. На Common Lisp же вместо этого можно применить передающие продолжения макросы из 20-ой главы. Вариант /выбора/, полученный с их помощью, будет несколько безобразнее написанного ранее на Scheme, но на деле эквивалентен ему.

Рисунок 22.5 демонстрирует реализацию /неудачи/ и двух вариантов /выбора/ на Common Lisp. Синтаксис этого =choose= немного отличен от предыдущего. Тот принимал один параметр: список вариантов выбора. Этот же совпадает по синтаксису с =progn=. За ним может следовать любое число выражений, из которых для вычисления выбирается только одно:

```
> (defun do2 (x)
        (choose (+ x 2) (* x 2) (expt x 2)))
DO2
> (do2 3)
5> (fail)
6
```

На верхнем уровне работа отступления, лежащего в основе недетерминированного поиска, заметнее. Переменная =*paths*= содержит ещё не пройденные пути. Когда вычисление достигает вызова =choose= с несколькими альтернативами, первая из них вычисляется, а остальные сохраняются в =*paths*=. Если программа в дальнейшем достигает =fail=, последнее сохранённое значение извлекается из =*paths*= и перезапускается. Когда список исчерпывается, =fail= возвращает специальное значение:

```
> (fail)
9> (fail)
```

На рисунке 22.5 константа =failsym=, обозначающая неудачу, определена как символ =. При желании использовать его в качестве обычного возвращаемого значения, можно в качестве =failsym= использовать =(gensym)=.

Второй оператор недетерминированного выбора, =choose-bind=, отличен по форме, принимая символ, список вариантов выбора и блок кода. Он /выберет/ одну из альтернатив, свяжет с ней символ и выполнит код.

```
(defparameter *paths* nil)
(defconstant failsym '@)

(defmacro choose (&rest choices)
   (if choices
          `(progn
             ,@(mapcar #'(lambda (c)
                           `(push #'(lambda () ,c) *paths*))
                       (reverse (cdr choices)))
             ,(car choices))
          '(fail)))

(defmacro choose-bind (var choices &body body)
   `(cb #'(lambda (,var) ,@body) ,choices))

(defun cb (fn choices)
   (if choices
         (progn
           (if (cdr choices)
               (push #'(lambda () (cb fn (cdr choices)))
                     *paths*))
           (funcall fn (car choices)))
         (fail)))

(defun fail ()
   (if *paths*
          (funcall (pop *paths*))
          failsym))
```

Рисунок 22-5: Недетерминированные операторы в Common Lisp.

```
> (choose-bind x '(marrakesh strasbourg vegas)
        (format nil "Let's go to ~A." x))
"Let's go to MARRAKESH."
> (fail)
"Let's go to STRASBOURG."
```

То, что на Common Lisp целых два оператора выбора - лишь вопрос удобства. Эффекта =choose= можно было бы добиться, всякий раз заменяя

```
(choose (foo) (bar))
```

на

```
(choose-bind x '(1 2)
   (case x
        (1 (foo))
        (2 (bar))))
```

но программы более читаемы, когда на этот случай имеется особый оператор.[1]

[1] Более того, внешний интерфейс мог бы состоять всего из одного оператора, потому что =(fail)= эквивалентен =(choose)=.

Операторы выбора на Common Lisp сохраняют связи соответствующих переменных в замыканиях с захватом переменных. Будучи макросами, =choose= и =choose-bind= раскрываются в лексической среде содержащих их выражений. Заметьте, что в =*paths*= помещается замыкание вокруг сохраняемой альтернативы, включающее в себя все связи имеющихся лексических переменных. К примеру, в выражении

```
(let ((x 2))
   (choose
        (+x1)
        (+ x 100)))
```

при перезапуске замыкания понадобится значение =x=. Вот почему =choose= оборачивает свои аргументы в лямбды. Выражение выше макрорасширяется до

```
(let ((x 2))
   (progn
        (push #'(lambda () (+ x 100))
              *paths*)
        (+ x 1)))
```

В =*paths*= сохраняется замыкание с указателем на =x=. Именно необходимость хранить переменные в замыканиях диктует различие синтаксиса между операторами выбора на Scheme и Common Lisp.

Если использовать =choose= и =fail= вместе с передающими продолжения макросами из главы 20, указатель на переменную продолжения =*cont*= тоже сохраняется. Определяя функции с помощью ~=defun~, вызывая с ~=bind~, получая возвращаемые значения с ~=values~, можно применять недетерминированность во всякой программе на Common Lisp.

С этими макросами можно успешно запустить примеры с недетерминированным выбором в подпрограммах. Рисунок 22.6 показывает версию =parlor-trick= на Common Lisp, работающую так же, как в Scheme:

```
(=defun two-numbers ()
   (choose-bind n1 '(012345)
        (choose-bind n2 '(0 12345)
          (=values n1 n2))))

(=defun parlor-trick (sum)
   (=bind (n1 n2) (two-numbers)
        (if (= (+ n1 n2) sum)
            `(the sum of ,n1 ,n2)
            (fail))))
```

Рисунок 22-6: Выбор в подпрограмме на Common Lisp

```
> (parlor-trick 7)
(THE SUM OF 2 5)
```

Это работает, потому что выражение

```
(=values n1 n2)
```

макрорасширяется до

```
(funcall *cont* n1 n2)
```

внутри =choose-bind=. Каждый =choose-bind= в свою очередь расширяется в замыкание, сохраняющее указатели на все переменные в теле кода, включая =*cont*=.

Ограничения применимости =choose=, =choose-bind= и =fail= совпадают с данными на рисунке 20.5 для кода с передающими продолжения макросами. Встречающееся выражение выбора должно вычисляться последним. Поэтому для последовательных выборов операторы выбора на Common Lisp должны быть вложены друг в друга:

```
> (choose-bind first-name '(henry william)
        (choose-bind last-name '(james higgins)
          (=values (list first-name last-name))))
(HENRY JAMES)
> (fail)
(HENRY HIGGINS)
> (fail)
(WILLIAM JAMES)
```

что приведёт, как обычно, к поиску в глубину.

Операторы, определённые в главе 20, нуждались в том, чтобы вычисляться последними. Это право теперь унаследовано новым слоем макросов; ~=values~ должно встречаться в =choose=, а не наоборот. То есть,

```
(choose (=values 1) (=values 2))
```

будет работать, а

```
(=values (choose 1 2))                                          ; wrong
```

нет. (В последнем случае расширение =choose= не захватит употребление =*cont*= в расширении ~=values~.)

До тех пор, пока эти требования, как и указанные на рисунке 20.5, будут соблюдаться, недетерминированный выбор на Common Lisp будет работать как и на Scheme. Рисунок 22.7 показывает вариант недетерминированного поиска по дереву с рисунка 22.2 на Common Lisp. Функция =descent= - результат прямого преобразования, правда, чуть более длинный и неприятный.

Теперь мы располагаем в Common Lisp средствами для недетерминированного поиска без явного отступления. Озаботившись написанием этого кода, теперь можно пожинать плоды, несколькими строками описывая в противном случае большие и спутанные программы. Построив ещё один уровень макросов над этими, можно будет написать ATN-компилятор на одной странице кода (глава 23) и набросок Пролога на двух (глава 24)

Программы с использованием /выбора/ на Common Lisp стоит компилировать с оптимизацией хвостовой рекурсии - не только ради ускорения, но и чтобы предотвратить исчерпание места на стеке. Программы, «возвращающие» значения вызовом продолжений, в действительности не возвращаются до последней /неудачи/. Без хвостовой оптимизации стек будет расти и расти.

22.5 22-5 Отсечения

Этот раздел рассказывает про использование отсечений в недетерминированных программах на Scheme. Хотя слово /отсечение/ пришло из Пролога, сама идея принадлежит недетерминированности вообще. Она может пригодиться во всякой программе с недетерминированным выбором.

Отсечения легче понять независимо от Пролога. Представим жизненный пример. Производитель шоколадных конфет решает провести рекламную кампанию. Небольшое число коробок будут содержать жетоны, обмениваемые на ценные призы. Ради справедливости, никакие две выигрышные коробки не отправляются в один город.

```
> (=defun descent (n1 n2)
       (cond ((eq n1 n2) (=values (list n2)))
             ((kids n1) (choose-bind n (kids n1)
                                  (=bind (p) (descent n n2)
                                      (=values (cons n1 p)))))
             (t (fail))))
DESCENT
> (defun kids (n)
       (case n
          (a '(b c))
          (b '(d e))
          (c '(d f))
          (f '(g))))
KIDS
> (descent 'a 'g)
(ACFG)
> (fail)
@> (descent 'a 'd)
(ABD)
> (fail)
(ACD)
> (fail)
@> (descent 'a 'h)
```

Рисунок 22-7: Недетерминированный поиск в Common Lisp

После начала рекламной кампании выясняется, что жетоны достаточно малы, чтобы быть проглочены детьми. Преследуемые видениями грядущих исков, юристы компании начинают неистово выискивать все выигрышные коробки. В каждом городе их продают во многих магазинах, в каждом магазине - много коробок. Но юристам может не понадобиться открывать каждую: как только они найдут нужную в каком-либо городе, им не придётся больше в нём искать, потому что таких в каждом городе — не более одной. Осознать это значит сделать отсечение.

/Отсекается/ часть исследуемого дерева. Для нашей компании это дерево существует физически: его корень - в главном офисе; дочерние узлы - магазины в каждом городе; от них - коробки

```
(define (find-boxes)
  (set! *paths* ())
  (let ((city (choose '(la ny bos))))
    (newline)
    (let* ((store (choose '(1 2)))
              (box (choose '(1 2))))
       (let ((triple (list city store box)))
          (display triple)
          (if (coin? triple)
              (display 'c))
          (fail)))))

(define (coin? x)
  (member x '((la 1 2) (ny 1 1) (bos 2 2))))
```

Рисунок 22-8: Исчерпывающий поиск Chocoblob.

в соответствующих магазинах. Когда юристы находят одну из коробок с жетоном, они отрезают все неисследованные ветви в том же городе.

Отсечение в действительности совершается в две операции: да, требуется знать бесполезную часть дерева, но сначала нужно /отметить/ точку дерева, в которой можно произвести отсечение. В примере с шоколадной компанией здравый смысл подсказывает, что дерево отмечается по приходу в город. В абстрактных понятиях сложно объяснить, как работает отсечение в Прологе, потому что отметки расставляются неявно. С явным же отмечающим оператором действие отсечения понять будет легче.

Программа на рисунке 22.8 недетерминированно ищет в уменьшенном подобии дерева шоколадной компании. При открывании каждой коробки она отображает список (/город/ /магазин/ /коробка/). Если в коробке оказывается жетон, выводится =c=:

```
> (find-boxes)
(LA 1 1)(LA 1 2)C(LA 2 1)(LA 2 2)
(NY 1 1)C(NY 1 2)(NY 2 1)(NY 2 2)
(BOS 1 1)(BOS 1 2)(BOS 2 1)(BOS 2 2)C
```

Для реализации техники оптимизированного поиска, открытой юристами, нужны два новых оператора: =mark= и =cut=. Одна из возможных реализаций представлена на рисунке 22.9. Тогда как недетерминированность сама по себе не зависит от реализации, сокращение дерева поиска, будучи приёмом оптимизации, определяется способом реализации =choose=. Данные операторы =mark= и

```
(define (mark) (set! *paths* (cons fail *paths*)))

(define (cut)
   (cond ((null? *paths*))
         ((equal? (car *paths*) fail)
           (set! *paths* (cdr *paths*)))
         (else
           (set! *paths* (cdr *paths*))
           (cut))))
```

Рисунок 22-9: Разметка и обрезка деревьев поиска.

```
(define (find-boxes)
   (set! *paths* ())
   (let ((city (choose '(la ny bos))))
       (mark)                                                    ;
       (newline)
       (let* ((store (choose '(1 2)))
                 (box (choose '(1 2))))
         (let ((triple (list city store box)))
            (display triple)
            (if (coin? triple)
                  (begin (cut) (display 'c)))                   ;
            (fail)))))
```

Рисунок 22-10: Сокращённый поиск Chocoblob.

=cut= подходят для =choose=, ищущего в глубину (рисунок 22.4).

Общая идея в том, что =mark= сохраняет маркеры в списке неисследованных точек выбора =*paths*=. Вызов =cut= вынимает из =*paths*= элементы вплоть до маркера, положенного туда последним. Что бы использовать в качестве маркера? Скажем, символ =m=; но тогда пришлось бы переписать =fail=, чтобы он игнорировал встречающиеся символы =m=. К счастью, поскольку функции — тоже объекты данных, один маркер позволит использовать =fail= без изменений: это сама функция =fail=. Тогда, если =fail= натолкнётся на маркер, она просто вызовет саму себя.

Рисунок 22.10 показывает использование этих операторов для сокращения дерева поиска в случае шоколадной компании. (Изменённые строки отмечены точкой с запятой.) =mark= вызывается по выбору города. К этому моменту =*paths*= содержит одно продолжение,

Рисунок 22-11: Направленный граф с циклами.

соответствующее поиску в оставшихся городах.

Когда находится коробка с жетоном, вызывается =cut=, возвращающий =*paths*= к состоянию до вызова =mark=. Результат отсечения не проявляется до следующего вызова =fail=. Но когда он, после вызова =display=, наконец обнаруживает себя, следующий =fail= отбрасывает поиск вплоть до самого первого

=choose=, даже если ниже по дереву ещё остались неисчерпанные точки выбора. В результате, как только находится коробка с жетоном, поиск продолжается со следующего города:

```
> (find-boxes)
(LA 1 1)(LA 1 2)C
(NY 1 1)C
(BOS 1 1)(BOS 1 2)(BOS 2 1)(BOS 2 2)C
```

Так, открытыми оказались семь коробок вместо двенадцати.

22.6 22-6 Настоящая недетерминированность

Детерминированной программе поиска по графу пришлось бы предпринимать явные шаги, чтобы не застрять в циклическом пути. Рисунок 22.11 изображает направленный граф, содержащий цикл. Поиск пути между =a= и =e= рискует попасться в круг =<a b c>=. Без рандомизации, поиска в ширину или отмечания циклических путей, детерминированная программа может и не завершиться. Реализация =path= на рисунке 22.12 избегает зацикливания поиском в ширину.

В принципе, недетерминированность избавляет от всякого беспокойства о циклических путях. Да, реализация /выбора/ и /неудачи/ поиском в глубину из раздела 22.3 уязвима для циклических путей, но, будучи более требовательным, можно было бы ожидать, чтобы недетерминированный /выбор/ отбирал объект,

```
(define (path node1 node2)
   (bf-path node2 (list (list node1))))

(define (bf-path dest queue)
   (if (null? queue)
        '          (let* ((path (car queue))
                          (node (car path)))
           (if (eq? node dest)
                 (cdr (reverse path))
                 (bf-path dest
                           (append (cdr queue)
                                        (map (lambda (n)
                                               (cons n path))
                                             (neighbors node))))))))
```

Рисунок 22-12: Детерминированный поиск.

```
(define (path node1 node2)
   (cond ((null? (neighbors node1)) (fail))
              ((memq node2 (neighbors node1)) (list node2))
              (else (let ((n (true-choose (neighbors node1))))
                     (cons n (path n node2))))))
```

Рисунок 22-13: Недетерминированный поиск.

удовлетворяющей всякой вычислимой спецификации, без исключения и в этом случае. Корректный /выбор/ позволил бы написать =path= короче и яснее, как показано на рисунке 22.13.

Этот раздел показывает, как реализовать /выбор/ и /неудачу/, безопасные от циклических путей. По-настоящему недетерминированная версия на Scheme показана на рисунке 22.14. Программы, использующие её, найдут решение для всякого недетерминированного алгоритма, лишь бы хватило аппаратных ресурсов.

Реализация =true-choose= на рисунке 22.14 работает со список сохранённых путей как с очередью. Программы, использующие =true-choose=, будут искать свои пространства состояний в ширину. При достижении точки выбора, продолжения каждой альтернативы добавляются в конец списка сохранённых путей.

```
(define *paths* ())
(define failsym '@)

(define (true-choose choices)
      (call-with-current-continuation
        (lambda (cc)
           (set! *paths* (append *paths*
                                 (map (lambda (choice)
                                        (lambda () (cc choice)))
                                      choices)))
           (fail))))

(define fail)

(call-with-current-continuation
      (lambda (cc)
        (set! fail
                (lambda ()
                   (if (null? *paths*)
                        (cc failsym)
                        (let ((p1 (car *paths*)))
                           (set! *paths* (cdr *paths*))
                           (p1)))))))
```

Рисунок 22-14: Корректный choose на Scheme.

(=map= в Scheme возвращает то же, что и =mapcar= в Common Lisp.) После этого вызывается =fail=, определение которой осталось тем же.

Эта версия /выбора/ позволит реализации =path= на рисунке 22.13 найти путь (причём кратчайший) от =a= до =e= на графе на рисунке 22.11.

Хотя ради полноты здесь и была приведена корректная реализация /выбора/ и /неудачи/, исходной обычно будет достаточно. Ценность языковой абстракции не уменьшается уже от того лишь, что её реализация не является формально корректной: в некоторых языках мы поступаем так, будто нам доступны все целые числа, тогда как наибольшим может быть всего лишь 32767. До тех, пока мы отдаём себе отчёт в том, сколь долго можно предаваться иллюзии, в ней мало опасности; во всяком случае, достаточно мало, чтобы абстракция оставалась выгодной. Краткость программ в следующих двух главах в значительной мере обусловлена использованием недетерминированных /выбора/ и /неудачи/.

23 23 Разбор с ATN

В этой главе показано, как писать недетерминированный синтаксический анализатор как встроенный язык. Первая часть объясняет, что такое парсеры ATN и как они представляют граматические правила. Во второй части представлен компилятор ATN который использует недетерминированные операторы определенные в предыдущей главе. В заключительных разделах представлена небольшая граматика ATN и она показана в действии при разборе входных данных.

23.1 23-1 История Вопроса

Расширенная Сеть Переходов(Augmented Transition Networks), илиг ATNs, является видом синтаксического анализатора, описанного Bill Woods в 1970. С тех пор они стали широко используемым формализмом для анализа естественного языка. Через час вы сможете написать граматику ATN, которая анализирует интересные предложения на английском языке. По этой причине людей часто держат в некоем заклинании, когда они впервые сталкиваются с ними.

В 1970-х некоторые люди думали, что ATN смогут однажды стать компонентами действительно интеллектуальных программ. Хотя немногие занимают эту позицию сегодня, ATN нашли свою нишу. Они не так хороши, как вы при разборе английского, но они все же могут анализировать впечатляющее разнообразие предложений.

ATN полезны если вы соблюдаете следующие четыре ограничения:

1. Используйте их в семантически ограниченном домене - например, во внешнем интерфейсе конкретной базы данных.
2. Не скармливайте им слишком сложный ввод. Помимо прочего, не ожидайте, что они поймут дико не грамотные предложения так, как это могут люди.
3. Используйте их только для английского или других языков, в которых порядок слов определяет грамматическую структуру. ATN не будут полезны при разборе таких языков как латынь.
4. Не ожидайте, что они будут работать все время. Используйте их в предложениях, где это полезно, если они работают 90 процентах случаев, а не в тех, где важно, чтобы они работали в 100 процентах случаев.

В этих пределах есть много полезных приложений. Канонический пример - это внешний интерфейс базы данных. Если к такой системе подключить интерфейс, управляемый ATN, то вместо формального запроса пользователи могут задавать вопросы в ограниченной форме английского языка.

23.2 23-2 Формализм

Чтобы понять что делалетт ATNs, мы должны вспомнить их полное название: расширенные сети переходов. Сеть переходов - это набор узлов, содединенных между собой направленными дугами, по сути, диаграмма-потоков. Один узел обозначен как начальный узел, а некоторые другие узлы обозначены как конечные узлы(terminal node). К каждой дуге прикрепляются условия, которые должны быть выполнены до того, как можно будет проследовать по дуге. Теперь возмем входное предложение, с указателем на текущее слово. Следование некоторым дугам приведет к продвижению

указателя. Разобрать предложение в сети переходов - это найти путь, от начального узла к некоторому конечному/терминальному(terminal) узлу, для которого выполнены все условия.

ATN добавляет две функции к этой модели:

1. ATN имеет регистры - именованные слоты для хранения информации в процессе анализа. Помимо выполнения тестов, дуги могут изменять содержимое регистров.
2. ATN является рекурсивным. Дуги могут потребовать, чтобы для прохождения по ним, анализ(парсер) прошел через некоторую подсеть.

Терминальные узлы используют информацию, накопленную в регистрах, для построения списковых структур, которые они возвращают почти так же, как функции возвращают значения. Фактически, за исключением того, что ATN являются недетерминированными, они ведут себя во многом как функциональный язык программирования.

ATN определенный на рисунке 23-1 является почти самым простым из возмжных. Он анализирует высказывания существительное-глагол в форме "Spot(место) runs(беги)." Сетевое представление этого ATN показано на рисунке 23-2.

Что делает этот ATN когда получает на вход предложение (spot runs)? Первый узел имеет одну отходящую дугу - кошку или дугу категории, ведущую к узлу s2. Он говорит, по сути,

```
(defnode s
   (cat noun s2
       (setr subj *)))

(defnode s2
   (cat verb s3
       (setr v *)))

(defnode s3
   (up `(sentence
             (subject ,(getr subj))
             (verb ,(getr v)))))
```

Рисунок 23-1: Очень маленький ATN.

Рисунок 23-2: Граф маленькой ATN.

что вы можете следовать за мной, если текущее слово является существительным, и если вы это делаете, вы должны сохранить текущее слово(обозначенное *) в регистре subj. Таким образом, мы оставляем этот узел со spot(местом) сохраненным в регистре subj.

Здесь всегда есть указатель на текущее слово. В начале он указывает на первое слово в предложении. Когда мы следуем по дуге-категории(кошке), этот указатель перемещается вперед на одно слово. Поэтому когда мы добираемся до узлв s2, текущим словом становиться второе слов - runs. Вторая дуга, такая же как и первая, за

исключением того, что она ищет глагол. Она находит runs, сохраняет его в регистре v, и переходит к s3.

Последний узел s3, имеет только входящую или конечную/терминальную дугу. (Узлы с терминальным дугами обозначаются пунктирной линией.) Поскольку мы достигаем терминальной дуги, когда у нас закончился ввод, мы имеем успешно выполненный анализ. Терминальная дуга возвращает выражение в кавычках с ним:

```
(sentence (subject spot)
          (verb runs))
```

ATN соответствует граматике языка, который она(сеть) должна анализировать Приличный размер ATN для разбора английского языка будет иметь основную сеть для разбора предложений, и подсети для разбора именных фраз, предлоговых фраз, групп модификаторов, и так далее. Необходимость рекурсии очевидна, если учесть, что именные фразы(noun-phrases) могут содержать предлоговые фразы, которые могут содержать именные фразы, и так до бесконечности, как в

> "the key on the table in the hall of the house on the hill" "ключ на столе в холле дома на холме"

23.3 23-3 Недетерминизм

Хотя мы не видели в этом небольшом примере, ATNs недетерминированы. Узел может иметь несколько исходящих дуг, более чем по одной из которых может следовать заданный ввод. Например, достаточно хороший ATN должен уметь анализировать как императивные, так и декларативные высказывания. Таким образом, первый узел может иметь исходящие дуги-категории как для существительных(в операторах), так и для глаголов (в командах).

Что если первое слово в предложении время("time"), которое в английском является и существительным и глаголом? Как анализатор(парсепр) узнает, какой дуге следовать? Когда ATN описывается как недетерминированный, это означает, что пользователи могут предполагать, что анализатор будет правильно угадывать, какой дуге следовать. Если некоторые дуги приводят только к неудачным анализам, по ним нельзя будет проходить.

В действительности парсер не может заглядывать в будущее. Он имитирует правильное угадывание путем обратного отслеживания, когда у него заканчиваются дуги или ввод. Но все механизмы обратного отслеживания автоматически вставляются в код, сгенерированный компилятором ATN. Мы можем написать ATN так, как будто парсер действительно может угадать, по какой дуге надо следовать.

Как и многие(возможно большинство) программы, которые используют недетерминизм, ATNs использует реализацию в глубину(принцип - сначала в глубь(depth-first)). Опыт разбора английского языка быстро учит, что в любом предложении есть множество законных разборов(legal parsings), большинство из которых не желательны. На обычной однопроцессорной машине, лучеше попытаться быстро получить хороший анализ. Вместо того, чтобы получать все разборы одновременно, мы получаем несколько наиболее вероятных. Если они имеют разумное толкование, то мы сэкономили усилия на поиске других разборов; если нет мы можем вызвать fail, чтобы получить больше анализов.

Чтобы управлять порядком в котором генерируются синтаксические разборы, программист должен иметь какой-то способ управления порядком, в котором choose(выбор) выбирает альтернативы. Реализация в глубину не единственный способ управления порядком поиска. Любая реализация, за исключением случайной, накладывает какой-то порядок. Тем не менее ATN, такие как Prolog, имеют концептуально встроенную реализацию с поиском в глубину. В ATN дуги, покидающие узел, проверяются в порядке, в котором они были определены. Это соглашение позволяет программисту упорядочивать дуги по приоритету.

23.4 23-4 Компилятор ATN

Обычно синтаксическому анализатору на основе ATN необходимы три компонента: сама ATN, интерпретатор для ее обхода, и словарь, который может сказать, например. что "runs" это глагол(verb). Словари - отдельная тема, здесь мы будем использовать элементарную ручную работу. Нам также не нужно иметь дело с интерпретатором сети, потому что мы переведем ATN непосредственно в код на Lisp. Описанная здесь программа называется компилятором ATN, поскольку она преобразует всю ATN в код. Узлы превращаются в функции, а дуги становяться блоками кода внутри них.

Глава 6 ввела использование функций в качестве формы представления. Эта практика обычно делает программы быстрее. Здесь это означает, что не будет никаких затрат на интерпретацию сети во время выполнения. Недостатком является то, что происходит меньше проверок, когда что то идет не так, особенно если вы используете реализацию Common Lisp, которая не предоставляет функцию лямбда-выржения.

На Рисунке 23-3 показан весь код преобразования узлов ATN в код Lisp. Макрос defnode используется для определения узлов. Он сам генерирует небольшой код, просто выбирая выражения, сгенерированные для каждой из дуг. Два параметра функции node получают следующие значения: pos это текущий указатель ввода (целое число), и regs это текущий набор регистров (список ассоциативных списков(assoc-lists)).

Макрос defnode определяет макрос с тем же именем, что и соответствующий узел. Узел s будет определен как макрос s. Это соглашение позволяет дугам знать, как ссылаться на их узлы назначения - они просто вызывают макрос с этим именем. Это также означает, что вы не должны давать узлам имена существующих функций или макросов, иначе они будут переопределены.

Отладка ATN требует своего рода средства трассировки. Поскольку узылы становятся функциями, нам не нужно писать свою собственную трассировку. Мы можем использовать встроенную функцию трассировки Lisp trace. Как упомянуто на стр. 266, использование =defun для определения узлов означает, что мы можем отслеживать анализ, проходящий через узлы mods просто сказав (trace =mods).

Дуги в теле узла - это просто вызовы макросов, возвращающие код, который внедряется в функцию узла, создаваемую defnode. Парсер использует недетерминизм в каждом узле, выполняя выбор кода, представляющего каждую из дуг, покидающих этот узел. Узел с несоклькими исходящими дугами, скажем

```
(defnode foo
  <arc 1>
  <arc 2>)
```

транслируется в определение функции следующего вида:

```
(=defun foo (pos regs)
  (choose
        <translation of arc 1>
        <translation of arc 2>))
```

```
(defmacro defnode (name &rest arcs)
      `(=defun ,name (pos regs) (choose ,@arcs)))

(defmacro down (sub next &rest cmds)
      `(=bind (* pos regs) (,sub pos (cons nil regs))
         (,next pos ,(compile-cmds cmds))))

(defmacro cat (cat next &rest cmds)
      `(if (= (length *sent*) pos)
           (fail)
           (let ((* (nth pos *sent*)))
             (if (member ',cat (types *))
                   (,next (1+ pos) ,(compile-cmds cmds))
                   (fail)))))

(defmacro jump (next &rest cmds)
      `(,next pos ,(compile-cmds cmds)))

(defun compile-cmds (cmds)
      (if (null cmds)
         'regs
         `(,@(car cmds) ,(compile-cmds (cdr cmds)))))

(defmacro up (expr)
      `(let ((* (nth pos *sent*)))
         (=values ,expr pos (cdr regs))))

(defmacro getr (key &optional (regs 'regs))
      `(let ((result (cdr (assoc ',key (car ,regs)))))
         (if (cdr result) result (car result))))

(defmacro set-register (key val regs)
      `(cons (cons (cons ,key ,val) (car ,regs))
             (cdr ,regs)))

(defmacro setr (key val regs)
      `(set-register ',key (list ,val) ,regs))

(defmacro pushr (key val regs)
      `(set-register ',key
                     (cons ,val (cdr (assoc ',key (car ,regs))))
                     ,regs))
```

Рисунок 23-3: Компиляция узлов и дуг.

```
(defnode s
   (down np s/subj
       (setr mood 'decl)
       (setr subj *))
   (cat v v
       (setr mood 'imp)
       (setr subj '(np (pron you)))
       (setr aux nil)
       (setr v *)))
```

макро расширяетс в:

```
(=defun s (pos regs)
   (choose
       (=bind (* pos regs) (np pos (cons nil regs))
        (s/subj pos
                   (setr mood 'decl
                           (setr subj * regs))))
       (if (= (length *sent*) pos)
           (fail)
           (let ((* (nth pos *sent*)))
              (if (member 'v (types *))
                   (v (1+ pos)
                       (setr mood 'imp
                               (setr subj '(np (pron you))
                                       (setr aux nil
                                                  (setr v * regs)))))
                   (fail))))))
```

Рисунок 23-4: Макрорасширение функции узла.

На рисунке 23-4 показано макроразложение первого узла в примере для ATN с Рисунка 23-11. При вызове во время выполнения, функция узла, такая как s недетерминированно выбирает дуга для следования. Параметр pos будет текущей позицией во входящем предложении и regs текущими регистрами.

Категориальные дуги, как мы видели в нашем первоначальном примере, настаивают на том, что текущее слово входного предложения относиться к определенной граматической категории. В теле категориальной дуги, символ * будет связан с текущим входным словом.

Push дуги, определенные с помощью down, требуют успешных вызовов в подсетях. Они принимают два узла назначения, подсеть назначения - sub, и следующий узел в текущей сети - next. Обратите внимание, что хотя код сгенерированный для категориальной дуги просто вызывает следующий(next) узел в сети, код генерируемый для push дуги использует =bind. push дуга должна успешно вернуться из подсети, прежде чем перейдет к узлу следующему за ней. Чистый набор регистров (nil) попадает на фронт regs прежде чем они будут переданы в подсеть. В телах други типов дуг, символ * будет связан с текущим входным словом, но в push дугах он будет связан с выражением возвращаемым подсетью.

Jump дуги похожи на короткое замыкание в схеме. Синтаксический анализатор пропходит прямо к узлу назначения, никаких тестов не требуется, и указатель входного слова не передвигается.

Последний тип дуги это pop дуга, определяемая с помощью up. Pop дуги необычны тем, что у них нет места назначения. Точно также как команда return Lisp не приводит к вызову подпорограммы, а к переходу в вызывающую функцию, pop дуга не ведет к новому узлу, а ведет обратно к "вызывающейg" push дуге. =values в pop дуге "возвращают" значение для =bind в самой последней push дуге. Но, как пояснено в Разделе 20-2 происходящее не является обычным возвратом Lisp: тело =bind было заключено в подолжение и передано в качестве параметра через любое количество дуг, пока значения =values pop дуги, накоенц, не вызовут его для "возврата" значений.

В главе 22 описаны две версии недетерминированного выбора: быстрый выбор (стр.293), который не гарантированно завершиться при наличии циклов в циклов в пространстве поиска, и более медленный истинный выбор(true-choose) (стр. 304), который был более безопасен для таких циклов. Конечно, в ATN могут быть циклы, но пока по крайней мере одна дуга в каждом цикле опережает входной указатель, синтаксический анализатор в конечном итоге переходит за конец предложения. Проблема возникает с циклами, которые не перемещают указатель ввода. Здесь у нас есть две альтернативы:

1. Использовать медленный, правильный недетерминированный оператор выбора(поиск в глубину версия данная на стр. 396).
2. Использовать быстрый выбор, и указать, что ошибочно определять сети содержащие циклы, которые можно пройти следуюя простым jump дугам (перехода).

Код определенныый на рисунке 23-3 использует второй подход.

Последние четыре определения на рисунке 23-3 определяют макросы, используемые для чтения и установки регистров внутри тела дуги. В этой программе наборы регистров представлени в виде ассоциативных списков(assoc-lists). ATN имеет дело не с набором регистров, а с наборами наборов регистров. Когда анализатор(парсер) перемещается вниз в подсеть, он получает чистый набор регистров, помещенный поверх существующих. Таким образом, вся коллекция регистров в любой момент времени представляет собой список ассоциативных спиков(assoc-lists).

Предопределенные операторы регистров работают с текущим или самым верхним набором регистров: getr читает регистр; setr устанавливает один; и pushr помещает одно значение. Оба getr и pushr используют макрос set-register упавляющий примитивом регистра.

Обратите внимание что регистры не должны объявляться. Если set-register посылает определенное имя, оно создает регистр с этим именем.

Операторы работающие с регистрами все совершенно не разрушающие. Cons, cons, cons, говорит set-register. Это замедляет их работу и создает много мусора, но как объясняется на стр. 261, объекты используемые в части программы, в которой создаются продолжения, не должны подвергаться деструктивному изменению. Объект в одном потоке управления может использоваться другим потоком, который в данный момент приостановлен. В этом случае регистры, найденные в одном разборе, будут иметь общую структуру с регистрами во многих других разборах. Если бы скорость

была приемлемой, мы могли бы хранить регистры в векторах, вместо ассоциативных списков(assoc-lists), и повторно использовать векторы в общем пуле.

Push, cat, и jump дуги, все могут содержать тела выражений. Обычно это просто сеттеры. Вызывая compile-cmds для своих тел, функция расширения этих типов дуг объединяет серию сеттеров(setrs) в одно выражение:

```
> (compile-cmds '((setr a b) (setr c d)))
(SETR A B (SETR C D REGS))
```

Каждое выражение имеет следующее выражение, вставленное в качестве последнего аргумента, кроме последнего, который получает regs. Таким образом, серия выражений в теле дуги будет преобразована в одно выражение возвращающее новые регистры.

Этот подход позволяет пользователям вставлять произвольный код Lisp в тела дуг, помещая его в progn. Например:

```
> (compile-cmds '((setr a b)
                        (progn (princ "ek!"))
                        (setr c d)))
(SETR A B (PROGN (PRINC "ek!") (SETR C D REGS)))
```

Некоторые переменные остаются видимыми для кода, встречающегося в телах дуг. Предложение(sentence) будет в глобальном *sent*. Также будут видны, две лексические переменные: pos, содержащая текущий указатель на входное слово и regs, содержащий текущие регистры. Это еще один пример преднамеренного захвата переменных. Если было бы желательно, чтобы пользователь не ссылался на эти переменные, их можно было бы заменить на gensyms.

Макрос with-parses, определенный на Рисунке 23-5, дает нам возможность не вызвать ATN. Он должен вызываться с именем начального узла, выражением, которое нужно проанализировать, и телом кода, описывающим, что делать с возвращенными значениями парсера. Тело кода в выражении with-parses будет вычисляться один раз для каждого успешного анализа. Внутри тела, символ parse будет привязан к текущему анализу/парсеру. Внешне with-parses напоминает оператор, такой как dolist, но под ним используется поиск с возвратом, вместо простой итерации. Выражение with-parses вернет потому что это то, что возвращает fail когда у него заканчивается выбор.

```
(defmacro with-parses (node sent &body body)
  (with-gensyms (pos regs)
    `(progn
         (setq *sent* ,sent)
         (setq *paths* nil)
         (=bind (parse ,pos ,regs) (,node 0 '(nil))
           (if (= ,pos (length *sent*))
                (progn ,@body (fail))
                (fail))))))
```

Рисунок 23-5: Марос верхнего уровня.

Прежде чем перейти к рассмотрению более представительного ATN, давайте рассмотрим синтаксический анализ, созданный из крошечного ATN, определенного ранее. Компилятор ATN (Рисунок 23-3) генерирует код, который вызывает types для определения граматических ролей слов, поэтому сначала мы должны дать ему некоторое определение:

```
(defun types (w)
     (cdr (assoc w '((spot noun) (runs verb)))))
```

Теперь мыы просто вызываем with-parsesc именем начального узла в качестве первого аргумента:

```
> (with-parses s '(spot runs)
        (format t "Parsing: ~A~%" parse))
Parsing: (SENTENCE (SUBJECT SPOT) (VERB RUNS))
```

23.5 23-5 Пример ATN

Теперь, когда был описан весь компилятор ATN, мы можем попробовать некоторые синатксические разборы, используя примеры сетей. Чтобы синтаксический анализатор ATN обрабатывал более разнообразные предложения, вы усложняеете сам ATN, а не компилятор ATN. Представленный здесь компилятор, представляет собой игрушку, в основом в том смысле, что он медленный, а не в смысле ограниченной мощности.

Мощь (в отиличие от скорости) синтаксического анализатора заключается в грамматике, и здесь ограниченное пространство действительно заставляет нас использовать игрушечную версию. Рисунки с 23-8 по 23-11 определяют ATN (или набор ATNs) представленную на Рисунке 23-6. Эта сеть достаточно велика, чтобы можно было выполнить несколько разборов для классического корма парсеров "Время летит как стрела"("Time flies like an arrow.")

Рисунок 23-6: Грав большой ATN.

```
(defun types (word)
   (case word
       ((do does did) '(aux v))
       ((time times) '(n v))
       ((fly flies) '(n v))
       ((like) '(v prep))
       ((liked likes) '(v))
       ((a an the) '(det))
       ((arrow arrows) '(n))
       ((i you he she him her it) '(pron))))
```

Рисунок 23-7: Номинальный словарь

Нам нужен немного больший словарь для разбора более сложного ввода. Функция types (Рисунок 23-7) предоставляет словарь самого примитивного вида. Он опреде-

ляет словарь из 22-слов, и связывает каждое слово со списком одной или нескольких простых граматических ролей.

```
(defnode mods
  (cat n mods/n
    (setr mods *)))

(defnode mods/n
  (cat n mods/n
    (pushr mods *))
  (up `(n-group ,(getr mods))))
```

Рисунок 23-8: Подсеть для модификаторов строк.

Компоненты ATN сами являются ATNs. Самый маленький ATN в нашем наборе - тот, что на рисунке 23-8. Он анализирует строковые модификаторы, что в данном случае означает стороки существительных. Первый узел, mods, принимает существительное. Второй узел, mods/n, может либо искать больше существительных, или возвращать разбор.

В разделе 3-4 объясняется как написание программ в функциональном стиле облегчает их тестирование:

1. В функциональной программе компоненты можно тестировать индивидуально.
2. В Lisp, функции можно тестировать интерактивно, в цикле верхнего уровня.

Вместе эти два принципа позволяют проводить интерактивную разработку: когда мы пишем функциональные программы на Lisp, мы можем тестировать каждый фрагмент по мере его написания.

ATNs настолько похожи на функциональные программы - в этой реализации они макрорасширяются в функциональные прогарммы, так что возможноть интерактивной разработки распространяется и на них. Мы можем проверить ATN начиная с любого узла, просто указав его имя в качестве первого аргумента with-parses:

```
> (with-parses mods '(time arrow)
        (format t "Parsing: ~A~%" parse))
Parsing: (N-GROUP (ARROW TIME))
```

Следующие два сети должны обсуждаться вместе, потому что они взаимно рекурсивны. Сеть определенная на рисунке 23-9, которая начинается с узла np, используется для разбора фраз существительных(noun phrases). Сеть определенная на рисунке 23-10 анализирует предлоги(prepositional phrases). Фразы существительных могут содержать фразы предлоги и наоборот, поэтому каждая из двух сетей содержит push дугу, которая вызывает другую сеть.

Сеть существительных фраз содержит шесть узлов. Первый узел np имеет три варианта. Если он читает метоимение(pronoun), он может перейти к узлу местоимению(pron), которое является выходом из сети:

```
(defnode np
 (cat det np/det
    (setr det *))
 (jump np/det
    (setr det nil))
 (cat pron pron
    (setr n *)))

(defnode pron
 (up `(np (pronoun ,(getr n)))))

(defnode np/det
 (down mods np/mods
    (setr mods *))
 (jump np/mods
    (setr mods nil)))

(defnode np/mods
 (cat n np/n
    (setr n *)))

(defnode np/n
 (up `(np (det ,(getr det))
           (modifiers ,(getr mods))
           (noun ,(getr n))))
 (down pp np/pp
    (setr pp *)))

(defnode np/pp
 (up `(np (det ,(getr det))
           (modifiers ,(getr mods))
           (noun ,(getr n))
           ,(getr pp))))
```

Рисунок 23-9: Подсеть фраз существительных.

```
> (with-parses np '(it)
        (format t "Parsing: ~A~%" parse))
Parsing: (NP (PRONOUN IT))
```

```
(defnode pp
  (cat prep pp/prep
      (setr prep *)))

(defnode pp/prep
  (down np pp/np
      (setr op *)))

(defnode pp/np
  (up `(pp (prep ,(getr prep))
                (obj ,(getr op)))))
```

Рисунок 23-10: подсеть фраз предлогов.

Обе другие дуги ведут к узлу np/det: одна дуга считывает определитель (например "the"), и другае дуга просто переходит(jumps), не считывая ввод. В узле np/det, обе дуги ведут к np/mods; У np/det есть возможность входа(pushing) в подсеть модификаторво(mods) чтобы получить строку модификаторов, или переход. Узел np-mods читает существительное и продлжает np/n. Этот узел может либо выдать(pop) результат, либо войти(push) в сеть фраз предлогов, чтобы попытаться подобрать фразу предлог. Последний узел, np/pp, выдает(pop) результат.

Различные типы существительных фраз буду иметь различные пути синтаксичекского анализа. Вот два анализа в сети фраз существительных:

```
> (with-parses np '(arrows)
        (pprint parse))
(NP (DET NIL)
        (MODIFIERS NIL)
        (NOUN ARROWS))
@> (with-parses np '(a time fly like him)
        (pprint parse))
(NP (DET A)
        (MODIFIERS (N-GROUP TIME))
        (NOUN FLY)
        (PP (PREP LIKE)
             (OBJ (NP (PRONOUN HIM)))))
```

Первый анализ завершается успешно, если перейти к np/det, снова перейти к np/mods, прочитать существительнрое, а затем выйти через np/n. Второй никогда не переходит (jumps), входя(pushing) вначале для поиска модификаторов строки в сеть mods

```
(defnode s
   (down np s/subj
       (setr mood 'decl)
       (setr subj *))
   (cat v v
       (setr mood 'imp)
       (setr subj '(np (pron you)))
       (setr aux nil)
       (setr v *)))

(defnode s/subj
   (cat v v
       (setr aux nil)
       (setr v *)))

(defnode v
   (up `(s (mood ,(getr mood))
               (subj ,(getr subj))
               (vcl (aux ,(getr aux))
                     (v ,(getr v)))))
   (down np s/obj
       (setr obj *)))

(defnode s/obj
   (up `(s (mood ,(getr mood))
               (subj ,(getr subj))
               (vcl (aux ,(getr aux))
                     (v ,(getr v)))
               (obj ,(getr obj)))))
```

Рисунок 23-11: Сеть предложений.

и возвращаясь снова к фразе предлогов. Как часто случается с синтаксическими анализаторами, выражение котоыре правильно синтаксически сформированы, семантически настолько бессмысленны, что людям даже трудно обнаружить в них синтаксическую структуру. Здесь фраза существительного "a time fly like him" имеет ту же форму, что и "a Lisp hacker like him."

Теперь все, что нам нужно, это сеть для распознавания структуры предложений. Сеть показанная на Риснуке 23-11 анализирует как команды, так и операторы. Начальный узел, покидающий ее, входит в сеть фраз существительных,

```
> (with-parses s '(time flies like an arrow)
          (pprint parse))

(S (MOOD DECL)
       (SUBJ (NP (DET NIL)
                    (MODIFIERS (N-GROUP TIME))
                    (NOUN FLIES)))
       (VCL (AUX NIL)
              (V LIKE))
       (OBJ (NP (DET AN)
                   (MODIFIERS NIL)
                   (NOUN ARROW))))

(S (MOOD IMP)
       (SUBJ (NP (PRON YOU)))
       (VCL (AUX NIL)
              (V TIME))
       (OBJ (NP (DET NIL)
                   (MODIFIERS NIL)
                   (NOUN FLIES)
                   (PP (PREP LIKE)
                        (OBJ (NP (DET AN)
                                    (MODIFIERS NIL)
                                    (NOUN ARROW)))))))
```

Рисунок 23-12: Два анализа для предложения.

которое будет предметом предложения. Вторая исходящая дуга читает глагол. Когда предложение синтаксически не однозначно, обе дуги могут быть успешными, в конечно итоге приводя к двум разборам, как на Рисунке 23-12. Первый разбор аналогичен "Island nations like a navy,"(островные нации, подобны флоту) и второй аналогичен "Find someone like a policeman."(найти кото-то похожего на полицейского). Более сложные ATN могут найти шесть или более разборов для "Time flies like an arrow."

Компилятор ATN в этой главе представлен скорее как воплощение идеи ATN, чем как производственная программа. Несколько очевидных изменений сделают это код намного более эффективным. Когда важна скорость, ама идея симуляции недетерминизма с помощью замыканий может быть слишком медленной. Но когда это не важно, описанные здесь методы программирования приводят к очень кратки программам.

24 24 Пролог

В этой главе описывается как написать Prolog как встроенный язык. Глава 19 показала как написать программу, которая отвечала бы на сложные запросы в базах данных. Здесь мы добавляем один новый инградиент: правила, которые позволяют выводить факты из уже известных. Набор правил определяет дерево следствий. Чтобы использовать правила, котоыре в противном случае подразумевали бы неограниченное количество фактов, мы будем искать это дерево следствий недетерминированным образом.

Prolog дает отличный пример встроенного языка. Он объединяет три компонента: сопоставление с образцом(pattern-matching), недетерминизм и правила. Главы 18 и 22 дают нам первые два независимо. Посторив Prolog поверх уже существующих операторов сопоставления с образцом и недетерминированного выбора, мы получим пример реальной многослойной(уровневой) восходящей системы. На рисунке 24-1 показаны соответствующие уровни абстракци.

Вторичной целью этой главы является изучение самого пролога. Для опытных программистов наиболее удобным объяснением Prolog-а может быть набросок его реализации. Написание Пролога на Лиспе особенно интересно, потому что оно выявляет сходство между двумя языками.

24.1 24-1 Концепции(идеи)

Глава 19 показала, как написать систему базы данных, которая будет принимать сложные запросы, содержащие переменные, и генерировать все привязки, которые сделали запрос истинным в базе данных. В следующем примере (после вызова clear-db) мы утверждаем два факта и затем запрашиваем базу данных:

Рисунок 24-1: Уровни абстракции.

```
> (fact painter reynolds)
(REYNOLDS)
> (fact painter gainsborough)
(GAINSBOROUGH)
> (with-answer (painter ?x)
         (print ?x))
GAINSBOROUGH
REYNOLDS
NIL
```

Концептуально Prolog - это программа базы данных с добавлением правил, которые позволяют удовлетворить запрос не только путем поиска его в базе данных, но и путем вывода его из других известных фактов. Например, если у на есть правило, подобное:

```
If       (hungry ?x) and (smells-of ?x turpentine)
Then (painter ?x)
если ?x голодный и пахнет скипидаром, тогда он художник! (глубокомысленно)
```

тогда запрос (painter ?x) будет удовлетворен для ?x = raoul, если база данных содержит как (hungry raoul), так и (smells-of raoul turpentine), даже если она не содержит (painter raoul).

В Prolog-е, if-часть правила(условие) называется телом, а then-часть(следствие) - головой. (В логике, имена антседент(antecedent)/предшествие и консекуент(consequent)последствие, но с тем же успехом можно было присвоить и другие имена, подчеркивают то, что в Prolog логический вывод это не тоже самое что логический.) При попытке установить привязки[1] для запроса, программа смотрит вначале на голову правила. Если голова соответствует запросу, на который пытается ответить программа, она попытается установить привязки для тела правила. Привязки которые удовлетворяют тело, по определению, удовлетворяют и голову.

Факты используемые в теле правила, могут быть в свою очередь, выведены из других правил:

```
If      (gaunt ?x) or (eats-ravenously ?x)
Then (hungry ?x)
```

и правила могут быть рекурсивными, как в:

```
If      (surname ?f ?n) and (father ?f ?c)
Then (surname ?c ?n)
```

Prolog сможет установить привязки для запроса, если он сможет найти какой-то путь через правила, который в конечном итоге приведет к известным фактам. Таким образом, это по сути поисковая система: она обходит дерево логических следствий, сформированных правилами, ища успешный путь.

Хотя правила и факты звучат как отдельные типы объектов, концептуально они взаимозаменяемы. Правила можно рассматривать как виртуальные факты. Если мы хотим, чтобы наша база данных отражала открытие, что большие(big), жестокие (fierce) животные(animals) встречаются редко(rare), мы могли бы найти все x, такие, что все факты (species x), (big x), и (fierce x), и добавить новый факт (rare x). Однако, определив правило говорящее

```
If      (species ?x) and (big ?x) and (fierce ?x)
Then (rare ?x)
```

мы получаем тот же самый эффект, фактически не добавляя всем признак (rare x) в базу данных. Мы даже можем определить правила, которые подразумевают бесконечное количество фактов. Таким образом, правила уменьшают базу данных за счет дополнительной обработки, когда приходит время отвечать на вопросы.

Тем временем, Факты, являются вырожденным случаем правил. Эффект от любого факта F может дублироваться правилом, тело которого всегда истинно:

```
If      true
Then F
```

Чтобы упростить нашу реализацию, мы воспользуемся этим принципом и представим факты в виде правил не имеющих тела.

[1] Многие понятия, используемые в этой главе, включая значение привязка, объяснены в Главе 18-4.

24.2 24-2 Интерпретатор

Раздел 18-4 показывает два способа определения if-match. Первый был простым, но неэффективным. Его преемник был быстрее, потому что делал большую часть своей работы во время компиляции. Мы будем следовать аналогичной стратегии и здесь. Чтобы предоставить некоторые из затронутых тем, мы начнем с простого интерпретатора. Позже мы покажем, как написать ту же программу более эффективно.

```
(defmacro with-inference (query &body body)
 `(progn
        (setq *paths* nil)
        (=bind (binds) (prove-query ',(rep_ query) nil)
           (let ,(mapcar #'(lambda (v)
                             `(,v (fullbind ',v binds)))
                         (vars-in query #'atom))
             ,@body
             (fail)))))

(defun rep_ (x)
      (if (atom x)
           (if (eq x '_) (gensym "?") x)
           (cons (rep_ (car x)) (rep_ (cdr x)))))

(defun fullbind (x b)
      (cond ((varsym? x) (aif2 (binding x b)
                               (fullbind it b)
                               (gensym)))
            ((atom x) x)
            (t (cons (fullbind (car x) b)
                     (fullbind (cdr x) b)))))

(defun varsym? (x)
      (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))
```

Рисунок 24-2: Макрос верхнего уровня.

Рисунки 24-224-4 содержат код для простого интерпретатора Prolog-a. Он принимает те же запросы, что и интерпретатор запросов в Разделе 19-3, но использует правила вместо базы данных для генерации привязок. Интерпретатор запросов вызывался с помощью макроса названного with-answer. Интерфейсом для интерпретатора Prolog-a будет похожий макрос, называемый with-inference. Подобно with-answer, with-inference предоставляется запрос и последовательность выражений Lisp. Переменные в запросе - это символы, начинающиеся со знака вопроса:

```
(with-inference (painter ?x)
  (print ?x))
```

Вызов with-inference расширяется до кода, который будет вычислять врыважения Lisp длял каждого набора привязок, сгенерированных запросом. Например, приве-

денный выше вызов напечатает каждый x для которого можно сделать вывод, что (painter x).

На Рисунке 24-2 показано определение with-inference, вместе с функцией, которую он вызывает для получения привязок. Одно заметное отличие между with-answer и with-inference заключается в том, что первый просто собирал все допустимые привязки. Новая программа ищет недетерминированно. Мы видим это в определении with-inference: вместо расширения в цикл, оно расширяется в код, который возвращает один набор привязок, следующий за ним fail перезапускает поиск. Это дает нам, неявно, итерацию, как в:

```
> (choose-bind x '(0 1 2 3456789)
        (princ x)
        (if (= x 6) x (fail)))
0123456
6
```

Функция fullbind указывает на другое различие между with-answer и with-inference. Обратная трассировака последовательности правил може тсоздать списки привязок, в которых привязка переменной является списком других переменных. Чтобы использовать результаты запроса, нам теперь нужна рекурсивная функция для плучения привязок. Это цель fullbind:

```
> (setq b '((?x . (?y . ?z)) (?y . foo) (?z . nil)))
((?X ?Y . ?Z) (?Y . FOO) (?Z))
> (values (binding '?x b))
(?Y . ?Z)
> (fullbind '?x b)
(FOO)
```

Привязки для запроса генерируются вызовом prove-query в расширении with-inference. На Рисунке 24-3 показано определение этой функции и функций, которые она вызывает. Этот код структурно изоморфен интерпретатору запросов, описанному в разделе 19-3. Обе программы используют одинаковые функции для сопоставления, но там где интерпретатор запросов использовал отображение(mapping) или итеарцию, интерпретатор Prolog-а использует эквивалентные варианты chooses.

Использование недетерминированного поиска вместо итерации делает интерпретацию отритцающих(negated) запросов немного более сложной. Данный запрос, подобный

```
(not (painter ?x))
```

интерперетатор запросов может просто попытаться установить привязки для (painter ?x) возвращающие nil, если таковые были найдены. При недетерминированном поиске мы должны быть более осторожными: мы не хотим, чтобы интерепретация (painter ?x) в ложь возвращалась за пределы действия области отритцания(not), и поэтому мы не хотим, чтобы она оставляла сохраненные пути, которые могли бы

```
(=defun prove-query (expr binds)
      (case (car expr)
        (and (prove-and (cdr expr) binds))
        (or     (prove-or (cdr expr) binds))
        (not (prove-not (cadr expr) binds))
        (t      (prove-simple expr binds))))

(=defun prove-and (clauses binds)
      (if (null clauses)
           (=values binds)
           (=bind (binds) (prove-query (car clauses) binds)
               (prove-and (cdr clauses) binds))))

(=defun prove-or (clauses binds)
      (choose-bind c clauses
        (prove-query c binds)))

(=defun prove-not (expr binds)
      (let ((save-paths *paths*))
        (setq *paths* nil)
        (choose (=bind (b) (prove-query expr binds)
                     (setq *paths* save-paths)
                     (fail))
                  (progn
                    (setq *paths* save-paths)
                    (=values binds)))))

(=defun prove-simple (query binds)
      (choose-bind r *rlist*
        (implies r query binds)))
```

Рисунок 24-3: Интерпретация запросов.

быть перезапущены позже. Итак, теперь тест для (painter ?x) выполняется с временно пустым списком сохраненных состояний, а старый список восстанавливается при выходе.

Другое отличие между этой программой и интерпретатором запросов заключается в интерпретации простых образцов - выражений таких как (painter ?x) которые состоят только из предиката и нескольких аргументов. Когда интерпретатор запросов генерировал привязки для простых образцов, он вызывал lookup (стр. 251). Теперь, вместо вызова lookup, мы должны получить любые привязки , подразумеваемые правилами.

```
(defvar *rlist* nil)

(defmacro <- (con &rest ant)
   (let ((ant (if (= (length ant) 1)
                  (car ant)
                  `(and ,@ant))))
       `(length (conc1f *rlist* (rep_ (cons ',ant ',con))))))

(=defun implies (r query binds)
   (let ((r2 (change-vars r)))
       (aif2 (match query (cdr r2) binds)
             (prove-query (car r2) it)
             (fail))))

(defun change-vars (r)
   (sublis (mapcar #'(lambda (v)
                              (cons v (symb '? (gensym))))
                          (vars-in r #'atom))
           r))
```

Рисунок 24-4: Code involving rules.

```
rule        : (<-  sentence  query )
query       : (not  query )
            : (and  query *)
            : (or  query *)
            :  sentence
sentence  : ( symbol  argument *)
argument  :  variable
            :  symbol
            :  number
variable  : ? symbol
```

Рисунок 24-5: Синтаксис правил.

Код для определения и использования правил показан на рисунке 24-4. Правила храняться в глобальном списке *rlist*. Каждое правило представлено в виде точечной пары тела и головы. Во время определения правила все подчеркивания заменяются уникальными переменными.

Определение <- следует трем соглашениям, часто используемым в программах подобного типа:

1. Новые правила добавляются в конец, а не в начало списка, так что они будут применяться в том порядке, в котором они были определены.
2. Правила выражены так, что голова(head) стоит первой, так как это порядок в котором программа их проверяет(рассматривает).
3. Множественные выражения в теле находятся внутри неявных и(and).

Самый внешний вызов length в расширении <- нужен просто для того, чтобы не печатать длинный список, когда <- вызывается из верхнего уровня.

Синтаксис правил приведен на Рисунке 24-5. Голова(head) правила должна быть образцом для факта: список предиката, за которым следует ноль или более аргументов. Тело(body) может быть любым запросом, который может быть обработан интерпретатором запросов главы 19. Вот правило ранее упоминавшееся в данной главе:

```
(<- (painter ?x) (and (hungry ?x)
                                    (smells-of ?x turpentine)))
```

или просто

```
(<- (painter ?x) (hungry ?x)
                              (smells-of ?x turpentine))
```

Как и в интерпретаторе запросов, аргументы типа turpentine(скипидара) не вычисляются, поэтому их надо квотировать(ставить кавычку).

Когда prove-simple запрашивается для создания привязко для запроса, он недетерминированно выбирает правило и посылает как правило, так и запрос в implies. Последняя функция(implies) затем пытается сопоставить(найти соответсвие) запрос с головой(head) правила. Если сопоставление выполнено успешно, implies вызовет prove-query, чтобы установить привязки для тела(body). Таким образом, мы рекурсивно ищем дерево следствий.

Функция change-vars заменяет все переменные в правиле на новые. ?x используемый в одном правиле, должен быть независимым от ?x используемого в другом правиле. Чтобы измежать конфликтов с существующими привязками, change-vars вызывается каждый раз, когда используется правило.

Для удобства пользователя в правлах можно использовать _(подчеркивание) в качестве безразличной переменной в правиле. Когда правило определено, вызывается функция rep_is чтобы изменить подчерквивание на реальную переменную. Подчеркивания также могут использоваться в запросах, переданных with-inference.

24.3 24-3 Правила

В этом разделе показано, как написать правила для нашего Prolog-a. В начале, вот два правилаа из Раздела 24-1:

```
(<- (painter ?x) (hungry ?x)
                         (smells-of ?x turpentine))

(<- (hungry ?x) (or (gaunt ?x) (eats-ravenously ?x)))
```

Если мы также утвердим следующие факты:

```
(<- (gaunt raoul))
(<- (smells-of raoul turpentine))
(<- (painter rubens))
```

Тогда мы получим привязки, которые они генериуют в соответствии с порядком, в котором они были определены:

```
> (with-inference (painter ?x)
```

```
        (print ?x))
RAOUL
RUBENS
```

Макрос with-inference имеет те же ограничения на привязку переменных, что и with-answer. (См. Раздел 19-4.)

Мы можем написать правила, которые подразумевают, что факты данной формы верны для всех возможных привязок. Это происходит, например, когда некоторая переменная встречается в голове(head) правила, но не в его теле. Правило

```
(<- (eats ?x ?f) (glutton ?x))
```

говорит, что если ?x это glutton(обжора), то ?x есть всё(eats everything). Поскольку ?f не встречается в теле, мы можем доказать любой факт вида (eats ?x y), просто установив привязк для ?x. Если мы сделаем запрос с буквальным значением в качестве второго аргумента в eats(есть),

```
> (<- (glutton hubert))
7> (with-inference (eats ?x spinach)
        (print ?x))
HUBERT
```

тогда любое литеральное значение будет работать. Когда мы дадим переменную в качестве второго аргумента:

```
> (with-inference (eats ?x ?y)
        (print (list ?x ?y)))
(HUBERT #:G229)
```

мы получаем gensym назад. Возврат gensym в качестве привязки переменной в запросе - это способ показать, что любое значение будет истинным. Программы могут быть написаны явно, чтобы использовать в своих интересах это соглашение:

```
> (progn
        (<- (eats monster bad-children))
        (<- (eats warhol candy)))
9> (with-inference (eats ?x ?y)
        (format t "~A eats ~A.~%"
                  ?x
                  (if (gensym? ?y) 'everything ?y)))
HUBERT eats EVERYTHING.
MONSTER eats BAD-CHILDREN.
WARHOL eats CANDY.
```

Наконец, если мы хотим указать, что факты определенной формы будут верны для любых аргументов, мы делаем тело соединением(conjunction) без аргументов. Выражение и(and) всегда будет вести себя как истинный факт. В макросе <- (рисунок 24-4), тело по умолчанию имеет значение (and), поэтому для таких правил мы можем просто опустить(не писать) тело:

```
> (<- (identical ?x ?x))
```

```
10
> (with-inference (identical a ?x)
        (print ?x))
A
```

Для читателей знакомых с Prolog-ом, на Рисунке 24-6 показан перевод синтаксиса Prolog-а в синтаксис нашей программы. Традиционно, первая программа пролога Prolog это append, которая будет написана в конце рисунка 24-6. В случае добавления, два коротких списка объединяются в один более длинны. Любые два из этих списков определяют, каким должен быть третий. Функция Lisp append принимает два коротких списка как аргументы и возвращает более длинный. append из Пролога является более общим; два правила на Рисунке 24-6 определяют программу, которая, учитывая любые два из задействованных списков, может найти третий.

Наш синтаксис отличается от традиционного синтаксиса Prolog-а следующим:

1. Переменные представлены символами, начинающимися с вопросительных знаков вместо заглавных букв. Common Lisp по умолчанию не учитывает регистр букв, поэтому от использования заглавных букв было бы больше проблем.
2. [] становиться nil.
3. Выражения вида [x | y] становятся (x . y).
4. Выражения вида [x, y, ...] становятся (x y . . .).
5. Предикаты перемещаются внутрь круглых скобок, и запятые не являются разделителями аргументов: pred(x, y, ...) становится (pred x y ...).

Таким образом Prolog определение append:

```
append([ ], Xs, Xs).
append([X | Xs], Ys, [X | Zs]) <- append(Xs, Ys, Zs).
```

становится:

```
(<- (append nil ?xs ?xs))
(<- (append (?x . ?xs) ?ys (?x . ?zs))
     (append ?xs ?ys ?zs))
```

Рисунок 24-6: синтаксическая эквивалентность Prolog-а

```
> (with-inference (append ?x (c d) (a b c d))
        (format t "Left: ~A~%" ?x))
Left: (A B)
@> (with-inference (append (a b) ?x (a b c d))
        (format t "Right: ~A~%" ?x))
Right: (C D)
@> (with-inference (append (a b) (c d) ?x)
        (format t "Whole: ~A~%" ?x))
Whole: (ABCD)
```

Мало того, учитывая последний список, он может найти всеп возможности для первых двух:

```
> (with-inference (append ?x ?y (a b c))
        (format t "Left: ~A Right: ~A~%" ?x ?y))
Left: NIL Right: (A B C)
Left: (A) Right: (B C)
Left: (A B) Right: (C)
Left: (A B C) Right: NIL
```

Случай append указывает на большую разницу между Prolog-ом и другими языками. Набор правил Prolog-а не обязательно должен давать конкретное значение. Вместо этого он может давать ограничения(constraints), которые в сочетании с ограничениями генерируемыми другими частями программы, дают конкретное значение. Например, если мы определим member таким образом:

```
(<- (member ?x (?x . ?rest)))
(<- (member ?x (_ . ?rest)) (member ?x ?rest))
```

тогда мы сможем использовать его для проверки членства(membership) в списке, как если бы использовали функцию Lisp member:

```
> (with-inference (member a (a b)) (print t))
T
```

но мы также можем использовать ее для установления ограничения на членство, которе в сочетании с другими ограничениями приводит к определенному списку. Если мы так же имеем предикат cara

```
(<- (cara (a _)))
```

который истинен для двух-элементного списка, чье начало(car) равно a, тогда между ним и member у нас достаточно ограничений для Prolog-а, чтобы построить определенный ответ:

```
> (with-inference (and (cara ?lst) (member b ?lst))
        (print ?lst))
(A B)
```

Это довольно тривиальный пример, но большие программы могут быть построены по тому же принципу. Когда мы хотим программировать, комбинируя частичные решения, Prolog может быть полезен. Действительно, удивительное разнообразие проблем может быть выражено в таких терминах: например, на рисунке 24-14 показан алгоритм сортировки, выраженный в виде набора ограничений на решение.

24.4 24-4 Необходимость Недетерминизма

Глава 22 объяснила связь между детерминированным и недетерминированным поиском. Детерминированная поисковая программа может принять запрос и сгенерировать все решения, которые его удовлетворяют. Недетерминированная программа поиска будет использовать выбор(choose) для генерации решений по одному, или если потребуется больше, вызовет fail для перезапуска поиска.

Когда у нас есть правила, которые все дают конечные наборы привязок, и мы хотим, чтобы они были все сразу, нет причин предпочитать недетерминированный поиск. Разница между этими двумя стратегиями становиться очевидной, когда у нас

есть запросы, которые генерируют бесконечное количество привязок, из которых мы хотим получить конечное подмножество. Например, правила

```
(<- (all-elements ?x nil))
(<- (all-elements ?x (?x . ?rest))
        (all-elements ?x ?rest))
```

подразумевают все факты вида (all-elements x y), где каждый член y равен x. Без обратной трассировки мы могли бы обрабатывать такие запросы, подобно:

```
(all-elements a (a a a))
(all-elements a (a a b))
(all-elements ?x (a a a))
```

Тем не менее, запрос (all-elements a ?x) выполняется для бесконечного числа возможных ?x: nil, (a), (a a), и т.д. Если мы попытаемся сгенерировать ответы для этого запроса с помощью итерации, итерация никогда не прекратиться. Даже если бы нам нужен был только один из ответов, мы бы никогда не получили разультата от реализации, которая должна была бы сгенерировать все привязки для запроса, прежде чем он мог бы начать перебирать выражения Lisp, следующие за ним.

Вот почему with-inference перемежает генерацию привязок с вычислением своего тела. Там, где запросы могут привести к бесконечному количеству ответов, единственным успешным подходом будет генерировать ответы по одному и возвращаться, чтобы получить новые ответы, перезапустив приостановленный поиск. Поскольку он использует choose и fail, наша программа может обработать этот случай:

```
> (block nil
        (with-inference (all-elements a ?x)
          (if (= (length ?x) 3)
               (return ?x)
               (princ ?x))))
NIL(A)(A A)
(AAA)
```

Как и любая другая реализация Prolog-а, наша имитирует недетерминизм, выполняя поиск в глубину с возвратами. В теории, "логические программы" работают в условиях истинного недетерминизма. По факту, реализации Prolog-а всегда используют поиск в глубину. Отнюдь не будучи скованными этим выбором, обычные программы на Prolog-е зависят от него. В истинно недетерминированном мире, запрос

```
(and (all-elements a ?x) (length ?x 3))
```

имеет ответ, но вам понадобится сколь угодно времени, чтобы выяснить, что это такое.

Prolog не только использует поиск в глубину реализованны на основе недетерминированности, он также использует версию эквивалентну определенной на странице 293. Как объяснялось там, данная реализация не всегда гарантирует прекращение. Поэтому программисты Prologo-а должны предпринять преднамеренные шаги, чтобы избежать петель в пространстве поиска. Например, если мы определим member в обратном порядке

```
(<- (member ?x (_ . ?rest)) (member ?x ?rest))
(<- (member ?x (?x . ?rest)))
```

тогда логически это будет иметь то же значение, но как программа на Prolog-е это будет иметь другой эффект. Первоначальное определение member дало бы бесконечный поток ответов на запрос(member 'a ?x), но обратное определение даст бесконечную рекурсию, и никаких ответов.

24.5 24-5 Новая Реализация

В этом разделе мы увидим еще один пример знакомого шаблона. В разделе 18-4, мы обнаружили после написания исходной версиии, что if-match может работать намного быстрее. Используюя информацию известную во время компилияции, мы смогли написать новую версию, которая выполняла меньше работы во время выполнения. Мы увидели то же явление в более широком маштабе в главе 19. Наш интерпретатор запросов был заменен эквивалентной, но более быстрой версией. То же самое случиться и с нашим интерпретатором Prolog-а.

Рисунки 24-7, 24-8, и 24-10 определяют Prolog новым способом. Макрос with-inference был просто интерфейсом для интерпретатора Prolog-а. Сейчас он, это большая часть программы. Новая программа имеет ту же общую форму, что и старая, но из функций определенных на Рисунке 24-8, только prove вызывается во время выполнения. Другие вызываются с помощью with-inference, чтобы сгенерировать его расширение.

На рисунке 24-7 показано новое определение with-inference. Как в случае с if-match и with-answer, переменные образца изначально связаны с gensyms, чтобы указать, что им не были присвоены реальные значения при сопоставлении. Таким образом функция varsym?, которую используют match и fullbind для обнраружения переменных, должна быть изменена для поиска gensyms.

```
(defmacro with-inference (query &rest body)
   (let ((vars (vars-in query #'simple?)) (gb (gensym)))
       `(with-gensyms ,vars
          (setq *paths* nil)
          (=bind (,gb) ,(gen-query (rep_ query))
               (let ,(mapcar #'(lambda (v)
                                 `(,v (fullbind ,v ,gb)))
                             vars)
                 ,@body)
               (fail)))))

(defun varsym? (x)
   (and (symbolp x) (not (symbol-package x))))
```

Рисунок 24-7: Новый макрос верхнего уровня.

Чтобы сгенерировать код для установления привязок для запроса, with-inference вызывает gen-query (Рисунок 24-8). Первое что делает gen-query, это проверяет, является ли первый аргумент сложным запросом, начинающимся с оператора подобного and или or. Этот процесс продолжается рекурсивно, пока не достигнет простых запросов, которые расширяются в вызовы prove. В первоначальной реализации такая

логическая струкутра, анализировалась во время выполнения. Сложное выражение, встречающееся в теле правила, нужно было анализировать заново, каждый раз, когда оно использовалось. Это расточительно, потому что логическая структура правил и запросов известна заранее. Новая реализация разбирает(декопозирует) сложные выражения во время компиляции.

Как и в предыдущей реализации, выражение with-inference расширяется в код, который выполняет итерацию по Lisp коду следующему за запросом с переменными образца, связываемыми с последовательными значениями, устанавливаемыми правилами. Расширение with-inference завершается с fail, который перезапускает любые сохраненные состояния.

Остальные функции на Рисунке 24-8 генерируют расширения для сложных запросов - запросов, объединенных с помощью операторов подобных and, or, и not. Если у нас есть запрос подобный

```
(and (big ?x) (red ?x))
```

мы хотим, чтобы код на Lisp-е вычислялся только с теми ?x, для которых могут быть доказаны оба коньюнкта(conjuncts/члена выражения and). Таким образом, чтобы создать расширение and, мы вкладываем расширение второго коньюнкта в расширение первого. Когда (big ?x) успешно, мы попытаемся вычислить (red ?x), и если оно тоже успешно, мы вычисллим выражения Lisp. Таким образом, всё выражение расширяется, как показано на Рисунке 24-9.

```
(defun gen-query (expr &optional binds)
      (case (car expr)
       (and (gen-and (cdr expr) binds))
       (or (gen-or (cdr expr) binds))
       (not (gen-not (cadr expr) binds))
       (t     `(prove (list ',(car expr)
                            ,@(mapcar #'form (cdr expr)))
                      ,binds))))

(defun gen-and (clauses binds)
      (if (null clauses)
         `(=values ,binds)
         (let ((gb (gensym)))
              `(=bind (,gb) ,(gen-query (car clauses) binds)
                  ,(gen-and (cdr clauses) gb)))))

(defun gen-or (clauses binds)
      `(choose
        ,@(mapcar #'(lambda (c) (gen-query c binds))
                  clauses)))

(defun gen-not (expr binds)
      (let ((gpaths (gensym)))
       `(let ((,gpaths *paths*))
             (setq *paths* nil)
             (choose (=bind (b) ,(gen-query expr binds)
                       (setq *paths* ,gpaths)
                       (fail))
                     (progn
                      (setq *paths* ,gpaths)
                      (=values ,binds))))))

(=defun prove (query binds)
       (choose-bind r *rules* (=funcall r query binds)))

(defun form (pat)
      (if (simple? pat)
         pat
         `(cons ,(form (car pat)) ,(form (cdr pat)))))
```

Рисунок 24-8: Компиляция запросов.

```
(with-inference (and (big ?x) (red ?x))
   (print ?x))
```

расширяется в:

```
(with-gensyms (?x)
   (setq *paths* nil)
   (=bind (#:g1) (=bind (#:g2) (prove (list 'big ?x) nil)
                           (=bind (#:g3) (prove (list 'red ?x) #:g2)
                              (=values #:g3)))
       (let ((?x (fullbind ?x #:g1)))
          (print ?x))
       (fail)))
```

Рисунок 24-9: Расширение коньюнкции(соединения - and).

and означает вложение; а or означает выбор(choose). Получая запрос, подобный

```
(or (big ?x) (red ?x))
```

мы хотим, чтобы выражения Lisp вычислялись для значений ?x установленных любым из подзапросов. Функция gen-or расширяется в choose поверх gen-query каждого из аргументов. Что касается not, gen-not почти идентичен prove-not (Рисунок 24-3).

На рисунке 24-10 показан код для определения правил. Правила переводятся непосредственно в код Lisp сгенерированный rule-fn. Поскольку <- теперь расширет правила в код на Lisp-е, компиляция фала, содержащего определения правил, приведет к тому, что правила будут скомпилированными функциями.

Когда rule-function отправляет шаблон, она пытается сопоставить его головой(head) правила, которое он представляет. Если сопоставление выполнено успешно, rule-function пытается установить привязки для тела. Эта задача, по сути, та же, что и with-inference, и на самом деле rule-fn заканчивается вызовом gen-query. В конечном итоге, rule-function возвращает привязки, установленные для переменных, встречающихся в начале(head) правила.

24.6 24-6 Добавление свойств(функций) Prolog-а

Уже представленный код может запускать большинство "чистых" Prolog программ. Последний шаг - добавить такие дополнения, как cuts, arithmetic, и ввод/вывод(I/O).

Помещение cut в правило Prolog-а приводит к обрезке/сокращению дерева поиска. Обычно, когда наша программа сталкиваестя с fail, он возвращется к последней точке выбора(choice).

```
(defvar *rules* nil)

(defmacro <- (con &rest ant)
      (let ((ant (if (= (length ant) 1)
                     (car ant)
                     `(and ,@ant))))
        `(length (conc1f *rules*
                         ,(rule-fn (rep_ ant) (rep_ con))))))

(defun rule-fn (ant con)
      (with-gensyms (val win fact binds)
        `(=lambda (,fact ,binds)
            (with-gensyms ,(vars-in (list ant con) #'simple?)
               (multiple-value-bind
                    (,val ,win)
                    (match ,fact
                           (list ',(car con)
                                 ,@(mapcar #'form (cdr con)))
                           ,binds)
                 (if ,win
                     ,(gen-query ant val)
                     (fail)))))))
```

Рисунок 24-10: Код для определиния правил.

Реализация choose в Разделе 22-4 хранит точки выбора в глобальной переменной *paths*. Вызов fail возобновляет поиск с самой последней точки выбора, которая является началом(car) списка *paths*. Оператор Cut вносит новое усложнение. Когда программа встречает cut, он отбрасывает некоторые самые последние точки выбора, хранящиеся в *paths*, в частности, все те, котоыре были сохранены с момента последнего вызова prove.

Эффект состоит в том, чтобы сделать правила взаимоисключающими. Мы можем использвать cut, чтобы получить эффект оператора case в программах Prolog. Например, если мы пределим minimum следующим образом:

```
(<- (minimum ?x ?y ?x) (lisp (<= ?x ?y)))
(<- (minimum ?x ?y ?y) (lisp (> ?x ?y)))
```

он будет работать правильно, но не эффективно. Получив запрос

```
(minimum 1 2 ?x)
```

Prolog сразу установит, что ?x = 1 из первого правила. Человек на этом остановится, но программа будет тратить время на поиск овтетов для второго правила, потому что небыло никаких указаний, на то, что эти два правила взаимоисключающие. В среднем эта версия minimum будет выполнять на 50% больше работы, чем нужно. Мы можем решить проблему, добавив cut после первого теста:

```
(<- (minimum ?x ?y ?x) (lisp (<= ?x ?y)) (cut))
(<- (minimum ?x ?y ?y))
```

Теперь, когда Prolog завершит работу с первым правилом, он потерпит неудачу обрабатывая все возможные пути, не переходя к следующему правилу.

Очень легко изменить нашу программу для обработки cut. При каждом вызове prove текущее состояние *paths* передается в качестве параметра. Если программа встречает cut, она просто устанавливает *paths* обратно на старое значение, переданное в параметре. На рисунках 24-11 и 24-12 показан код, который необходимо изменить для обработки cut. (Измененные строки отмечены точкой с запятой. Не все изменения происходят из-за cut.)

Cut(срезы), которые просто делают программу более эффективной, называются зелеными cut. Cut как минимум будет зеленым cut. Cut, которые заставляют программу вести себя по-другому, называются красными cut. Например, если мы определим предикат artist следующим образом:

```
(<- (artist ?x) (sculptor ?x) (cut))
(<- (artist ?x) (painter ?x))
```

результат состоит в том, что, если есть какие-либо sculptor, тогда запрос может на этом и закончиться. Если sculptor нет, тогда painter будет рассматриваться как artist:

```
> (progn (<- (painter 'klee))
             (<- (painter 'soutine)))
4> (with-inference (artist ?x)
         (print ?x))
KLEE
SOUTINE
```

Но если есть sculptor, cut останавливает вывод на первом правиле:

```
> (<- (sculptor 'hepworth))
5> (with-inference (artist ?x)
         (print ?x))
HEPWORTH
```

```
(defun rule-fn (ant con)
      (with-gensyms (val win fact binds paths)                    ;
       `(=lambda (,fact ,binds ,paths)                            ;
             (with-gensyms ,(vars-in (list ant con) #'simple?)
              (multiple-value-bind
                  (,val ,win)
                  (match ,fact
                            (list ',(car con)
                                  ,@(mapcar #'form (cdr con)))
                            ,binds)
                (if ,win
                     ,(gen-query ant val paths)                   ;
                     (fail)))))))

(defmacro with-inference (query &rest body)
      (let ((vars (vars-in query #'simple?)) (gb (gensym)))
       `(with-gensyms ,vars
             (setq *paths* nil)
             (=bind (,gb) ,(gen-query (rep_ query) nil '*paths*) ;
              (let ,(mapcar #'(lambda (v)
                                `(,v (fullbind ,v ,gb)))
                            vars)
                ,@body)
              (fail)))))

(defun gen-query (expr binds paths)                               ;
      (case (car expr)
       (and (gen-and (cdr expr) binds paths))                     ;
       (or      (gen-or (cdr expr) binds paths))                  ;
       (not (gen-not (cadr expr) binds paths))                    ;
       (lisp (gen-lisp (cadr expr) binds))                        ;
       (is      (gen-is (cadr expr) (third expr) binds))          ;
       (cut `(progn (setq *paths* ,paths)                         ;
                    (=values ,binds)))                            ;
       (t       `(prove (list ',(car expr)
                              ,@(mapcar #'form (cdr expr)))
                        ,binds *paths*))))                        ;

(=defun prove (query binds paths)                                 ;
       (choose-bind r *rules*
         (=funcall r query binds paths)))                         ;
```

Рисунок 24-11: Добавление поддержки для новых операторов.

```
(defun gen-and (clauses binds paths)                                    ;
  (if (null clauses)
        `(=values ,binds)
        (let ((gb (gensym)))
         `(=bind (,gb) ,(gen-query (car clauses) binds paths);
            ,(gen-and (cdr clauses) gb paths)))))                       ;

(defun gen-or (clauses binds paths)                                     ;
  `(choose
        ,@(mapcar #'(lambda (c) (gen-query c binds paths))              ;
                  clauses)))

(defun gen-not (expr binds paths)                                       ;
  (let ((gpaths (gensym)))
       `(let ((,gpaths *paths*))
          (setq *paths* nil)
          (choose (=bind (b) ,(gen-query expr binds paths)              ;
                     (setq *paths* ,gpaths)
                     (fail))
                  (progn
                    (setq *paths* ,gpaths)
                    (=values ,binds))))))

(defmacro with-binds (binds expr)
  `(let ,(mapcar #'(lambda (v) `(,v (fullbind ,v ,binds)))
                 (vars-in expr))
        ,expr))

(defun gen-lisp (expr binds)
  `(if (with-binds ,binds ,expr)
         (=values ,binds)
         (fail)))

(defun gen-is (expr1 expr2 binds)
  `(aif2 (match ,expr1 (with-binds ,binds ,expr2) ,binds)
           (=values it)
           (fail)))
```

Рисунок 24-12: Добавление поддержки для новых операторов.

```
rule       : (<-  sentence  query )
query      : (not  query )
           : (and  query *)
           : (lisp  lisp expression )
           : (is  variable  lisp expression )
           : (cut)
           : (fail)
           :  sentence
sentence   : ( symbol  argument *)
argument   :  variable
           :  lisp expression
variable   : ? symbol
```

Рисунок 24-13: Новый синтаксис правил.

Иногда cut используется вместе с оператором Prolog fail. Наша функция fail делает то же самое. Помещение cut в правило превращает его в улицу с односторонним движением: когда вы входите, вы обязуетесь использовать только это правило. Помещение в правило комбинации cut-fail превращает ее в улицу с односторонним движением в опасном районе: как только вы входите, вы полны решимости уйти ни с чем. Типичный пример - реализация not-equal:

```
(<- (not-equal ?x ?x) (cut) (fail))
(<- (not-equal ?x ?y))
```

Первое правило здесь - ловушка для самозванцев. Если мы пытаемся доказать факт в виде (not-equal 1 1), он будет соответствовать голове(head) первого правила и таким образом, будет обречен. Запрос (not-equal 1 2), с другой стороны, не будет соответствовать голове(head) первого правила и перейдет ко второму, где он завершится успешно:

```
> (with-inference (not-equal 'a 'a)
          (print t))
@> (with-inference (not-equal '(a a) '(a b))
          (print t))
T
```

Код, показанный на рисунках 24-11 и 24-12, также дает нашей программе арифметику, ввод-вывод и оператор Prolog is. На рисунке 24-13 показан полный синтаксис правил и запросов.

Мы добавляем арифметику (и многое другое), добавляя люк в Лисп. Теперь в дополнение к таким операторам, как and и or, у нас есть оператор lisp. После него может следовать любое выражение Lisp, которое будет вычисляться с переменными внутри границ привязок, установленными для них запросом. Если выражение вычисляется как nil, тогда Lisp выражение в целом эквивалентно (fail); в противном случае оно эквивалентно (and).

В качестве примера использования оператора lisp рассмотрим определение ordered в Прологе, которое верно для списков, элементы которых расположены в порядке возрастания:

```
(<- (ordered (?x)))
(<- (ordered (?x ?y . ?ys))
```

```
        (lisp (<= ?x ?y))
        (ordered (?y . ?ys)))
```

В английском языке список одного элемента упорядочен, а список двух или более элементов упорядочен, если первый элемент списка меньше или равен второму, а список из второго элемента упорядочен.

```
> (with-inference (ordered '(1 2 3))
        (print t))
T@> (with-inference (ordered '(1 3 2))
        (print t))
```

С помощью оператора lisp мы можем предоставить другие функции, предлагаемые Типичными реализациями Пролога. Предикаты ввода/вывода Prolog-a можно дублировать, помещая вызовы Lisp I/O в выражение lisp. Утверждение Prolog-a, которое в качестве побочного эффекта определяет новые правила, может быть продублировано путем вызова макроса <- в выражениях lisp.

Оператор is предлагает форму присваивания. Он принимает два аргумента, образец и выражение Lisp, и пытается сопоставить образец с результатом, возвращаемым выражением. Если совпадение не удается, то вызовы программы завершаются неудачно; в противном случае оно продолжается с новыми привязками. Таким образом, выражение (is? X 1) имеет эффект установки ?X в 1, или, точнее, настаивая на том, чтобы ?X было 1. Нам нужно вычислить, например, factorials(факториалы):

```
(<- (factorial 0 1))
(<- (factorial ?n ?f)
        (lisp (> ?n 0))
        (is ?n1 (- ?n 1))
        (factorial ?n1 ?f1)
        (is ?f (* ?n ?f1)))
```

Мы используем это определение, делая запрос с номером n в качестве первого аргумента и переменной в качестве второго:

```
> (with-inference (factorial 8 ?x)
             (print ?x))
40320
```

Обратите внимание, что переменные, используемые в выражении lisp или во втором аргументе is, должны иметь привязки, чтобы выражение возвращало значение. Это ограничение действует в любом Прологе. Например, запрос:

```
(with-inference (factorial ?x 120)                                    ; wrong
     (print ?x))
```

не будет работать с этим определением факториала, потому что ?n будет неизвестно при вычислении выражения lisp. Так что не все программы Prolog похожи на append: многие настаивают, как факториал, на том, что некоторые их аргументы являются реальными значениями.

24.7 24-7 Примеры

В этом последнем разделе показано, как написать несколько примеров программ на Prolog в нашей реализации. Правила на Рисунке 24-14 определяют быструю сортировку - quicksort. Эти правила подразумевают факты вида (quicksort xy), где x это список, а y это список тех же элементов, отсортированных в порядке возрастания. Переменные могут появляться в позиции второго аргумента:

```
> (with-inference (quicksort '(3 2 1) ?x)
        (print ?x))
(123)
```

Цикл ввода/вывода является тестом для нашего Prolog, поскольку он использует операторы cut, lisp, и is. Код показан на рисунке 24-15. Эти правила следует вызывать, пытаясь доказать (echo), без аргументов. Этот запрос будет соответствовать первому правилу, которое связывает ?x с результатом возвращенным read, и затем пытаестя установить (echo ?x). Новый запрос может соответствовать любому из двух вторых правил. Если ?x = done, то запрос завершиться на втором правиле. В противном случае запрос будет соответствовать только третьему правилу, котое печатает прочитанное значение и запускает процесс заново.

```
(setq *rules* nil)

(<- (append nil ?ys ?ys))
(<- (append (?x . ?xs) ?ys (?x . ?zs))
       (append ?xs ?ys ?zs))

(<- (quicksort (?x . ?xs) ?ys)
       (partition ?xs ?x ?littles ?bigs)
       (quicksort ?littles ?ls)
       (quicksort ?bigs ?bs)
       (append ?ls (?x . ?bs) ?ys))
(<- (quicksort nil nil))

(<- (partition (?x . ?xs) ?y (?x . ?ls) ?bs)
       (lisp (<= ?x ?y))
       (partition ?xs ?y ?ls ?bs))
(<- (partition (?x . ?xs) ?y ?ls (?x . ?bs))
       (lisp (> ?x ?y))
       (partition ?xs ?y ?ls ?bs))
(<- (partition nil ?y nil nil))
```

Рисунок 24-14: Быстрая сортировка(Quicksort).

```
(<- (echo)
       (is ?x (read))
       (echo ?x))
(<- (echo 'done)
       (cut))
(<- (echo ?x)
       (lisp (prog1 t (format t "~A~%" ?x)))
       (is ?y (read))
       (cut)
       (echo ?y))
```

Рисунок 24-15: Цикл ввода/вывода(I/O) в Prolog-е.

В совокупности правила определяют программу, которая будет продолжать повторять, то что вы печатаете, до тех пор, пока вы не напечатаете done:

```
> (with-inference (echo))
hi
HI
ho
HO
done
```

Такие программы трудно читать, потому что они разрушают абстрактную модель Prolog-а. Может быть, легче понять echo если мы посмотрим на буквальную трансляцию в код Lisp.

```
(defun echo (&rest args)
     (cond ((null args) (echo (read)))
             ((eq (car args) 'done) nil)
             (t (format t "~A~%" (car args))
                 (echo (read)))))
```

который в идиоматическом Common Lisp будет:

```
(defun echo (&optional (arg (read)))
     (unless (eq arg 'done)
        (format t "~A~%" arg)
        (echo)))
```

24.8 24-8 Смыслы Компиляции

Слово "компиляция"(compile) имеет несколько смыслов. В самом общем смысле, компиляция это преобразование некоторого абстрактного описания программы в код более низкого уровня. Программа, описанная в этой главе, безусловно является компилятором, в этом смысле, потому что она переводит правила в функции Lisp.

В более конкретном смысле, компиляция означает преобразование программы в машинный язык. Хороший Common Lisp-ы компилируют функции в машинный код. Как упомянуто на стр. 25, если код, который генерирует замыкания, скомпилирован, он даст скомпилированные замыкания. Таким образом, описанная здесь программа,

также является более строгим компилятором. В хорошем Lisp, наши Prolog программы будут переведены на машинный язык.

Вместе с тем, описанная здесь программа, все еще не является компилятором Prolog-a. Для языков программирования существует еще более конкретный смысл "компиляции", и простого создания машинного кода недостаточно для удовлетворения этого определения. Компилятор для языка программирования должен как оптимизировать код, так и транслировать его. Например, если компилятору Lisp предлагается скомпилировать выражение вроде

```
(+ x (+ 2 5))
```

он должен быть достаточно умен, чтобы понять, что нет причин ждать, пока во время выполнения вычислиться (+ 2 5). Программу можно оптимизировать, заменив это выражение на 7, и вместо него компилировать

```
(+ x 7)
```

В нашей программе, вся компиляция выполняется компилятором Lisp, и он ищет оптимизацию для Lisp-as, а не оптимизацию Prolog-a. Ее оптимизация будет действенной, но слижком низкоуровневой. Компилятор Lisp не знает, что код, который он компилирует, предназначен для представления правил. В то время как настоящий компилятор Prolog будет искать правила, которые можно преобразовывать в циклы, наша программа ищет выражения, которые выдают констатны, или замыкания, которые могут быть размещены в стеке.

Встроенные языки позволяют максимально использовать доступные абстракции, но они не волшебны. Если вы хотите пройти путь от очень абстрактного представления до быстрого машинного кода, кто-то еще длолжен сказать компьютеру, как это сделать. В этой главе мы прошли большую часть этого расстояния с удивительно небольшим кодом, но это не тоже самое, что написиать настоящий компилятор Prolog-a.

25 25 Объектно Ориентированный Lisp

В этой главе обсуждается объектно-ориентированное программирование на Lisp. Common Lisp включает в себя набор операторов для написания объектно-ориентриованных программ. В совокупности они называются Common Lisp Object System, или CLOS. Здесь мы рассматриваем CLOS не просто как способ написания объектно ориентированных программ, а как саму программу на Lisp. Видение CLOS в этом свете является ключом к пониманию связи между Lisp и объектно ориентированным программированием.

25.1 25-1 Plus ca Change

Объектно-ориентированное программирование означает изменение способа организации программ. Это изменение аналогично тому, какое имело место при распределении мощности процессора. В 1970, многопользовательская компьютерная система означала один или два больших мэйн фрейма, подключенных к большому количеству "глупых"(dumb) терминалов. Теперь это скорее всего означает большое количество рабочих станций, соединенных друг с другом сетью. Теперь вычислительная мощность системы распределяется между отдельными пользователями, а не централизованно на одном большом компьютере.

Объектно-ориентированное программирование разбивает традиционные программы почти таким же образом: вместо единой программы, которая работает с инертной массой данных, самим данным сообщается, как себя вести, и программа является неявным взаимодействием этих новых "объектов" данных.

Например, предположим, что мы хотим написать программу для поиска площадей двумерных фигур. Одним из способов сделать это было бы написать функцию, которая смотрела бы на тип свого аргумента и вела бы себя соответственно:

```
(defun area (x)
  (cond ((rectangle-p x) (* (height x) (width x)))
        ((circle-p x) (* pi (expt (radius x) 2)))))
```

Объектно-ориентированный подход заключается в том, чтобы каждый объект мог сам рассчитывать свою собственную площадь. Функция area разбита на части и каждое предложение распространяется на соответствующий класс объекта; метод area класса прямоугольника(rectangle) может быть

```
#'(lambda (x) (* (height x) (width x)))
```

и для класса круга(circle),

```
#'(lambda (x) (* pi (expt (radius x) 2)))
```

В этой модели, мы спрашиваем объект, какова его площадь, и он отвечает в соответствии с методом, предоставленным для его класса.

Появление CLOS может показаться признаком того, что Lisp меняется, чтобы принять объектно-ориентированную парадигму. На самом деле, было бы точнее сказать, что Lisp остается неизменным, чтобы принять объектно-ориентированную парадигму. Но принципы лежащие в основе Lisp не имеют названия, как у объектно-ориентированного программирования, поэтому сейчас существует тенденция описывать Lisp как объектно-ориентированный язык. Но было бы ближе к правде сказть,

что Lisp является расширяемым языком, на котором легко могут быть написаны конструкции для объектно-ориентированного программирования.

Поскольку CLOS поставляется предварительно написанным, описание Lisp как объектно- ориентированного языка не является ложной рекламой. Однако было бы ограничением смотреть так на Lisp. Да, Lisp это объектно-ориентированный язык, но не потому, что он принял объектно ориентированную модель. Скроее, эта модель оказывается еще одной перестановкой абстракций, лежащих в основе Lisp. И чтобы доказть это, у нас есть CLOS, программа написанная на Lisp, которая делает Lisp объектно- ориентированным языком.

Цель этой главы - выявить связь между Lisp и объектно-ориентированным программированием, изучая CLOS как пример встроенного языка. Это также хороший способ понять сам CLOS: в конце концов, ничто не объясняет языковую функцию более эффективно, ем набросок её реализации. В разделе 7-6, таким образом объяснялись макросы. В следующем разделе дается аналогичный эскиз того, как создавать объектно-ориентированные абстракции поверх Lisp. Эта программа обеспечивает ориентир для описания CLOS в Разделах 25.3 - 25-6.

25.2 25-2 Объекты в простом/чистом Lisp

Мы можем лепить из Lisp множество видов языков. Существует прямое соответствие между концепциями объектно-ориентированного программирования и фундаментальными абстракциями Lisp. Размер CLOS имеет тенденцию скрывать этот факт. Итак, прежде чем посмотреть, что мы можем сделать с CLOS, давайте посмотрим, что мы можем сделать с простым Lisp. Многое из того, что мы хотем от объектно-ориентированного программирования, уже есть в Lisp. И мы можем получить остальное с удивительно небольшим кодом. В этом разделе мы определим объектную систему, достаточную для многих реальных приложений, на двух страницах кода. Объектно ориентированное программирование, как минимум, подразумевает

1. объекты, которые имеют свойства
2. и отвечают на сообщения,
3. и которые наследуют свойства и методы от своих родителей.

В Lisp уже есть несколько способов хранения коллекций свойств. Один из способов - представить объекты в виде хеш-таблиц и сохранить их свойства в виде записепй внутри них. Затем у нас есть доступ к отдельным свойствам через gethash:

```
(gethash 'color obj)
```

Поскольку функции являются объектами данных, мы также можем хранить их как свойства. Это означает, что у нас также могут быть методы; Вызов данного метода объекета означает передачу funcall свойства с этим именем:

```
(funcall (gethash 'move obj) obj 10)
```

Основываясь на этой идее мы можем определить Smalltalk стиль передачи сообщений:

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

так чтобы указать obj, чтобы он переместился на 10 мы можем сказать:

```
(tell obj 'move 10)
```

Фактически, единственный компонент отсутствующий в простом Lisp это наследование, и мы можем предоставить его элементарную версию в шести строках кода, определяющих рекурсивную версию gethash:

```
(defun rget (obj prop)
  (multiple-value-bind (val win) (gethash prop obj)
        (if win
              (values val win)
              (let ((par (gethash 'parent obj)))
                (and par (rget par prop))))))

(defun rget (obj prop)
   (some2 #'(lambda (a) (gethash prop a))
            (get-ancestors obj)))

(defun get-ancestors (obj)
   (labels ((getall (x)
                  (append (list x)
                            (mapcan #'getall
                                      (gethash 'parents x)))))
        (stable-sort (delete-duplicates (getall obj))
                       #'(lambda (x y)
                            (member y (gethash 'parents x))))))

(defun some2 (fn lst)
   (if (atom lst)
        nil
        (multiple-value-bind (val win) (funcall fn (car lst))
           (if (or val win)
                (values val win)
                (some2 fn (cdr lst))))))
```

Рисунок 25-1: Множественное наследование.

Если мы просто используем rget вместо gethash, мы получим наследование свойств и методов. Мы указываем объект-родитель следующим образом:

(setf (gethash 'parent obj) obj2)

Поскольку у нас только одно наследование у объекта может быть только один родитель(parent). НО мы можем иметь множественное наследование, сделав свойство parent списком, и определивg rget как показано на Рисунке 25-1. При одиночном наследовании, когда мы хотели получить какое-либо свойство объекта, мы просто рекурсивно искали его предков. Если сам объект не имел информации о желаемом свойстве, мы смотрели на его родителя и так далее. При множественном наследовании мы хотим выполнять поиск того же рода, но наша работа осложняется тем фактом, что предки объекта могут формировать граф, вместо простого списка. Мы не можем просто искать в этом графе в глубину. С множественным наследованием мы можем иметь иерархию показанную на Рисунке 25-2: a происходит от b и c, которые оба

происходят от d. Обход в глубину (точнее, в высоту), будет идти по a, b, d, c, d. Если бы желаемое свойство присутствовало как в d, так и c, мы бы

Figure 25-2: Multiple paths to a superclass.

получили значение хранящееся в d, а не то, которое хранится в c. Это нарушило бы принцип, согласно которому подклассы переопределяют значения по умолчанию, предоставленные их родителями. Если мы хотим реализовать обычную идею наследования, мы никогда не должны исследовать объект прежде исследования его потомков. В этом случае правильный порядок поиска будет: a, b, c, d. Как мы можем гарантировать, что поиск всегда вначале проверяет потомков? Самый простой способ - собрать всех предков исходного объекта, отсортировать этот список так, чтобы ни один объект не появился перед одним из своих потомков, а затем посмотреть на каждый элемент по очереди. Эта стратегия используется get-ancestors, которая возвращает правильно упорядоченный список объекта и его предков. Чтобы отсортировать список, get-ancestors вызывает stable-sort, а не sort, чтобы избежать возможного переупорядочивания параллельных предков. Как только список отсортирован, rget просто ищет первый объект с желаемым свойством. (Утилита some2 это версия some для использования с такими функциями как gethash, которые указывают на успех или неудачу во втором возвращаемом значении.) Список предков объекта проходит от наиболее специфического к наименее конкретному: если апельсин это плод цитрусовых, которыйй является фруктом, тогда список будет иметь порядок (orange citrus fruit). Когда у объекта несколько родителей, их приоритет идет слева направо. То есть если мы скажем

(setf (gethash 'parents x) (list y z))

тогда y будет рассматриваться перед z когда мы будем искать унаследованное свойство. Например, мы можем сказать что патриот-негодяй(patriotic scoundrel) это сначала негодяй (scoundrel), а потом патриот(patriot):

> (setq scoundrel (make-hash-table) patriot (make-hash-table) patriotic-scoundrel (make-hash-table)) #<Hash-Table C4219E>

```
(defun obj (&rest parents)
 (let ((obj (make-hash-table)))
    (setf (gethash 'parents obj) parents)
    (ancestors obj)
    obj))

(defun ancestors (obj)
 (or (gethash 'ancestors obj)
      (setf (gethash 'ancestors obj) (get-ancestors obj))))

(defun rget (obj prop)
 (some2 #'(lambda (a) (gethash prop a))
          (ancestors obj)))
```

Рисунок 25-3: Функции для создания объектов.

```
> (setf (gethash 'serves scoundrel) 'self
            (gethash 'serves patriot)             'country
            (gethash 'parents patriotic-scoundrel)
                        (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget patriotic-scoundrel 'serves)
SELF
T
```

Давайте внесем некоторые улучшения в эту скелетную систему. Мы могли бы начать с функции для создания объектов. Эта функция должна составлять список предков объекта во время создания объекта. Текущий код строит эти списки при выполнении запросов, но нет никаких причин не сделать это раньше. На рисунке 25-3 определена функция с именем obj, которая создает новый объект, сохраняя в нем список его предков. Чтобы воспользоваться преимуществами хранимых предков, мы также переопределяем rget.

Другое место для улучшения - это синтаксис вызовов сообщений. Само по себе сообщение(tell) является ненужным беспорядком, поскольку оно заставляет глаголы стоять на втором месте, а это означает, что наши программы больше не могут читаться как обычные префиксные выражения Lisp: (tell (tell obj 'find-owner) 'find-owner)

Мы можем избавиться от синтаксиса tell, определив каждое имя свойства как функцию, как показано на рисунке 25-4. Необязательный аргумент meth?, если истина(true), означает, что это свойство следует рассматривать как метод. В противном случае оно будет рассматриваться как слот, а значение, полученное с помощью rge, будет просто возвращено. После того, как мы определили название любого вида свойств,

```
(defmacro defprop (name &optional meth?)
  `(progn
      (defun ,name (obj &rest args)
        ,(if meth?
             `(run-methods obj ',name args)
             `(rget obj ',name)))
      (defsetf ,name (obj) (val)
        `(setf (gethash ',',name ,obj) ,val))))

(defun run-methods (obj name args)
  (let ((meth (rget obj name)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))
```

Рисунок 25-4: Функциональный синтаксис.

```
(defprop find-owner t)
```

мы можем обратиться к нему с помощью вызова функции, и наш код снова будет выглядеть как Lisp:

```
(find-owner (find-owner obj))
```

Наш предыдущий пример теперь стал более читабельным:

```
> (progn
        (setq scoundrel (obj))
        (setq patriot (obj))
        (setq patriotic-scoundrel (obj scoundrel patriot))
        (defprop serves)
        (setf (serves scoundrel) 'self)
        (setf (serves patriot) 'country)
        (serves patriotic-scoundrel))
SELF
T
```

В текущей реализации объект может иметь не более одного метода для определенного имени. У объекта либо есть собственный метод, либо он наследуется. Было бы удобно иметь больше гибкости в этом вопросе, чтобы мы могли соединять локальный и унаследованный методы. Например, мы могли бы захотеть, чтобы метод move некоторых объектов, был методом move его родителя, но с некоторым дополнительным кодом, выполняемым до него, или после.

Чтобы предоставить такие возможности, мы изменим нашу программу, включив в неё методы before-, after- и around- . Before- методы позволяют нам сказать "Но сначала, сделаете это.". Они вызываются впереди всех, как прелюдия к остальной части вызова метода. After- методы позволяют нам сказать "P.S. Сделай то тоже." Они вызываются, позже всех, как эпилог вызова метода. Между ними, мы запускаем то, что раньше было методом, и теперь называется первичным методом. Значение вызова этого метода будет возвращаться, даже если after- методы будут вызваны позже. Before- и after-методы позволяют нам обернуть новое поведение вокруг вызова

первичного метода. Around-методы обеспечивают более радикальный способ сделать тоже самое. Если существует метод around-, он будет вызываться вместо первичного метода. Затем, по своему усмотрению, метод around- может сам вызвать первичный метод (через call-next, который будет представлен на Рисунке 25-7). Чтобы разрешить вспомогательные методы, мы изменим run-методы и rget как на Рисунках 25-5 и 25-6. В предыдущей версии, когда мы запускали какой-то метод объекта, мы просто запускали одну функцию: наиболее конкретный первичный метод. Мы запускали первый метод, с которым мы сталкивались при поиске по списку предков. С вспомогательными методами, вызывающая последовательность теперь выглядит следующим образом:

1. Наиболее конкретный метод around- если он есть.

2. Otherwise, in order:

(a) Все методы before-, от наиболее конкртетных до наименее конкретных. (b) Наиболее конкретный первичный метод (то что мы привыкли вызывать). (c) Все методы after-, от наименее конкретных до наиболее конкретных.

Обратите внимание, что вместо того, чтобы быть единой функцией, метод становиться структурой из четырех частей. Чтобы определить (основной/первичный) метод, вместо того чтобы сказать:

(setf (gethash 'move obj) #'(lambda ...))

мы скажем:

(setf (meth-primary (gethash 'move obj)) #'(lambda ...))

По этой и другим причинам, нашим следующим шагом должно стать определение макроса для определения методов. На Рисунке 25-7 показано определение такого макроса. Большая часть этого кода занята реализацией двух функций, которые методы могут использовать для ссылки на другие методы. Методы around- и primary могут использовать call-next для вызова следующего метода, который должен был бы выполниться, если бы текущего метода не существовало. Например, если текущий запущенный метод является единственным around- методом,

```
(defstruct meth around before primary after)

(defmacro meth- (field obj)
      (let ((gobj (gensym)))
       `(let ((,gobj ,obj))
          (and (meth-p ,gobj)
                  (,(symb 'meth- field) ,gobj)))))

(defun run-methods (obj name args)
      (let ((pri (rget obj name :primary)))
       (if pri
            (let ((ar (rget obj name :around)))
             (if ar
                   (apply ar obj args)
                   (run-core-methods obj name args pri)))
            (error "No primary ~A method for ~A." name obj))))

(defun run-core-methods (obj name args &optional pri)
      (multiple-value-prog1
       (progn (run-befores obj name args)
              (apply (or pri (rget obj name :primary))
                     obj args))
       (run-afters obj name args)))

(defun rget (obj prop &optional meth (skip 0))
      (some2 #'(lambda (a)
                  (multiple-value-bind (val win) (gethash prop a)
                    (if win
                         (case meth (:around (meth- around val))
                                    (:primary (meth- primary val))
                                    (t (values val win))))))
            (nthcdr skip (ancestors obj))))
```

Рисунок 25-5: Вспомогательные методы.

следующим методом будет сендвич(набор следующих друг за другом) из before-методов, наиболее конкретного первичного метода и методов after-. В пределах наиболее конкретного метода, следующий метод будет вторым наиболее конкретным методом. Поскольку поведение call-next зависит от того, где он вызывается, он никогда не определяется глобально с помощью defun, а определяется локально в каждом методе, определённом с помощью defmeth.

```
(defun run-befores (obj prop args)
 (dolist (a (ancestors obj))
    (let ((bm (meth- before (gethash prop a))))
       (if bm (apply bm obj args)))))

(defun run-afters (obj prop args)
 (labels ((rec (lst)
               (when lst
                  (rec (cdr lst))
                  (let ((am (meth- after
                                   (gethash prop (car lst)))))
                    (if am (apply am (car lst) args))))))
    (rec (ancestors obj))))
```

Рисунок 25-6: Вспомогательные методы (продолжение).

Метод around- или primary может использовать next-p, чтобы проверить, существует ли следующий метод. Например, если текущий метод является первичным методом объекта не имеющим родителей, то следующего метода не будет. Так как call-next выдает ошибку, когда нет следующего метода, обычно следует вызвать next-p чтобы проверить его наличие. Как и call-next, next-p определяется локально в отдельных методах. Новый макрос defmeth используется следующим образом. Если мы просто хотим определить метод area объекта rectangle, мы скажем

(setq rectangle (obj)) (defprop height) (defprop width) (defmeth (area) rectangle (r) (* (height r) (width r)))

Теперь площадь(area) экземпляра рассчитывается как метод класса:

> (let ((myrec (obj rectangle))) (setf (height myrec) 2 (width myrec) 3) (area myrec)) 6

```
(defmacro defmeth ((name &optional (type :primary))
                   obj parms &body body)
      (let ((gobj (gensym)))
       `(let ((,gobj ,obj))
          (defprop ,name t)
          (unless (meth-p (gethash ',name ,gobj))
            (setf (gethash ',name ,gobj) (make-meth)))
          (setf (,(symb 'meth- type) (gethash ',name ,gobj))
                ,(build-meth name type gobj parms body)))))

(defun build-meth (name type gobj parms body)
      (let ((gargs (gensym)))
       `#'(lambda (&rest ,gargs)
            (labels
              ((call-next ()
                     ,(if (or (eq type :primary)
                              (eq type :around))
                          `(cnm ,gobj ',name (cdr ,gargs) ,type)
                          '(error "Illegal call-next.")))
                   (next-p ()
                     ,(case type
                       (:around
                        `(or (rget ,gobj ',name :around 1)
                             (rget ,gobj ',name :primary)))
                       (:primary
                        `(rget ,gobj ',name :primary 1))
                       (t nil))))
              (apply #'(lambda ,parms ,@body) ,gargs)))))

(defun cnm (obj name args type)
      (case type
       (:around (let ((ar (rget obj name :around 1)))
                  (if ar
                      (apply ar obj args)
                      (run-core-methods obj name args))))
       (:primary (let ((pri (rget obj name :primary 1)))
                  (if pri
                      (apply pri obj args)
                      (error "No next method."))))))
```

Рисунок 25-7: Определение методов.

```
(defmacro undefmeth ((name &optional (type :primary)) obj)
  `(setf (,(symb 'meth- type) (gethash ',name ,obj))
         nil))
```

Рисунок 25-8: Удаление методов.

В более сложном примере предположим, что мы определили метод backup для объекта filesystem:

(setq filesystem (obj)) (defmeth (backup :before) filesystem (fs) (format t "Remember to mount the tape.~%")) (defmeth (backup) filesystem (fs) (format t "Oops, deleted all your files.~%") 'done) (defmeth (backup :after) filesystem (fs) (format t "Well, that was easy.~%"))

Обычная последовательность вызовов будет следующей:

> (backup (obj filesystem)) Remember to mount the tape. Oops, deleted all your files. Well, that was easy. DONE

Позднее мы хотим узнать, сколько времени занимает выполнения резервирования(backup), поэтому определим следующий метод:

(defmeth (backup :around) filesystem (fs) (time (call-next)))

Теперь, всякий раз, когда вызвается backup для потомка filesystem (если не вмешиваются более конкретные around-методы), будет вызываться наш метод around-. Он вызовет код который обычно запускается при вызове backup, но в вызове time. Значение возвращаемое time, будет возвращено как значение вызова backup:

> (backup (obj filesystem)) Remember to mount the tape. Oops, deleted all your files. Well, that was easy. Elapsed Time = .01 seconds DONE

Как только мы закончим подсчитывать время выполнения метода backup, мы захотим удалить метод around-. Это можно сделать, вызвав undefmeth (Рисуноке 25-8), который принимает те же первые два аргумента, что и defmeth:

(undefmeth (backup :around) filesystem)

Еще одна вещь, которую мы могли бы изменить - это список родителей объекта. Но после любого такого изменения мы должны также обновить список предков объекта и всех его потомков. Пока что у нас нет возможности получить из объекта его потомков, поэтому мы также должны добавить свойство children(потомки). На рисунке 25-9 приведен код для работы с родителями и потомками. Вместо того, чтобы получать родителей и детей через gethash, мы используем операторы parents и children. Последний является макросом и поэтому прозрачен для setf. Первый это функция, инверсия которой определяется defsetf как set-parents, которая делает все необходимое для поддержания согласованности в новом двусвязном мире. Чтобы обновить предков всех объектов в поддереве, set-parents вызывает maphier, который похож на mapc для иерархий наследования. Как и mapc вызывает функцию для каждого элемента списка, maphier вызывает функцию для объекта и всех его потомков. Если они не образуют правильное дерево, функция может вызываться несколько раз для некоторых объектов. Здесь это безвредно, потому что get-ancestors делает тоже самое, когда вызывается несколько раз. Теперь мы можем изменить иерархию наследования, просто используя setf для родителей объекта:

> (progn (pop (parents patriotic-scoundrel)) (serves patriotic-scoundrel)) COUNTRY T

Когда иерархия изменена, затронутые списки потомков и предков будут обновлены автоматически. (Свойство потомки(children) не предназначено для прямого манипулирования, но оно бы могло быть, если бы мы определили set-children аналогично set-parents.) Последняя функция на рисунке 25-9 это obj переопределенная для использования нового кода. В качестве окончательного усовершенствования нашей системы, мы дадим возможность указать новые способы объединения методов. В настоящее время, единственный первичный метод, является наиболее конкретным(хотя он может вызвать другие используя call-next). Вместо этого нам может потребоваться объединить результаты первичных методов каждого из предков объекта. Например, предположим, что my-orange это потомок orange, который является потомком citrus. Если метод props возвращает (round acidic) для citrus и (orange sweet) для orange, и (dented) для my-orange, было бы удобно иметь возможность (props my-orange) возвращать объединение всех этих значений: (dented orange sweet round acidic).

```
(defmacro children (obj)
   `(gethash 'children ,obj))

(defun parents (obj)
   (gethash 'parents obj))

(defun set-parents (obj pars)
   (dolist (p (parents obj))
       (setf (children p)
                (delete obj (children p))))
   (setf (gethash 'parents obj) pars)
   (dolist (p pars)
       (pushnew obj (children p)))
   (maphier #'(lambda (obj)
                      (setf (gethash 'ancestors obj)
                            (get-ancestors obj)))
               obj)
   pars)

(defsetf parents set-parents)

(defun maphier (fn obj)
   (funcall fn obj)
   (dolist (c (children obj))
       (maphier fn c)))

(defun obj (&rest parents)
   (let ((obj (make-hash-table)))
       (setf (parents obj) parents)
       obj))
```

Рисунок 25-9: Поддержание родительских и дочерних ссылок.

Мы могли бы иметь это, если бы позволили методам применять некоторую функцию к значениям своих первичных методов, вместо того, чтобы просто возвращать значение наиболее конкретного метода. Рисунок 25-10 содержит макрос, который позволяет нам определить способ объединения методов и новую версию run-core-methods, которые могут выполнять объединение методов. Мы определяем форму комбинации для метода через defcomb, который принимает имя метода и второй аргумент, описывающий желаемую комбинацию.

```
(defmacro defcomb (name op)
      `(progn
        (defprop ,name t)
        (setf (get ',name 'mcombine)
                ,(case op
                    (:standard nil)
                    (:progn '#'(lambda (&rest args)
                                  (car (last args))))
                    (t op)))))

(defun run-core-methods (obj name args &optional pri)
      (let ((comb (get name 'mcombine)))
       (if comb
           (if (symbolp comb)
                  (funcall (case comb (:and #'comb-and)
                                      (:or #'comb-or))
                           obj name args (ancestors obj))
                  (comb-normal comb obj name args))
           (multiple-value-prog1
                 (progn (run-befores obj name args)
                        (apply (or pri (rget obj name :primary))
                               obj args))
                 (run-afters obj name args)))))

(defun comb-normal (comb obj name args)
      (apply comb
            (mapcan #'(lambda (a)
                        (let* ((pm (meth- primary
                                          (gethash name a)))
                               (val (if pm
                                        (apply pm obj args))))
                          (if val (list val))))
                    (ancestors obj))))
```

Рисунок 25-10: Комбинация Методов.

Обычно этот второй аргумент должен быть функцией. Тем не менее, он также может быть одним из :progn, :and, :or или :standard. С первыми тремя первичные методы будут объединены как будто согласно согласно соответствующим оператором, тогда как :standard указывает, что мы хотим традиционный способ выполнения операторов.

```
(defun comb-and (obj name args ancs &optional (last t))
 (if (null ancs)
        last
        (let ((pm (meth- primary (gethash name (car ancs)))))
           (if pm
                 (let ((new (apply pm obj args)))
                    (and new
                             (comb-and obj name args (cdr ancs) new)))
                 (comb-and obj name args (cdr ancs) last)))))

(defun comb-or (obj name args ancs)
 (and ancs
         (let ((pm (meth- primary (gethash name (car ancs)))))
            (or (and pm (apply pm obj args))
                   (comb-or obj name args (cdr ancs))))))
```

Рисунок 25-11: Комбинация методов (продолжение).

Центральной функцией на рисунке 25-10 является новый run-core-methods. Если вызываемый метод не имеет свойства mcombine, то вызов метода продолжается как и раньше. В противном случае mcombine метода это либо функция (такая как +) или ключевое слово (такое как :or). В перовм случае функция просто применяется к списку значений возвращенных всеми первичными методами.[1] Во втором случае, мы используем функцию связанную с ключевым словом, для итерации по первичным методам.

Операторы and и or должны обрабатываться специальнго, как показано на рисунке 25-11. Они получают специальное обращение не только потому, что они являются специальными формами, но и потому что они вычисляются по укороченной схеме.:

> (or 1 (princ "wahoo")) 1

Здесь ничего не печатается, потому что or возвращается как только оно видит не нулевой(не nil) аргумент. Точно так же первичный метод или комбинация не будут вызваны, если более конкретный метод вернет истину(true). Чтобы обеспечить такую укороченную схему для and и or, мы используем различные функции comb-and и comb-or. Чтобы реализовать наш предыдущий пример, мы бы написали:

```
(setq citrus (obj))
(setq orange (obj citrus))

(setq my-orange (obj orange))

(defmeth (props) citrus (c) '(round acidic))
(defmeth (props) orange (o) '(orange sweet))
(defmeth (props) my-orange (m) '(dented))

(defcomb props #'(lambda (&rest args) (reduce #'union args)))
```

[1] Более сложная версия этого кода может использовать reduce чтобы избежать создания списка(consinga) здесь.

после чего props будет возвращать объединение всех значений первичных методов:[2]

```
> (props my-orange)
(DENTED ORANGE SWEET ROUND ACIDIC)
```

Между прочим, этот пример предлалгает выбор, который вы имеете только при объектно-ориентированном программировании на Lisp: хранить информацию в слотах или методах.

Впоследствии, если мыхотим, чтобы метод props вернулся к повдению по умолчанию, мы просто установим комбинацию методов обратно в standard:

```
> (defcomb props :standard)
NIL
> (props my-orange)
(DENTED)
```

Обратите внимание, что методы before- и after- выполняются только в стандартной комбинации методов. Однако, методы around- работают так же, как и раньше.

Программа, представленная в этом разделе, задумана как модель, а не как реальная основа для объектно-ориентированного программирования. Этот код был написан для краткости, а не для эффективности. Тем не менее, это по крайней мере рабочая модель, и поэтому может быть использован для экспериментов и прототипов. Если вы захотите использовать программу для этих целей, одно небольшое изменение сделает её гораздо более эффективной: не вычисляйте и не сохраняйте списки предков для объектов с одним родителем.

25.3 25-3 Классы и Экземпляры

Программа в предыдущем разделе была написана так, чтобы походить на CLOS на столько близко, насколько могла бы такая маленькая программа. Поняв её, мы уже продвинулись к пониманию CLOS. В следующих нескольких разделах мы рассмотрим сам CLOS.

В нашем наброске мы не делали синтаксического различия между классами и экземплярами или между слотами и методами. В CLOS, мы используем макрос defclass для определения класса и одновременно объявляем слоты в списке:

2

(defclass circle () (radius center))

Это выражение говорит о том, что у класса circle нет суперклассов, а есть два слота, radius и center. Мы можем создать экземпляр класса circle сказав:

(make-instance 'circle)

К сожалению мы не определили способ обращаения к слотам circle, поэтому любой созданный нами экземпляр будет довольно инертным. Чтобы получить слот мы определяем функцию доступа для него:

(defclass circle () ((radius :accessor circle-radius) (center :accessor circle-center)))

Теперь если мы создаем экзепляр circle, мы можем установить его слоты radius и center исползуя setf с соответствующими функциями доступа:

[2] Так как объединение функций для props вызывает union, элементы списка не обязательно будут идти в этом порядке.

> (setf (circle-radius (make-instance 'circle)) 2) 2

Мы можем выполнить этот тип инициализации прямо в вызове make-instance слоты позволяющие это:

(defclass circle () ((radius :accessor circle-radius :initarg :radius) (center :accessor circle-center :initarg :center)))

Ключевое слово :initarg в определении слота говорит, что следующий аргумент должен стать ключевым параметром в make-instance. Значение параметра ключевого слова станет начальным значением слота:

> (circle-radius (make-instance 'circle :radius 2 :center '(0 . 0))) 2

Объявляя :initform, мы также можем определить слоты, которые инициализируются сами. Видимый слот класса shape

(defclass shape () ((color :accessor shape-color :initarg :color) (visible :accessor shape-visible :initarg :visible :initform t)))

по умолчанию будет установлено значение t: > (shape-visible (make-instance 'shape)) T Если в слоте есть и initarg, и initform, initarg имеет приоритет, если он указан: > (shape-visible (make-instance 'shape :visible nil)) NIL Слоты наследуются экземплярами и подклассами. Если класс имеет более одного суперкласса, он наследует объединение из их слотов. Так что, если мы определим класс screen-circle как подкласс circle и shape, (defclass screen-circle (circle shape) nil) тогда экземпляры screen-circle будут иметь четыре слота, по два унаследованых от каждого родителя. Обратите внимание, что этот класс не должен создавать никаких новых собственных слотов; этот класс существует только для того, чтобы обеспечить создания того, что наследует как от circle, так и от shape. Функция доступа и initargs работают для экземпляров screen-circle точно так же, как и для экземпляров circle или shape: > (shape-color (make-instance 'screen-circle :color 'red :radius 3)) RED Мы можем заставить каждый screen-circle иметь некоторый начальны цвет по умолчанию, указав initform для этого слота в определении класса defclass: (defclass screen-circle (circle shape) ((color :initform 'purple))) Теперь экземпляры screen-circle будут purple по умолчанию, > (shape-color (make-instance 'screen-circle)) PURPLE хотя все еще возможно инициализировать слот, в противном случае, задав явный ицинцализирующий аргумент :color initarg. В нашем наброске объектно-ориентированного программирования, экземпляры наследовали значения непсредственно из слотов в своих же родительстких классах. В CLOS, экземпляры не имеют слотов таких же как классы. Мы определяем наследуемое значение по умолчанию для экземпляров, определяя initform в родительском классе. В некотором смысле, это более гибко, потому что initform может быть не только константой, но и выражением, которое возвращает разное значение при каждом его вычислении:

(defclass random-dot () ((x :accessor dot-x :initform (random 100)) (y :accessor dot-y :initform (random 100))))

Каждый раз, когда мы создаем экземпляр случайной точки, её координаты x и y будут случайным целым числом от 0 до 99:

> (mapcar #'(lambda (name) (let ((rd (make-instance 'random-dot))) (list name (dot-x rd) (dot-y rd)))) '(first second third)) ((FIRST 25 8) (SECOND 26 15) (THIRD 75 59))

В нашем наброске мы также не делали различий между слотами, значения которых доложны были варьироваться от экземпляра к экземпляру, и слотами, которые долж-

ны быть постоянными по всему классу. В CLOS мы можем указать, что некоторые слоты должны быть общими, то есть их значение одинаково для каждого экземпляра. Мы сделаем это объявив слот имеющий :allocation :class. (Альтернативно это для слота должно быть :allocation :instance, но так как это значение по умолчанию, нет необходимости говорить об этом явнор.) Например, если все совы(owls) ведут ночной образ жизни(nocturnal), то мы можем сделать слот nocturnal класса owl разделяемым слотом, и задать ему начальное значение t:

(defclass owl () ((nocturnal :accessor owl-nocturnal :initform t :allocation :class)))

Теперь каждый экземпляр класса owl будет наследовать этот слот:

> (owl-nocturnal (make-instance 'owl)) T

Если мы изменим "локальное" значение этого слота в некотором экземпляре, мы фактически изменим значение, хранящееся в классе:

> (setf (owl-nocturnal (make-instance 'owl)) 'maybe) MAYBE > (owl-nocturnal (make-instance 'owl)) MAYBE

Это может вызвать некоторую путаницу, поэтому мы можем сделать такой слот доступным только для чтения. Когда мы определяем функцию доступа для слота, мы создаем способ чтения и записи значения слота. Если мы хотим, чтобы значение было читаемым, но не доступным для записи, мы можем сделать это, предоставив слоту только функцию чтения, а не полноценную функцию доступа:

(defclass owl () ((nocturnal :reader owl-nocturnal :initform t :allocation :class)))

Тепарь попытки изменить слот nocturnal для экземпляра приведут к генерации ошибки:

> (setf (owl-nocturnal (make-instance 'owl)) nil) >>Error: The function (SETF OWL-NOCTURNAL) is undefined.

25.4 25-4 Методы

Наш набросок подчеркивал сходство между слотами и методами в языке, которые предоставляет лексические замыкания. В нашей программе основной/первичный метод хранился и наследовался так же, как и значения слота. Единственная разница между слотом и методом заключалась в том, что определение имени как слота

(defprop area)

делало area функцией, которая будет просто извлекать и возвращаеть значение, определяя его как метод

```
(defprop area t)
```

делало area функцией, которая после получения значения, вызывась бы (funcall) с этим аргументом.

В CLOS функциональные блоки по прежнему называются методами, и их можно определить так, чтобы каждый из них казался свойством некоторого класса. Здесь мы определяем метод area для класса circle:

```
(defmethod area ((c circle))
  (* pi (expt (circle-radius c) 2)))
```

Список параметров для этого метода говорит, что это функция одного аргумента, которая применяется к экземплярам класса circle.

Мы вызываем этот метод как функцию, как в нашем наброске:

```
> (area (make-instance 'circle :radius 1))
3.14...
```

Мы также можем определить методы, которые принимают дополнительные аргументы:

```
(defmethod move ((c circle) dx dy)
  (incf (car (circle-center c)) dx)
  (incf (cdr (circle-center c)) dy)
  (circle-center c))
```

Если мы вызовем этот метод для экземпляра circle, его центр(center) будет сдвинут на dx,dy :

> (move (make-instance 'circle :center '(1 . 1)) 2 3) (3.4)

Значение возвращаемое методом, отражает новую позицию circle. Как и в нашем наброске, если есть метод для экземпляра класса, и для суперкласса этого класса, запускается самый конкретный метод. Так что, если unit-circle это подкласс circle, со следующим методом area

```
(defmethod area ((c unit-circle)) pi)
```

тогда этот метод, а не более общий, будет выполняться, когда мы вызовем area для экземпляра unit-circle. Когда у класса несколько суперклассов, их приоритет идет слева на право. Определив класс patriotic-scoundrel следующим образом

```
(defclass scoundrel nil nil)
(defclass patriot nil nil)
(defclass patriotic-scoundrel (scoundrel patriot) nil)
```

мы уточняем, что патриотичные негодяи(patriotic scoundrels) являются сначала негодяями (scoundrels), а потом патриотами(patriots). Когда есть подходящий метод для обоих суперклассов,

```
(defmethod self-or-country? ((s scoundrel))
  'self)

(defmethod self-or-country? ((p patriot))
  'country)
```

будет выполняться метод класса негодяев(scoundrel):

> (self-or-country? (make-instance 'patriotic-scoundrel)) SELF

Пока приведенные примеры поддерживают иллюзию, что методы CLOS являются методами некоторого объекта. На самом деле они являются чем-то более общим. В списке параметров метода move, элемент (c circle) называется особенным(специализированным) параметром; он говорит, что этот метод применяется когда первый аргумент для move является экземпляром класса circle. В методе CLOS, может быть специализировано более одного параметра. Следующий метод имеет два специализированных и один не обязательный не специализированный параметр:

```
(defmethod combine ((ic ice-cream) (top topping)
                                &optional (where :here))
   (append (list (name ic) 'ice-cream)
```

```
(list 'with (name top) 'topping)
(list 'in 'a
      (case where
          (:here 'glass)
          (:to-go 'styrofoam))
      'dish)))
```

Он вызывается, когда первые два аргумента, которые нужно объединить, являются экземплярами ice-cream(мороженное) и topping(начинка), соответственно. Если мы определим некоторые минимальные классы для создания экземпляров

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) nil)
(defclass topping (stuff) nil)
```

тогда мы сможем определить и запустить этот метод: > (combine (make-instance 'ice-cream :name 'fig) (make-instance 'topping :name 'olive) :here) (FIG ICE-CREAM WITH OLIVE TOPPING IN A GLASS DISH) Когда методы специализируются более чем одним из их параметров, трудно продолжать рассматривать их как свойства классов. Наш метод combine относиться к классу ice-cream или классу topping? В CLOS модель объектнов, отвечающих на сообщения, просто испаряется. Эта модель кажеться естественной, пока мы вызываем методы, говоря что-то вроде:

(tell obj 'move 2 3)

Здесь мы явно вызваем метод move объекта obj. Но как только мы отбросим этот синтаксис в пользу функционального эквивалента:

```
(move obj 2 3)
```

тогда мы имеем определениее move, такое что его поведение диспетчеризируется в зависимости от первого аргумента, т.е. просматривает первый аргумент и вызывает соответствующий метод.

После того, как мы сделали этот шаг, возникает вопрос: почему разрешается проводить диспетчеризацию только по первому аргументу? CLOS отвечает: действительно, почему? В CLOS, методы могут специализироваться любым количеством своих параметров - и не только на определяемых пользователем классах, но и на типах Common Lisp,[3] и даже на отдельных объектах. Вот метод combine который применяется к строкам(string):

```
(defmethod combine ((s1 string) (s2 string) &optional int?)
   (let ((str (concatenate 'string s1 s2)))
        (if int? (intern str) str)))
```

Это означает не только то, что методы больше не являются свойствами классов, но и то, что мы можем использовать методы вообще без определения классов.

> (combine "I am not a " "cook.") "I am not a cook."

Здесь второй параметр специализируется символом palindrome(палиндром):

```
(defmethod combine ((s1 sequence) (x (eql 'palindrome))
                                  &optional (length :odd))
```

[3] Или точнее, на типах подобных классам, которые определяет CLOS параллельно с иерархией типов Common Lisp.

```
(concatenate (type-of s1)
                        s1
                        (subseq (reverse s1)
                                        (case length (:odd 1) (:even 0)))))
```

Этот конкретный метод делает палиндромы последовательностей любого вида:[4]

```
> (combine '(able was i ere) 'palindrome) (ABLE WAS I ERE I WAS ABLE)
```

Здесь у нас больше нет объектно-ориентированного программирования. а есть нечто большее. CLOS разработан с пониманием того, что под методами существует такая концепция диспетчеризации(распределения вызовов), которая может быть сделана для более чем одного аргумента и может основываться больше чем на классе аргументов. Когда методы построены на этом более общем понятии, они становятся независимыми от отдельных классов. Вместо того, чтобы концептуально придерживаться классов, методы теперь придерживаются других методов с теми же именами. В CLOS такой набор методов называется обобщенной/родовой/общей функцией(generic function). Все наши методы combine неявно определяют обобщенную функцию combine.

Мы можем определить обобщенные функции явно с помощью макроса defgeneric. Нет необходимости вызывать defgeneric для определения обобщенной функции, но оно может быть удобным местом для размещения документации или какой-то защитной сеткой от ошибок. Здесь мы делаем обе вещи:

```
(defgeneric combine (x y &optional z)
   (:method (x y &optional z)
         "I can't combine these arguments.")
   (:documentation "Combines things."))
```

Поскольку метод, указанный здесь для объединения, не специализирует ни один из его аргументов, он будет вызываться в том случае, если ни один другой метод не применим.

```
> (combine #'expt "chocolate")
"I can't combine these arguments."
```

Ранее, этот вызов вызывал бы ошибку.

Обобщенные функции накладывают одно ограничение, которого у нас небыло, когда методы являлись свойствами объектов: когда все методы с одинаковым именем объединяются в одну обобщенную функцию, их списоки параметров должны быть согласованы. Вот почему все наши методы combine имеют дополнительный необязательный параметр. После определения первого метода combine, который может принимать до трех аргументов, это вызвало бы ошибку, если бы мы попытались определить другой метод, который бы принимал только два параметра.

CLOS требует, чтобы списки параметров всех методов с одинаковыми именами были конгруэнтными(соответствующими). Два списка параметров являются соответствующими, если они имеют одинаковое количество обязательных параметров, одинаковое количество необязательных параметров и совместимое использование &rest и &key. Фактические параметры ключевые слова, принимаемые различными методами,

[4] В одной(в остальном очень хорошей) реализации Common Lisp, concatenate не принимает cons в качестве первого аргумента, поэтому там этот вызов работать не будет.

не обязательно должны быть одинаковыми, но defgeneric может настаивать на том, что все его методы принимают определенный минимальный набор. Следующие пары списков параметров являются конгруэнтными(соответствующими):

```
(x)                    (a)
(x &optional y) (a &optional b)
(x y &rest z)          (a b &rest c)
(x y &rest z)          (a b &key c d)
```

а следующие пары таковыми не являются:

```
(x)                    (a b)
(x &optional y) (a &optional b c)
(x &optional y) (a &rest b)
(x &key x y)           (a)
```

Переопределение методов аналогично переопределению функций. Поскольку специализироваными могут быть только обязательные параметры, каждый метод уникально идентифицируется своей обобщенной функцией и типами его обязательных параметров. Если мы определяем другой метод с теми же специализациями, он перезаписывает исходный. Итак, сказав:

```
(defmethod combine ((x string) (y string)
                                &optional ignore)
  (concatenate 'string x "+"y))
```

мы переопределяем то, что combine делает, когда два первых аргумента являются строками(string).

К сожалению, если вместо переопределения метода мы хотим удалить его, встроенной функции обратной defmethod - нет. К счастью, это Lisp, поэтому мы можем его написать

```
(defmacro undefmethod (name &rest args)
   (if (consp (car args))
        (udm name nil (car args))
        (udm name (list (car args)) (cadr args))))

(defun udm (name qual specs)
   (let ((classes (mapcar #'(lambda (s)
                                      `(find-class ',s))
                                  specs)))
       `(remove-method (symbol-function ',name)
                       (find-method (symbol-function ',name)
                                              ',qual
                                              (list ,@classes)))))
```

Рисунок 25-12: Макрос для удаления методов.

Детали того, как можно удалить метод в ручную, приведены в реализации undefmethod на Рисунке 25-12. Мы используем этот макрос, передавая аргументы аналогичные тем, котоыре мы передаем в defmethod, за исключением того, что вместо того, чтобы передавать полный списко параметров в качестве второго и

третьего аргументов, мы передаем просто названия классов требуемых параметров. Итак, чтобы удалить метод combine для двух строк(string), мы скажем:

```
(undefmethod combine (string string))
```

Неспециализированные аргументы неявно относятся к классу t, поэтому, если бы мы определили метод с обязательными, но неспециализированными параметрами:

```
(defmethod combine ((fn function) x &optional y)
  (funcall fn x y))
```

мы бы могли избавиться от него, сказав

```
(undefmethod combine (function t))
```

Если мы хотим удалить всю обобщенную функцию, мы можем сделать это так же, как мы бы удаляли определение любой функции, вызывав fmakunbound:

```
(fmakunbound 'combine)
```

25.5 25-5 Вспомогательные Методы и Комбинации

Вспомогательные методы работали в нашем наброске в основном так же как они работаю и в CLOS. До сих пор мы видели только первичные методы, но у нас также могут быть методы before-, after- и around-. Такие вспомогательные методы определяются путем добавления ключевого слова после имени метода в вызове defmethod. Если мы определим первичный метод speak для класса speaker следующим образом:

```
(defclass speaker nil nil)

(defmethod speak ((s speaker) string)
  (format t "~A" string))
```

Затем вызов speak с экземпляром speaker просто напечатает второй аргумент:

> (speak (make-instance 'speaker) "life is not what it used to be") life is not what it used to be NIL

Определим подкласс intellectual, который оборачивает методы before- и after- вокруг первичного метода speak,

```
(defclass intellectual (speaker) nil)

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

мы можем создать подкласс ораторов(speakers), у которых всегда есть последнее (и первое) слово:

> (speak (make-instance 'intellectual) "life is not what it used to be") Perhaps life is not what it used to be in some sense NIL

В стандартной комбинации методов, методы вызываются так, как описано в нашем наброске% все методы before-, сначала наиболее конкретные, затем наиболее конкретный первичный метод, затем все методы after, где самый конкретный вызывается

последним. Поэтому, если мы определим методы before- или after- для суперкласса speaker,

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

они будут вызваны в середине "сендвича" из методов:

> (speak (make-instance 'intellectual) "life is not what it used to be") Perhaps I think life is not what it used to be in some sense NIL

Независимо от того, что вызываются методы before- или after-, значение возвращаемое обобщенной функцией будет значением наиболее конкретного первичного метода - в данном случае nil возвращаемый функцией format. Оно измениться если есть методы around-. Если у одного из классов в семейном дереве объектов есть метод around- или точнее, если есть метод around- специализированный для аргументов передаваемых в обобщенную функцию- сначала будет вызван метод around-, а остальные методы будут выполняться только в том случае, если метод around- решит ими воспользоваться. Как и в нашем наброске, метод around- или primary(первичный) может вызвать следующий метод вызывав функцию: функцию, которую мы определили как call-next в CLOS называется call-next-method. Существует также next-method-p, аналог нашего next-p. С помощью методов around- мы можем определить другой подкласс speaker, который является более осмотрительным:

```
(defclass courtier (speaker) nil)

(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A? " string)
  (if (eq (read) 'yes)
        (if (next-method-p) (call-next-method))
        (format t "Indeed, it is a preposterous idea.~%"))
  'bow)
```

Когда первым аргументом для speak является экземпляр класса courtier, язык придворного(courtier) теперь защищен методом around-:

> (speak (make-instance 'courtier) "kings will last") Does the King believe that kings will last? yes I think kings will last BOW > (speak (make-instance 'courtier) "the world is round") Does the King believe that the world is round? no Indeed, it is a preposterous idea. BOW

Обратите внимание, что в отличии от методов before- и after-, значение возвращаемое методом around-, возвращается как значение обобщенной функции.

Как правило, методы выполняются, как показано в этой схеме, которая перепечатана из раздела 25-2:

1. . Наиболее конкретный метод around- если он есть.

2. Иначе, в порядке:

(a) Все методы before-, от наиболее конкртетных до наименее конкретных. (b) Наиболее конкретный первичный метод (то что мы привыкли вызывать). (c) Все методы after-, от наименее конкретных до наиболее конкретных.

Этот способ объединения методов называется стандартным. Как и в нашем наброске, можно определять методы которые комбирируются другими способами: например, для обобщенной функции возвращать сумму всех применимых первичных методов.

В нашей программе, мы указали, как объединять методы, вызывая defcomb. По умолчанию, методы будут объединяться как в приведенной выше схеме, но скажем, например,

```
(defcomb price #'+)
```

мы могли бы заставить функцию price возвращать сумму всех применимых первичных методов.

В CLOS это называется комбинацией операторных методов. Как и в нашей программе, такая комбинация методов, может быть понята так, как если бы она приводила к вычислепнию выражения Lisp, первым элементом которого был бы некоторый оператор, аргументы которого были бы вызовами соответствующих первичных методов, в порядке конкретизации. Если мы определили обобщенную функцию price для объединения значений с помощью +, и небыло бы применимых методов around-, она будет вести себя так, как если бы она была определена:

```
(defun price (&rest args)
  (+ (apply  most specific primary method  args)
         ...(apply  leastspecificprimarymethod  args)))
```

Если существуют применимые методы around-, они имеют приоритет, как в стандартной комбинации методов. В комбинации с операторным методом, метод around-может вызывать следующий метод через call-next-method. Однако первичные методы больше не могут испольовать call-next-method. (Это отличие от нашего наброска, где мы оставляли call-next доступным для таких методов.)

В CLOS, мы можем указать тип комбинации методов, который будет использоваться обобщенной функцией, указав необязательный аргумент :аргумент method-combination для defgeneric:

```
(defgeneric price (x)
  (:method-combination +))
```

Теперь метод price будет использовать + для комбинации методов. Если мы определим некоторые классы с prices,

```
(defclass jacket nil nil)
(defclass trousers nil nil)
(defclass suit (jacket trousers) nil)

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

затем, когда мы запрашиваем price для экземпляра suit, мы получаем сумму применимых методов price:

> (price (make-instance 'suit)) 550

Следующие символы могут использоваться в качестве второго аргумента для defmethod или в опции :method-combination для defgeneric:

+ and append list max min nconc or progn

Вызывая define-method-combination вы можете определить другие виды комбинации методов, см. CLTL2, p. 830.

После указания метода комбинации, который должна использовать обобщенная функция, все методы для этой функции должны использовать один и тоже вид. Теперь это вызовет ошибку, если мы попытаемся испольовать другой оператор (или :before или :after) в качестве второго аргумента в defmethod для price. Если мы хотим изменить комбинацию методов price мы должны удалить всю обобщенную функцию, вызывав fmakunbound.

25.6 25-6 CLOS и Lisp

CLOS является хорошим примером встроенного языка. Такая программа обычно приносит двойную выгоду:

1. Встраиваемые языки могут быть концептуально хорошо интегрированы с их средой, так что в рамках встроенного языка мы можем продолжать думать о программах во многом в тех же терминах.

2. Встроенные языки могут быть мощными. потому что они используют все то, что базовый язык уже знает как делать.

CLOS побеждает по обоим пунктам. Он очень хорошо интегрирован с Lisp, и он хорошо использует абстракции, которые уже есть в лиспе. Действительно, мы часто можем видеть Lisp через CLOS, также как, мы можем видеть форму объектов через листы, покрывающие их.

Не случайно мы говорим с CLOS через слой макросов. Макросы выполняют преобразование. а CLOS по сути, программа, которая берет программы построенные из объектно-ориетированных абстракций и переводит их в программы, построенные из абстракций Lisp.

Как и предполагалось в первых двух разделах, абстракции объектно-ориентированного программироания настолько аккуратно отображаются на абстракции Lisp, что практически можно первые назвать частным случаем последнего. Объекты объектно-ориентированного программирования легко могут быть реализованы как объекты Lisp, а их методы - как лексические замыкания. Используя преимущество таких изоморфизмов, мы смогли представить элементарную форму объектно ориентированного программирования всего в нескольких строках кода, и эскиз CLOS на нескольких страницах.

CLOS намного крупнее и мощнее нашего эскиза, но не настолько велик, чтобы скрывать свои корни как встроенного языка. Возьмите defmethod в качестве примера. Хотя CLTL2 не упоминает это явно, методы CLOS обладают всей силой лексических замыканий. Если мы определим несколько методов в рамках некоторой переменной,

```
(let ((transactions 0))
  (defmethod withdraw ((a account) amt)
        (incf transactions)
        (decf (balance a) amt))
  (defmethod deposit ((a account) amt)
        (incf transactions)
        (incf (balance a) amt))
  (defun transactions ()
        transactions))
```

тогда во время выполнения они будут иметь доступ к переменной, как замыкания. Методы могут сделать это, потому что под синтаксисом они являются замыканиями. В расширении defmethod, его тело окажеться целиком в теле заквотированного с решёткой ('#) лямбда выражения.

В разделен 7-6 предполагается, что проще понять, как работают макросы, чем то, что они имеют в виду. Аналогично, секрет понимания CLOS заключается в том, чтобы понять как он отображается на фундаментальные абстракции Lisp.

25.7 25-7 Когда использовать Объекты

Объектно-ориентированный стиль обеспечивает несколько явных преимуществ. Разные программы нуждаются в этих преимуществах в разной степени. На одном конце континуума находяться программы, например симуляторы, которые наиболее естественным образом выражаются в абстракциях объектно-ориентированного программирования. На другом конце программы написанные в объектно-ориентированном стиле, главным образом, чтобы сделать их расширяемыми.

Расширяемость действительно является одним из больших преимуществ объектно-ориентированного стиля. Вместо того, чтобы быть единым монолитным блоком кода, программа написана небольшими кусочками, каждый из которых помечен своей целью. Поэтому позже, когда кто-то захочет изменить программу, будет легко найти ту часть, которую необходимо изменить. Если мы хотим изменить способ отображения объектов типа ob на экране, мы изменим метод отображения(display) класса ob. Если мы хотим создать новый класс объектов подобных ob, но отличающихся от него в нескольких отношениях, мы можем создать подкласс ob; в подклассе мы поменяем нужные нам свойства, а все остальное по умолчанию наследуется от класса ob. И если мы просто хотим создать один объект ob, который ведет себя не так как остальные, мы можем создать новый дочерний объект ob и напрямую изменить его свойства. Если программа была тщательно написана с самого начала, мы можем внести все эти типы изменний, дже не глядя на остальную часть кода. С этой точки зрения объектно-ориентированная программа - это программа организованная как таблица: мы можем быстро и безопасно изменить ее, посмотрев соответствующую ячейку.

Расширяемость требуется меньше всего от объектно-ориентированного стиля. Фактически, она требует так мало, что расширяемая программа вообще не должна быть объектно- ориентированной. Если в предыдущих главах и было чтот-то показано, то оно показало что программы на Lisp не обязательно являются монолитными блоками. Lisp предлагает целый ряд возможностей для расширения. Например, вы могли бы буквально иметь программу организованную как таблица: программу, состоящую из набора замыканий, хранящихся в массиве.

Если вам нужна расширяемость, вам не нужно выбирать между "объектно- ориентированной" и "традиционной" программой. Вы можете дать программе на Lisp именно ту степень расширямости, в которой она нуждается, часто не прибегая к объектно ориентированным техникам. Слот в классе это глобальная переменная. И также, как некрасиво использовать глобальную переменную, там где вы могли использовать параметр, также неэлегантно создавать мир классов и экземпляров, когда вы могли бы сделать тоже самое с меньшими усилиями в простом Lisp. С добавлением CLOS, Common Lisp стал самым мощным используемым широкорасространненым объектно

ориентированным языком. По иронии судьбы, это также язык, в котором объектно-ориентированное программирование наименее необходимо.

26 Приложение: Пакеты

Пакеты - это способ Common Lisp группировать код в модули. Ранние диалекты Lisp содержали таблицу символов, называемую oblist, в которой перечислялись все символы, которые до сих пор прочитала система. Через ввод символа в oblist, система получала доступ к таким вещам как его значение и список свойств. Указанный в oblist символ, как говорят, был интернирован(interned).

Недавние диалекты Lisp разделили концепцию oblist на несколько пакетов. Теперь символ нре просто интернирован, но иинтернирован в определенном пакете. Пакеты поддерживают модульность, потому что символы, содержащиеся в одном пакете, доступны в других пакетах(за исключением читерства(cheating)) только если они явно объявлены таковыми.

Пакет - это своего рода объект Lisp. Текущий пакет всегда храниться в глобальной переменной *package*. Когда запускается Common Lisp, текущим пакетом будет пользовательский пакет: либо user (в реализации CLTL1), либо common-lisp-user (в реализации CLTL2).

Пакеты обычно идентифицируются по их именам, которые являются строками. Чтобы найти имя текущего пакета, попробуйте:

> (package-name *package*) "COMMON-LISP-USER" Обычно прочитанный символ помещается в пакет, который был текущим на момент его прочтения. Чтобы определить пакет, в который помещается символ, мы можем использовать symbol-package: > (symbol-package 'foo) #<Package "COMMON-LISP-USER" 4CD15E>

Возвращаемое здесь значение - фактический объект пакета. Для использования в будущем, давайте дадим foo значение:

> (setq foo 99) 99

Вызвав in-package мы можем переключиться на новый покет, создав его, если это необходимо(создать можно с помощью команды (defpackage :mine (:use :common-lisp)):[1]

> (in-package 'mine :use 'common-lisp) #<Package "MINE" 63390E>

В этой точке должна быть жуткая музыка, потому что мы находимся в другом мире: foo здесь не то что раньше.

MINE> foo >>Error: FOO has no global value.

Почему это случилось? Поскольку foo, который мы выше установили в 99, является отличающимся символом от foo в mine.[2] Чтобы обратиться к оригинальному foo извне пользовательского пакета, мы должны поставить префикс имени пакета и двух двоеточий:

MINE> common-lisp-user::foo 99

Таким образом, разные символы с одинаково печатающимися именами могут сосуществовать в разных пакетах. Может быть один foo в пакете common-lisp-user и другой foo в пакете mine, и они будут разными символами. На самом деле, в этом и

[1] В старых реализациях Common Lisp, опустите аргумент :use.

[2] Некоторые реализации Common Lisp печатаю имя пакета перед запросом верхнего уровня всякий раз, когда мы не находимся в пользовательском пакете. Это не обязательно, но приятно.

заключается суть пакетов: если вы пишете свою программу в отдельном пакете, вы можете выбирать имена для своих функций и переменных, не беспокоясь о том, что кто-то будет использовать то же имя для чего-то другого. Даже если они используют одно и тоже имя, это не будет одним и тем же символом.

Пакеты также пердоставляют средства сокрытия информации. Программы должны ссылаться на функции и переменные по их именам. Если вы не делаете данное имя доступным за пределами вашего пакета, маловероятно, что код в другом пакете сможет использовать или изменить то, на что оно ссылается.

В программах обычно плохо использовать префиксы пакетов с двойными двоеточиями. Тем самым вы нарушаете модульность, которую должны предоставлять пакеты. Если вам нужно использовать двойное двоеточие для обозначения символа, это происходит потому, что кто-то не хочет чтобы вы это делали.

Обычно следует ссылаться только на символы, которые были экспортированы. Экспортируя символ из пакета, в котором он находиться, мы делаем его видимым для других пакетов. Чтобы экспортировать символ, мы вызываем (как вы уже догадались) export:

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T> (setq bar 5)
5
```

Теперь, когда мы вернемся к пакету mine, мы можем сослаться на bar только с одним двоеточием, потому что это общедоступное имя:

```
> (in-package 'mine)
#<Package "MINE" 63390E>
MINE> common-lisp-user:bar
5
```

Импортируя(importing) bar в mine мы можем сделать еще один шаг, и сделать так, чтобы mine фактически поделилась символом bar с пакетом пользователя:

MINE> (import 'common-lisp-user:bar) TMINE> bar 5

После импорта bar мы можем ссылаться на него без какого либо спецификатора пакета. Два пакета теперь имеют один и тот же символ; здесь уже нет отдельного mine:bar.

Что если такой символ уже был? В этом случае, вызов import вызвал бы ошибку, какую мы видим попытавшись имппортировать foo:

MINE> (import 'common-lisp-user::foo) >>Error: FOO is already present in MINE.

Раньше, когда мы безуспешно пытались вычислить foo в mine, мы тем самым вызывали интернирование туда символа foo. Он не имел глобального значения, и следовательно, генерировал ошибку, но интернирование происходило просто как следствие ввода его имени. Так что теперь, когда мы пытаемся импортировать foo в mine, там уже есть символ с тем же именем.

Мы также можем импортировать символы массово, определив один пакет для использования другим:

MINE> (use-package 'common-lisp-user) T

Теперь все символы экспортируемые пользовательским пакетом будут автоматически импортированы mine. (Если бы foo был экспортирован пользовательским пакетом, этот вызов также бы вызвал ошибку.)

Начиная с CLTL2, пакет, содержащий имена встроенных операторов и переменных, называется common-lisp вметос lisp, и новые пакеты больше не используют его по умолчанию. Так как мы использовали этот пакет в вызове in-package, который создал mine, все имена Common Lisp будут видны здесь:

MINE> #'cons #<Compiled-Function CONS 462A3E>

Вы практически вынуждены заставлять любой новый пакет использовать common-lisp (или какой либо другой пакет содержащий операторы Lisp). Иначе вы бы даже не смогли выбраться из нового пакета.

Как и в случае компиляции, операции над пакетами обычно не выполняются на таком верхнем уровне как этот. Чаще всего вызовы содержаться в исходныхs файлах. Обычно достаточно начать файл с in-package и defpackage. (Макрос defpackage является новым в CLTL2, но некоторые более старые реализации предоставляют его.) Вот что вы пожете поместить в начало файла, содержащего отдельный пакет кода:

(in-package 'my-application :use 'common-lisp)

(defpackage my-application (:use common-lisp my-utilities) (:nicknames app) (:export win lose draw))

Это приведет к тому, что код в файле- или точнее, имена в файле- будут в пакете my-application. Этот пакет использует как common-lisp, так и my-utilities, поэтому любые экспортируемые символы могут появиться без префикса пакета в этом файле.

Сам пакет my-application экспортирует только три символа: win, lose, и draw. Поскольку вызов in-package дал my-application прозвище app, код в других пакетах будет ссылаться на них как, например app:win.

Тип модульности, предоставляемый пакетами, на самом деле немного странный. У нас есть модули не объектов, а имен. Каждый пакет, который использует common-lisp импортирует имя cons, потому что common-lisp включает функцию с этим именем. Но в результате переменная с именем cons также будет видна в каждом пакете который использует common-lisp. И то же самое относиться к другим пространствам имен Common Lisp. Если пакеты сбивают с толку, это основная причина; они основаны не на объектах, а на именах.

Вещи имеющие отношение к пакетам, обычно проихсодят во время чтения, а не во время выполнения, что может привести к некоторой путанице. Вторым выражением мы наберем:

(symbol-package 'foo)

вернет значение, которое он прочитал, поскольку чтение запроса создало ответ. Чтобы выполнить это выражение, Lisp должен прочитать его, что означает интернирование foo.

В качестве другого примера рассмотрим этот обмен, который появился выше:

MINE> (in-package 'common-lisp-user) #<Package "COMMON-LISP-USER" 4CD15E> > (export 'bar)

Обычно два выражения введенные в верхнем уровне эквивалентны тем же двум выражениям, заключенным в progn. Дело, не в этом. Если мы попробуем сказать

MINE> (progn (in-package 'common-lisp-user) (export 'bar)) >>Error: MINE::BAR is not accessible in COMMON-LISP-USER.

вместо этого мы получаем ошибку. Это происходит потому, что перед вычислением все выражения progn обрабатываются чтением(read). Когда вызывается read, текущий пакет mine, поэтому принятый bar считается mine:bar. Это как если бы мы попросили экспортировать данный символ вместо common-lisp-user:bar, из пакета пользователя.

То как определяются пакеты создает неудобства при написании программ, которые используют символы в качестве данных. Например, если мы определим noise следующим образомs:

(in-package 'other :use 'common-lisp) (defpackage other (:use common-lisp) (:export noise))

(defun noise (animal) (case animal (dog 'woof) (cat 'meow) (pig 'oink)))

тогда если мы будем вызывать noise из другого пакета с неквалифицированным символом (без указания пакета символа) в качестве аргумента, он обычно будет попадать в конец предложений case и возвращать nil:

OTHER> (in-package 'common-lisp-user) #<Package "COMMON-LISP-USER" 4CD15E> > (other:noise 'pig) NIL

Это происходит потому, что то что мы передали в качестве аргумента, было common-lisp-user:pig(без обид), в то время как ключом для выражения case является other:pig. Чтобы noise работал так как следовало ожидать, мы должны экспортировать все шесть символов используемых в нем, и импортировать их в любой пакет, из которого мы собираемся вызывать noise.

В этом случае, мы можем избежать проблем, используя ключевые слова вместо обычных симолов. Если noise был определен

(defun noise (animal) (case animal (:dog :woof) (:cat :meow) (:pig :oink)))

тогда мы можем смело вызывать его из любого пакета:

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise :pig)
:OINK
```

Ключевые слова как золото, универсальные и само вычисляемые. Они видны повсюду, и их никогда не нужно цитировать(квотировать). Функция управляемая символами такая как defanaph (стр. 223) почти всеугда должна быть написана с использованием ключевых слов.

Пакеты являются богатым источником путаницы. Это введение в предмет едва приподняло завесу тайны. Все подноbности, см. CLTL2, Chapter 11.

27 Замечания

This section is also intended as a bibliography. All the books and papers listed here should be considered recommended reading.

v Foderaro, John K. Introduction to the Special Lisp Section. CACM 34, 9 (September 1991), p. 27. viii The final Prolog implementation is 94 lines of code. It uses 90 lines of utilities from previous chapters. The ATN compiler adds 33 lines, for a total of 217. Since Lisp has no formal notion of a line, there is a large margin for error when measuring the length of a Lisp program in lines. ix Steele, Guy L., Jr. Common Lisp: the Language, 2nd Edition. Digital Press, Bedford (MA), 1990. 5 Brooks, Frederick P. The Mythical Man-Month. Addison-Wesley, Reading (MA), 1975, p. 16. 18 Abelson, Harold, and Gerald Jay Sussman, with Julie Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, 1985. 21 More precisely, we cannot define a recursive function with a single lambda-expression. We can, however, generate a recursive function by writing a function to take itself as an additional argument,

(setq fact #'(lambda (f n) (if (= n 0) 1(* n (funcall f f (- n 1)))))) and then passing it to a function that will return a closure in which original function is called on itself:

(defun recurser (fn) #'(lambda (&rest args) (apply fn fn args)))

Passing fact to this function yields a regular factorial function,

> (funcall (recurser fact) 8) 40320

which could have been expressed directly as:

((lambda (f) #'(lambda (n) (funcall f f n))) #'(lambda (f n) (if (= n 0) 1(* n (funcall f f (- n 1)))))) Many Common Lisp users will find labels or alambda more convenient. 23 Gabriel, Richard P. Performance and Standardization. Proceedings of the First International Workshop on Lisp Evolution and Standardization, 1988, p. 60. Testing triangle in one implementation, Gabriel found that "even when the C compiler is provided with hand-generated register allocation information, the Lisp code is 17% faster than an iterative C version of this function." His paper mentions several other programs which ran faster in Lisp than in C, including one that was 42% faster. 24 If you wanted to compile all the named functions currently loaded, you could do it by calling compall: (defun compall () (do-symbols (s) (when (fboundp s) (unless (compiled-function-p (symbol-function s)) (print s) (compile s))))) This function also prints the name of each function as it is compiled. 26 You may be able to see whether inline declarations are being obeyed by calling (disassemble 'foo), which displays some representation of the object code of function foo. This is also one way to check whether tail-recursion optimization is being done. 31 One could imagine nreverse defined as:

(defun our-nreverse (lst) (if (null (cdr lst)) lst (prog1 (nr2 lst) (setf (cdr lst) nil))))

NOTES 389

(defun nr2 (lst) (let ((c (cdr lst))) (prog1 (if (null (cdr c)) c(nr2 c)) (setf (cdr c) lst)))) 43 Good design always puts a premium on economy, but there is an additional reason that programs should be dense. When a program is dense, you can see more of it at once. People know intuitively that design is easier when one has a broad view of one's work. This is why easel painters use long-handled brushes, and often step back from their work. This is why generals position themselves on high ground, even if they are thereby exposed to enemy fire.

And it is why programmers spend a lot of money to look at their programs on large displays instead of small ones. Dense programs make the most of one's field of vision. A general cannot shrink a battle to fit on a table-top, but Lisp allows you to perform corresponding feats of abstraction in programs. And the more you can see of your program at once, the more likely it is to turn out as a unified whole. This is not to say that one should make one's programs shorter at any cost. If you take all the newlines out of a function, you can fit it on one line, but this does not make it easier to read. Dense code means code which has been made smaller by abstraction, not text-editing. Imagine how hard it would be to program if you had to look at your code on a display half the size of the one you're used to. Making your code twice as dense will make programming that much easier. 44 Steele, Guy L., Jr. Debunking the "Expensive Procedure Call" Myth or, Procedu- ral Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. Proceedings of the National Conference of the ACM, 1977, p. 157. 48 For reference, here are simpler definitions of some of the functions in Figures 4-2 and 4-3. All are substantially (at least 10%) slower: (defun filter (fn lst) (delete nil (mapcar fn lst)))

(defun filter (fn lst) (mapcan #'(lambda (x) (let ((val (funcall fn x))) (if val (list val)))) lst))

(defun group (source n) (if (endp source) nil (let ((rest (nthcdr n source))) (cons (if (consp rest) (subseq source 0 n) source) (group rest n)))))

390 NOTES

(defun flatten (x) (mapcan #'(lambda (x) (if (atom x) (mklist x) (flatten x))) x))

(defun prune (test tree) (if (atom tree) tree (mapcar #'(lambda (x) (prune test x)) (remove-if #'(lambda (y) (and (atom y) (funcall test y))) tree))))

49 Written as it is, find2 will generate an error if it runs off the end of a dotted list:

> (find2 #'oddp '(2 . 3)) >>Error: 3 is not a list.

CLTL2 (p. 31) says that it is an error to give a dotted list to a function expecting a list. Implementations are not required to detect this error; some do, some don't. The situation gets murky with functions that take sequences generally. A dotted list is a cons, and conses are sequences, so a strict reading of CLTL would seem to require that

(find-if #'oddp '(2 . 3))

return nil instead of generating an error, because find-if is supposed to take a sequence as an argument. Implementations vary here. Some generate an error anyway, and others return nil. However, even implementations which follow the strict reading in the case above tend to deviate in e.g. the case of (concatenate 'cons '(a . b) '(c . d)), which is likely to return (ac.d)instead of (a c). In this book, the utilities which expect lists expect proper lists. Those which operate on sequences will accept dotted lists. However, in general it would be asking for trouble to pass dotted lists to any function that wasn't specifically intended for use on them. 66 If we could tell how many parameters each function had, we could write a version of compose so that, in f g, multiple values returned by g would become the correspond- ing arguments to f. In CLTL2, the new function function-lambda-expression returns a lambda-expression representing the original source code of a function. However, it has the option of returning nil, and usually does so for built-in func- tions. What we really need is a function that would take a function as an argument and return its parameter list. 73 A version of rfind-if which searches for whole subtrees could be defined as follows:

NOTES 391

(defun rfind-if (fn tree) (if (funcall fn tree) tree (if (atom tree) nil (or (rfind-if fn (car tree)) (and (cdr tree) (rfind-if fn (cdr tree)))))))

The function passed as the first argument would then have to apply to both atoms and lists:

> (rfind-if (fint #'atom #'oddp) '(2 (3 4) 5)) 3> (rfind-if (fint #'listp #'cddr) '(a (b c d e))) (B C D E)

95 McCarthy, John, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. Lisp 1.5 Programmer's Manual, 2nd Edition. MIT Press, Cam- bridge, 1965, pp. 70-71. 106 When Section 8-1 says that a certain kind of operator can only be written as a macro, it means, can only be written by the user as a macro. Special forms can do everything macros can, but there is no way to define new ones. A special form is so called because its evaluation is treated as a special case. In an interpreter, you could imagine eval as a big cond expression:

(defun eval (expr env) (cond ... ((eq (car expr) 'quote) (cadr expr)) ... (t (apply (symbol-function (car expr)) (mapcar #'(lambda (x) (eval x env)) (cdr expr))))))

Most expressions are handled by the default clause, which says to get the function referred to in the car, evaluate all the arguments in the cdr, and return the result of applying the former to the latter. However, an expression of the form (quote x) should not be treated this way: the whole point of a quote is that its argument is not evaluated. So eval has to have one clause which deals specifically with quote. Language designers regard special forms as something like constitutional amend- ments. It is necessary to have a certain number, but the fewer the better. The special forms in Common Lisp are listed in CLTL2, p. 73. The preceding sketch of eval is inaccurate in that it retrieves the function before evaluating the arguments, whereas in Common Lisp the order of these two operations is deliberately unspecified. For a sketch of eval in Scheme, see Abelson and Sussman, p. 299.

392 NOTES

115 It's reasonable to say that a utility function is justified when it pays for itself in brevity. Utilities written as macros may have to meet a stricter standard. Reading macro calls can be more difficult than reading function calls, because they can violate the Lisp evaluation rule. In Common Lisp, this rule says that the value of an expression is the result of calling the function named in the car on the arguments given in the cdr, evaluated left-to-right. Since functions all follow this rule, it is no more difficult to understand a call to find2 than to find-books (page 42). However, macros generally do not preserve the Lisp evaluation rule. (If one did, you could have used a function instead.) In principle, each macro defines its own evaluation rule, and the reader can't know what it is without reading the macro's definition. So a macro, depending on how clear it is, may have to save much more than its own length in order to justify its existence. 126 The definition of for given in Figure 9-2, like several others defined in this book, is correct on the assumption that the initforms in a do expression will be evaluated left-to-right. CLTL2 (p. 165) says that this holds for the stepforms, but says nothing one way or the other about the initforms. There is good cause to believe that this is merely an oversight. Usually if the order of some operations is unspecified, CLTL will say so. And there is no reason that the order of evaluation of the initforms of a do should be unspecified, since the evaluation of a let

is left-to-right, and so is the evaluation of the stepforms in do itself. 128 Common Lisp's gentemp is like gensym except that it interns the symbol it creates. Like gensym, gentemp maintains an internal counter which it uses to make print names. If the symbol it wants to create already exists in the current package, it increments the counter and tries again:

> (gentemp) T1 > (setq t2 1) 1> (gentemp) T3

and so tries to ensure that the symbol created will be unique. However, it is still possible to imagine name conflicts involving symbols created by gentemp. Though gentemp can guarantee to produce a symbol not seen before, it cannot foresee what symbols might be encountered in the future. Since gensyms work perfectly well and are always safe, why use gentemp? Indeed, for macros the only advantage of gentemp is that the symbols it makes can be written out and read back in, and in such cases they are certainly not guaranteed to be unique. 131 The capture of function names would be a more serious problem in Scheme, due to its single name-space. Not until 1991 did the Scheme standard suggest any official way of defining macros. Scheme's current provision for hygienic macros differs greatly from defmacro. For details, and a bibliography of recent research on the subject, see the most recent Scheme report.

NOTES 393

137 Miller, Molly M., and Eric Benson. Lisp Style and Design. Digital Press, Bedford (MA), 1990, p. 86. 158 Instead of writing mvpsetq, it would be cleaner to define an inversion for values. Then instead of

(mvpsetq (w x) (values y z) ...)

we could say

(psetf (values w x) (values y z) ...)

Defining an inversion for values would also render multiple-value-setq un- necessary. Unfortunately, as things stand in Common Lisp it is impossible to define such an inversion; get-setf-method won't return more than one store variable, and presumably the expansion function of psetf wouldn't know what to do with them if it did. 180 One of the lessons of setf is that certain classes of macros can hide truly enormous amounts of computation and yet leave the source code perfectly comprehensible. Eventually setf may be just one of a class of macros for programming with assertions. For example, it might be useful to have a macro insist which took certain ex- pressions of the form (predicate . arguments), and would make them true if they weren't already. As setf has to be told how to invert references, this macro would have to be told how to make expressions true. In the general case, such a macro call might amount to a call to Prolog. 198 Gelernter, David H., and Suresh Jagannathan. Programming Linguistics. MIT Press, Cambridge, 1990, p. 305. 199 Norvig, Peter. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Morgan Kaufmann, San Mateo (CA), 1992, p. 856. 213 The constant least-negative-normalized-double-float and its three cousins have the longest names in Common Lisp, with 38 characters each. The operator with the longest name is get-setf-method-multiple-value, with 30. The following expression returns a list, from longest to shortest, of all the symbols visible in the current package: (let ((syms nil)) (do-symbols (s) (push s syms)) (sort syms #'(lambda (x y) (> (length (symbol-name x)) (length (symbol-name y)))))) 217 As of CLTL2, the expansion function of a macro is supposed to be defined in the environment where the defmacro expression appears. This should make it possible to give propmacro the cleaner definition:

394 NOTES

(defmacro propmacro (propname) `(defmacro ,propname (obj) `(get ,obj ',propname))) But CLTL2 does not explicitly state whether the propname form originally passed to propmacro is part of the lexical environment in which the inner defmacro occurs. In principle, it seems that if color were defined with (propmacro color),it should be equivalent to: (let ((propname 'color)) (defmacro color (obj) `(get ,obj ',propname))) or (let ((propname 'color)) (defmacro color (obj) (list 'get obj (list 'quote propname)))) However, in at least some CLTL2 implementations, the new version of propmacro does not work. In CLTL1, the expansion function of a macro was considered to be defined in the null lexical environment. So for maximum portability, macro definitions should avoid using the enclosing environment anyway. 238 Functions like match are sometimes described as doing unification. They don't, quite; match will successfully match (f ?x) and ?x, but those two expressions should not unify. For a description of unification, see: Nilsson, Nils J. Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, New York, 1971, pp. 175-178. 244 It's not really necessary to set unbound variables to gensyms, or to call gensym? at runtime. The expansion-generating code in Figures 18-7 and 18-8 could be written to keep track of the variables for which binding code had already been generated. To do this the code would have to be turned inside-out, however: instead of generating the expansion on the way back up the recursion, it would have to be accumulated on the way down. 244 A symbol like ?x occurring in the pattern of an if-match always denotes a new variable, just as a symbol in the car of a let binding clause does. So although Lisp variables can be used in patterns, pattern variables from outer queries cannot-you can use the same symbol, but it will denote a new variable. To test that two lists have the same first element, it wouldn't work to write: (if-match (?x . ?rest1) lst1 (if-match (?x . ?rest2) lst2 ?x)) In this case, the second ?x is a new variable. If both lst1 and lst2 had at least one element, this expression would always return the car of lst2. However, since you can use (non-?ed) Lisp variables in the pattern of an if-match, you can get the desired effect by writing:

NOTES 395

(if-match (?x . ?rest1) lst1 (let ((x ?x)) (if-match (x . ?rest2) lst2 ?x)))

The restriction, and the solution, apply to the with-answer and with-inference macros defined in Chapters 19 and 24 as well. 254 If it were a problem that "unbound" pattern variables were nil, you could have them bound to a distinct gensym by saying (defconstant unbound (gensym)) and then replacing the line

`(,v (binding ',v ,binds)))

in with-answer with:

`(,v (aif2 (binding ',v ,binds) it unbound))

258 Scheme was invented by Guy L. Steele Jr. and Gerald J. Sussman in 1975. The language is currently defined by: Clinger, William, and Jonathan A. Rees (Eds.). Revised4 Report on the Algorithmic Language Scheme. 1991. This report, and various implementations of Scheme, were at the time of printing available by anonymous FTP from altdorf.ai.mit.edu:pub. 266 As another example of the technique presented in Chapter 16, here is the derivation of the defmacro template within the definition of =defun:

(defmacro fun (x) `(=fun *cont* ,x))

(defmacro fun (x) (let ((fn '=fun)) `(,fn *cont* ,x)))

‘(defmacro ,name ,parms (let ((fn ’,f)) ‘(,fn *cont* „@parms)))

‘(defmacro ,name ,parms ‘(,’,f *cont* „@parms))

267 If you wanted to see multiple return values in the toplevel, you could say instead:

(setq *cont* #’(lambda (&rest args) (if (cdr args) args (car args))))

273 This example is based on one given in: Wand, Mitchell. Continuation-Based Program Transformation Strategies. JACM 27, 1 (January 1980), pp. 166.

396 NOTES

273 A program to transform Scheme code into continuation-passing style appears in: Steele, Guy L., Jr. LAMBDA: The Ultimate Declarative. MIT Artificial Intelligence Memo 379, November 1976, pp. 30-38. 292 These implementations of choose and fail would be clearer in T, a dialect of Scheme which has push and pop, and allows define in non-toplevel contexts:

(define *paths* ()) (define failsym ’@)

(define (choose choices) (if (null? choices) (fail) (call-with-current-continuation (lambda (cc) (push *paths* (lambda () (cc (choose (cdr choices))))) (car choices)))))

(call-with-current-continuation (lambda (cc) (define (fail) (if (null? *paths*) (cc failsym) ((pop *paths*))))))

For more on T, see: Rees, Jonathan A., Norman I. Adams, and James R. Meehan. The T Manual, 5th Edition. Yale University Computer Science Department, New Haven, 1988. The T manual, and T itself, were at the time of printing available by anonymous FTP from hing.lcs.mit.edu:pub/t3-1. 293 Floyd, Robert W. Nondeterministic Algorithms. JACM 14, 4 (October 1967), pp. 636-644. 298 The continuation-passing macros defined in Chapter 20 depend heavily on the optimization of tail calls. Without it they may not work for large problems. For example, at the time of printing, few computers have enough memory to allow the Prolog defined in Chapter 24 to run the zebra benchmark without the optimization of tail calls. (Warning: some Lisps crash when they run out of stack space.) 303 It’s also possible to define a depth-first correct choose that works by explicitly avoiding circular paths. Here is a definition in T:

(define *paths* ()) (define failsym ’@) (define *choice-pts* (make-symbol-table))

(define-syntax (true-choose choices) ‘(choose-fn ,choices ’,(generate-symbol t)))

NOTES 397

(define (choose-fn choices tag) (if (null? choices) (fail) (call-with-current-continuation (lambda (cc) (push *paths* (lambda () (cc (choose-fn (cdr choices) tag)))) (if (mem equal? (car choices) (table-entry *choice-pts* tag)) (fail) (car (push (table-entry *choice-pts* tag) (car choices)))))))) In this version, true-choose becomes a macro. (The T define-syntax is like defmacro except that the macro name is put in the car of the parameter list.) This macro expands into a call to choose-fn, a function like the depth-first choose defined in Figure 22-4, except that it takes an additional tag argument to identify choice-points. Each value returned by a true-choose is recorded in the global hash-table *choice-pts*. Ifagiventrue-choose is about to return a value it has already returned, it fails instead. There is no need to change fail itself; we can use the fail defined on page 396. This implementation assumes that paths are of finite length. For example, it would allow path as defined in Figure 22-13 to find a path from a to e in the graph displayed in Figure 22-11 (though not

necessarily a direct one). But the true-choose defined above wouldn't work for programs with an infinite search-space: (define (guess x) (guess-iter x 0))

(define (guess-iter x g) (if (= x g) g(guess-iter x (+ g (true-choose '(-1 0 1)))))) With true-choose defined as above, (guess n) would only terminate for non- positive n. How we define a correct choose also depends on what we call a choice point. This version treats each (textual) call to true-choose as a choice point. That might be too restrictive for some applications. For example, if two-numbers (page 291) used this version of choose, it would never return the same pair of numbers twice, even if it was called by several different functions. That might or might not be what we want, depending on the application. Note also that this version is intended for use only in compiled code. In interpreted code, the macro call might be expanded repeatedly, each time generating a new gensymed tag. 305 Woods, William A. Transition Network Grammars for Natural Language Analysis. CACM 3, 10 (October 1970), pp. 591-606.

398 NOTES

312 The original ATN system included operators for manipulating registers on the stack while in a sub-network. These could easily be added, but there is also a more general solution: to insert a lambda-expression to be applied to the register stack directly into the code of an arc body. For example, if the node mods (page 316) had the following line inserted into the body of its outgoing arc,

(defnode mods (cat n mods/n ((lambda (regs) (append (butlast regs) (setr a 1 (last regs))))) (setr mods *)))

then following the arc (however deep) would set the the topmost instance of the register a (the one visible when traversing the topmost ATN)to1. 323 If necessary, it would be easy to modify the Prolog to take advantage of an existing database of facts. The solution would be to make prove (page 336) a nested choose:

(=defun prove (query binds) (choose (choose-bind b2 (lookup (car query) (cdr query) binds) (=values b2)) (choose-bind r *rules* (=funcall r query binds))))

325 To test quickly whether there is any match for a query, you could use the following macro:

(defmacro check (expr) `(block nil (with-inference ,expr (return t))))

344 The examples in this section are translated from ones given in: Sterling, Leon, and Ehud Shapiro. The Art of Prolog: Advanced Programming Techniques. MIT Press, Cambridge, 1986. 349 The lack of a distinct name for the concepts underlying Lisp may be a serious barrier to the language's acceptance. Somehow one can say "We need to use C++ because we want to do object-oriented programming," but it doesn't sound nearly as convincing to say "We need to use Lisp because we want to do Lisp programming." To administrative ears, this sounds like circular reasoning. Such ears would rather hear that Lisp's value hinged on a single, easily understood concept. For years we have tried to oblige them, with little success. Lisp has been described as a "list- processing language," a language for "symbolic computation," and most recently, a "dynamic language." None of these phrases captures more than a fraction of what Lisp is about. When retailed through college textbooks on programming languages, they become positively misleading. Efforts to sum up Lisp in a single phrase are probably doomed to failure, because the power of Lisp arises from the combination of at least five or six features. Perhaps

NOTES 399

we should resign ourselves to the fact that the only accurate name for what Lisp offers is Lisp. 352 For efficiency, sort doesn't guarantee to preserve the order of sequence elements judged equal by the function given as the second argument. For example, a valid Common Lisp implementation could do this:

> (let ((v #((2 . a) (3 . b) (1 . c) (1 . d)))) (sort (copy-seq v) #'< :key #'car)) #((1 . D) (1 . C) (2 . A) (3 . B))

Note that the relative order of the first two elements has been reversed. The built-in stable-sort provides a way of sorting which won't reorder equal elements:

> (let ((v #((2 . a) (3 . b) (1 . c) (1 . d)))) (stable-sort (copy-seq v) #'< :key #'car)) #((1 . C) (1 . D) (2 . A) (3 . B))

It is a common error to assume that sort works like stable-sort. Another common error is to assume that sort is nondestructive. In fact, both sort and stable-sort can alter the sequence they are told to sort. If you don't want this to happen, you should sort a copy. The call to stable-sort in get-ancestors is safe because the list to be sorted has been freshly made.

400 NOTES

28 Book's Index

404 INDEX

INDEX 405

406 INDEX

INDEX 407

408 INDEX

Concept Index

(Предметный указатель не существует)

Function Index

(Предметный указатель не существует)