LET OVER LAMBDA

LET OVER LAMBDA

50 лет Lisp-у

DOUG HOYTE

Let Over Lambda
50 Лет Лиспу

Даг Хойт

2008

λ

Let Over Lambda

50 лет Лиспу

Даг Хойт (Doug Hoyte)

Let Over Lambda

Редакция 1.0

ISBN: 978-1-4357-1275-1

http://letoverlambda.com

doug@hcsw.org

Производство HCSW и Hoytech

Сделано на Lisp-е

Автор перевода: Charlz_Klug

Русификация обложки: Илья (Zirom) Сёмин. E-mail: zirom@list.ru

Вёрстка, вычитка седьмой главы и перевод восьмой главы: Алексей Фокин.

Оглавление

1 Введение **5**
1.1 Макросы 5
1.2 U-Язык 11
1.3 Утилиты Лисп 15
1.4 Лицензия 18
1.5 Благодарности 18

2 Замыкания **21**
2.1 Замыкание — Ориентированное Программирование 21
2.2 Среды и Пространство 22
2.3 Области Видимости 24
2.4 Let — это Лямбда 29
2.5 Лямбда окружённая Let'ом 35
2.6 Lambda над Let над Lambda 36
2.7 Let над Lambda над Let над Lambda 39

3 Основы макросов **43**
3.1 Итеративная Разработка 43
3.2 Предметно-Ориентированные Языки 44
3.3 Управляющие Структуры 49
3.4 Свободные Переменные 52
3.5 Нежелательный Захват 55
3.6 Once Only 67
3.7 Двойственность Синтаксиса 73

4 Считывающие Макросы **79**
4.1 Время-Работы как Время-Чтения 79
4.2 Обратная Кавычка 82
4.3 Чтение Строк 87
4.4 CL-PPCRE 89
4.5 Циклические Выражения 98

4.6 Безопасность Считывателя 101

5 Программирующие Программы 109
5.1 Лисп — Не Функциональный Язык 109
5.2 Программирование Сверху-Вниз 113
5.3 Неявные Контексты 120
5.4 Проход по Коду с Macrolet 127
5.5 Рекурсивные Расширения 133
5.6 Рекурсивные Решения 141
5.7 Dlambda 147

6 Анафорические Макросы 153
6.1 Больше Фор? 153
6.2 Шарп-Обратное Закавычивание 156
6.3 Alet и Машины с Конечным Состоянием 159
6.4 Косвенные Цепи 166
6.5 Замыкания, Поддерживающие Горячую Замену 172
6.6 Суб-Лексическая Область Видимости 176
6.7 Пандорические Макросы 184

7 Темы эффективности макросов 203
7.1 Лисп быстр 203
7.2 Макросы Ускоряют Лисп 205
7.3 Знакомство с Вашим Дизассемблером 221
7.4 Область Видимости Указателя 227
7.5 Tlist-ы и Cons Пулы 238
7.6 Сортирующие Сети 249
7.7 Написание и Замеры Производительности Компиляторов . 264

8 Лисп, переходящий в Форт, переходящий в Лисп 281
8.1 Причудливая Архитектура 281
8.2 Cons Шитый Код 285
8.3 Двойственность Синтаксиса, Определённая 292
8.4 Работающий Форт 300
8.5 Более Фортово 310
8.6 Переходим в Лисп 322

Глава 1

Введение

1.1 Макросы

> Ядро Lisp'а занимает в некотором роде локальный оптимум в пространстве языков программирования.
>
> — Скромные слова Джона МакКарти, создателя Lisp'а

Эта книга о программировании *макросов* на Lisp-е. Большинство книг о программировании дают лишь беглый обзор материала. Эта книга построена в виде руководств и примеров, написанных таким образом, чтобы вы могли максимально быстро и эффективно программировать сложные макросы. Освоение макросов — это финальный шаг, отделяющий посредственного Lisp-программиста от Lisp-профессионала.

Макросы — это единственное самое большое преимущество Lisp-а и единственная важнейшая деталь, которая может присутствовать в других языках программирования. С помощью макросов вы можете делать то, что попросту невозможно в других языках. Поскольку макросы могут использоваться для преобразования Lisp-а в другие языки программирования и обратно, то программисты, изучившие макросы, могут обнаружить, что все остальные языки являются просто оболочкой поверх Lisp-а. Это *далеко идущий* вывод. Lisp примечателен тем, что программирование на нём — это программирование на высочайшем уровне. В то время, когда большинство языков изобретают и применяют синтаксические и семантические правила, Lisp универсален и пластичен. С Lisp-ом вы создаёте правила.

История Lisp'а более богата и обширна, нежели история других языков программирования. Над разработкой этого языка работали одни из лучших компьютерных учёных. Благодаря их трудам возник самый мощный и универсальный язык программирования. Lisp содержит много

стандартов, несколько замечательных реализаций с открытым исходным кодом и макросы, с которыми не сравнятся другие языки программирования. В этой книге используется только COMMON LISP [ANSI—CL] [CLTL2], но большое количество идей можно легко портировать на такие Lisp'ы, как Scheme[R5RS]. Тем не менее, ознакомившись с этой книгой, вы увидите, что для написания макросов стоит использовать COMMON LISP. В то время, когда остальные Lisp'ы хороши для других целей, COMMON LISP — это выбор профессионала, работающего с макросами.

Архитекторы COMMON LISP-а проделали замечательную работу при проектировании языка. Учитывая качество реализации COMMON LISP, на данный момент, — это лучшая среда для разработки программ с удивительно малым количеством оговорок. Как программист вы всегда можете рассчитывать на COMMON LISP и будьте уверены что в этом языке всё сделано так, как надо. Несмотря на то, что архитекторы и авторы реализаций выполнили свою работу на отлично, есть пробелы в объяснении того, почему язык реализован таким, а не другим способом. Для многих программистов COMMON LISP — это огромное коллекция непонятных особенностей, поэтому эти программисты переходят к более привычным, для них, языкам, так и не познав истинной мощи макросов. Эта книга может быть использована в качестве путеводителя по многим замечательным особенностям удивительного языка программирования — COMMON LISP. Большинство языков спроектированы так, чтобы их можно было легко реализовать; COMMON LISP спроектирован для создания мощных программ. Я искренне надеюсь, что создатели COMMON LISP оценят эту книгу как наиболее завершённый и доступный источник сведений об особенностях макросов и что эта книга будет ощутимой каплей в океан тем о макросах.

История макросов начинается почти с историей Lisp-а. Макросы изобретены Тимоти Хартом [MACRO-DEFINITIONS] в 1963 году. Тем не менее, большинство Lisp программистов не используют всю мощь макросов, а остальные программисты вообще не используют макросы. Это всегда остаётся загадкой для продвинутых лисперов. Если макросы настолько хороши, то почему их не используют все программисты? Самые умные и наиболее решительные программисты в конечном счёте приходят к макросам Лиспа. Для того, чтобы понять возможности макросов, необходимо понять что есть в Лиспе и чего нет в других языках. А это в свою очередь требует знания других, менее мощных языков. К сожалению многие программисты теряют желание учиться после того, как они освоят несколько других языков и таким образом никогда не узнают что такое макросы и их возможности. Но, несмотря на это, некоторый процент программистов задумывается над возможностью создания про-

грамм, пишущих другие программы — то есть, они приходят к макросам. А поскольку Лисп — это лучший язык для создания макросов, самые умные, самые решительные и самые любознательные программисты всегда приходят к Лиспу.

Программисты из числа верхнего процентиля всегда будут малым числом, несмотря на то, что общая популяция программистов растёт. Люди из мира программирования видят несколько примеров мощи макросов, а число людей понимающих макросы ещё меньше, но всё постепенно меняется. Поскольку макросы увеличивают производительность в разы, *век макросов* наступает, независимо от того, готов к ним мир или нет. Цель этой книги — стать первой линией подготовки к неизбежному будущему: миру макросов. Будьте готовы.

Вокруг макросов распространено следующее мнение — использовать их только при необходимости. Причина этому — некоторые макросы сложно понимать, в макросы могут закрасться очень трудно определяемые ошибки и в случае, когда вы обо всём думаете как о функциях, то макросы могут вас неприятно удивить. Это не дефекты Лисповской системы макросов, а общие черты программирования макросов. Как и в случае с любой технологией: чем более мощен инструмент — тем больше способов его неправильного использования. А среди всех программных конструкций макросы Лиспа — это наиболее мощный инструмент.

Можно провести параллель между изучением макросов в Лиспе и изучением указателей в языке программирования C. Большинство начинающих программистов C легко усваивают большую часть языка. Функции, типы, переменные, арифметические выражения: все они имеют параллели с предыдущим интеллектуальным опытом начинающих программистов, начиная с математики уровня начальной школы и заканчивая экспериментами с простейшими языками программирования. Но, большинство новичков в C упираются в кирпичную стену при изучении указателей.

Указатели являются второй натурой опытного C программиста. Многие опытные C программисты считают что полное понимание указателей необходимо для правильного использования C. Поскольку указатели являются фундаментальной идеей, многие опытные C программисты не рекомендуют ограничивать использование указателей при обучении или для стилистической красоты. Несмотря на это многие новички в C считают указатели ненужным усложнением и избегают их использования, так возникает *симптом FORTRAN-а (пренебрежение полезными особенностями "вне зависимости от языка")*. Болезнью является игнорирование особенностей языка, а не плохой программистский стиль. Если же особенности языка освоены в полном объёме, то корректный стиль

программирования становится очевидным. Не нужно стремиться выработать какой-либо стиль программирования, это касается всех языков, — вспомогательная тема этой книги. Стиль нужен только в том случае, когда отсутствует понимание[1].

Подобно указателям C, макросы — это особенность Лиспа, которая часто остаётся плохо понятой, и поэтому распространены не совсем верные представления о макросах. Если при работе с макросами вы полагаетесь на такие высказывания как:

> Макросы изменяют синтаксис Лисп кода.
>
> Макросы работают в дереве разбора ваших программ.
>
> Используйте макросы только тогда, когда с задачей не справляются функции.

то скорее всего вы упустили из виду общую картину, что даёт программирование макросов. Именно это и исправляет эта книга.

Хорошие справочники и руководства по макросам можно пересчитать по пальцам. Одним из хороших книг по макросам является книга Пола Грэма *On Lisp* [On-Lisp]. Рекомендуется к прочтению от корки до корки всем, кто интересуется макросами. *On Lisp* и остальные труды Грэма послужили толчком к написанию книги, которую вы сейчас читаете. Благодаря Полу Грэму и остальным людям, писавшим о Лиспе, мощь макросов широко обсуждается, но, к сожалению, всё же остаётся также широко не понятой. Несмотря на то, что просто прочитав книгу *On Lisp* можно узнать много интересного о макросах, некоторые программисты связывают проблемы программирования с макросами. В то время, когда *On Lisp* показывает вам различные виды макросов, эта книга расскажет вам как использовать эти макросы.

Написание макросов — это итеративный процесс, связанный с размышлениями. Все сложные макросы начинаются из простых макросов, прошедших через долгую серию улучшений и тестирований. Кроме того, знание о том, где применить макросы приходит с накоплением опыта написания макросов. При написании программ человек следует некоторой системе и процессу. Каждый программист представляет себе некую концептуальную модель работы инструментов программирования и создаёт код исходя из результатов вытекающих из этой модели. Программист, обладающий интеллектом начнёт задумываться о программировании, как о логической процедуре, и придёт к мысли, о процессе

[1] Следствие этого — эффективное использование чего-либо одного, подсмотренного где-либо, способом при отсутствии должного понимания природы вещей.

автоматизации программирования. После этого программист будет готовиться к процессам автоматизации.

Важным шагом к пониманию макросов является следующее: если писать код без тщательного планирования и без приложения значительных усилий, то в результате код будет нашпигован множественными шаблонами и негибкими абстракциями. Такое положение вы можете увидеть в любых больших программах. Дублируемые участки кода и слишком усложнённый код — это отсутствие правильных абстракций у авторов кода. Эффективное использование макросов подразумевает под собой признание проблемы в виде повторяющихся шаблонов и абстракций, после этого следует создание *кода, помогающего вам писать код*. Но этого не достаточно для того, чтобы понять как писать макросы; профессиональный Лисп программист хочет знать зачем писать макросы.

Программисты на C, только пришедшие в Лисп, часто делают ошибку считая что главное предназначение макросов в улучшении эффективности кода в момент выполнения[2]. Да, довольно часто макросы используются именно для этой задачи, но наиболее общей целью использования макросов является упрощение процесса программирования. В большинстве программах шаблоны просто избыточно копируются, а абстракции используются в не достаточной мере, правильно спроектированные макросы могут вывести выразительность программирования на ещё больший уровень. Там, где остальные языки оказываются ограниченными и конкретными, Лисп остаётся универсальным и гибким.

Эта книга не введение в Лисп. Темы и материал подобраны для профессиональных программистов в не-Лисп языках, интересующихся пользой, которую можно извлечь из макросов и для студентов, посредственно знающих Лисп, которые готовы по настоящему изучить самую сильную сторону Лиспа. Подразумевается, что вы посредственно знаете Лисп, и от вас не требуется глубокого знания замыканий и макросов.

Эта книга не только о теории. Все примеры, приведённые здесь, полностью функциональны, пригодны для использования и могут помочь вам улучшить ваше программирование здесь и сейчас. Эта книга о применении передовых программистских технологий для улучшения вашего программирования. Во многих книгах посвящённых программированию сознательно применяется простой стиль программирования в угоду доступности. В этой книге материал преподаётся с полным применением всего языка. Кроме этого приведённые примеры кода используют эзотерические особенности COMMON LISP, большинство из которых бу-

[2]Программисты на C делают эту ошибку по причине того, что "макро система"действительно хороша для этих целей, но главное назначение макросов не в этом.

дут описываться при использовании. Если вы прочитали и поняли[3] всё, что описано в *главе 2, Замыкания, на странице 21* и в *главе 3, Основы Макросов, на странице 43*, то можете считать что вы прошли среднюю стадию понимания Лиспа.

Одной из частей Лиспа является самостоятельные открытия, эта книга не лишит вас удовольствия от экспериментов. Имейте в виду, что материал в этой книге подаётся очень быстро, много быстрее, чем вы сможете усвоить. Для того, чтобы понять некоторые участки кода, приведённые здесь, вам придётся обращаться к другим справочникам и руководствам по COMMON LISP'у. После изучения базовых понятий мы перейдём прямо к последним исследованиям, посвящённым макросам, большинство из которых граничат с большой неизведанной областью. Эта книга фокусируется на *комбинировании макросов*. Эта тема имеет пугающую репутацию, а хорошо понять эту тему способны не все программисты. Комбинирование макросов включает в себя наиболее обширные и плодородные исследования в языках программирования. Исследователями было выжато почти всё из типов, объектов и пролого-подобной логики, но программирование макросов остаётся огромной, зияющей чёрной дырой. Никто не знает до конца что лежит по ту сторону. Всё, что мы знаем на данный момент — это то, что макросы сложны, пугающи и проявляют неограниченный потенциал. В отличие от многих программистских идей макросы не являются ни академической концепцией для производства бесполезных теоретических публикаций, ни пустым модным словом относящемуся к программному обеспечению компаний. Макросы — это лучшие друзья хакеров. Макросы делают ваши программы умнее, а не труднее. Большинство программистов, начавших изучение макросов приходят к выводу что программирование без макросов — это не программирование.

Многие книги посвящённые Лиспу написаны в пропагандистском ключе, но я совершенно равнодушно отношусь к публичным призывам использовать Лисп. Лисп не собирается уходить. Я был бы счастлив если бы мог использовать Лисп в качестве *секретного оружия* на протяжении всей своей карьеры программиста. Эта книга имеет только одну цель — вдохновить на изучение и исследование, также, как я был вдохновлён *On Lisp*'ом. Я надеюсь, что смогу вдохновить читателей этой книги и смогу насладиться ещё более лучшими Лисповскими макро инструментами и ещё более интересными книгами о макросах Лиспа.

Да пребудет с вами сила Лиспа,

ваш покорный автор,

[3]Конечно, вы можете не соглашаться с этим утверждением.

Листинг 1.1: ПРИМЕР-ЛИСТИНГА-ПРОГРАММЫ

```
(defun example-program-listing ()
  '(this is
    (a (program
        (listing)))))
```

Даг Хойт.

1.2 U-Язык

Обсуждение макросов влечёт за собой обсуждение обсуждения, поэтому мы должны чётко обозначить соглашения, используемые в этой книге. Всё, что я сейчас пишу и всё что вы получаете из книги через чтение и интерпретацию прочитанного, что в свою очередь представляет из себя систему выражений подлежащих формализации и анализу.

Никто не понимает этого лучше, чем Хаскелл Карри, автор *Основ Математической Логики* [FOUNDATIONS]. Карри, который пытался не просто формализовать идеи, а формализовать ещё и само выражение идей, посчитал необходимым абстрагировать эту концепцию в коммуникативный язык между писателем и читателем. Он назвал его U-Язык (U-Language).

> Каждое исследование, включая эту книгу, должны передаваться от одного человека к другому через язык. Целесообразно начать наше обучение обратив внимание на тот очевидный факт, что нужно дать некоторое имя нашему языку и явно обозначить его функции. Мы будем называть наш используемый язык U-Языком. [...] Если бы язык не был так тесно связан с нашей работой, то можно было бы не уделять ему такого внимания.

На протяжении всей книги мы будем вводить новые ключевые концепции или будем обращать внимание на какой-либо момент *вот таким шрифтом*. При цитировании специальных форм, функций, макросов и других идентификаторов присутствующих в уже встреченных или ещё не описанных программах, мы будем использовать **этот специальный шрифт** (учтите что некоторые слова имеют множественные значения, например макрос **lambda** в COMMON LISP и концепция лямбда; специальная форма **let** и список, представляющую формы).

В этой книге новые куски кода представлены в форме *листингов программ*. Код, который можно использовать в своих программах или примеры соответствующих реализаций, выделены в определении наших функций так: **example-program-listing**. Но иногда мы будем демонстрировать использование некоторого кода или просто обсуждать свойства некоторых выражений с шрифтом текста книги[4]. В этих случаях код или примеры использования кода будут отображаться так:

```
(this is
  (demonstration code))
```

В большинстве книг о программировании приводятся изолированные, искусственные примеры, и в то же время авторы забывают связывать эти примеры с реальной жизнью. Примеры в этой книге очень короткие и связаны с общей картиной преподаваемых идей программирования. Некоторые авторы пытаются скрасить скуку примеров используя забавные и хитрые идентификаторы имён или поверхностные аналогии. Наши примеры служат лишь для иллюстрации идей. Но это не значит что вся книга позиционируется как очень серьёзная. Здесь присутствует юмор, но нужно суметь его найти.

Поскольку Лисп имеет интерактивную природу, результат вычисления простого выражения часто передаёт больше чем соответствующее количество информации в U-Языке. В этих случаях мы будем применять вывод из COMMON LISP-овского Прочитать Вычислить Напечатать Повторить (Read Evaluate Print Loop — сокращается до REPL):

```
* (это
   (выражение
     (для вычисления)))

ЭТО-РЕЗУЛЬТАТ
```

Обратите внимание: текст кода введён в нижнем регистре, а текст возвращённый Лиспом в верхнем регистре. Эта особенность COMMON LISP, что позволяет нам легко вычленять вывод REPL от введённых выражений. Точнее, эта особенность позволит нам быстро просматривать Лисп формы, содержащие символы — в каком-либо файле или экране — и сразу узнать во что они были вычислены Лисп считывателем. Кроме того, заметьте, что в приглашении присутствует символ астериска (*).

[4]А это сноска, имеющая отношение к основному тексту, но, уводящая повествование в сторону.

Этот символ идеален, поскольку его нельзя перепутать со сбалансированным символом и его начертание позволяет быстро определить его в сессии REPL.

Написание сложных Лисп макросов — это *итеративный* процесс. Ни один человек не может просто так взять и выдавать очередями уже готовые макросы величиной с целую страницу. Этому есть две причины: при сравнении большинства других языков, Лисп программа при равном объёме кода содержит гораздо больше информации, вторая причина в том, что Лисп поощряет программистов в итеративной разработке программ: выполнении серии улучшений, продиктованной потребностями создаваемой программы.

В этой книге используется разделение между такими диалектами Лиспа, как COMMON LISP и Scheme и более абстрактными Лиспами, используемыми в качестве строительного материала. Другое важное отличие, используемое в этой книге — это проведение границы между диалектами Лиспа и не-Лисп языками программирования. Иногда, нам нужно будет говорить о не-Лисповских языках программирования, для того чтобы нажить как можно больше врагов, мы будем избегать упоминания какого-либо конкретного языка. Мы используем следующее необычное утверждение:

> Язык без Лисп макросов называется Блаб.

Слово Блаб в U-Язык пришло из эссе Пола Грэма, *Побеждая Посредственность* [BEATING-AVGS], где Блаб — это гипотетический язык, используемый для выделения того факта, что Лисп отличается от других языков: Лисп — другой. Блаб характеризуется инфиксным синтаксисом, раздражающей системой типов, увечной объектной системой, но самая главная черта, объединяющая Блаб языки — это отсутствие Лисп макросов. Ещё одна полезная сторона Блаба — в том, что простейший способ изучения продвинутой технологии макросов — это рассмотрение причин по которой макросы невозможно использовать в Блабе. Цель Блаб терминологии не в издевательствах над не Лисп языками[5].

Для иллюстрации итеративного процесса создания макроса, в этой книге используется следующее соглашение: символ процента (%), добавляемый к именам обозначает не завершённость или возможность какого-либо улучшения функций или макросов. С каждым улучшением к имени будет добавляться ещё один символ % и так до тех пор, пока мы не придём к финальному варианту. Финальный вариант будет без символов процента.

[5]Хотя в этом есть доля юмора.

Листинг 1.2: ПРИМЕР-ИТЕРАТИВНОГО-ПРОЦЕССА

```
(defun example-function% () ; первая версия
  t)

(defun example-function%% () ; вторая версия
  t)

(defun example-function () ; финальная версия!
  t)
```

В терминологии Карри макросы описываются как *метапрограммирование*. Метапрограммирование — это программа, созданная с единственной целью — дать возможность программисту писать ещё более лучшие программы. Метапрограммирование, в том или ином виде присутствует во всех языках программирования, но ни в одном языке метапрограммирование не реализовано так хорошо как в Лиспе. Ни в одном другом языке нельзя так удобно писать программы с помощью техник метапрограммирования. Именно поэтому Лисп программы выглядят так *странно* для не-Лисп программистов: Лисп код выражен как прямое следствие нужд метапрограммирования. В этой книге мы попытаемся описать это архитектурное решение Лиспа — создание метапрограмм на самом же Лиспе — деталь, которая даёт Лиспу потрясающее превосходство в продуктивности. Однако, поскольку мы будем создавать метапрограммы в Лиспе, мы должны помнить, что метапрограммирование отличается от спецификаций U-Языка. Мы могли бы обсуждать метаязыки с различных точек зрения, включая остальные метаязыки, но здесь у нас есть только U-Язык. Комментарий Карри по поводу своей системы:

> Мы можем продолжить формировать иерархию языков с любым числом уровней. Однако, вне зависимости от количества уровней U-Язык будет находиться на самом высоком уровне: если есть два уровня, то это будет мета-язык; если есть три уровня, то будет мета-мета-язык; и т.д. Таким образом термины U-Языка и мета-языка должны различаться.

Эта книга о Лиспе, конечно, логическая система Лиспа очень отличается от системы, которую описал Карри, поэтому мы будем использовать ещё и несколько других идей из его работы. Но, вклад Карри в логику и метапрограммирование продолжает вдохновлять нас и по сей день.

Не только благодаря его фундаментальным идеям о символическом закавычивании (symbolic quotation), но и благодаря красиво описанном и выполненном U-Языке.

1.3 Утилиты Лисп

On Lisp — это одна из тех книг, которую вы либо поймёте, либо не поймёте. *On Lisp* вы либо будете обожать, либо будете бояться. Начиная с самого названия *On Lisp* рассказывает о создании программных абстракций, находящихся *на верхнем слое Лиспа*. После создания этих абстракций мы можем создать ещё один слой абстракций над абстракциями.

В большинстве языках большая часть функциональности языка реализована на самом языке; обычно Блаб языки имеют при себе стандартную библиотеку написанную на Блабе. Если авторы языка не хотят программировать на самом языке, то, возможно, вы тоже не захотите писать программы на таком языке.

Даже если в другие языки добавить стандартные библиотеки, Лисп всё равно остаётся другим. Другие языки составлены из примитивов, а Лисп составлен из мета-примитивов. Как только макросы стандартизируются, как в COMMON LISP'е, то весь остальной язык можно *создать* из, практически, ничего. Для гибкости в большинстве языках используются наборы примитивов, Лисп же предоставляет нам систему мета-программирования, в свою очередь дающую доступ к любым разновидностям примитивов. Можно взглянуть и с другой точки зрения: Лисп вообще устраняет всю концепцию примитивов. В Лиспе система мета-программирования не останавливается на каких-либо примитивах. В Лиспе возможно, а по факту — желательно, используя технику макро программирования, развить язык в пользовательское приложение. Приложение, доработанное до самого высокого пользовательского уровня, по прежнему остаётся слоями из макросов на луковице Лиспа, выращенными через итерации.

В этом свете присутствие примитивов в языке выглядит как проблема. В архитектуре системы всегда присутствуют примитивы, барьеры, не ортогональность. Конечно, иногда это оправдано. Для большинства программистов не является проблемой обработка отдельных машинных кодов как примитивов в C или Лисп компиляторах. Но Лисп пользователям нужен контроль над всем. Ни в одном другом языке не предоставляется такой полный контроль программисту над языком как в Лиспе.

По совету из книги *On Lisp*, книга, которую вы сейчас читаете спроектирована как ещё один слой луковицы. Так же, как одни програм-

Листинг 1.3: MKSTR-SYMB

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

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

мы представляют слой над другими программами, эта книга базируется на *On Lisp*. Это центральная тема книги Грэма: при сочетании хорошо спроектированных *утилит* общая производительность может оказаться большей чем сумма производительностей каждой утилиты. Этот раздел описывает коллекцию полезных утилит. Кроме утилит из On Lisp в этот раздел вошли ещё и сторонние утилиты.

Утилита **symb**, слой поверх **mkstr** — основной способ создания символов. Поскольку символы могут ссылаться на любые строки, а программное создание символов чрезвычайно удобно, то **symb** — это основная утилита для макро программирования и очень часто используется на протяжении всей книги.

Group — это другая утилита, постоянно всплывающая при написании макросов. Отчасти это связано с необходимостью в таких отражающих операторах как COMMON LISP'овские **setf** и **psetf**, группирующими аргументы, а отчасти и с тем, что часто группировка — это наилучший путь структурирования связанных данных. Поскольку мы будем очень часто использовать эту функциональность, то эту абстракцию нужно сделать как можно более универсальной. **Group** Грэма группирует все предоставленное количество групп, определённое в параметре **n**. В отличие от **setf**, где аргументы группируются в пары, а **n** равно 2.

Flatten — одна из наиболее важных утилит в *On Lisp*. Получает вложенную списковую структуру и возвращает новый список, содержащий списковую структуру всех атомов из начального списка. Если же мы будем думать о списковой структуре как о дереве, то **flatten** будет возвращать список всех листьев дерева. Если дерево представляет Лисп код, то выполняя некоторую проверку присутствующих объектов в выражении, **flatten** будет представлять из себя некоторую разновидность *прохода-по-коду (code-walking)* — тема рекурсии, проходящая через всю книгу.

Fact и **choose** — это очевидные реализации функций факториала и биномиального коэффициента.

Листинг 1.4: GROUP

```
(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)))
```

Листинг 1.5: FLATTEN

```
(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)))
```

Листинг 1.6: FACT-AND-CHOOSE

```
(defun fact (x)
  (if (= x 0)
    1
    (* x (fact (- x 1)))))

(defun choose (n r)
  (/ (fact n)
     (fact (- n r))
     (fact r)))
```

1.4 Лицензия

Поскольку я считаю что концепции представленные в коде в этой книге являются такими же фундаментальными, как физические наблюдения или математические доказательства, то даже если бы я захотел присвоить их себе, сделать это я бы не смог. По этой причине вы вольны делать с кодом присутствующим в этой книге всё, что угодно. Эта очень либеральная лицензия распространяется вместе с кодом:

```
;; Это исходный код для книги
;; _Let_Over_Lambda_ за авторством Дага Хойта.
;; Правообладателем кода является Даг Хойт. 2002-2008 годы.
;;
;; Вы можете свободно использовать, модифицировать и
;; распространять
;; этот код по вашему желанию, но любые
;; модификации должны быть явно описаны перед
;; распространением. Нет никакой гарантии,
;; явной или косвенной.
;;
;; Приветствуется, но не является обязательным
;; указывание меня, Дага Хойта, в качестве автора кода.
;; Если вы нашли
;; полезным код или хотите получить документацию,
;; пожалуйста, рассмотрите вариант приобретения книги!
```


1.5 Благодарности

Брайан Хойт (Brian Hoyte), Нэнси Холмс (Nancy Holmes), Розали Холмс (Rosalie Holmes), Ян (Ian), Алекс (Alex) и вся остальная моя семья; syke, madness, fyodor, cyb0rg/asm, theclone, blackheart, d00tz, rt, magma, nummish, zhivago, defrost; Майк Конрой (Mike Conroy), Сильвия Рассел (Sylvia Russell), Алан Пэт (Alan Paeth), Роб МакАртур (Rob McArthur), Сильви Десярдинс (Sylvie Desjardins), Джон МакКарти (John McCarthy), Пол Грэм (Paul Graham), Дональд Кнут (Donald Knuth), Лео Броди (Leo Brodie), Брюс Шнайер (Bruce Schneier), Ричард Столлман (Richard Stallman), Эди Вейтз (Edi Weitz), Питер Норвиг (Peter Norvig), Питер

Сибель (Peter Seibel), Кристиан Куиннек (Christian Queinnec), Кейт Бостик (Keith Bostic), Джон Гэмбл (John Gamble); проектировщики и создатели COMMON LISP, особенно Гай Стил (Guy Steele), Ричард Гэбриел (Richard Gabriel) и Кент Питман (Kent Pitman), разработчики и сопроводители CMUCL/SBCL, CLISP, OpenBSD, GNU/ Linux.

Отдельная благодарность Яну Хойту (Ian Hoyte) за дизайн обложки и Лео Броди (Leo Brodie) за картинку на обороте обложки.

Эта книга предназначена всем, кто любит программирование.

Глава 2

Замыкания

2.1 Замыкание — Ориентированное Программирование

> Один из выводов к которому мы пришли был следующим: “объект” не должен быть примитивным понятием в языке программирования; объекты и их поведение должны строиться из значений ячеек и старых добрых лямбда выражений.
>
> — Гай Стил во время разработки Scheme

Иногда это называется *замыканием (closure)*, в других случаях — сохранённым лексическим окружением. Или, как любят говорить некоторые из нас — *let, окружающий lambda (let over lambda)*. Вне зависимости от используемой терминологии, усвоение концепции замыканий — это первый шаг в становлении профессионального Лисп программиста. По факту, это умение является жизненно необходимым для правильного использования многих современных языков программирования, кроме этого, сюда можно включить языки явно не поддерживающие let или lambda, например Perl или Javascript.

Замыкания входят в то небольшое число любопытных концепций, парадоксальная сложность которых связана с их чрезвычайной простотой. Когда программист начинает использовать сложные решения проблем, то простые решения начинают казаться ему незавершёнными и неудобными. Далее мы увидим что замыкания могут быть более простым решением нежели решение проблемы в лоб, например организация данных и кода через замыкания может оказаться более простой чем использование объектов. Но, в замыканиях есть ещё более важное свойство чем

простота: абстракции, используемые для конструирования макросов — центральная тема этой книги.

Тот факт, что мы можем строить объекты и классы с помощью примитивов замыканий не означает что объектные системы бесполезны для Лисп программистов. Это далеко не так. На самом деле COMMON LISP включает в себя одну из наиболее мощных объектных систем: *CLOS*, COMMON LISP Object System (Объектная Система COMMON LISP). Несмотря на то, что я очень восхищён гибкостью и функциональностью CLOS, его расширенные особенности мне редко бывают нужны[1], благодаря ячейкам с присваиваемыми значениями и старым добрым лямбда выражениям.

В целом эта книга предназначена для программистов со средним уровнем знания Лиспа, но в этой главе мы попытаемся обучить вас с самых основ теории и использованию замыканий, для того, чтобы ознакомить вас с общей терминологией замыканий, используемой на протяжении всей книги. В этой главе рассматривается эффективность замыканий и как современные компиляторы оптимизируют замыкания.

2.2 Среды и Пространство

Под ячейками с присваиваемым значением Стил определяет среду для сохранения ссылок к данным, где среда — это субъект чего-либо, называемый *неопределённым пространством (indefinite extent)*. Это причудливый способ определения возможности ссылаться на такую среду в любой момент времени. Будучи единожды определённой эта среда и её ссылки будут существовать столько, сколько нам нужно. Рассмотрим следующую C функцию:

```
#include <stdlib.h>
int *environment_with_indefinite_extent(int input) {
  int *a = malloc(sizeof(int));
  *a = input;
  return a;
}
```

После вызова этой функции и получения возвращённого указателя, мы, в течении неопределённого срока, по прежнему можем ссылаться

[1] В COMMON LISP практически невозможно программировать без CLOS, поскольку CLOS занимает центральное место в COMMON LISP.

на выделенную память. В C новые среды создаются при вызове функции, но C программисты указывают в **malloc()** требуемую память при возвращении её для использования за пределами функции.

Для контраста: ниже приведён бесполезный пример. C программисты рассматривают **a** как автоматически собираемую переменную, происходящее при возвращении функции поскольку среда выделяется в *стеке*. С точки зрения Лисп программистов **a** создаётся во *вре́менном пространстве*.

```
int *environment_with_temporary_extent(int input) {
  int a = input;
  return &a;
}
```

Разница между средами C и Лисп в следующем: если вы явно не укажете что вам нужно, то Лисп будет предполагать что вы хотите использовать неопределённое пространство. Другими словами: Лисп всегда предполагает, что вы хотите вызвать **malloc()**, как в первом примере. Можно сказать, что это по своей природе менее эффективно, чем использование временного пространства, но преимущества почти всегда превышают незначительные издержки производительности. Более того, Лисп сам часто определяет, когда данные могут быть безопасно размещены в стеке и будет делать это автоматически. Вы даже можете использовать *декларации* для того, чтобы Лисп сам явно выполнял эту операцию. Более подробно мы будем обсуждать декларации в *главе 7, Темы эффективности макросов*.

Но из-за динамичного характера, Лисп не имеет явных значений указателя или типов, как в C. Это может ввести вас в заблуждение, если вы, как программист C, используете вызов указателей и значения для определения типов. Lisp думает обо всем этом немного по-другому. В Лиспе удобно применять следующую мантру:

Переменные не имеют типов. Только значения имеют типы.

Тем не менее, мы должны возвращать что-то для хранения указателей. В Лиспе существуют множество структур данных, которые могут хранить указатели. Одна из самых любимых Лисп программистами простейшей структурой является *ячейка cons (cons cell)*. Каждая cons ячейка содержит ровно два указателя, ласково называемые как *car* и *cdr*. При вызове **environment-with-indefinite-extent** ячейка *cons* будет вызвана с *car*, указывающим на переданный аргумент **input**, и *cdr*, указывающий

на **nil**. И что наиболее важно, эта *cons* ячейка (и ссылка на введённый аргумент **input**) обладает неопределённым пространством, таким образом, мы можем ссылаться на эту ячейку столько, сколько нам нужно:

```
(defun environment-with-indefinite-extent (input)
  (cons input nil))
```

Эффективность недостатков неопределённого пространства достигает неуместности как состояние искусства улучшения технологии Лисп компилирования. Среды и пространства очень тесно связаны с замыканиями. Мы ещё не раз коснёмся темы сред и состояния в этой главе.

2.3 Лексическая и Динамическая Области Видимости

Технический термин, обозначающий доступность ссылки к переменной называется *областью видимости (scope)*. Наиболее общий тип области видимости в современных языках программирования называется *лексической (lexical)* областью. Когда фрагмент кода окружён лексической привязкой к переменной, то говорят что переменная расположена в лексической области привязки. С помощью формы **let**, наиболее универсальная форма для создания привязок, можно создавать переменные в лексической области:

```
* (let ((x 2))
    x)

2
```

Доступ к **x** внутри тела формы **let** осуществляется через лексическую область. Аналогичным образом аргументы в функциях, определённых с помощью **lambda** или **defun** являются лексически связанными переменными в тексте определения функции. Лексические переменные — это переменные, доступ к которым возможен только из внутреннего кода контекста, для примера выше, это форма **let**. Поскольку лексическая область — это чрезвычайно интуитивный способ ограничения доступа к переменной, то может показаться, что это единственный способ разграничения. Есть ли другие другие способы создания областей видимости?

Комбинация неопределённого пространства и лексической области довольно долгое время не использовались в широко распространённых

языках программирования. Первая реализация была разработана Стивом Расселом (Steve Russell) для Lisp 1.5 [HISTORY-OF-LISP] и впоследствии спроектирована для таких языков, как Algol-60, Scheme, и COMMON LISP. В Блабах некоторые полезные элементы лексической области видимости стали медленно реализовываться только спустя некоторое время и это несмотря на долгую и познавательную историю лексической области видимости.

Хотя способы реализации областей видимости в C-подобных языках ограничены, C программистам тоже приходится программировать в различных средах. Для этого они используют неявно определённую область видимости, также известную как *область указателя (pointer scope)*. Область указателя известна трудностью в отладке, многочисленными рисками безопасности и, несколько искусственной эффективностью. Идея, лежащая за областью указателей — это создание проблемно-ориентированного языка для контроля регистров и памяти машины фон Неймана, похожей на большинство современных процессоров [PAIP-PIX], с последующим использованием этого языка для доступа и манипулирования структурами данных с непосредственными командами процессора. Область указателей была необходима по причинам производительности во времена, когда Лисп компиляторы ещё не были изобретены, но, теперь, в современных языках программирования область указателей скорее рассматривается как проблема, а не как особенность.

И хотя Лисп программисты редко думают в терминах указателей, понимание областей указателей бывает весьма полезным при создании эффективного Лисп кода. В *разделе 7.4, Область Указателя* мы будем исследовать реализацию области видимости указателя для тех редких случаев, когда нам нужно уведомить компилятор о создании специфичного кода. Но, в данный момент нам нужно только обсудить механику области указателей. В C нам иногда нужно получать доступ к переменной, определённой за пределом создаваемой функции:

```
#include <stdio.h>

void pointer_scope_test() {
  int a;
  scanf("%d", &a);
}
```

В вышеприведённой функции мы использовали оператор **&** в языке C для того, чтобы передать адрес занимаемый в памяти переменной **a** в функцию **scanf**, после этого функция **scanf** знает куда записывать отсканированные данные. Лексическая область видимости в Лиспе запрещает

нам напрямую реализовать это действие. В Лиспе нам придётся передать подобную анонимную функцию в гипотетическую Лисп функцию **scanf**, что позволит ей изменить нашу лексическую переменную **a**, несмотря на то, что **scanf** определена за пределами нашей лексической области:

```
(let (a)
  (scanf "%d" (lambda (v) (setf a v))))
```

Лексическая область видимости — это благоприятная среда для замыканий. По факту, замыкания настолько связаны с концепцией лексической области, что часто их более точно называют как *лексические замыкания (lexical closures)*, для того, чтобы отделить их от других видов замыканий. Если не указано что-то другое, то по-умолчанию все замыкания в этой книге являются лексическими.

В дополнение к лексической области в Common Lisp есть ещё и *динамическая область (dynamic scope)*. Это *сленг (slang)* Лиспа обозначающий комбинацию временного пространства и глобальную область. Динамическая область видимости — это одна из разновидности областей видимости. Динамическая область видимости уникальна для Лиспа тем, что предполагает отличающееся поведение с аналогичным синтаксисом лексической области. В Common Lisp переменные, доступные через динамическую область видимости, называются *специальными переменными*. Сделано это для того, чтобы привлечь внимание к таким переменным. Специальные переменные могут быть объявлены с помощью **defvar**. Некоторые программисты придерживаются соглашения, по которому имена специальных переменных следует начинать и заканчивать символом астериска, например, ***temp-special***. Это соглашение называется *"наушник" (earmuff)*. По причинам описанным в *разделе 3.7, Дуализм Синтаксиса*, эта книга не использует наушники, поэтому наше объявление специальных переменных выглядит так:

```
(defvar temp-special)
```

При таком определении **temp-special** будет помечен как специальный[2], но не будет инициализирован каким-либо значением. В этом состоянии специальная переменная называется *несвязанной (unbound)*. Только специальные переменные могут быть несвязанными — лексические переменные всегда связаны и поэтому всегда имеют значения. На это можно взглянуть с другой точки зрения: по-умолчанию все символы представляют из себя лексически не связанные переменные. Также как

[2]Также мы можем указать с помощью определений локальную особенность.

и с лексическими переменными мы можем присвоить значение специальным переменным с помощью **setq** или **setf**. Некоторые Лиспы, такие как Scheme, не имеют динамической области видимости. Другие Лиспы, такие как EuLisp [SMALL-PIECES-P46], используют один синтаксис для доступа к лексическим переменным и другой синтаксис для доступа к специальным переменным. Но в Common Lisp синтаксис общий. Многие лисперы считают это особенностью языка. Так мы присваиваем значение к нашей специальной переменной **temp-special**:

```
(setq temp-special 1)
```

До сих пор мы не увидели ничего такого, что выделяло бы особенность специальной переменной. Она выглядит как обычная переменная, некоторым образом привязанная к глобальному пространству имени. Всё это потому, что мы только единожды выполнили привязку — специальная глобальная привязка по умолчанию. Специальные переменные интересны тем, что по отношению к ним возможно выполнение повторной привязки, или *затенения (shadowed)*, с помощью новой среды. Мы определим функцию просто вычисляющую и возвращающую **temp-special**:

```
(defun temp-special-returner ()
  temp-special)
```

Функция может быть использована для получения значения, в которое Лиспом вычисляется **temp-special** в момент вызова:

```
* (temp-special-returner)

1
```

Иногда это называют вычислением формы в *нулевую лексическую среду (null lexical environment)*. Очевидно, что нулевая лексическая среда не содержит никаких лексических привязок. В данном примере значение **temp-special** возвращается из глобального специального значения, 1. Но, если мы вычислим его в не-нулевой лексической среде — в той, которая содержит привязку для нашей специальной переменной — то здесь проявится специальность **temp-special**[3]:

[3]Поскольку мы создаём динамическую привязку, но не лексическую среду. Они просто похожи друг на друга.

```
* (let ((temp-special 2))
     (temp-special-returner))

2
```

Обратите внимание, что в качестве значения была возвращена 2, это означает, что значение для **temp-special** было получено из нашей среды **let**, а не из специального глобального окружения. Если вам это всё ещё не кажется интересным, то взгляните как это невозможно выразить в большинстве других традиционных языков программирования. Ниже показан пример из псевдокода на Блабе:

```
int global_var = 0;

function whatever() {
  int global_var = 1;
  do_stuff_that_uses_global_var();
}

function do_stuff_that_uses_global_var() {
  // global_var is 0
}
```

Расположение в памяти или значения регистров для лексических привязок становятся известными при компиляции[4], а привязки для специальных переменных определяются при работе программы. Вам может показаться что специальные переменные не эффективны, но это не так. Специальная переменная всегда ссылается на одно и тоже место в памяти. При использовании **let**, для привязки специальной переменной, вы на самом деле компилируете код, который сохраняет копию переменной, перезаписывает участок памяти новым значением, вычисляет формы в теле **let** и, наконец, восстанавливает оригинальное значение из копии.

Специальные переменные постоянно связаны с символом, применяемым для их обозначения. Место в памяти, на которое ссылается специальная переменная называется ячейкой **символ-значение (symbol-value)** символа. Это прямое противопоставление лексическим переменным. Лексические переменные указываются символами только в момент компилирования. Поскольку доступ к лексическим переменным возможен только изнутри лексической области видимости их привязок, то

[4]По тем же причинам лексическая область видимости называется как “статическая область видимости”.

компилятору нет нужды помнить о символах, применяемых в качестве ссылки на лексические переменные, поэтому компилятор удаляет их из скомпилированного кода. Мы приукрасим истинность этого высказывания в *разделе 6.7, Пандорические Макросы.*

Хотя Common Lisp и предоставляет нам неоценимые возможности динамической области, но, лексические переменные являются более распространёнными. Раньше динамическая область использовалась для определения особенностей Лиспа, но сейчас, с появлением Common Lisp, почти полностью заменена лексической областью.Поскольку лексическая область позволяет использовать такие приёмы, как лексические замыкания (в дальнейшем мы кратко рассмотрим их), а также более эффективную оптимизацию компилятора, то замена динамической области оказывается хорошей идеей. Однако, архитекторы Common Lisp оставили нам очень прозрачное окно в мир динамической области, которая ныне используется в более узких целях.

2.4 Let — это Лямбда

Let — это специальная Лисп форма для создания среды с именами (привязками), инициализированными в результаты вычисления соответствующих форм. Эти имена доступны для кода внутри тела **let**, формы в теле **let** вычисляются последовательно, финальным результатом **let** является результат вычисления последней формы. Первоочередная задача **let** — это сам процесс преднамеренной неопределённости. Результат работы **let** отделён от самой работы **let**. Так или иначе **let** нужен для предоставления структуры данных для хранения указателей на значения.

Как мы уже увидели, cons ячейки, несомненно, удобны для хранения указателей, но кроме cons ячеек для хранения указателей можно использовать ещё и другие структуры. Один из лучших путей для хранения указателей в Лиспе — это дать Лиспу возможность самому позаботиться о хранении указателей с помощью формы **let**. С помощью **let** вы только указываете имя (привязку) этих указателей, а об остальном позаботится сам Лисп. Иногда мы можем помочь компилятору в создании более эффективного кода, через передачу некоторой информации, добавляемой в форму определения.

```
(defun register-allocated-fixnum ()
  (declare (optimize (speed 3) (safety 0)))
  (let ((acc 0))
    (loop for i from 1 to 100 do
```

```
          (incf (the fixnum acc)
                (the fixnum i)))
    acc))
```

В примере **register-allocated-fixnum** мы дали некоторые подсказки компилятору, что позволяет очень эффективно просуммировать целые числа с 1 до 100. При компиляции эта функция будет располагать данные в регистрах, в целом устраняя необходимость в указателях. Несмотря на то, что с нашей точки зрения мы попросили Лисп создать неопределённую среду для хранения **acc** и **i**, Лисп компилятор может оптимизировать эту функцию сохраняя значения прямо в регистрах процессора (CPU). Сгенерированный код может быть таким:

```
; 090CEB52:      31C9        XOR ECX, ECX
;       54:      B804000000  MOV EAX, 4
;       59:      EB05        JMP L1
;       5B: L0:  01C1        ADD ECX, EAX
;       5D:      83C004      ADD EAX, 4
;       60: L1:  3D90010000  CMP EAX, 400
;       65:      7EF4        JLE L0
```

Заметьте, что 4 обозначает 1, а 400 обозначает 100, причина — сдвиг целых чисел на два бита в скомпилированном коде. Это связано с *маркировкой (tagging)* — способ представления чего-либо указателем, а на самом деле использование этого чего-либо для хранения информации. Схема маркировки нашего Лисп компилятора обладает хорошими преимуществами, благодаря которым компилятор не использует сдвиг если индексное слово помещается в памяти [DESIGN-OF-CMUCL]. Подробнее разбор Лисп компилятора будет произведён в *главе 7, Тема Эффективности Макросов*.

Но, если Лисп определит что позже вы можете обратиться к этой среде, то в этом случае будет использоваться что-то менее непостоянное чем регистр. Общей структурой, используемой для хранения указателей в средах, является массив. Если каждая среда имеет массив и все переменные принадлежат этой среде и ссылаются в этот массив, то мы получаем эффективную среду с потенциально неопределённым пространством.

Как было замечено выше, **let** будет возвращать вычисление последней формы в своём теле. Эта черта объединяет многие специальные формы и макросы Лиспа, такое поведение является настолько общим, что

этот шаблон часто называют *неявным progn (implicit progn)* поскольку специальная форма **progn** реализует именно это поведение[5]. Иногда весьма удобно использовать **let** форму через возвращение анонимной функции, пользующейся той-же лексической средой, что доступна из формы **let**. Для создания этих функций в Лиспе мы используем *лямбду (lambda)*.

Лямбда — это простая концепция, способная напугать своей гибкостью и важностью. Лямбда, применяемая в Лиспе и Схеме, восходит к логической системе Алонзо Чёрча, но, лямбда получила развитие под спецификацией Лиспа. Лямбда — это короткий способ повторного присваивания временных имён (привязок) к значениям для определённого лексического контекста и лежит в основе концепции Лисп функции. Лисп функция очень отличается от Чёрчевского описания математической функции. Это произошло по причине того, что лямбда развивалась как мощный, практичный инструмент в руках многих поколений лисперов, совершенствующих и расширяющих лямбду до такой степени, предвидеть которую не могли ранние логики.

В лямбде, несмотря на благоговение Лисп программистов, нет ничего особенного. В дальнейшем мы увидим, что лямбда всего лишь один из многих способов выражения этой разновидности именования переменных. В частности, мы увидим, что макросы позволяют нам изменять переименование переменных такими способами, которые абсолютно невозможны в других языках программирования. После изучения таких макросов мы вернёмся к лямбде и обнаружим что лямбда наиболее удобна для оптимального выражения такого рода именования. Это не случайно. С точки зрения программных сред нашего времени Чёрч может показаться устаревшим и неуместным, но на самом деле его вклад нельзя умалить. Его математическая нотация с многочисленными улучшениями в руках многих поколений Лисп профессионалов развилась в универсальный и гибкий инструмент[6].

Лямбда, как и многие другие особенности Лиспа, очень удобна, самые современные языки начинают импортировать идеи из Лиспа в их собственные системы. Некоторым разработчикам языков кажется что “lambda” — это очень длинно, и они начинают использовать такие аббревиатуры как **fn** и другие. С другой стороны, концепция лямбды настолько фундаментальна, что скрытие этого названия менее длинным именем граничит с ересью. В этой книге мы будем описывать и изучать многие

[5]Progn используется для кластеризации форм и наделяет их высокоуровневым поведением.

[6]Классический пример макроса — это реализация **let** в лямбда форме. В этой книге я не буду утомлять вас этим примером.

разновидности лямбды и по доброй традиции будем называть лямбду лямбдой так же, как и поколения Лисп программистов до нас.

Но как выглядит лямбда в Лиспе? Прежде всего, как и все имена в Лиспе, лямбда — это *символ (symbol)*. Мы можем закавычивать его, сравнивать его и хранить его в списках. Лямбда обретает специальный смысл только когда появляется в виде первого элемента списка. Если это случится, то список называют *лямбда формой (lambda form)* или *определением функции (function designator)*. Но эта форма — не функция. Эта форма — списковая структура данных, которая может быть преобразована в функцию с помощью специальной формы **function**:

```
* (function '(lambda (x) (+ 1 x)))
#<Interpreted Function>
```

7

Для удобства работы с предыдущим выражением в Common Lisp есть сокращение в виде считывающего макроса `#'` (шарп-кавычка). Вместо того, чтобы писать **function** вы можете получить тот-же эффект с помощью этого сокращения:

```
* #'(lambda (x) (+ 1 x))
#<Interpreted Function>
```

Для большего удобства, лямбда определяется как макрос, расширяющийся в специальную форму с вызовом **function**, как в примере выше. Стандарт Common Lisp ANSI требует [ANSI-CL-ISO-COMPATIBILITY] чтобы макрос **lambda** определялся примерно так:

```
(defmacro lambda (&whole form &rest body)
  (declare (ignore body))
  `#' ,form)
```

В данный момент не следует обращать внимание на определение игнорирования[8]. Этот макрос всего лишь простой способ автоматического применения специальной формы **function** к вашему определению функции. Этот макрос позволяет нам вычислить определение функции для создания функций, поскольку они расширяются в формы шарп-кавычка:

```
* (lambda (x) (+ 1 x))
#<Interpreted Function>
```

[7]Примечание переводчика: Это выражение приводит к ошибке, возможно здесь подразумевается “(function (lambda (x) (+ 1 x)))”.

[8]Определение U-Языка.

Есть несколько веских причин, по которым нужно предварять лямбда формы шарп-кавычкой `#'`. Поскольку эта книга не ставит целью поддержку предыдущих сред ANSI Common Lisp, то обратная совместимость соблюдаться не будет. А как же стилистические возражения? Пол Грэм, в ANSI Common Lisp [GRAHAM-ANSI-CL], высказался что краткость этого макроса является "в лучшем случае показной элегантностью". Возражение Грэма сводится к тому, что пока вам приходится использовать шарп-кавычку для функций, на которые ссылаются символы, то система становится асимметричной. Однако, я считаю, что не шарп-закавыченные лямбда формы являются стилистическим улучшением, поскольку они подсвечивают асимметрию, присутствующую в спецификации второго пространства имён. Использование шарп-кавычки для символов — это способ сослаться ко второму пространству имён, в то время когда функции, созданные лямбда формами, конечно, являются безымянными.

Даже не ссылаясь на макрос **lambda** вы можете использовать лямбда формы в виде первого аргумента в вызове функции. Тут Лисп действует также, как и в случае с символом: если символ будет обнаружен первым элементом списка, то считается что мы ссылаемся на ячейку **symbol-function (функция-символа)**, если же будет обнаружена лямбда, то предполагается подстановка анонимной функции:

```
* ((lambda (x) (+ 1 x)) 2)
3
```

Следует помнить, что также, как вы не можете вызывать функцию для динамического возвращения символа, используемого в обычном вызове функции, вы не можете вызывать функцию, возвращающую лямбда форму в позиции функции. Для обоих задач используйте или **funcall** или **apply**.

Большое превосходство лямбда выражений перед функциями в C и других языках в том, что Лисп компиляторы часто могут полностью оптимизировать их. Например, **compiler-test** выглядит так, как будто он применяет инкрементирующую функцию к числу 2 и возвращает результат, но тут хороший компилятор должен быть достаточно умным, чтобы определить, что функция всегда возвращает значение 3 и просто прямо возвращать это значение, не вызывая никаких функций в процессе работы. Это называется *лямбда сворачиванием (lambda folding)*:

```
(defun compiler-test ()
  (funcall
```

```
    (lambda (x) (+ 1 x))
    2))
```

При наблюдении можно выявить эффективность скомпилированной лямбда формы: скомпилированная лямбда форма является постоянной формой. Это означает что если ваша программа будет скомпилирована, все ссылки на эту функцию будут простыми указателями на некоторый кусок машинного кода. Этот указатель может быть возвращён из функций и встроен в новые среды без накладных расходов, связанных с созданием функции. Накладные расходы будут устранены в момент компилирования программы. Другими словами, функция, возвращающая другую функцию будет просто постоянным указателем, возвращающим функцию:

```
(defun lambda-returner ()
  (lambda (x) (+ 1 x)))
```

Прямой противоположностью является форма **let**, спроектированная для создания новой среды во время выполнения программы, как правило **let** не постоянная операция, поскольку возникают накладные расходы из-за сборки мусора в лексических замыканиях, которые представляют неопределённое пространство.

```
(defun let-over-lambda-returner ()
  (let ((y 1))
    (lambda (x)
      (incf y x))))
```

При каждом вызове **let-over-lambda-returner** необходимо создавать новую среду, вставлять указатель на константу в код лямбда формы в этой новой среде, затем вернуть результирующее *замыкание*. Для того, чтобы увидеть насколько мала эта среда вы можете использовать **time**.

```
* (progn
    (compile 'let-over-lambda-returner)
    (time (let-over-lambda-returner)))
; Evaluation took:
; ...
; 24 bytes consed.
; ...
#<Closure Over Function>
```

Если вы вызовите компиляцию замыкания, то получите сообщение об ошибке, гласящее о невозможности компилирования функций определённых в не нулевой лексической среде [CLTL2-P677]. Вы не можете компилировать замыкания, компилируются только функции, которые создают замыкания. Когда вы компилируете функцию, создающую замыкания, то эти замыкания также будут скомпилированы [ON-LISP-P25].

Использование **let** в купе с лямбдой является настолько важной идеей, что оставшуюся часть этой главы мы проведём обсуждая их шаблоны и вариации.

2.5 Лямбда окружённая Let'ом

Лямбда, окружённая Let'ом (Let over lambda), — это прозвище данное лексическому замыканию. Лямбда окружённая **Let**'ом, по сравнению с другими терминологиями, наиболее полно отражает Лисп код, используемый для создания замыканий. В сценарии лямбды, окружённой **let**'ом, последняя форма, возвращаемая **let** конструкцией — это **lambda** выражение. Буквально это можно представить как **let** находящийся над **lambda**:

```
* (let ((x 0))
    (lambda () x))
#<Interpreted Function>
```

Напомним, что форма **let** возвращает результат вычисления его последней формы внутри своего тела, поэтому вычисление этой формы лямбды, окружённой **let**'ом приводит к созданию функции. Однако, в последней форме **let** есть нечто особенное. Это **lambda** форма с **x** в качестве *свободной переменной (free variable)*. Лисп достаточно умён и сам определяет куда, в этой функции, должен ссылаться **x**: **x** из внешнего лексического окружения создан **let** формой. А поскольку в Лиспе, по умолчанию, всё является неопределённым пространством, то эта среда будет доступна функции так долго, сколько нам это будет необходимо.

Лексическая область видимости — это инструмент для точного определения действительности ссылок на переменные и указания куда именно ссылается каждая ссылка. Простым примером замыкания является *счётчик (counter)*, замыкание, которые сохраняет целочисленное значение в среде, инкрементирует и возвращает значение при каждом обращении. Ниже представлена типичная реализация с помощью лямбды, окружённой **let**'ом:

```
(let ((counter 0))
  (lambda () (incf counter)))
```

Это замыкание вернёт 1 при первом вызове, 2 при последующем вызове и так далее. Ещё один способ думать о замыканиях — это представить их в виде функций с *состоянием*. Эти функции нельзя назвать математическими функциями, это скорее процедуры, в каждой из которых есть своя маленькая память. Иногда структуры данных, объединяющих вместе код и данные называются *объектами (objects)*. Объект — это коллекция процедур и некоторого связанного состояния. Поскольку объекты очень близки к замыканиям, то часто они могут рассматриваться как одно и тоже. Замыкание похоже на объект, имеющий только один метод: **funcall**. Объект похож на замыкание, к которому можно применить **funcall** несколькими способами.

Хотя замыкания являются единственной функцией, но, окружающая среда, множественные методы, внутренние классы и статические переменные объектной системы, все они обладают своими замыкающими двойниками. Один из возможных путей эмуляции множественных методов — это просто вернуть множественные **lambda**'ы изнутри той же лексической области видимости:

```
(let ((counter 0))
  (values
    (lambda () (incf counter))
    (lambda () (decf counter))))
```

Этот шаблон, *Let, окружающий две лямбды*, возвращает две функции, они оба получают доступ к одной и той же окружающей их переменной — счётчику. Первая функция инкрементирует счётчик, а вторая функция декрементирует счётчик. Есть много других способов реализации такого поведения. Один из этих способов, **dlambda**, обсуждён в *разделе 5.7, Dlambda*. По причинам, которые будут разъяснены позже, код в этой книге будет структурировать данные с помощью замыканий, а не объектов. Подсказка: Это связано с макросами.

2.6 Lambda над Let над Lambda

В некоторых объектных системах есть чёткая граница между объектами, набором процедур со связанным состоянием и классами, структурами данных, используемыми для создания объектов. В замыканиях такой границы не существует. Мы увидим примеры форм, которые вы можете

вычислять для создания замыканий, большинство из них следуют шаблону **let**, окружающий лямбду, но каким образом наши программы могут создавать эти объекты по мере надобности?

Ответ на этот вопрос чрезвычайно прост. Если мы можем вычислить что-то в REPL, то мы можем вычислить это что-то и в функции. Что если мы создадим функцию, единственным назначением которой будет вычисление **let**, окружающий лямбду, и возвращение получившегося результата? Поскольку для представления функции мы используем **lambda**, то у нас получится нечто, похожее на такой код:

```
(lambda ()
  (let ((counter 0))
    (lambda () (incf counter))))
```

При вызове *лямбды, окружающей let, окружающий лямбду, (lambda over let over lambda)* создаётся и возвращается новое замыкание, содержащее привязку счётчика. Помните, что **lambda** выражения являются константами: простыми указателями на машинный код. Это выражение — простой кусок кода, создающий новые среды, замыкающие внутреннее **lambda** выражение (само по себе являющееся константой, скомпилированной формой) — то-же самое, что мы выполняли в REPL'е.

В объектной системе, часть кода, создающего объекты называется классом. Но, лямбда, окружающая **let**, окружающий лямбду, немного отличается от классов во многих языках. В то время, когда во многих языках классам требуется давать имена, этот шаблон полностью избегает именования. Формы вида лямбды, окружающей **let**, окружающий лямбду, можно называть *анонимными классами (anonymous classes)*.

Хотя анонимные классы часто бывают полезны, мы, обычно, даём имена классам. Самый лёгкий путь именования классов — это рассматривать классы как обычные функции. Как мы, обычно, именуем функции? Конечно, с помощью формы **defun**. После именования вышеприведённый анонимный класс приобретает следующий вид:

```
(defun counter-class ()
  (let ((counter 0))
    (lambda () (incf counter))))
```

Где начинается первая **lambda**? **Defun** создаёт *неявную лямбду (implicit lambda)* вокруг формы в своём теле. Когда вы пишете обычную функцию с помощью **defun**, то, внутренне, вы по прежнему используете те же лямбда формы, но, этот факт скрывается за поверхностью синтаксиса **defun**.

К несчастью большинство книг, посвящённых программированию на Лиспе, не приводят реалистичных примеров использования замыканий, и читатель заблуждается, считая, что замыкания — это такая идея, которую можно применять только для таких игрушечных приёмов, как счётчики. Ничто не может быть так далеко от истины, как это высказывание. Замыкания — это строительные блоки Лиспа. Среды, функции, определённые внутри этих сред и макросы как **defun**, упрощающие их использование — это всё, что нужно для моделирования любой проблемы. Цель этой книги — научить начинающих Лисп программистов, ранее работавших с объектно-ориентированными языками, не идти на поводу своей интуиции к таким системам, как CLOS. Не стоит использовать CLOS там, где достаточно лямбды, даже несмотря на то, что CLOS предлагает ряд преимуществ профессиональным Лисп программистам.

Для того, чтобы мотивировать использование замыканий, представляю вам реалистичный пример: **block-scanner**. Для какой задачи предназначен **block-scanner**? В некоторых формах передачи данных данные передаются в группах (блоках) неопределённых размеров. В основном, эти размеры удобны для нижележащей системы, но, неудобны для программиста приложения поскольку часто размеры определяются такими факторами как буферы операционной системы, блоки жёсткого диска или сетевые пакеты. При этом сканирование потока данных на определённую последовательность значительно усложняется по сравнению со сканированием отдельно взятого блока с помощью обычной процедурой без состояния. Нам приходится сохранять состояние между сканированием каждого блока, поскольку есть вероятность что искомая последовательность будет разделена между двумя (или более) блоками.

Самым простым и естественным способом реализации этого хранимого состояния в современных языках программирования является замыкание. Начальным эскизом сканера блоков, основанного на замыкании является вышеприведённый **block-scanner**. Так же, как и всё остальное в Лисп разработке, создание замыканий представляет из себя итеративный процесс. Мы должны начать с кода в **block-scanner** и постепенно улучшать его. Например: мы можем увеличить его эффективность отказавшись от преобразования строк в списки, или мы можем улучшить информативность, подсчитывая количество появления искомых последовательностей в блоках.

И хотя **block-scanner** является всего лишь начальной реализацией и её ещё предстоит улучшить, она вполне пригодна для демонстрации использования лямбды, окружающей **let**, окружающий лямбду. Вот демонстрация его использования, в роли коммуникативного фильтра, наблюдающего за определёнными словами, находящимися в чёрном списке,

Листинг 2.1: BLOCK-SCANNER

```
(defun block-scanner (trigger-string)
  (let* ((trig (coerce trigger-string 'list))
         (curr trig))
    (lambda (data-string)
      (let ((data (coerce data-string 'list)))
        (dolist (c data)
          (if curr
            (setq curr
                  (if (char= (car curr) c)
                    (cdr curr) ; следующий символ
                    trig))))   ; начать снова
        (not curr))))) ; вернуть t при обнаружении
                       последовательности
```

например *jihad (джихад)*:

```
* (defvar scanner
    (block-scanner "jihad"))
SCANNER

* (funcall scanner ‘‘We will start ")
NIL

* (funcall scanner "the ji")
NIL

* (funcall scanner "had tomorrow.")
T
```

2.7 Let над Lambda над Let над Lambda

Пользователи объектных систем сохраняют значения, предназначенные для общего использования всеми объектами определённого класса, в так называемые *переменные класса (class variables)* или *статические переменные (static variables)*[9]. В Лиспе эта концепция разделения состояния

[9]Термин статический (static) — это один из наиболее перегруженных терминов в языках программирования. Значения, разделяемые всеми объектами класса называ-

между замыканиями обрабатывается средами тем же способом, каким замыкания сохраняют свои состояния. Поскольку доступность среды не ограничена и доступ к ней возможен до тех пор пока мы ссылаемся к ней, то наша среда будет доступна столько, сколько нам будет нужно.

Если мы хотим реализовать глобальное изменение для всех счётчиков, **up** для инкрементирования счётчика каждого замыкания и **down** для декрементирования, то нам нужно использовать шаблон **let**, окружающий лямбду, окружающую **let**, окружающий лямбду:

```
(let ((direction 'up))
  (defun toggle-counter-direction ()
    (setq direction
          (if (eq direction 'up)
            'down
            'up)))

(defun counter-class ()
  (let ((counter 0))
    (lambda ()
      (if (eq direction 'up)
          (incf counter)
        (decf counter))))))
```

В вышеприведённом примере мы расширили **counter-class** из предыдущего раздела. Теперь вызов замыканий, созданных с **counter-class** либо инкрементирует значение счётчика привязки, либо декрементирует его в зависимости от значения привязки направления, общего для всех счётчиков. Заметьте, что мы также воспользовались другой **lambda**, внутри среды направления, создав новую функцию под названием **toggle-counter-direction**, которая изменяет текущее направление для всех счётчиков.

И хотя комбинация **let** и **lambda** оказываются настолько полезными, что другие языки адаптируют их в форме классовых или статических переменных, существуют другие комбинации **let** и **lambda**, позволяющие вам структурировать код и состояния такими способами, которые не имеют прямых аналогов в объектных системах[10]. Объектные системы — это формализация подмножества комбинаций **let** и **lambda**, иногда в эту формализацию включаются такие трюки, как *наследование*[11].

ются статическими переменными в таких языках, как Java, что отдалённо связано со значениями статичности в C.

[10] Но, иногда, эти аналоги можно создавать на основе объектных систем.

[11] Доступность макросов неизмеримо важнее доступности наследования.

По этой причине Лисп программисты редко мыслят в рамках классов и объектов. **Let** и лямбда — фундаментальны; объекты и классы — производные. Как говорит Стил: “объекты” не должны быть примитивами в языках программирования. Если нам доступны ячейки с присваиваемыми значениями и старые добрые лямбда выражения, то объектные системы, в лучшем случае, иногда оказываются полезными, а в худшем случае, узко специализированы и избыточны.

Глава 3

Основы макросов

3.1 Итеративная Разработка

> Лисп помог целому ряду одарённых людей размышлять таким образом, какой ранее не был им доступен.
>
> — Эдсгер Дейкстра

Конструирование макросов — это итеративный процесс: все сложные макросы начинались из более простых макросов. Отталкиваясь от этой идеи, можно сказать что макросы создаются подобно скульптуре из куска камня. Если реализация макроса оказывается недостаточно гибкой, либо результатом оказывается недостаточно эффективное или опасное расширение, то в этом случае профессиональный программист макросов слегка модифицирует макрос, добавляя функционал или исправляя ошибки до тех пор, пока макрос не будет удовлетворять всем требованиям.

Необходимость итеративного процесса в конструировании макросов возникает отчасти потому, что итеративный процесс программирования — это наиболее универсальный и эффективный способ программирования, а другая причина заключается в том, что программирование макросов — это наиболее сложный вид программирования. Поскольку при программировании макросов программисту приходится думать сразу о нескольких уровнях кода, исполняемого в различные моменты времени, то вопросы сложности увеличиваются гораздо быстрее, чем в остальных типах программирования. Итеративный процесс позволяет убедиться в том, что ваша концептуальная модель наиболее полно соответствует тому, что планируется создать. Если бы мы создавали макросы без такой обратной связи, то конструирование макросов стало бы значительно более трудным делом.

В этой главе мы напишем несколько базовых макросов и ознакомимся с двумя основными концепциями макросов: *предметно-ориентированные языки (domain specific languages)* и *структуры управления (control structures)*. После того, как мы изучим эти основные понятия о макросах, мы вернёмся назад и обсудим процесс создания макросов с помощью макросов. На протяжении всей книги мы будем использовать такие техники как захват переменной и введение свободной переменной, кроме этого мы ознакомимся с новым, более удобным синтаксисом определения Лисп макросов.

3.2 Предметно-Ориентированные Языки

Common Lisp, наряду с большинством программных сред, предоставляет функцию **sleep**, которая приостанавливает исполнение процесса на **n** секунд, где **n** — не отрицательный, не комплексный, числовой аргумент. Например, мы можем захотеть уснуть на 3 минуты (180 секунд), в этом случае мы можем вычислить эту форму:

```
(sleep 180)
```

Или, если мы предпочитаем думать о засыпании в терминах минут, то взамен последнего выражения мы можем использовать

```
(sleep (* 3 60))
```

Поскольку компиляторы знают как *перемножать константы*, эти два вызова одинаково эффективны. Для того, чтобы более явно описать наши действия, мы можем определить функцию **sleep-minutes**:

```
(defun sleep-minutes (m)
  (sleep (* m 60)))
```

Подход с определением новой функции для каждой единицы времени оказывается неуклюжим и неудобным. Что нам нужно — так это некая абстракция, которая позволяла бы использовать единицу времени вместе со значением. Нам нужен *предметно-ориентированный язык (domain specific language)*.

Решение, которое мы можем использовать в Лиспе может быть таким же, как и в остальных языках программирования: создать функцию, которая будет принимать значение и единицу времени и возвращать значение умноженное на некоторую константу, относящуюся к конкретной

единице времени. И тут проявляются удобства, предоставляемые Лиспом при реализации единицы времени. В языках, подобных C принято использовать нижележащие типы данных (например: int) и присваивать произвольные значения, относящиеся к различным единицам времени:

```
#define UNIT_SECONDS 1
#define UNIT_MINUTES 2
#define UNIT_HOURS 3

int sleep_units(int value, int unit) {
  switch(value) {
    case UNIT_SECONDS: return value;
    case UNIT_MINUTES: return value*60;
    case UNIT_HOURS: return value*3600;
  }
}
```

Но в Лиспе самый очевидный путь обозначения желаемой единицы — это использование символа. Символ в Лиспе — это нечто, не **eq**'альное другим символам. **Eq** — это самый быстрый оператор сравнения в Лиспе и примерно соответствует сравнению указателей. Поскольку сравнение указателей производится очень быстро, то символы предоставляют нам очень быстрый и удобный способ определения равенства двух и более различных Лисп выражений. В Лиспе мы должны определить функцию **sleep-units%**[1], после этого мы можем использовать единицы времени в наших формах:

```
(sleep-units% 2 'm)
(sleep-units% 500 'us)
```

[1]Примечание переводчика: При компилировании этого кода происходит вывод предупреждений. Я предлагаю использовать следующий вариант:

```
(defun sleep-units% (value unit)
  (defun unit-to-second (unit)
    (case unit
      ((s) 1)
      ((m) 60)
      ((h) 3600)
      ((d) 86400)
      ((ms) 1/1000)
      ((us) 1/1000000)))
  (sleep (* value (unit-to-second unit))))
```

Листинг 3.1: SLEEP-UNITS-1

```
(defun sleep-units% (value unit)
  (sleep
   (* value
      (case unit
        ((3) 1)
        ((m) 60)
        ((h) 3600)
        ((d) 86400)
        ((ms) 1/1000)
        ((us) 1/1000000)))))
```

Поскольку сравнение символов требует сравнения только одного указателя, **sleep-units%** будет скомпилирован в очень быструю диспетчеризацию во времени выполнения:

```
...
524: CMP ESI, [#x586FC4D0] ; 'S
52A: JEQ L11
530: CMP ESI, [#x586FC4D4] ; 'M
536: JEQ L10
538: CMP ESI, [#x586FC4D8] ; 'H
53E: JEQ L9
540: CMP ESI, [#x586FC4DC] ; 'D
546:
...
```

Заметьте, как должна закавычиваться единица времени, указанная в **sleep-units%**. Это происходит потому, что Лисп начинает вычисление функции с вычисления всех аргументов функции и привязывает результаты к переменным, использующимся внутри функции. Числа, строки и некоторые другие примитивы вычисляются в самих себя, поэтому нам не нужно закавычивать числовые значения, переданные в **sleep-units%**. Если же вы хотите чтобы и эти значения не вычислялись, то вы можете закавычить и их:

```
(sleep-units% '.5 's)
```

Листинг 3.2: SLEEP-UNITS

```
(defmacro sleep-units (value unit)
  `(sleep
    (* ,value
     ,(case unit
            ((s) 1)
            ((m) 60)
            ((h) 3600)
            ((d) 86400)
            ((ms) 1/1000)
            ((us) 1/1000000)))))
```

Однако, как правило, символы не вычисляются в самих себя[2]. Когда Лисп вычисляет символ, то он предполагает, что вы ссылаетесь на переменную и пытается найти значение, ассоциированное с этой переменной в заданном лексическом контексте (если же переменная объявлена как специальная, то в этом случае поиск происходит в динамическом окружении).

Если мы хотим избежать закавычивания единицы времени, то нам нужно использовать макрос. В отличие от функции макрос не вычисляет свои аргументы. Для того, чтобы воспользоваться этим фактом, мы заменим функцию **sleep-units%** макросом **sleep-units**. Теперь нам не нужно закавычивать единицу времени:

```
(sleep-units .5 h)
```

И хотя основным предназначением макроса было избежание закавычивания аргумента **unit**, этот макрос оказывается даже более эффективным чем функция, поскольку не производится диспетчеризация во время выполнения: единица времени и, следовательно, множитель известны в момент компилирования. Конечно, когда мы сталкиваемся с ситуацией слишком-хорошо-чтобы-быть-правдой, то чаще всего всё действительно оказывается слишком хорошим, чтобы быть правдой. Выигрыш в эффективности не даётся бесплатно. Отказавшись от диспетчеризации во время выполнения мы теряем способность определять единицу времени во время выполнения. Используя наш макрос мы не можем выполнить следующий код:

[2]Как правило, нет правил без исключения. Некоторые символы вычисляются в самих себя, например: **t**, **nil** и ключевые слова.

Листинг 3.3: UNIT-OF-TIME

```
(defmacro unit-of-time (value unit)
  `(* ,value
      ,(case unit
             ((s) 1)
             ((m) 60)
             ((h) 3600)
             ((d) 86400)
             ((ms) 1/1000)
             ((us) 1/1000000))))
```

```
(sleep-units 1 (if super-slow-mode 'd 'h))
```

Это не сработает, поскольку **sleep-units** ожидает что второй аргумент будет одним из символов, в нашем случае вторым аргументом выступает список в котором первым элементом является **if**.

Напомним, что большинство макросов писались для того, чтобы создать более удобные и полезные программные абстракции, и не предназначались для улучшения эффективности нижележащего кода. Возможно ли извлечь некоторые идиомы из этого кода, с тем, чтобы сделать его более полезным для остальной части нашей программы (и, возможно, для наших будущих программ)? Уже сейчас можно предсказать, что мы можем захотеть выполнять разные операции с временными значениями, чем простой вызов **sleep**. Макрос **unit-of-time** абстрагирует функциональность из макроса **sleep-units**, возвращая значение вместо вызова **sleep**. Параметр **value** может определяться во время выполнения программы, поскольку он является вычисляемым, но **unit** не является вычисляемым, поскольку нам нужна информация во время компиляции, также, как и при **sleep-units**. Вот пример:

```
* (unit-of-time 1 d)
86400
```

Такой простой макрос как **unit-of-time** даёт нам более лучший синтаксис для решения конкретных областей некоторых проблем и может значительно улучшить продуктивность работы и корректность решения задачи. В *разделе Программирование Сверху-Вниз* мы продолжим разработку языка единиц. В отличие от большинства языков программирования, в Лиспе вам доступны те же инструменты, что и у людей создавших вашу программную среду. Для того, чтобы реализовать язык Common

Листинг 3.4: NLET

```
(defmacro nlet (n letargs &rest body)
  `(labels ((,n ,(mapcar #'car letargs)
                ,@body))
     (,n ,@(mapcar #'cadr letargs))))
```

Lisp, достаточно использовать макросы, этих же макросов достаточно, чтобы реализовать ваш собственный предметно-ориентированный язык.

3.3 Управляющие Структуры

Хотя эта книга ориентирована на Common Lisp, идеи, описанные здесь можно применить и для языка программирования Scheme. Scheme — это замечательный язык, и хотя в Scheme отсутствуют многие особенности воспринимаемые Лисп программистами как должное, в нём есть достаточно гибкое ядро, которое при необходимости могут расширить профессиональные Лисп программисты[3]. Аналогичным образом, Scheme программисты приводят, в качестве преимущества, особенности на которых не делается ставка в Common Lisp. Но, сравнение особенностей всех языков, кроме некоторых особенностей, бессмысленно. Часто мы в состоянии перебросить мост между двумя языками. Само собой разумеется, что в роли моста, с помощью которых мы объединяем два языка выступают макросы.

Scheme'овская форма **let** является более мощной чем её коллега в Common Lisp. Scheme'овская форма **let** поддерживает нечто, под названием *именованный let (named let)*. В Scheme вы можете вставить символ перед списком привязок **let** формы и Scheme привяжет функцию под указанным символом вокруг тела **let**[4]. Эта функция принимает новые аргументы для значений, указанных в привязках **let**, предоставляя очень удобный способ выражения циклов.

К счастью, мы можем построить мост между Scheme и Common Lisp с помощью макроса **nlet**. **Nlet** даёт нам возможность писать код в Scheme стиле эмулируя Scheme'овские именованные **let**'ы. В **nlet-fact**, **nlet** ис-

[3]В основном, Scheme и Common Lisp различаются в своих сообществах. Scheme программисты любят говорить о том, как здорово иметь короткую спецификацию языка; Common Lisp программисты любят писать программы.

[4]В Scheme есть только одно пространство имён, поэтому функция привязывается именно здесь.

пользуется для определения функции факториала с помощью именованного **let**:

```
(defun nlet-fact (n)
  (nlet fact ((n n))
        (if (zerop n)
            1
            (* n (fact (- n 1))))))
```

Поскольку **nlet** — это наш первый макрос, сделаем паузу и изучим этот макрос глубже. Иногда для того чтобы понять макрос полезно применить *macroexpand* к примеру, показывающему использование этого макроса[5]. Для этого функции **macroexpand** мы передадим список, представляющий вызов макроса. Учтите, что **macroexpand** будет расширять макросы, в которых первый символ списка является макрос символом и не расширяет вложенные макро вызовы[6]. Теперь мы скопируем вызов **nlet** прямо из **nlet-fact**, закавычим его и передадим в **macroexpand**:

```
CL-USER> (macroexpand
          '(nlet fact ((n n))
            (if (zerop n)
                1
                (* n (fact (- n 1))))))
(LABELS ((FACT (N)
           (IF (ZEROP N)
               1
               (* N (FACT (- N 1))))))
  (FACT N))
T
```

Расширение использует специальную форму **labels**, предназначенную для привязки функции вокруг указанного тела. Именованная функция соответствует символу, используемому в форме именованного **let**. Он получает аргументы значений, привязанных с помощью **nlet**, в данном

[5]Терминология расширения на самом деле довольно неудачна. Никто не говорит что при макрорасширении чего-либо результатом будет больший, расширенный код. Иногда формы расширяются даже в ничто (т.е. **nil**).

[6]Но, **macroexpand** будет продолжать расширение макроса до тех пор, пока первый элемент не будет представлять макрос. **Macroexpand-1** полезен для наблюдения первого шага в этом процессе.

случае только **n**. Поскольку эта функция может быть рекурсивной, **nlet** реализует полезную итерационную конструкцию.

В простых макросах обычно достаточно использовать обратные кавычки, но более сложные макросы должны использовать хотя бы Лисповские функции обработки списков. **Mapcar**, применяет функцию к каждому элементу списка и возвращает список из получившихся значений, особенно часто появляется в макросах. Что характерно, **mapcar** довольно часто появляется и в обычных Лисп программах. Лисп можно настроить так, чтобы его можно было по максимуму использовать для обработки списков. Во всех разновидностях Лисп программирования мы занимаемся тем, что объединяем, разделяем, уменьшаем, отображаем и фильтруем списки. Единственная разница в том, что при программировании макросов вывод непосредственно переходит к компилятору или интерпретатору. Программирование макросов в Лиспе — это, по сути, то же самое что и обычное Лисп программирование.

Но что даёт нам право сказать что **nlet** — это новая управляющая структура? Управляющая структура — это всего лишь забавный способ определения некоторой конструкции, которая не следует поведению функции. Функция вычисляет каждый аргумент слева направо, привязывает результаты в среде и выполняет машинный код, определённый в некоторой **lambda** форме. Поскольку **nlet** не вычисляет напрямую свои аргументы, а вместо этого объединяет их в некоторый кусок Лисп кода, мы изменили выполнение вычисления для **nlet** форм и тем самым создали новую управляющую структуру.

Руководствуясь этим грубым определением, можно сказать что все макросы — по крайней мере все интересные макросы — определяют новые управляющие структуры. Когда люди говорят “используйте макросы только тогда, когда недостаточно функций”, они имеют ввиду определения, где вы не хотите допустить вычисление некоторых аргументов или вы хотите вычислить аргументы в другом порядке или хотите чтобы аргументы вычислялись несколько раз, то в этом случае вам нужно использовать макросы. Функции, вне зависимости от того насколько хорошо они написаны, просто не будут выполнять эти задачи.

Common Lisp очень удобен для создания макросов, макрос **nlet** показывает одно из этих преимуществ. В таких формах привязки как **let**, существует общее соглашение, которое гласит: если переменной не привязано какое-либо значение, то эта переменная будет привязана к **nil**. Другими словами результатом вычисления `(let ((a)) a)` будет **nil**[7]. В

[7]Common Lisp позволяет нам писать даже так: `(let (a) a)` что в итоге даст все тот-же эффект.

Scheme, а этот язык менее дружественный к созданию макросов, этот случай должен проверяться как частный случай при итерации с подобными привязками, по причине того, что `(car nil)` и `(cdr nil)` вернут ошибки типа. В Common Lisp `(car nil)`, `(cdr nil)` и, следовательно, `(car (cdr nil))` и `(cadr nil)` вернут **nil**, позволяя второму **mapcar** в **nlet** продолжать работу даже если используется соглашение о пустом **let** значении. Эта особенность Common Lisp взята из Interlisp [INTERLISP].

Наш макрос **nlet** отличается от именованного **let** в Scheme очень маленькой деталью. В этом случае интерфейс макроса является приемлемым, но расширения может и не быть. В общем случае, при программировании через несколько уровней, наша ментальная модель кода может немного отличаться от реальности. В Scheme, при выполнении хвостовой рекурсии именованного **let** гарантируется неупотребление дополнительного стекового пространства, поскольку Scheme гарантирует, согласно стандарту, выполнение специальной оптимизации. Это не тот случай в Common Lisp, поэтому не исключается переполнение стека в Common Lisp'овской версии **nlet**, чего не произойдёт с именованным **let** в Scheme. В *разделе Проход по Коду с Macrolet* мы рассмотрим как написать версию **nlet** с идентичным интерфейсом но, с потенциально более эффективным расширением[8].

3.4 Свободные Переменные

Свободная переменная (free variable) — это любая переменная или функция, ссылающаяся на выражение, не имеющее специальной глобальной привязки или ограничивающей лексической привязки. В следующем выражении **x** является свободной переменной:

```
(+ 1 x)
```

В следующем примере мы создаём привязку вокруг формы, эта привязка *захватывает (captures)* переменную **x**, лишая её свободы:

```
(let ((x 1))
  (+ 1 x))
```

На первый взгляд терминология свободы и захвата может показаться странной. В конце концов, свобода подразумевает сознательность и

[8]На практике, обычно бывает достаточно и этой версии **nlet**, поскольку компилятор Common Lisp почти наверняка оптимизирует хвостовые вызовы в скомпилированном коде.

возможность принятия решений — очевидно, что выразить всё простым выражением не получится. Но свобода не относится к тому, что может выполнять выражение, скорее, это относится к тому, что мы, как программист, можем выполнять с этим выражением. Например, мы можем взять выражение `(+ 1 x)` и встроить его куда-либо ещё, позволяя нашему выражению получить доступ к **x**, привязанному к окружающему коду. Затем мы можем сказать что код *захватил (captured)* нашу свободную переменную. После того, как свободные переменные в выражении были захвачены, как в предыдущей форме **let**, другой окружающий код не может захватить нашу переменную **x**. Наша в прошлом свободная переменная уже была захвачена. Теперь вполне однозначно ясно куда ссылается **x**. По этой причине понятно почему Лиспу не надо хранить значение, на которое ссылается символ **x** на протяжении всего кода. Как мы описали в деталях в *разделе Лексическая и Динамическая Области Видимости*, Лисп компиляторы забывают символы, которые были использованы для представления лексических переменных.

Возможности макросов означают что свободные переменные в Лиспе гораздо более полезны чем в других языках, хотя в любом, поддерживающим выражения, языке может быть код со свободными переменными. В большинстве языках мы вынуждены подчиняться *ссылочной прозрачности (referential transparency)*. Если в Блаб языке не объявлена глобальная или объектная переменная **x**, то следующий код безусловно неверен:

```
some_function_or_method() {
  anything(1 + x);
```

Нет никакого способа, с помощью которого **some_function_or_method** мог бы создать *неявную привязку* к **x**. В Блабе любое использование переменной должно быть текстово очевидным определением[9]. Языки с примитивной макро системой (такие как C) могут реализовывать некоторые приёмы, но, реализация этих приёмов весьма ограничена. Создание макросов в C непрактично или невозможно, также, как и частные случаи использования свободных переменных.

В Лиспе мы можем оперировать выражениями со свободными переменными так, как нам заблагорассудится и либо объединить их в новое выражение, захватываемое окружающим кодом, или определить специальные глобальные переменные для последующего захвата. Также мы можем написать макрос, который модифицирует и освобождает переменные в выражении или переписывает выражение с целью уменьшения

[9]Или, иногда, в объектно-ориентированном Блабе, определения класса или объекта.

количества свободных переменных (например, как представлено выше, оборачивая выражение в **let** форму) или модифицируя выражение с целью добавления новых свободных переменных. Такое добавление свободных переменных является противоположностью захвата переменных и называется *инъекцией свободной переменной (free variable injection)*.

Простейшей инъекцией свободной переменной является макрос, расширяющийся в ссылку на символ:

```
(defmacro x-injector ()
  'x)
```

Поскольку макрос — это та же функция, то он исполняет своё тело как обычную Лисп форму. Вышеприведённый макрос — инжектор вычисляет закавыченный символ и, конечно, возвращает символ — свободную переменную — которая будет объединена с любым, использующим макрос **x-injector**, выражением. Обсуждая такие инъекции свободных переменных в *On Lisp*, Пол Грэм написал

> *Подобная разновидность лексических взаимодействий обычно рассматривается как источник проблем, нежели как источник удовольствия. Обычно написание макросов в таком ключе является плохим стилем. Из всех макросов в этой книге, только [два изолированных случая] используют вызов среды подобным способом.*

В противоположность *On Lisp* в этой книге получают большое удовольствие от таких разновидностей лексических взаимодействий. Инъекция свободной переменной — это создание макроса с полным знанием лексической среды, в которую он будет расширен — это всего лишь другой подход к программированию Лисп макросов, который особенно полезен при наличии нескольких немного отличающихся лексических контекстов, в которых вы можете захотеть написать почти идентичный код. Хотя главным преимуществом вызова функций является вынос за пределы функции вашей лексической среды, иногда, для Лисп программистов, это всего лишь руководство к действию, которое может быть проигнорировано через использование макросов. Фактически, только привыкнув к этому, некоторые Лисп программисты по возможности всегда стараются писать макросы, расширяя лексический контекст настолько, насколько это возможно, и используют функции только когда им нужно вычислить аргументы или просто для того, чтобы поджать хвост при желании нового лексического контекста. В *разделе Once Only* мы рассмотрим способ

не допустить вывода наружу вашей лексической среды при вычислении аргументов. Сохранение окружающей лексической среды до тех пор, пока это возможно, позволяет нам выполнять очень интересные *комбинации (combinations)* макросов, где макрос добавляет окружающий лексический контекст для использования несколькими другим макросами. Расширения в коде, которые очень сильно используют макросы будут определены как частный случай комбинации макросов и будут рассмотрены в *разделе Рекурсивные Расширения.*

Кратчайший путь между двумя точками — это прямая линия. Свободные переменные и, более глобально, расширенные лексические контексты — это наиболее простейший способ программного построения программ. Здесь применение макросов может показаться хаком и нежелательным со стилистической точки зрения, но даже здесь использование макросов удобно и надёжно. Особенно после того, как мы рассмотрим **macrolet** в *разделе Проход по Коду с помощью Macrolet*, этот стиль программирования — комбинирование макросов — покажется вам более комфортабельным. Только помните, что программирование макросов — это не вопрос стиля; это вопрос мощи. Макросы дают нам возможность выполнять такие вещи, реализация которых невозможна в других языках. Одна из них — это инъекция свободной переменной.

3.5 Нежелательный Захват

Существуют две точки зрения на захват переменной. Захват переменной — это источник некоторых непредсказуемых багов, но, при правильном использовании может быть весьма желаемой особенностью макросов. Начнём рассмотрение захвата переменной с простого макроса, определённого Грэмом в *On Lisp*: **nif**. **Nif** — это *числовой if*, который имеет четыре обязательных пункта, по сравнению с обычным **if**, который имеет два обязательных пункта и необязательный третий пункт. **Nif**, или вернее код, в который расширяется **nif**, вычисляет первый пункт и предполагает, что результат будет не комплексным числом. Далее число вычисляется в одно из трёх пунктов, которое в зависимости от результата может быть положительным (**plusp**), нулём (**zerop**) или отрицательным (3-й пункт). Мы можем использовать его для проверки переменной **x**, следующим способом:

```
(nif x "positive" "zero" "negative")
```

Nif — это идеальная функция для обсуждения захвата переменной и мы будем использовать его в качестве иллюстрации некоторых ключе-

вых пунктов, а также, тестового варианта для новых обозначений при конструировании макросов. Прежде чем перейти к версии **nif** созданной Грэмом, определим почти правильную, но, немного забагованную версию:

```
(defmacro nif-buggy (expr pos zero neg)
  `(let ((obscure-name ,expr))
     (cond ((plusp obscure-name) ,pos)
           ((zerop obscure-name) ,zero)
           (t ,neg))))
```

Niff-buggy расширяется в маленький кусок кода, который использует **let** для привязки результата вычисления предоставленной пользователем формы **expr**. Мы делаем это потому, что вычисление **expr** может привести к *побочным эффектам (side-effects)*, а полученное значение нужно для двух раздельных операций: передача значения в **plusp** и передача значения в **zerop**. Но зачем мы вызываем эту временную привязку? Маленькая ошибка появляется из-за введения постороннего символа, **obscure-name**. Если никто не будет смотреть на расширение макроса, то никто не заметит это имя, вроде бы это не проблема, не так-ли?

В большинстве случаев **nif-buggy** будет работать также, как и **nif**. Если символ **obscure-name** не будет использоваться в формах, передаваемых в **nif-buggy**[10], то не будет и нежелательного захвата переменной. Но что произойдёт если **obscure-name** появится в переданных формах? В большинстве случаев, баг по прежнему не будет проявляться:

```
(nif-buggy
  x
  (let ((obscure-name 'pos))
    obscure-name)
  'zero
  'neg)
```

Даже если **x** будет положительным, и даже если присутствует инъецированный запрещённый символ в макро расширении **nif-buggy**, этот код будет работать так, как и было задумано. Если созданы новые привязки, а код внутри привязки всегда ссылается на созданную привязку, то в этом случае нежелательный захват переменной не возникает. Проблема возникнет только тогда, когда использование **obscure-name** *пересечёт своё* использование в расширении. Вот пример нежелательного захвата переменной:

[10]Или в макро расширениях переданного кода. Смотрите сублексическую область видимости.

```
(let ((obscure-name 'pos))
  (nif-buggy
    x
    obscure-name
    'zero
    'neg))
```

В этом случае, **obscure-name** будет привязываться к результату вычисления **x**, поэтому, вопреки задумке, символ **pos** не будет возвращаться[11]. Происходит это по причине того, что наше использование символа пересекается с невидимым использованием привязки. Иногда упоминая код с такими невидимыми привязками, говорят что он не *ссылочно прозрачен*.

Но разве это не академический вопрос? Конечно, мы можем подумать о достаточно редких именах, таких, при использовании которых эта проблема никогда не появится. Да, во многих случаях пакеты и грамотное именование переменных может решить проблему захвата переменной. В коде, непосредственно написанном программистом, редко возникают баги, связанные с захватом переменной. Большинство проблем с захватом переменной проявляются тогда, когда другие макросы начинают использовать ваши макросы (комбинируются с вашими макросами) самым неожиданным образом. Пол Грэм даёт прямой ответ на вопрос о защите от нежелательного захвата переменной:

> Зачем писать программы с маленькими багами, когда вы можете писать программы без багов?

Я думаю мы можем развить этот вопрос дальше: вне зависимости от тонкости проблемы, зачем делать что-то неправильно, когда вы можете делать это правильно?

К счастью, оказывается, что захват переменной, в тех случаях, когда это является проблемой, — это задача с простым решением. Последнее утверждение — это спорный вопрос для многих людей, особенно для тех, кто не любит очевидные решения и посвящают большую часть своего времени на поиски других решений. Как профессиональному программисту макросов, вам придётся столкнуться со многими решениями проблемы захвата переменной. Текущее популярное решение — это использование так называемых *гигиенических макросов (hygienic macros)*[12]. Эти

[11] Конечно, на самом деле это неправильное поведение было намеренным. Проблемы с захватом переменных очень редко бывают такими очевидными и запланированными. Гораздо чаще они бывают менее очевидными и неожиданными.

[12] Другой популярный термин — это *"макросы по примеру"("macros by example")*.

решения пытаются ограничить или устранить воздействие нежелательных захватов переменной, но, к несчастью выполняют это за счёт желательных захватов переменной. Почти все подходы, используемые для уменьшения воздействия захвата переменной, также ограничивают вас в использовании **defmacro**. Гигиенические макросы, в лучшем случае, — это ограждение, предназначенное для безопасности новичков; а в худшем случае гигиенические макросы принимают форму электрического забора, помещающего свои жертвы в стерильную, захвато-безопасную тюрьму. Более того, недавние исследования показали, что гигиенические макро системы, подобные тем, что реализованы в различных ревизиях Scheme, могут быть уязвимы ко многим интересным проблемам захвата [SYNTAX-RULES-INSANE] [SYNTAX-RULES-UNHYGIENIC].

Настоящее решение проблемы захвата переменных известно как *генерированный символ (generated symbol)* или, кратко, *gensym*. **Gensym** — это способ, с помощью которого Лисп получает имя для нашей переменной. Но, вместо того, чтобы использовать такие неудачные имена как наш **obscure-name**, Лисп использует хорошие имена. Действительно хорошие имена. Эти имена настолько хороши и уникальны, что нет такого способа (кроме самого **gensym**) по которым можно было бы создать такое имя снова. Каким образом это становится возможным? В Common Lisp символы (имена) ассоциируются с *пакетами (packages)*. Пакет — это коллекция символов, из которых вы можете получить указатели к строкам их **symbol-name (имя-символа)**. Самым важным свойством этих указателей (обычно просто называемых символами) в том что они будут **eq**'абельны остальным указателям (символам), обнаруживаемым в этом пакете с тем же самым **symbol-name**. **Gensym** — это символ, который не существует ни в одном пакете, поэтому не существует такого **symbol-name**, который бы вернул символ **eq**'абельный **gensym**'у. **Gensym**'ы нужны тогда, когда вам нужно дать знать Лиспу что некоторый символ должен быть **eq**'абелен некоторым другим символам в выражении без необходимости их именования. Поскольку вы ничему не даёте имён, то, соответственно, не происходит никаких столкновений имён.

Таким образом, следуя этим трём несложным, но очень важным правилам, решение проблемы нежелательного захвата переменной в Common Lisp становится простой задачей:

> Всякий раз, когда вы оборачиваете ваш код в макросе лексической или динамической привязкой, то для именования этой привязки используйте **gensym**, исключая те случаи, когда вам нужен захват переменной.
>
> Каждый раз, когда вы оборачиваете привязку функции или

> **macrolet** или макрос **symbol-macrolet** вокруг кода вашего макроса, и вы хотите чтобы эти элементы не захватывались в оборачиваемом коде, то именование этого макроса или функции нужно выполнять с помощью **gensym**. Убедитесь, что эта привязка не конфликтует с остальными специальными формами, макросами или функциями, определёнными по стандарту.
>
> Никогда не определяйте или пере-привязывайте специальную форму, макрос или функцию, определённую в Common Lisp.

Существуют некоторые, отличающиеся от Common Lisp, Лиспы, такие как Scheme, обладащие неудачным свойством комбинирования пространства имён переменных с пространством имён функций/макросов. Иногда эти Лиспы называются *lisp-1*, в отличие от Common Lisp с его разделённым пространством имён, который называется *lisp-2*. При работе с макросами в гипотетическом *lisp-1* нам придётся следовать дополнительным двум правилам:

> Убедитесь, что добавленные лексические или динамические привязки не конфликтуют с добавленными функциями или макро привязками, или любыми другими специальными формами, макросами или функциями, определёнными по стандарту.
>
> Убедитесь, что добавленная функция или макро привязка не конфликтует с добавленными лексическими или динамическими привязками.

Мудрое архитектурное решение отделения пространства имён переменных от пространства имён функций в Common Lisp устраняет большую часть проблем связанных с нежелательным захватом переменной. Конечно, *lisp-1* Лиспы не содержат никаких теоретических барьеров в создании макросов: если мы будем следовать предыдущим двум правилам, то мы можем избежать захвата переменной тем же способом что и в Common Lisp. Однако, отслеживание символов в единственном и изолированном пространстве имён может значительно усложниться при программировании сложных макросов. Присутствие взаимного обмена именами позволяет сделать вывод о том, что процесс создания макросов станет сложнее.

Scheme — замечательный язык, но проблема незавершённого стандарта[13], это также касается единственного пространства имён, перевешивает все достоинства этого языка и делает его непригодным к созданию серьёзных макро конструкций[14]. Ричард Гэбриэл и Кент Питман обобщили эту проблему в следующей запоминающейся цитате [LISP2-4LIFE]:

> Есть две точки зрения на проблему макросов и пространств имён. Первая в том, что единственное пространство имён носит фундаментальный характер, следовательно, макросы являются проблемой. Вторая заключается в том, что макросы — фундаментальны, следовательно, единое пространство имён является проблемой.

Поскольку число пространств имён менее важно возможности конструирования макросов, то можно сделать вывод, что Scheme — это *неправильный* выбор, а Common Lisp — это *правильный* выбор.

Тем не менее, постоянный вызов **gensym** каждый раз, когда нам нужен безымянный символ — это неудобно и неуклюже. Нет ничего удивительного, в том, что разработчики Scheme экспериментировали с так называемой *гигиенической* макро системой для того, чтобы не печатать **gensym** постоянно. В Scheme конструирование макросов построено через предметно-ориентированный язык. Это был неправильный выбор. Scheme — это чрезвычайно мощный мини-язык, но в этом был потерян смысл макросов: макросы мощны тем, что написаны на Лиспе, а не на упрощённом пре-процессорном языке.

Эта книга представляет новый синтаксис для **gensym**'ов, преимущество этого синтаксиса в том, что он до разумных размеров короче чем традиционные Лисп выражения. Наш новый синтаксис для **gensym**'ов, который будет использоваться как основа для большинства макросов в этой книге, можно более точно охарактеризовать как снятие слоёв с простого макроса, использующего наш синтаксис. Продолжим рассматривать пример **nif**, с предыдущего раздела. Вот как Грэм определяет захвато-безопасный **nif**:

```
(defmacro nif (expr pos zero neg)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ((plusp ,g) ,pos)
```

[13]Это касается и макросов и исключений.

[14]На протяжении всей книги мы увидим что есть множество причин предпочесть Common Lisp, а не Scheme.

```
            ((zerop ,g) ,zero)
            (t ,neg)))))
```

Это пример корректного использования **gensym**. Как мы увидели в предыдущем разделе, макрос, который может расширить ввод пользователя в нечто, могущее наложиться на одну из его переменных, должен заботиться о захвате переменной. Грэм представляет макрос аббревиатуру **with-gensyms**, сокращающий текст в случаях, когда нужно создать несколько **gensym**'ов:

```
(with-gensyms (a b c)
  ...)
```

Расширяется в

```
(let ((a (gensym))
      (b (gensym))
      (c (gensym)))
  ...)
```

Поскольку **gensym** очень широко используется в форме **defmacro**, мы решили продолжить дальнейшее сокращение. В частности, заметьте, что для каждого **gensym**'а мы печатаем временное имя (такое, как **a**, **b** и **c**) по крайней мере дважды: первый раз когда мы объявляем его **gensym**'ом и другой раз, когда используем его. Можем ли мы устранить эту избыточность?

Для начала рассмотрим как макрос **nif** использует **gensym**'ы. При расширении макроса **nif** происходит вызов **gensym**, который возвращает сгенерированный символ. Поскольку гарантируется уникальность этого символа, мы можем безопасно использовать его в расширении макроса, будучи уверенными в том, что этот символ никогда не будет захвачен через какую-либо непреднамеренную ссылку. Но, нам по прежнему необходимо как то называть этот **gensym** в определении макроса для того, чтобы мы могли использовать его в нужных местах в расширении. Грэм, в пределах макроса **nif** называет этот **gensym** как **g**. Учтите, что это имя никогда не появится в макро расширении **nif**:

```
CL-USER> (macroexpand '(nif x 'pos 'zero 'neg))
(LET ((#:G1194 X))
  (COND ((PLUSP #:G1194) 'POS)
        ((ZEROP #:G1194) 'ZERO)
        (T 'NEG)))
T
```

Название **g** исчезает в расширении макроса. Поскольку **g** был привязан только к нашей среде расширителя, имя, данное такой переменной не имеет никакого отношения к захваченному имени в расширениях. Все **g** в расширении будут замещены символом с именем **G1605**. Перед именем стоит префикс `#:` и обозначает, что символ не *входит* ни в один пакет — это **gensym**. При печати форм, обязательно следует использовать такие префиксы, поскольку Лисп должен прекратить работу если мы захотим использовать (вычислить) эту форму после его прочтения. Мы должны прервать работу Лиспа по той причине, что невозможно просто глядя на два имена **gensym**'ов сказать, являются ли они **eq**'абельными или нет — это остаётся на усмотрении системы. Лисп прерывает работу интересным способом: поскольку каждый раз символ `#:` считывает новый созданный символ, а поскольку `(eq '#:a '#:a)` никогда не будет истинным значением, по той причине, что внутренние символы `#:G1605` в предыдущем расширении не соответствуют привязкам, созданным формой **let**, Лисп считает что выражение обладает свободной переменной, давая нам знать что эта форма обладает **gensym**'ами.

Несмотря на **gensym**'овское поведение по-умолчанию, всё же возможно сохранение и перезагрузка макро расширений. Для более точного отображения напечатанных отображений формы с **gensym**'ами, мы можем включить режим ***print-circle**** при печати результатов[15]:

```
CL-USER> (let ((*print-circle* t))
           (print
            (macroexpand '(nif x 'pos 'zero 'neg)))
           t)

(LET ((#1=#:G1195 X))
  (COND ((PLUSP #1#) 'POS) ((ZEROP #1#) 'ZERO) (T 'NEG)))
T
```

В вышеприведённой форме Лисп принтер использует считывающие макросы `#=` и `##`. Эти считывающие макросы позволяют нам создать формы, ссылающиеся на *самих себя (self-referential)*, более подробно об этих формах мы поговорим в *разделе Циклические Выражения*. Если мы прочитаем предыдущую форму, символы, используемые внутри будут теми же самыми, что и символы, используемые в привязке **let** и поэтому выражение будет работать. Кажется, что в вышеприведённом выражении удалось избежать избыточности с двойным именованием. Есть ли способ добавления этого способа в шаблон макроса, пишущего макросы?

[15]Мы возвращаем **t**, поэтому мы не видим формы, возвращённой **print**'ом. Возвращение **(values)** — это обычное явление.

Листинг 3.5: G-BANG-SYMBOL-PREDICATE

```
(defun g!-symbol-p (s)
  (and (symbolp s)
       (> (length (symbol-name s)) 2)
       (string= (symbol-name s)
                "G!"
                :start1 0
                :end1 2)))
```

Помните, что в определении макроса мы можем именовать наши **gensym**'ы во что угодно, даже, как это сделал Грэм, в простые имена, такие как **g**, и они исчезнут при расширении макроса. Так как у нас есть свобода в именовании, то следует как-то стандартизировать соглашение об именовании **gensym**'ов. В виде компромисса между краткостью и уникальностью, считается что любой символ, начинающийся с двух символов **G!** и последующим после них одним или более символами рассматривается как специальный ссылающийся **gensym**'овский символ называемый *G-банг символ (G-bang symbol)*. Мы определили предикат, **g!-symbol-p**, определяющий является ли переданный атом G-банг символом.

Теперь, после того как мы стандартизировали G-bang символы, мы можем создать макрос, пишущий для нас макро определения и использующий сокращение написания макроса известное как *автоматические gensym'ы (automatic gensyms)*. Макрос **defmacro/g!** определяет предметно — ориентированный язык, предназначенный для решения задачи по написанию макросов, но в этом языке удаётся сохранить всю мощь Лиспов. **Defmacro/g!** прост, но его использование и то, как он работает может быть не очевидным. По этой причине, а также по причине того, что это первый настоящий макрос впервые представленный вам в этой книге, мы не спеша проанализируем **defmacro/g!**.

При исследовании любого макроса, первый шаг, который вам следует сделать — это *остановиться*. Не думайте о макросе как о трансформации синтаксиса или любой другой абстракции. Думайте о макросе как о функции. В основе макроса лежит функция и макрос работает таким же способом. Функция — это заданные не вычисленные выражения, переданные как аргументы и ожидается возвращённый код для Лиспа с последующей вставкой в другие выражения.

Итак, размышляя о **defmacro/g!** как о функции, рассмотрим его исполнение. Поскольку мы программируем обычную Лисп функцию, то

Листинг 3.6: DEFMACRO-WITH-G-BANG

```
(defmacro defmacro/g! (name args &rest body)
  (let ((syms (remove-duplicates
               (remove-if-not #'g!-symbol-p
                              (flatten body)))))
    `(defmacro ,name ,args
       (let ,(mapcar
              (lambda (s)
                `(,s (gensym ,(subseq
                               (symbol-name s)
                               2))))
              syms)
         ,@body))))
```

это значит что мы имеем доступ ко всем особенностям Лиспа и даже утилитам, добавленными позже к языку. В **defmacro/g!** мы используем утилиту Грэма **flatten**, Лисповские функции **remove-if-not** и **remove-duplicates** и наш предикат G-bang символа **g!-symbol-p**, для создания нового списка состоящего из всех G-bang символов, найденных внутри тела формы, переданной в наш макрос. Дальше мы используем шаблон обратную кавычку для того, чтобы вернуть список представляющий код, в который нам нужно расширить макрос. Поскольку в нашем случае мы пишем улучшение для **defmacro**, то нам нужно чтобы наш код самостоятельно расширился в форму **defmacro**. Но, мы добавили новые удобные особенности в язык **defmacro** и нужно создать чуть более сложное расширение. Для того, чтобы применить gensym к каждому G-bang символу, найденному в теле макроса, мы используем **mapcar** для применения функции ко всему списку из собранных G-bang символов, создавая новый список, который может быть объединён в форму **let**, устанавливая привязки для каждого gensym'а[16].

Заметьте как лямбда, используемая в отображении, содержит выражение, созданное с помощью оператора обратной кавычки, эта ситуация должна была бы являться, но не является, вложенной *обратной кавычкой (nested backquote)*. Поскольку **mapcar**, применяющий эту функцию, не *закавычен (unquoted)*, то не закавыченные выражения во вложенной обратной кавычке по прежнему вычисляются в нашем исходном контек-

[16]Функции **gensym** может быть передан аргумент в виде единственной строки. Это меняет печатаемое имя gensym'а, что может помочь при чтении расширений. Для этих целей **defmacro/g!** использует печать имён символов в G-bang символе.

сте. Вложенные обратные кавычки известны своей чрезвычайной сложностью в освоении и мы вернёмся к рассмотрению этой концепции в *главе Считывающие Макросы.*

Итак, что конкретно позволяет нам выполнять **defmacro/g!**? Он позволяет нам использовать технику автоматических gensym'ов — способ проверки существования определённых символов в лексической области видимости кода, переданного макросу[17]. Если мы не будем использовать никаких G-bang символов, то **defmacro/g!** будет работать также, как и **defmacro**. Но, появление любого G-bang символа в теле расширения макроса будет интерпретироваться следующим образом:

> Я хочу, чтобы была создана привязка вокруг этого выражения и я уже дал символ. Сделай это.

Мы можем использовать **defmacro/g!** для явного создания gensym'а в таком определении **nif**:

```
(defmacro/g! nif (expr pos zero neg)
  `(let ((,g!result ,expr))
     (cond ((plusp ,g!result) ,pos)
           ((zerop ,g!result) ,zero)
           (t ,neg))))
```

Когда нам нужно использовать gensym мы просто его используем. Конечно, нам нужно соблюдать осторожность, потому что все ссылки на G-bang символы вычисляются только макро расширением, поскольку это единственное место, где привязан gensym[18]. Разкавычивание G-bang символов, как это показано выше, происходящее внутри обратной кавычки — наиболее очевидный способ выполнения этой операции и мы можем увидеть в этом параллель с разкавычиванием символа **g** в исходном определении **nif** Грэма.

Мы определили макрос **nif**, который вроде как должен работать также как и Грэмовский **nif**, но это улучшение выглядит слишком хорошим чтобы быть правдой. Это действительно работает? Взглянем на макро расширение[19] прежде чем выносить какое-либо суждение:

---

[17]На данный момент это просто упрощение. Читайте раздел о суб-лексической области видимости.

[18]G-bang символы не должны появляться в расширении самих себя — именно этого мы пытаемся избежать с помощью gensym'ов.

[19]Мы использовали **macroexpand-1**, благодаря чему произвели единственное расширение макроса **defmacro/g!** не вызвав последующее расширение **defmacro**.

```
CL-USER> (macroexpand-1
          '(defmacro/g! nif (expr pos zero neg)
             `(let ((,g!result ,expr))
                (cond ((plusp ,g!result) ,pos)
                      ((zerop ,g!result) ,zero)
                      (t ,neg)))))
(DEFMACRO NIF (EXPR POS ZERO NEG)
  (LET ((G!RESULT (GENSYM "RESULT")))
       `(LET ((,G!RESULT ,EXPR))
             (COND ((PLUSP ,G!RESULT) ,POS)
                   ((ZEROP ,G!RESULT) ,ZERO)
                   (T ,NEG)))))
T
```

Кажется, что **defmacro/g!** написал почти тот же код что написал Грэм в исходной версии **nif**. Рассматривая этот пример использования **defmacro/g!** мы видим что в этом расширении создаются только gensym привязки. **Nif**, определённый с помощью **defmacro/g!** свободен от проблем захвата переменных.

Так как **defmacro/g!** сам является макросом, могут ли возникнуть проблемы в среде расширения макроса с нежелательным захватом или заменой? Как и в любой достаточно сложной абстракции такое поведение возможно. В том же смысле в каком захват переменной — это недостаток, некоторые особенности **defmacro/g!** могут оказаться недостатками, существующими в самой архитектуре конструкции[20]. Как всегда, самое лучшее решение — это полное понимание абстракции.

Ещё одним *интересным случаем* является применение **defmacro/g!** в G-bang макросе, определяющим G-bang макрос. **Defmacro/g!** вводит набор привязок в среду расширения, каждая привязка связана с gensym'ом, который при необходимости может использовать макрос. Если возникает такой случай, когда есть несколько возможных привязок gensym'а, то их всегда можно различить по контексту. Другими словами, вы всегда можете указать какую среду следует использовать gensym'у, основываясь на среде в которой он вычисляется. Вот один синтетический пример:

```
(defmacro/g! junk-outer ()
  `(defmacro/g! junk-inner ()
     `(let ((,g!abc))
        ,g!abc)))
```

[20]Хотя не безопасно исключать ошибку самого программиста.

Здесь создаётся два gensym'а. Использованию **g!abc** предшествует только одно разкавычивание (запятая) поэтому мы знаем, что расширение относится к внутреннему gensym'у, созданному с помощью расширения **junk-inner**. Если же вместо одного разкавычивания было бы два разкавычивания, то они стали бы относиться к gensym'у созданному в расширении **junk-outer**.

Defmacro/g! использует функцию Грэма **flatten**. **Flatten**, как было описано в *разделе Лисп Утилиты*, получает дерево cons структуры — наш Лисп код — и возвращает новый список из всех листьев/атомов. Использование **flatten** в **defmacro/g!** — это простой пример *прохода по коду (code-walking)*, тема, к которой мы будем возвращаться на протяжении этой книги.

Упражнение: Что произойдёт в предыдущем G-bang макросе, определяющим G-bang макрос, если первый gensym будет разкавычен один раз, а другой два раза?

3.6 Once Only

Питер Норвиг — блестящий программист и автор. Его книги по ИИ, особенно *Искусственный Интеллект: Современный Подход (Artificial Intelligence: A Modern Approach [AIMA])*, обязательны к прочтению всем людям, работающим с наиболее трудными проблемами в области информатики. Скорее всего, Норвиг известен Лисп программистам по книге *Парадигмы Программирования Искусственного Интеллекта: Обучающие Примеры на Common Lisp (Paradigms Of Artificial Intelligence Programming: Case Studies in COMMON LISP [PAIP])*. И хотя эта книга довольно старая, она обязательна к прочтению всеми серьёзными Лисп студентами, поскольку содержит описание многих важных идей Лиспа[21]. Этот раздел посвящён Питеру Норвигу, а название раздела взято из названия макроса, описанного в PAIP. Тему последних нескольких страниц, описывающих реализацию функции последовательности, можно выразить так

> Once-only: Урок Макрологии

Далее следует чрезвычайно интригующее выражение:

[21] Один из, вечно актуальных, советов PAIP по Common Lisp следующий: никогда не смешивайте аргументы *&optional* и *&key* в деструктурирующих формах lambda или defmacro. Иначе будет больно!

> Если вы можете понять как писать и когда использовать once-only, то это значит что вы полностью освоили макросы.

Как мы уже знаем, никто не в состоянии полностью понять макросы. Понимание одного макроса, пусть даже такого важного как **once-only** приближает вас к пониманию макросов также как и понимание важной теоремы может помочь в полном освоении математики. Поскольку их возможности (математики и макросов) безграничны, то полное понимание математики или макросов абсолютно невозможно.

Здесь мы не будем давать определение **once-only** Норвига, но, поскольку это сложный макрос с некоторыми интересными свойствами, мы реализуем его несколько по-другому. Изначально **once-only** был реализован для программной среды канувшей в лету *Лисп машины* и не был внесён в Common Lisp по несущественным причинам.

Идея лежащая в основе **once-only** — это окружение макро расширения кодом, который создаст новую привязку. После вычисления макро расширения, эта привязка будет инициализирована результатом вычисления одной формы, переданной макросу в виде аргумента. Затем код в теле **once-only** может использовать привязку, которой, конечно, не нужно пересчитывать форму, переданную макросу. Форма, переданная макросу в виде аргумента вычисляется один единственный раз. Once-only (только единожды).

В качестве примера **once-only** Норвиг приводит макрос **square**. Его расширение получает один аргумент и возвращает результат умножения этого аргумента на самого себя:

```
(defmacro square (x)
  `(* ,x ,x))
```

Такой подход будет работать со многими вещами: большинство переменных, чисел и других форм, которые могут свободно вычисляться столько раз, сколько это необходимо. Но, стоит только передать этой версии **square** формы с *побочными эффектами (side-effects)*, как все ставки будут отменены. Конечно, поведение по прежнему будет детерминированным, но теперь трудно предсказать к чему оно приведёт. При работе с этим макросом, форма будет вычислена дважды. Но, поскольку эти сущности усложняются очень быстро, то в целом, все ставки будут отменены. Назначение **once-only** — создание удобства и возможность избежать этих нежелательных побочных эффектов. Заметьте, что если бы мы использовали функцию, то такое поведение досталось бы нам бесплатно. Мы уйдём с области надуманных книжных примеров, опустимся

на грешную землю и определим **square** в виде функции, функция будет выглядеть примерно так:

```
(defun square (x)
  (* x x))
```

Мы можем использовать любую форму в качестве аргумента в этом определении функции **square**. Поскольку этот аргумент будет вычисляться только один раз, то будут удовлетворены наши представления и концептуальные модели побочных эффектов. Во многих примерах мы ожидаем, что выражение, написанное нами один раз будет вычислено только единожды. С другой стороны, одна из составляющих частей макроса — нарушение этого предположения с помощью манипулирования частотой и порядком выполнения вычисления. Например, в таких вещах как циклы (loops), мы можем захотеть вычисления выражений более одного раза. Кроме того, мы даже можем захотеть чтобы эти выражения вообще не вычислялись, поскольку нам может понадобиться от них что-то другое, а не результат вычисления.

Once-only позволяет нам указать конкретные параметры в нашем макро расширении, которые должны вычисляться только один раз с порядком вычисления слева-направо, как в лямбде. Вот как будет выглядеть макрос **square** вкупе с макросом **once-only**:

```
(defmacro square (x)
  (once-only (x)
    `(* ,x ,x)))
```

Конечно, если вы хотите подвергнуть **once-only** все аргументы вашего макроса, то в этом случае взамен макроса следует использовать функцию (лямбду). Мы ещё вернёмся к этому вопросу, но, поскольку эта книга не содержит прямой реализации **once-only**, то мы введём альтернативную реализацию этой функциональности в нашу нотацию макроса. Хотя существуют множество интересных реализаций **once-only** [PAIP-P853] [PRACTICAL-CL-P95], в этом разделе мы введём новую технику, созданную в комбинации с **defmacro/g!**.

Первый шаг в нашей реализации once-only — это создание некоторых новых предикатов и функций-утилит. С целью нахождения компромисса между краткостью и уникальностью мы зарезервируем другой набор символов для нашего персонального использования. Все символы начинающиеся с сочетания символов **O!** называются *O-bang символами.*

Листинг 3.7: O-BANG-SYMBOLS

```
(defun o!-symbol-p (s)
  (and (symbolp s)
       (> (length (symbol-name s)) 2)
       (string= (symbol-name s)
                "O!"
                :start1 0
                :end1 2)))

(defun o!-symbol-to-g!-symbol (s)
  (symb "G!"
        (subseq (symbol-name s) 2)))
```

Листинг 3.8: DEFMACRO-BANG

```
(defmacro defmacro! (name args &rest body)
  (let* ((os (remove-if-not #'o!-symbol-p args))
         (gs (mapcar #'o!-symbol-to-g!-symbol os)))
    `(defmacro/g! ,name ,args
       `(let ,(mapcar #'list (list ,@gs) (list ,@os))
          ,(progn ,@body)))))
```

Предикат, используемый для того, чтобы отличать O-bang символы от других объектов будет называться так: **o!-symbol-p**. Его определение очень близко к **g!-symbol-p**. Кроме того мы создадим удобную функцию-утилиту для смены O-bang символа на G-bang символ, сохраняя остальные символы после bang'а: **o!-symbol-to-g!-symbol**. Эта функция-утилита будет использовать удобную функцию-утилиту Грэма **symb** для создания новых символов.

Defmacro! представляет последний шаг в нашем языке определения макросов — он добавляет свойство once-only. **Defmacro!** комбинируется с **defmacro/g!** из предыдущего раздела. Поскольку **defmacro!** расширяется прямо в форму **defmacro/g!**, то происходит *наследование (inherits)* поведения автоматических gensym'ов. Для создания сложных сочетаний очень важно понимать работу всех скомбинированных частей. Напомним, что **defmacro/g!** ищет символы начинающиеся с G-bang и автоматически создаёт gensym'ы. Расширяясь в форму с G-bang символами, **defmacro!** может избежать дублирования gensym'а при реализации once-only.

Defmacro! создаёт сокращение известное как *автоматический once-only*. С помощью автоматического once-only мы можем предварить один или более символов в аргументах макроса O-bang'ом, таким образом создав O-bang символы определённые с помощью **o!-symbol-p**. Благодаря этому **defmacro!** понимает что мы хотим создать привязку в создаваемом коде. Эта привязка будет содержать результат вычисления кода, переданного в виде аргумента макроса. Эта привязка будет доступна расширению макроса через gensym. Но как нам сослаться на gensym при создании расширения? Через использование эквивалентного G-bang символа определённого с помощью **o!-symbol-to-g!-symbol**.

Реализация зависит от возможностей **defmacro/g!**. С помощью утилиты **o!-symbol-to-g!-symbol** мы создаём новые G-bang символы для добавления в форму **defmacro/g!**. Поскольку у нас уже есть автоматические gensym'ы, то once-only легко реализовать, об этом свидетельствует краткость определения **defmacro!**.

Вернёмся на короткое время на землю синтетических книжных примеров и реализуем макрос **square**, на этот раз с помощью **defmacro!**:

```
(defmacro! square (O!x)
  `(* ,g!x ,g!x))
```

Применим macroexpand к этому выражению:

```
CL-USER> (macroexpand
  `(square (incf x)))
(LET ((#:X1207 (INCF X)))
  (* #:X1207 #:X1207))
T
```

В предыдущем разделе я упомянул что мы передаём строковое значение в **gensym** для всех G-bang символов. Это значительно облегчает изучение подобных форм. Также нет ничего удивительного в таком имени gensym'ов как **#:X1633**, если бы мы отладили определение **defmacro!** в последнем **square**, то обнаружили бы связь между этим символом и символом, используемом в определении макроса: **X**. Возможность сравнивания символов определения и расширения и наоборот становится проще если информация находится в печатаемых именах используемых gensym'ов, как это было сделано в расширениях **defmacro/g!**[22].

[22]Это же является причиной возникновения чисел в печатаемых именах gensym'а, числа определяются с помощью ***gensym-counter***. Этот счётчик позволяет нам различать экземпляры gensym'ов с одинаковыми печатаемыми именами.

Листинг 3.9: NIF

```
(defmacro! nif (o!expr pos zero neg)
  `(cond ((plusp ,g!expr) ,pos)
         ((zerop ,g!expr) ,zero)
         (t ,neg)))
```

Помимо менее выразительного использования и более удобного вывода расширения по сравнению с традиционным **once-only**, в **defmacro!** есть ещё одна ключевая особенность. В обычном **once-only** привязка gensym используется для получения доступа к созданной лексической переменной с тем же именем, что и аргумент макро расширения, *затеняющие (shadows)* аргументы макроса, в результате, к ним нельзя получить доступ из определения макроса. Поскольку **defmacro!** разделяет это на два различных типа символов, G-bang символы и O-bang символы, то мы можем писать макро расширения, использующие оба значения. В качестве примера, вот ещё одно определение макроса **square**:

```
(defmacro! square (O!x)
  `(progn
     (format t "[~a gave ~a]~%"
             ',o!x ,g!x)
     (* ,g!x ,g!x)))
```

Это может использоваться так:

```
CL-USER> (defvar x 4)
4
CL-USER> (square (incf x))
[(INCF X) gave 5]
25
```

Обратите внимание: мы *закавычиваем* разкавыченный O-bang символ в последнем определении **square**. Мы делаем это потому, что нам не нужно вычислять эту форму снова. Расширение сгенерированное от **defmacro!** уже вычислило его. Мы просто хотим получить форму, переданную в square и использовать её по другому назначению, в данном случае мы используем её в роли сырого отладочного выражения. Однако, даже если выражение уже было вычислено один раз, ничто не мешает нам вычислить эту форму ещё раз если этого требуют наши абстракции.

Язык **defmacro!** позволяет нам осуществлять удобный, гранулированный контроль над вычислением аргументов, переданных в наши макросы. Если мы добавим префиксы в виде O-bang ко всем символам, представляющим аргументы в макро определении, а в макро определении будем использовать только соответствующие G-bang символы, то наши выражения будут теми же лямбда выражениями — каждая форма вычисляется только один раз, слева направо. Без этих символов в **args** и без использования G-bang символов в расширении, **defmacro!** будет работать также, как и обычный **defmacro** Common Lisp'a.

Defmacro! — наиболее удобен при итеративной разработке макросов. Поскольку мы простым добавлением двух символов в аргумент макроса получаем вычисление в лямбда стиле, а использование gensym'ов становится таким же лёгким как написание этого слова, то мы можем легко менять наше представление о макросе. **Defmacro!** можно представить как более плотно подогнанную оболочку над лямбдой, чем Common Lisp'овский **defmacro**. По этим причинам в итеративной разработке мы будем использовать **defmacro!** в качестве главного определителя макросов на протяжении всей оставшейся части книги.

Вернёмся с макросу **nif** Грэма. При переписывании этого макроса с использованием **defmacro!** мы упомянули, что аргумент **expr**, для которого мы создавали gensym вычисляется только один раз. Здесь мы использовали **defmacro!** для того, чтобы этот аргумент вычислялся только один раз назвав аргумент как **o!expr**. Эта реализация **nif** представляет последний шаг в нашей эволюции этого макроса.

Defmacro! размывает границу между макросом и функцией. Именно эта особенность, возможность использования некоторых O-bang символов в аргументах макроса наряду с обычными символами делает **defmacro!** особенно полезным. Как обратная кавычка позволяет вам перевернуть поведение закавычивания, также **defmacro!** позволяет вам перевернуть семантики вычисления аргументов макроса из обычных не вычисляемых макро форм в однократно вычисляемые справа налево лямбда аргументы.

3.7 Двойственность Синтаксиса

Одна из наиболее важных концепций Лиспа называется *двойственностью синтаксиса*. Понимание того, как использовать эту двойственность и почему эта двойственность так важна — общая тема создания макросов и этой книги. Эта двойственность отчасти была спроектирована, а отчасти случайно открыта. Для программистов на не-Лисп языках ре-

альность двойственного синтаксиса будет очень нереальной, поэтому на данный момент мы уклонимся от прямого определения этого явления. Для того, чтобы избежать шока вы, читатель, будете изучать её шаг за шагом, постепенно и неторопливо. Если вы при чтении этой книги будете испытывать какой-либо дискомфорт и/или головную боль, то я рекомендую вам немедленно выполнить цикл сборки мусора (немного поспать), и только потом вернуться к чтению с ясным и незамутнённым умом.

Ссылочная прозрачность (Referential transparency) иногда определяется как следующее свойство кода: любое выражение может быть вставлено куда угодно и везде это выражение будет иметь один и тот же смысл. Вводя синтаксическую двойственность мы искусственно нарушаем ссылочную прозрачность и получаем плоды новых возможностей, от языка, в котором возможны такие нарушения. В то время когда другие языки позволяют вам создавать полупрозрачные стеклянные панели, Лисп позволяет использовать набор из дыма, зеркал и призм. Магическая пыль создаёт макросы и самые лучшие трюки макросов основаны на синтаксической двойственности.

Этот раздел описывает важность двойного синтаксиса, о котором мы только что говорили, но ещё недостаточно изучили: Common Lisp использует тот же синтаксис для получения доступа к главным типам переменных, динамическим и лексическим. Эта книга пытается проиллюстрировать реальную мощь динамической и лексической области видимости и важность решения использовать двойственный синтаксис в Common Lisp.

Цель динамической области видимости — это предоставление способа получения значений внутри и из Лисп выражений, основанном на вычислении выражения, а не местом где это выражение было определено или скомпилировано. Это возможно благодаря синтаксису, с помощью которого Common Lisp определяет идентичность, используемую для получения доступа к лексическим переменным, являющимися прямой противоположностью динамическим переменным в том, что они всегда относятся к тем местам, в которых они были скомпилированы, будучи независимыми от места использования. На самом деле без внешнего контекста в форме определения вы не можете ничего сказать о типе переменной, на которую ссылается выражение. Двойственный синтаксис нарушает ссылочную прозрачность, но вместо того, чтобы пытаться избежать эту, казалось бы, проблему, Лисп программисты приветствуют её, поскольку также как вы не можете различать выражение без контекста, также вы не можете писать макросы без двойственности. Поразмыслите секунду над этой мыслью. Это объясняет почему создание привязок для динамических переменных не создаёт лексических замыканий. В качестве

примера привяжем снова переменную **temp-special**, которую ранее мы объявили специальной:

```
CL-USER> (let ((temp-special 'whatever))
   (lambda () temp-special))
#<FUNCTION (LAMBDA ()) {C9E452D}>
```

И хотя мы видим *lambda'у окружённую let'ом*, это не лексическое замыкание. Это простое вычисление макро формы **lambda** в некотором динамическом контексте, результатом чего является, конечно же, анонимная функция. Эта функция, при её применении, будет работать в текущей динамической среде и будет получать значение **temp-special**. При вычислении макроса **lambda** существовала динамическая привязка **temp-special** к символу **whatever**, но кому это будет интересно? Помните что **lambda** формы — это объекты-константы, всего лишь простой машинный код, возвращающий указатели, поэтому вычисление этой лямбда формы никогда не получит доступа к динамической среде. Что случится с нашим символом **whatever**? После вычисления лямбда формы Лисп посчитает символ **whatever** не нужным и удалит его из динамической среды.

Некоторые ранние Лиспы поддерживали *динамические замыкания*, это обозначало что каждая функция определялась в не-null'евой динамической среде, обладавшей своим (возможно частично общим) стеком динамических привязок. Это был эффект, похожий на Common Lisp'овскую лексическую область видимости и реализовался с помощью так называемого *спагетти стека* [SPAGHETTI-STACKS] [INTERLISP-TOPS20]. Эта структура уже не представляла из себя стековую структуру данных, но была скорее множеством путей, этакой собирающей мусор сетью. Common Lisp не использует спагетти стеки и предоставляет только лексические замыкания [MACARONI].

Таким образом, лексические и динамические переменные — это абсолютно различные концепции, с одним синтаксисом в коде Common Lisp. Зачем нам нужна эта, так называемая, двойственность синтаксиса? Ответ на этот вопрос может понять меньшая часть Лисп программистов, но этот ответ настолько фундаментален, что заслуживает пристального изучения. Эта двойственность синтаксиса позволяет нам писать макросы, имеющие общий, единый интерфейс для создания расширений, полезных и в динамическом и в лексическом контексте. И хотя значение расширений макроса может абсолютно различаться в зависимости от заданного контекста и каждое расширение может иметь в своём основании абсолютно разные вещи, мы по прежнему можем использовать

тот же самый макрос и те же самые комбинации этого макроса с другими макросами. Другими словами макросы могут быть *двойственными* не только по отношению к аргументам макросов, но также обладать различными значениями по отношению к их расширениям. Мы можем использовать макрос только ради понимания трансформации кода, игнорируя семантическое обозначение кода, всё это потому, что код имеет значение только тогда, когда мы его где то используем — и не имеет значения при работе макроса. Чем больше двойственность синтаксиса, тем более мощным становится макрос. На протяжении этой книги вы увидите много примеров, в деталях показывающих преимущества двойственного синтаксиса. Двойственность между динамическими и лексическими переменными является не очень выразительным (но полезным) примером Лисп философии. Некоторые макросы были созданы на основе мощи двойственности и предназначены для специфических целей, иногда в расширениях таких макросов могут быть заложены более чем два различных обозначения.

Согласно традиционному соглашению в Common Lisp имена специальных переменных следует предварять и завершать символами астериска. Например, для нашей переменной **temp-special** мы должны выбрать имя ***temp-special***. Поскольку это соглашение выглядит как обладание другим пространством имени для динамических переменных тем самым снижая двойственность с лексическими переменными, то в этой книге не выдерживается строгое следование данному соглашению. Астериски — это всего лишь соглашение и, к счастью, Common Lisp не строго следует ему. Мы можем не только не использовать астериски в именах специальных переменных, но даже использовать астериски в именах лексических переменных. Возможно это вопрос стиля. Что будет менее преступным с точки зрения моды: лексические переменные с астерисками или специальные переменные без астерисков? Я склоняюсь к более короткому варианту. Кроме того, имена лексических и специальных переменных могут быть gensym'ами, концепцией, выводящей печать имён за пределы символов.

Как уже упоминалось эта книга нарушает традиционное соглашение об астерисках. Вместо

> Имя переменной с астериском обозначает специальную переменную.

эта книга использует

> Имена переменных с астерисками обозначают специальные переменные, определённые по стандарту.

Моя главная мотивация в отбрасывании наушников с имён переменных проста и субъективна: я считаю, что постоянная печать астерисков раздражает программиста и ухудшает внешний вид программы. Я не буду заходить далеко и предлагать вам следовать этому в своих программах, я просто упомяну, что в своих программах я уже несколько лет не использую наушники и очень доволен этим.

Глава 4

Считывающие Макросы

4.1 Время-Работы как Время-Чтения

> Синтаксический сахар вызывает рак точек с запятой
>
> — Алан Перлис

Не один только Лисп даёт прямой доступ к коду, который может разбираться на структуры из cons ячеек, но, кроме этого Лисп предоставляет доступ к символам, составляющим ваши программы, ещё до того, как программы будут представлять из себя структуру. Обычный макрос работает с программой в форме дерева. Кроме обычного макроса существует специальный тип макросов, называемых *считывающими макросами (read macro)*, оперирующими сырыми символами, из которых состоит ваша программа.

Если нам понадобиться определить не-Лисповский синтаксис в Лиспе, то в этом случае не имеет смысла использовать Лисповский считыватель — он предназначен только для чтения Лиспа. Считывающий макрос — это устройство, используемое для обработки не-Лисп синтаксиса, после считывающего макроса в дело вступает Лисп считыватель. Причина, по которой Лисп считыватель является более мощным чем считыватели в других языках заключается в том, что Лисп даёт нам возможность *перехватывать (hooks)* каждый аспект поведения считывателя. В частности, Лисп позволяет вам *расширять* считыватель, таким образом, что не-Лисп объекты будут прочитываться и преобразовываться в Лисп объекты. Также, как вы строите ваши приложения поверх Лиспа, расширяя его с помощью макросов и функций, также и Лисп приложения могут, а чаще всего и просачиваются в это измерение расширяемости. Когда это происходит, то становится возможным чтение любого символа на ос-

нове синтаксиса с помощью Лисп считывателя, а это означает что вы добавили этот синтаксис в Лисп.

В то время, когда трансформация кода, выполняемая обычными макросами используется для вставки Лисп кода в новый Лисп код, считывающие макросы могут быть написаны для вставки не-Лисп кода в Лисп код. Подобно обычным макросам, считывающие макросы реализуются с помощью функций, благодаря этому мы имеем доступ ко всей мощи Лисп окружения. Подобно макросам, увеличивающим продуктивность через создание более удобного предметно ориентированного языка, считывающие макросы увеличивают продуктивность позволяя сокращать выражения в точку где они больше не являются выражениями.

Если всё что нам нужно для того, чтобы разобрать эти не-Лисповские предметно-ориентированные языки — это написать короткий считывающий макрос, то может быть эти не-Лисп языки на самом деле являются умно замаскированным Лиспом. Если XML может прямо считываться Лисп считывателем [XML-AS-READ-MACRO], то может быть XML — это просто видоизменённый Лисп. Подобным образом считывающие макросы могут использоваться для чтения регулярных выражений и SQL запросов прямо в Лисп, то может быть эти языки на самом деле являются Лиспом. Это нечёткое различие между кодом и данными, Лиспом и не-Лиспом, являются источником многих интересных философских проблем, возникающих перед Лисп программистами с самого начала зарождения Лиспа.

Базовый считывающий макрос, встроенный в Common Lisp — это *#.* — вычисляющий макрос в момент чтения. Этот считывающий макрос позволяет вам встраивать объекты в читаемые, не сериализируемые формы, но создаваемые с помощью Лисп кода. Один забавный пример — это создание формы, содержащей изменяющиеся значения при каждом вызове.

```
CL-USER> '(football-game
           (game-started-at
            #.(get-internal-real-time))
           (coin-flip
            #.(if (zerop (random 2)) 'heads 'tails)))
(FOOTBALL-GAME (GAME-STARTED-AT 177030) (COIN-FLIP TAILS))
```

Несмотря на то, что это то-же самое выражение, эта форма считывается по разному раз за разом:

```
CL-USER> '(football-game
```

```
           (game-started-at
            #.(get-internal-real-time))
           (coin-flip
            #.(if (zerop (random 2)) 'heads 'tails)))
(FOOTBALL-GAME (GAME-STARTED-AT 385713) (COIN-FLIP HEADS))
```

Заметьте, что две формы окружённые *#.* вычисляются в момент чтения, а не тогда, когда вычисляется форма. Полный список был сформирован после того, как они были вычислены, а предыдущая и последующая эквивалентность (определённая **equal**'ом) может быть обнаружена через повторное вычисление последней прочитанной формы и сравнения её с предыдущими результатами, с помощью вспомогательных переменных ***, *+* в REPL[1]:

```
CL-USER> (equal * (eval +))
T
```

Следует обратить внимание на то, что эти формы вычисляются в момент чтения, этим они отличаются от использования обратной кавычки, более подробно мы рассмотрим обратную кавычку в следующем разделе. Мы можем вычислить подобную форму с помощью использования обратных кавычек:

```
CL-USER> `(football-game
           (game-started-at
            ,(get-internal-real-time))
           (coin-flip
            ,(if (zerop (random 2)) 'heads 'tails)))
(FOOTBALL-GAME (GAME-STARTED-AT 722283) (COIN-FLIP TAILS))
```

но в этом случае будет происходить вычисление в различные результаты, поскольку обратная кавычка производит считывание в виде кода для вычисления:

```
CL-USER> (equal * (eval +))
NIL ;если вы не очень быстрый и удачливый
```

[1]Переменная *** содержит значение, являющееся результатом предыдущей формы, а переменная *+* содержит значение последней формы.

4.2 Обратная Кавычка

Обратная кавычка, иногда называемая как *квазикавычка (quasiquote)*[2] и отображаемая как ` — это относительно новый элемент в промышленных диалектах Лиспа. Эта концепция по-прежнему абсолютно чужда для не-Лисп языков программирования.

У обратной кавычки странная история развития. Обратная кавычка развивалась параллельно с Лиспом. Есть сведения [QUASIQUOTATION] о том, что раньше никто не верил в то, что вложенные обратные кавычки будут работать правильно, пока умные программисты не выяснили что обратные кавычки правильно выполняют свою работу — идеи людей о том, что было верным оказались неверными. Известно, что трудно понять работу вложенной обратной кавычки. Даже Стил (Steele), отец Common Lisp'а, жалуется на это [CLTL2—P530].

В принципе Лисп не нуждается в обратной кавычке. Всё, что можно выполнить с помощью обратной кавычки — можно выполнять с помощью других функций, предназначенных для постройки списков. Однако, обратная кавычка настолько полезна при программировании макросов, а в Лиспе под программированием понимается программирование макросов, что Лисп профессионалы очень активно ею пользуются.

Вначале мы должны разобраться с обычным закавычиванием. В Лиспе, мы используем префикс формы в виде символа кавычки (') для того, чтобы информировать Лисп о том, что следующая форма должна рассматриваться как сырые данные, а не как код, который нужно вычислить. Или, если выразиться иначе, результатом вычисления кавычки в коде будет возвращение формы. Иногда мы говорим что кавычка *останавливает* или *отключает* вычисление формы.

В Лиспе обратная кавычка может использоваться как замена кавычки. Исключая некоторые специальные символы, называемые символами *раскавычивания (unquote)* и могущие появиться в форме, обратная кавычка останавливает вычисление тем-же способом, что и кавычка. Как следует из названия символы раскавычивания изменяют семантику вычисления. Иногда мы говорим что раскавычивание *перезапускает* или *включает* вычисление формы.

Есть три главных типа раскавычивания: обычное раскавычивание, объединяющее раскавычивание и деструктивное, объединяющее раскавычивание.

Для выполнения обычного раскавычивания, мы используем оператор

[2]Scheme программисты называют её квазикавычкой а Common Lisp программисты — обратной кавычкой.

запятую:

```
CL-USER> (let ((s 'hello))
           `(,s world))
(HELLO WORLD)
```

Хотя выражение, которое мы раскавычили является простым вычислением символа, **s**, на месте этого символа может быть любое Лисп выражение, вычисляемое в нечто значимое для любого контекста, в котором оно появляется в шаблоне обратной кавычки. Каким бы ни был результат, он будет вставлен в результирующий список в car позиции того места в котором он появился в шаблоне обратной кавычки.

В нотации Лисповской формы мы можем использовать **.** если мы хотим явно вставить что-либо cdr'ное в создаваемую списковую структуру. Если здесь мы вставим список, то результирующей формой обратной кавычки будет корректный список. Но, если здесь мы вставим что-нибудь другое, то мы получим новую не-списковую структуру.

Такое-же поведение доступно нам везде, в том числе внутри обратной кавычки[3]. Благодаря архитектуре обратной кавычки мы можем раскавычивать элементы даже в такой позиции:

```
CL-USER> (let ((s '(b c d)))
           `(a .,s))
(A B C D)
```

Вставка списков в *cdr* позиции создаваемого списка из шаблона обратной кавычки оказалось весьма универсальной операцией, поэтому был сделан следующий шаг названный как объединяющее раскавычивание. Вышеприведённая комбинация **.,** полезна, но не способна вставлять элементы в середину списка. Для этого у нас есть оператор объединяющее раскавычивание:

```
CL-USER> (let ((s '(b c d)))
           `(a ,@s e))
(A B C D E)
```

Ни **.,** ни **,@** не модифицируют сращиваемый список. Для примера, после вычисления обратной кавычки в обеих предыдущих формах, **s** по прежнему будет привязан к трёх элементному списку **(B C D)**. Хотя

[3]Поскольку обратная кавычка использует (почти ту же) стандартную функцию **read** что и везде.

это не является строго определяемым стандартом, в форме **(A B C D)**, **(B C D)** может являться общей структурой с объединённым списком **s**. Однако, структура списка **(A B C D E)** гарантирует, что она будет свежесозданной при вычислении обратной кавычки, поскольку **,@** запрещено модифицировать сращенные списки. Объединяющее раскавычивание — не деструктивная операция, поскольку в целом нам нужно думать об обратной кавычке как о компонуемом шаблоне для создания списков. Деструктивная модификация списковой структуры не свежесозданных данных при каждом вычислении обратно закавыченного кода может иметь нежелательные эффекты в плане будущих расширений.

Однако, в Common Lisp есть деструктивная версия сращиваемого раскавычивания, которое вы можете использовать везде, где можно использовать сращивающее раскавычивание. Для деструктивного раскавычивания используйте **,.** . Деструктивное сращивание работает также, как и обычное сращивание, за исключением того, что сращиваемый список может быть модифицирован в процессе вычисления шаблона обратной кавычки. И хотя отличие от обычного раскавычивания выражается всего лишь в одном символе, эта нотация более умно использует символ **.** из *cdr* позиционного раскавычивания **.,** рассмотренного выше.

Для того, чтобы увидеть это в действии, в этом примере мы деструктивно модифицируем список указанный в *to-splice*:

```
CL-USER> (defvar to-splice '(B C D))
TO-SPLICE
CL-USER> `(A ,.to-splice E)
(A B C D E)
CL-USER> to-splice
(B C D E)
```

Выполнение деструктивной модификации сращиваемых списков может быть опасной операцией. Рассмотрим следующее применение деструктивного сращивания:

```
(defun dangerous-use-of-bq ()
  `(a ,. '(b c d) e))
```

При первом вызове **dangerous-use-of-bq** возвращается ожидаемый ответ: **(A B C D E)**. Но, поскольку эта функция использует деструктивное сращивание и модификацию не свежесгенерированных списков — закавыченный список — то мы можем ожидать возникновения различных нежелательных последствий. В этом случае при втором вычислении

dangerous-use-of-bq форма **(B C D)** будет представлять из себя форму **(B C D E)** и в момент, когда обратная кавычка попытается деструктивно срастить этот список с остатком шаблона обратной кавычки **(E)** — его собственный хвост — будет создан список, содержащий *цикл (cycle)*. Более детально мы обсудим циклы в *разделе Циклические Выражения*.

Однако, есть много случаев в которых деструктивное сращивание является чрезвычайно безопасным. Не позволяйте **dangerous-use-of-bq** напугать вас если вам нужна повышенная эффективность в ваших формах обратной кавычки. Есть множество операций, создающих свежие списковые структуры, которые вам так или иначе могут не понадобиться. Например, сращивание результатов *mapcar* настолько распространено и безопасно, что вполне может претендовать на программную идиому:

```
(defun safer-use-of-bq ()
  `(a
    ,.(mapcar #'identity '(b c d))
    e))
```

Но, есть деталь, которая мешает этому. Чаще всего обратная кавычка используется для создания макросов, часть программирования на Лиспе, где менее важна скорость и более важна ясность. Если думать о *побочных эффектах* в ваших операциях сращивания, то при создании и интерпретации макросов вам придётся часто отвлекаться на них, а это не стоит свеч. Эта книга придерживается обычных сращиваний. Чаще всего обратная кавычка используется для конструирования макросов, но, это не единственное их использование. Обратная кавычка — это удобный предметно ориентированный язык для смешивания списков, и ещё более удобным он становится с возможностью деструктивного сращивания.

Как работает обратная кавычка? Обратная кавычка — это считывающий макрос. Формы обратной кавычки читаются как код, который при вычислении, становится желаемым списком. Возвращаясь к примеру из предыдущего раздела о вычислении во время выполнения, мы можем отключить *красивую печать (pretty printing)*, закавычить значение формы обратной кавычки и вывести его на печать для того, чтобы увидеть как читаются формы с обратной кавычкой[4]:

```
CL-USER> (let (*print-pretty*);привязка к nil
           (print
            '`(football-game
```

[4]Мы возвращаем **t** и поэтому мы не видим значения, возвращаемого от **print**. **(values)** также универсальны.

```
             (game-started-at
              ,(get-internal-real-time))
             (coin-flip
              ,(if (zerop (random 2))
                   'heads
                   'tails))))
           t)

(SB-IMPL::BACKQ-LIST
 (QUOTE FOOTBALL-GAME)
 (SB-IMPL::BACKQ-LIST
  (QUOTE GAME-STARTED-AT)
  (GET-INTERNAL-REAL-TIME))
 (SB-IMPL::BACKQ-LIST
  (QUOTE COIN-FLIP)
  (IF (ZEROP (RANDOM 2))
      (QUOTE HEADS)
      (QUOTE TAILS))))
T
```

В этой, *некрасиво напечатанной* форме, функция **LISP::BACKQ-LIST** идентична **list**, за исключением поведения, связанного с красивой печатью. Заметьте, что операторы запятая исчезли. Common Lisp очень либерален и поэтому позволяет читать обратные кавычки, также, как и допускает операции с использованием общих структур.

Кроме того обратная кавычка предоставляет много интересных решений забавной *не совсем проблемы* написания Лисп выражений, вычисляющихся в самих себя. Эти выражения часто называются *куайнами* в честь Уилларда Куайна, который занимался их широким изучением и кто, по сути, ввёл термин квазицитирования — альтернативное название обратной кавычки [FOUNDATIONS-P31-FOOTNOTE3]. Вот забавный пример куайна, от Майка МакМахонома (Mike McMahon) [QUASIQUOTATION]:

```
CL-USER> (let ((let '`(let ((let ',let))
                         ,let)))
           `(let ((let ',let)) ,let))
(LET ((LET
       '`(LET ((LET ',LET))
           ,LET)))
  `(LET ((LET ',LET))
     ,LET))
```

Для сохранения вашего *ментального прохода по коду*:

```
CL-USER> (equal * +)
T
```

Упражнение: Почему в выражении ниже обратная кавычка расширяется в обычную кавычку? Разве оно не закавычено?

```
CL-USER> `'q
'Q
```

4.3 Чтение Строк

В Лиспе строки разграничиваются символами двойной кавычки ("). Хотя строки могут содержать любые символы из символьного набора в вашей Лисп реализации, но вы не можете непосредственно вставлять определённые символы в строку. Если вы хотите вставить символ " или символ \, то вам придётся использовать префикс в виде символа \. Это называется *экранированием* символов. Ниже показан пример с введением строки, содержащей символы " и \:

```
 * "Contains \" and \\."
"Contains \" and \\."
```

Принцип работы очевиден, но иногда печать символов \ становится утомительным и порождает ошибки. Это Лисп, и если нам что-то не нравится, то мы в силах внести наши изменения, это не просто легко сделать, но, ещё и приветствуется. Придерживаясь этой мысли, эта книга представляет считывающий макрос под названием *#"* или *sharp-double-quote*. Этот считывающий макрос предназначен для создания строк, содержащих символы " и \ без необходимости экранирования.

Sharp-double-quote[5] начинает чтение строки непосредственно после вызова следующих символов: *#* и *"*. Чтение будет продолжаться символ за символом до тех пор, пока в последовательности не встретятся два символа *"* и *#*. Когда будет обнаружена завершающая последовательность, то будет возвращена строка, содержащая все символы между *#"* и *"#*. Считывающий макрос *sharp-double-quote* создан для работы с битовыми строками, но Common Lisp позволяет нам использовать этот

[5]Наше соглашение об именовании нижележащих функций считывающих макросов с символами основывается на символах считывающего макроса, похожих на Стиловский считыватель #**reader** в CLtL2.

Листинг 4.1: SHARP-DOUBLE-QUOTE

```
(defun |#"-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let (chars)
    (do ((prev (read-char stream) curr)
         (curr (read-char stream) (read-char stream)))
        ((and (char= prev #\") (char= curr #\#)))
      (push prev chars))
    (coerce (nreverse chars) 'string)))

(set-dispatch-macro-character
 #\# #\" #'|#"-reader|)
```

макрос передавая битовую строку в считывающий макрос *#** [EARLY-CL-VOTES].

Вот пример использования нашего нового *sharp-double-quote*:

```
 * #"Contains " and \."#

"Contains \" and \\."
```

Учтите, что при печати строки REPL по прежнему будет использовать символ " в качестве разграничителя, поэтому символы " и \ по прежнему будут экранироваться в печатаемой строке. Эти строки по прежнему будут прочитываться так, как будто символы в них были экранированы вручную.

Но, иногда *#"* оказывается недостаточно хорошим. Например: в этом, только что прочитанном вами, параграфе U-Языка я вставил следующую последовательность символов: *"#*. По этой причине этот параграф не будет ограничиваться *#"* и *"#*. А поскольку я ненавижу экранируемые элементы, то поверьте мне на слово, здесь не будет разграничения обычными двойными кавычками.

Нам нужен макрос, который бы мог дать нам возможность модифицировать разграничитель для каждого используемого контекста. Как это часто бывает нам не нужно идти далеко за примером, достаточно взглянуть на язык Ларри Уолла — Perl чтобы почерпнуть вдохновения для архитектуры программных сокращений. Perl — это прекрасный, чудесно спроектированный язык, содержащий большое количество идей, которые могут быть *украдены* Лиспом. В некотором роде, Лисп — это снежный

ком, который катится по идеям из других языков программирования и вбирает их в себя, делая эти идеи своей частью[6].

Считывающий макрос #> непосредственно вдохновлён оператором Perl'a «.

Этот оператор позволяет Perl программистам определять строку текста, которая будет служить разграничителем для цитируемой строки. #> читает символы до тех пор, пока не найдёт символ перехода на новую строку, затем читает символы один-за-другим, до тех пор, пока не встретиться последовательность символов идентичная символам обнаруженным сразу после #> и до новой строки.

Например:

```
 * #>END
I can put anything here: ", \, "#, and ># are
no problem. The only thing that will terminate
the reading of this string is...END

"I can put anything here: \", \\, \"#, and ># are
no problem. The only thing that will terminate
the reading of this string is..."
```

4.4 CL-PPCRE

CL-PPCRE [CL-PPCRE] — это высокопроизводительная библиотека регулярных выражений, написанная на Common Lisp. Она была создана широко известным и уважаемым Лисп хакером Эди Вейтзом (Edi Weitz). Этот раздел посвящается Эди Вейтзу. В то время, когда другие люди болтают Эди пишет программы; код говорит громче чем все аргументы.

PPCRE, для тех, кто ещё не знаком с этой библиотекой, — это Portable Perl Compatible Regular Expressions (Переносимые Perl Совместимые Регулярные Выражения). CL-PPCRE, также как и код в этой книге, является *переносимым*, поскольку может запускаться в любой ANSI — совместимой Common Lisp среде. CL-PPCRE, также как код в этой книге является открытым и доступным бесплатно. Хотя CL-PPCRE в основном прекрасно совместим с Perl'ом, всё же существуют некоторые важные различия от Perl. CL-PPCRE содержит несколько лисповых улучшений

[6]Наиболее цитируемым примером являются объекты, но кроме них существуют ещё бесчисленное множество других примеров, таких как функция **format** из FORTRAN'a.

Листинг 4.2: SHARP-GREATER-THAN

```
(defun |#>-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let (chars)
    (do ((curr (read-char stream)
               (read-char stream)))
        ((char= #\newline curr))
      (push curr chars))
    (let* ((pattern (nreverse chars))
           (pointer pattern)
           (output))
      (do ((curr (read-char stream)
                 (read-char stream)))
          ((null pointer))
        (push curr output)
        (setf pointer
              (if (char= (car pointer) curr)
  (cdr pointer)
                 pattern))
        (if (null pointer)
    (return)))
      (coerce
       (nreverse
(nthcdr (length pattern) output))
       'string))))

(set-dispatch-macro-character
 #\# #\> #'|#>-reader|)
```

в регулярных выражениях. Есть три различия, которыми CL-PPCRE отличается от реализации регулярных выражений в Perl.

Первое, CL-PPCRE быстр. Действительно быстр. Замеры производительности CL-PPCRE, скомпилированного с помощью хорошего, родного компилятора кода, показали что для большинства регулярных выражений CL-PPCRE оказывается в два раза быстрее Perl'а, а довольно часто ещё быстрее. И это при том, что Perl обладает одним из наиболее быстрейших не-лисповых движков регулярных выражений: высоко оптимизированный движок, написанный на C. Почему это возможно? Конечно, Perl'овская низко-уровневая реализация должна обладать производительностью, превосходящую всё, что написано на таком высокоуровневом языке, как Лисп.

Есть заблуждение, известное как *миф производительности*, основная версия следующая: использование низкоуровневых языков приводит к возникновению быстрого кода, поскольку вы можете программировать наиболее близко к железу. С помощью этой книги я хочу показать вам что для сложных систем этот миф является ложным. Примеры, подобные CL-PPCRE демонстрируют это. Чем более низкоуровневым является язык, тем больше он мешает вам и вашему компилятору эффективно оптимизировать код.

Техническая причина быстрой производительности CL-PPCRE следующая: Common Lisp, язык на котором реализовали CL-PPCRE, более мощный чем C, язык, использованный для реализации Perl. Когда Perl считывает регулярное выражение, он может выполнять анализ и оптимизацию, но в конечном счёте регулярное выражение будет сохранено в некоторую структуру данных C, после чего движок статичных регулярных выражений будет производить сравнивание. Но в Common Lisp — наиболее мощном языке — нет ничего сложного в том, чтобы получить это регулярное выражение, конвертировать его в Лисп программу и передать эту Лисп программу на оптимизацию в родной компилятор Лисп кода, используемый в том числе для построения всей остальной Лисп системы[7]. Поскольку программы, скомпилированные с помощью C компилятора не имеют доступ к C компилятору, Perl не может компилировать регулярные выражения в машинный код. Модель компиляции Лиспа отличается от модели C. В Common Lisp компиляция кода во время выполнения (как и в любое другое время) является переносимой, бесшовной, выполняющейся в том же процессе, что и ваша Лисп машина,

[7]На деле CL-PPCRE более запутан чем описано здесь. Эта библиотека содержит свою собственную функцию компилирования и, как правило, (если вы не строите регулярные выражения во время выполнения) гарантирует что она будет вызвана когда ваша программа будет скомпилирована.

отпадает необходимость в сборке мусора и, по причине инкрементальной природы, высокоэффективна.

Второе главное отличие между CL-PPCRE и Perl заключается в том, что CL-PPCRE не привязан к нотации регулярных выражений, основанной на строке. CL-PPCRE свободен от символьного представления и позволяет нам кодировать регулярные выражения в Лисп формах (иногда называемых *S-выражениями*). Поскольку такие же формы мы используем для написания Лисп программ и макросов, то мы можем использовать больше возможностей для *сплочения* в наших абстракций. За деталями использования такой нотации обращайтесь к документации и коду CL-PPCRE [CL-PPCRE], кроме того, код и документация CL-PPCRE может служить примером хорошо спроектированного, Лиспобразного предметно-ориентированного языка.

Конечно, CL-PPCRE — это замечательная библиотека, но почему мы обсуждаем эту библиотеку в главе, посвящённой считывающим макросам? Ответ заключается в третьей и последней разнице между CL-PPCRE и Perl. В Perl регулярные выражения тесно связаны с языком. Также, как синтаксис Лиспа — это способ работы с мета-программированием, синтаксис Perl'а — это способ работы с регулярными выражениями и другими видами синтаксических сокращений. Одна из причин по которой мы так часто используем регулярные выражения в Perl коде обусловлена тем, что использование регулярных выражений позволяет кратко и быстро писать программы.

Считывающие макросы очень удобны для реализации программного интерфейса в Perl стиле. Поскольку программирование считывающих макросов — это программирование Лиспа, то мы начнём с функции-утилиты: **segment-reader**. Получая поток, разделительный символ и счётчик, **segment-reader** будет читать символы из потока до тех пор, пока не встретится символ-разделитель. Если счётчик больше 1, **segment-reader** будет возвращать *cons*. *Car* этого *cons*-а будет представлять строку и *cdr* будет результатом рекурсивного вызова **segment-reader** с декрементированным параметром *count* для получения следующего сегмента[8].

Например, чтение 3 сегментов из потока t[9] с ограничительным символом / будет осуществляться так:

[8] В Common Lisp принято следующее правило: если вариант *else* отсутствует в форме *if* , а проверяемое выражение вернуло *false*, то возвращаемое значение будет *nil*. Опытные Common Lisp программисты часто полагаются именно на такое поведение, поэтому этот поведенческий шаблон, как базовый вариант рекурсивного построения списка, был применён при написании **segment-reader**.

[9] Поток **t** соответствует стандартному вводу при работе в REPL'е.

Листинг 4.3: SEGMENT-READER

```
(defun segment-reader (stream ch n)
  (if (> n 0)
      (let ((chars))
        (do ((curr (read-char stream)
                   (read-char stream)))
            ((char= ch curr))
          (push curr chars))
        (cons (coerce (nreverse chars) 'string)
              (segment-reader stream ch (- n 1))))))
```

```
CL-USER> (segment-reader t #\/ 3)
abc/def/ghi/
("abc" "def" "ghi")
```

Скорее всего, Perl программисты уже поняли что здесь происходит. Идея заключается в том, чтобы *свороватъ*, приношу искренние извинения Ларру Уоллу, синтаксис двух удобных операторов регулярных выражений Perl. В Perl, если нам нужно применить регулярное выражение к переменной, мы можем написать

```
$my_boolean = ($var =~ m/^\w+/);
```

для того, чтобы увидеть начинается ли содержимое **$var** с одной или более цифр или букв. Подобным образом, если нам нужно применить *подстановку* регулярного выражения мы можем использовать Perl оператор =~ изменяющий первое вхождение *dog* на *cat* в нашей строчной переменной **$var**:

```
$var =~ s/dog/cat/;
```

Замечательной деталью Perl синтаксиса является возможность использования любого символа в качестве разделителя для удобства программирования. Если мы хотим использовать регулярные выражение или подстановку, содержащую символ / , то мы должны использовать какой-либо отличающийся символ для избежания конфликтов[10]:

```
$var =~ s|/usr/bin/rsh|/usr/bin/ssh|;
```

[10]Это может быть не связано с Perl'ом; TEX'овские дословные закавычивания предоставляют нечто подобное.

Листинг 4.4: MATCH-MODES

```
#+cl-ppcre
(defmacro! match-mode-ppcre-lambda-form (o!args)
  ``(lambda (,',g!str)
      (cl-ppcre:scan
       ,(car ,g!args)
       ,',g!str)))

#+cl-ppcre
(defmacro! subst-mode-ppcre-lambda-form (o!args)
  ``(lambda (,',g!str)
      (cl-ppcre:regex-replace-all
       ,(car ,g!args)
       ,',g!str
       ,(cadr ,g!args))))
```

Определение считывающего макроса для копирования этих двух Perl синтаксисов даёт нам шанс для демонстрации интересных техник макросов — двойная обратная кавычка. Идея та же, что и в макросах **match-mode-ppcre-lambda-form** и **subst-mode-ppcre-lambda-form** — желание писать код, генерирующий списки. Учтите, что когда вы определяете макрос без прикрас и используете единственную обратную кавычку, то в этом случае вы генерируете список, представляющий код и возвращаете его из макроса для последующего объединения в выражения и вычисления. С двойными обратными кавычками вы по прежнему генерируете список, представляющий код, но, этот код при вычислении будет использовать код, созданный обратной кавычкой для возвращения списка. В нашем случае, эти два макроса расширяются в код, который вы можете вычислить для создания лямбда форм, пригодных для использования регулярных выражений CL-PPCRE.

Перед этими макросами, как и с некоторыми другими выражениями ниже, мы используем считывающий макрос `#+`. Этот считывающий макрос выясняет где нам требуется CL-PPCRE[11] перед вычислением следующей формы. Если CL-PPCRE не будет доступен при загрузке исходного кода из этой книги, то этот код не будет работать.

И наконец мы можем определить считывающую функцию для объединения этих утилит, а затем добавить эту функцию в нашу таблицу

[11]Этот макрос определяет необходимость CL-PPCRE с помощью поиска ключевого символа **:CL-PPCRE** в списке, сохранённом в переменной ***features***.

Листинг 4.5: CL-PPCRE-READER

```
#+cl-ppcre
(defun |#~-reader| (stream sub-char numarg)
  (declare (ignore sub-char numarg))
  (let ((mode-char (read-char stream)))
    (cond
      ((char= mode-char #\m)
       (match-mode-ppcre-lambda-form
        (segment-reader stream
                        (read-char stream)
                        1)))
      ((char= mode-char #\s)
       (subst-mode-ppcre-lambda-form
        (segment-reader stream
                        (read-char stream)
                        2)))
      (t (error "Unknown #~~ mode character")))))

#+cl-ppcre
(set-dispatch-macro-character #\# #\~ #'|#~-reader|)
```

диспетчеризации макроса. Мы решили использовать считывающий макрос `#~` поскольку это хороший аналог Perl'овского `#~` — источника, вдохновившего нас на такой синтаксис.

Считывающий макрос `#~` спроектирован с упором на удобство. Вот как мы можем создать функцию, производящую сравнение с регулярным выражением:

```
CL-USER> #~m/abc/
#<FUNCTION (LAMBDA (#:STR178)) {CAB9B35}>
```

Теперь применение этой функции к строке не будет отличаться от обычного вызова функции[12]:

```
CL-USER> (funcall * "123abc")
3
```

[12]Переменная * привязана к значению, возвращённому последним вычислением в REPL. В данный момент эта переменная привязана к нашей функции регулярного выражения.

```
6
#()
#()
```

Значения, возвращаемые от функции **cl-ppcre:scan**, задокументированы в [CL-PPCRE]. Если же вы интересуетесь только тем, совпадали строка с регулярным выражением, то вам понадобится только первое значение, истинное значение обозначает что обнаружено совпадение. Обобщённые логические значения и их важность в Common Lisp обсуждается позже в *главе Анафорические Макросы*.

Также мы можем создавать функции, производящие замену на базе регулярных выражений. Маленькая разница между Perl и нашим считывающим макросом в том, что функции замены на базе регулярных выражений не модифицируют свои аргументы. Они будут возвращать новые строки, являющиеся копиями исходных строк с выполненной заменой. Другим отличием является то, что, по умолчанию, этот считывающий макрос производит замену всех соответствий вместо замены первого соответствия в строке. Для того, чтобы получить такое поведение в Perl, вам понадобится добавить глобальный модификатор в ваше регулярное выражение, но в нашем случае глобального модификатора не нужно:

```
CL-USER> (funcall #~s/abc/def/ "Testing abc testing abc")
"Testing def testing def"
T
```

Итак, как всё это работает? Что делают выражения `#~`, которые не являются Лисп выражениями? При поверхностном взгляде кажется что они считываются как функции, но оказывается что это не так. Давайте закавычим одну из этих форм и посмотрим как эта форма выглядит для Лисп считывателя:

```
CL-USER> '#~m|\w+tp://|
(LAMBDA (#:STR178) (CL-PPCRE:SCAN "\\w+tp://" #:STR178))
```

Подстановки выглядят также:

```
CL-USER> '#~s/abc/def/
(LAMBDA (#:STR179) (CL-PPCRE:REGEX-REPLACE-ALL "abc"
                                               #:STR179 "def"))
```

Они считываются как лямбда формы. Поскольку мы работаем с Лисп считывателем, то мы не можем написать их в каком-либо не-Лисп языке. Это обозначение функции. По той причине, что наши выражения

являются простыми списками, в котором первым элементом выступает символ **lambda**, а в *разделе Let — это Lambda* обсуждается как мы можем использовать лямбда формы в первом аргументе вызова функции для порождения анонимных функций:

```
CL-USER> (if (#~m/^[\w-.]+$/ "hcsw.org")
     'kinda-looks-like-a-domain
     'no-chance!)
KINDA-LOOKS-LIKE-A-DOMAIN
```

Когда мы применяем **funcall** или **apply** для использования объектов считанных с помощью `#~`, то в этом случае мы используем ANSI макрос **lambda**, но только не в том случае, когда форма является первой в списке: польза *двойственности синтаксиса*. Если наши `#~` выражения считываются как шарп-закавыченные лямбда формы, то мы не сможем использовать их в качестве функций в выражениях — здесь могут использоваться только имена функций и лямбда формы. Для этих обеих задач нам понадобится только один считывающий макрос, по счастливой случайности этот макрос большой и сложный. Преимущества двойственного синтаксиса позволит нам сфокусироваться на получении корректного расширения вместо отслеживания требований различного синтаксиса. Вместо одного интересного макроса мы получаем два интересных макроса. Для экономии усилий, сделаем ваш синтаксис наиболее близким к оригиналу.

Общей проблемой использования CL-PPCRE является *экранирование* обратной косой черты в регулярных выражениях — программисты постоянно забывают о необходимости экранирования. Посмотрите, что получится при выполнении такого кода:

```
CL-USER> "\w+"
"w+"
```

Длина этой строки равна 2. Куда пропала обратная косая черта? Использование двойных кавычек обозначает что мы хотели экранировать символ *w* вместо печати символа \. Для нашего считывающего макроса `#~`, который просто считывает символы, это не проблема и мы можем писать регулярные выражения также, как это мы делаем в Perl — без экранирования. Смотрите закавычивание регулярных выражений URL выше.

И хотя считывающий макрос `#~` определённый в этом разделе уже очень удобен, всё ещё есть место для его улучшения и расширения.

Упражнение: Улучшите этот макрос. Наиболее очевидным первым шагом будет поддержка таких модификаторов регулярных выражений, как нечувствительность к регистру при сравнивании. Если продолжать реализовывать всё в том же синтаксисе что и в Perl, то работа будет связана с функцией **unread-char**, широко применяемой в считывающих макросах для избежания непреднамеренного *сведания* символа, которое может ожидать какой-либо другой считывающий макрос.

4.5 Циклические Выражения

Все наши разговоры о том, что Лисп программы являются *деревьями* из *cons* ячеек — маленькая ложь. Прошу прощения за это. Лисп программы на самом деле не деревья, но *направленные ациклические графы* — деревья, ветви в которых могут быть общими. Поскольку вычислителя не волнует откуда произошла ветвь, то нет ничего неправильного в вычислении кода с общими структурами.

У нас есть полезный считывающий макрос `#=`. Мы уже увидели как лисп может создавать формы с макросом `#=` при сериализации макро расширений в *разделе Нежелательный Захват*. `#=` и его партнёр `##` позволяют вам создавать S-выражения, ссылающиеся на самих себя. Это позволяет вам использовать такие приёмы, как общие ветви в направленном ациклическом графе и другие интересные структуры данных практически без усилий.

Но, что более важно — этот приём позволяет вам сериализовать данные без необходимости разбирать и пересобирать структуры данных в памяти, где общим может быть большой массив данных. Вот пример того, как два лисп списка считываются в различных объектах (не *eq*-абельные):

```
CL-USER> (defvar not-shared '((1) (1)))
NOT-SHARED
CL-USER> not-shared
((1) (1))
CL-USER> (eq (car not-shared) (cadr not-shared))
NIL
```

Но, в следующем примере с данными, сериализованными с помощью считывающего макроса `#=`, два списка на самом деле представляют один и тот-же список:

```
CL-USER> (defvar shared '(#1=(1) #1#))
```

```
SHARED
CL-USER> shared
((1) (1))
CL-USER> (eq (car shared) (cadr shared))
T
```

Как было замечено, мы без всяких проблем можем передавать вычислителю общие, ациклические списковые структуры:

```
CL-USER> (list #1=(list 0) #1# #1#)
((0) (0) (0))
```

Если мы напечатаем последнюю вычисленную форму, то мы увидим то же, что и выполнил вычислитель: обычный список с тремя отдельными ветвями:

```
CL-USER> +
(LIST (LIST 0) (LIST 0) (LIST 0))
```

Но, если мы присвоим специальной переменной `*print-circle*` не-nil значение и выведем на печать нашу структуру, то увидим что выражение на самом деле не является деревом, а является направленным ациклическим графом:

```
CL-USER> (let ((*print-circle* t))
           (print ++)
           t)

(LIST #1=(LIST 0) #1# #1#)
T
```

В качестве забавного примера, ниже показано как напечатать бесконечный список указав в качестве `cdr` сам `cons`, сформировав тем самым так называемый *цикл (cycle)* или *кольцо (circle)*:

```
* (print '#1=(hello.#1#))

(HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO ...
```

13

Вне зависимости от того как бы вы не продумывали ваши действия, убедитесь, что вы установили `*print-circle*` в истинное значение при *сериализации* структур циклических данных:

```
CL-USER> (let ((*print-circle* t))
           (print '#1=(hello.#1#))
           nil)

(|HELLO.#1#|)
NIL
```

Есть ли простой способ сказать является ли какая-либо часть списковой структуры циклической или содержит общую структуру? Да, предикат `cyclic-p` использует наиболее очевидный алгоритм для решения этого вопроса: рекурсивный проход через структуру с сохранением в *хэш-таблицу (hash-table)* всех обнаруженных `cons` ячеек. Если вы обнаружили `cons` ячейку, уже существующую в вашей хэш-таблице, то это однозначно говорит о том, что вы уже были здесь раньше и что это является циклом или общей веткой. Учтите, что поскольку рекурсия осуществляется через `cons` ячейки, то это значит что `cyclic-p` не в состоянии определить подобные ссылки в таких структурах данных, как векторы.

И наконец, поскольку большинство (смотрите [SYNTACTICALLY-RECURSIVE]) лисп компиляторов запрещает вам передавать циклические формы в компилятор, то неизвестно к чему приведёт выполнение следующего кода, но, скорее всего этот код сломает ваш компилятор введя его в бесконечный цикл компиляции:

```
(progn
  (defun ouch ()
```

[13] Примечание переводчика: этот приём у меня не сработал. Зато сработал вот такой:

```
CL-USER> (let ((a (list 'hello)))
           (setf (cdr a) a)
           (let ((*print-circle* nil))
             (write a)
             :done))
(HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO
 HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO ...
```

Листинг 4.6: CYCLIC-P

```
(defun cyclic-p-aux (l seen)
  (if (consp l)
      (or (gethash l seen)
          (progn
            (setf (gethash l seen) t)
            (or (cyclic-p-aux (car l) seen)
                (cyclic-p-aux (cdr l) seen))))))

(defun cyclic-p (l)
  (cyclic-p-aux l (make-hash-table)))
```

```
    #1=(progn #1#))
  (compile 'ouch))
```

4.6 Безопасность Считывателя

Расширяемость — возможность создавать новый функционал, который не был изначально задуман или реализован, — это почти всегда хорошая черта. В самом деле, поощрение расширяемости везде, где только это возможно сделало лисп тем замечательным языком, каким он является. Однако, есть моменты, когда мы не можем позволить расширять наши инструменты. В частности, мы не хотим чтобы посторонние могли расширить себя в нашу систему без нашего ведома или согласия. Это называется *взломом* или получением *root доступа (r00ted)*. На сегодняшний день наиболее интересные компьютерные вычисления в основном связаны с коммуникациями и сетями. Когда вы полностью контролируете обе программы, обменивающиеся данными, то очевидно, что вы доверяете всей системе. Но, как только возникает возможность контроля одной из программ некоторой не доверенной стороной, как вся доверенная система полностью разваливается подобно карточному домику.

Самым большим источником проблем *безопасности* является то, что программисты в шутку относят к *несоответствию импеданса*. Когда вы используете то, что не полностью понимаете, всегда есть вероятность неправильного использования. Есть две пути борьбы с несоответствиями импеданса: стиль (не используйте `strcpy(3)`) и понимание (то есть, чтение руководства). Лисп — это хороший язык для написания безопасного программного обеспечения. Причиной этому является то, что лисп

всегда выполняет то, что от него ожидают. Если вы всегда будете следовать предположению о том, что лисп всегда работает *верно*, то вы очень редко будете ошибаться. Например, если вы попытаетесь произвести запись за пределами привязок строки или вектора, что является очевидной проблемной ситуацией, то лисп вызовет исключение и немедленно и громко уведомит вас о проблеме. На самом деле, лисп выполняет эту операцию гораздо правильнее, чем вы можете себе представлять: после столкновения с исключением у вас есть возможность *перезапустить (restarting)* вашу программу с другого места тела программы сохранив большую часть ваших вычислений. Другими словами, система исключений Common Lisp не выполняет автоматическое разрушение стека вычислений при возникновении исключения: вы можете задействовать этот стек. В основном, из-за ограничений, связанных с объёмом этой книги, мы не будем описывать в деталях систему исключений[14]. Вместо этого я рекомендую вам книгу Питера Сибеля "Practical Common Lisp" [PRACTICAL-CL].

Но частью изучения лиспа является открытие о том, что всё расширяемо. Но как мы можем ограничить это? Оказывается мы думаем об этой проблеме в неверном направлении. Как и во всех областях компьютерной безопасности, вы не сможете рассматривать вопросы защиты до тех пор, пока вы не рассмотрите преступления. Во всех других областях программирования вы можете достичь хороших результатов с помощью конструктивного мышления, например выстраивая и используя абстракции. В области безопасности вы должны мыслить деструктивно. Вместо того, чтобы ждать возникновения ошибок с последующим их исправлением, вы должны пытаться найти ошибки ломая ваш код.

Так какими же атаками мы должны озаботиться? Нет никакой возможности атаковать программу, если у вас есть способ контроля *вводимых* в программу данных. Конечно, в нашем сетевом мире большинство программ становятся попросту бесполезными если не давать возможности вводить в них данные. По всему интернету существуют множество протоколов для перемешивания информации[15]. Разнообразность выполняемых операций в интернете попросту слишком велико для создания универсального стандарта для обмена данными. Лучшее что можно сделать — это предоставить расширяемый каркас и позволить программистам модифицировать протокол для соответствия создаваемому прило-

[14]Более точно это называется системой состояния, поскольку это более удобно чем просто исключения.

[15]Файл `nmap-service-probes`, который я помогаю поддерживать для Сканера Безопасности Nmap, — это одна из наиболее полных и часто обновляемых баз данных о подобных сервисах.

жению. В целом это означает уменьшение сетевой нагрузки, улучшенные алгоритмы передачи и большую надёжность. Однако главным достижением создания протокола является то, что мы можем уменьшить или устранить несоответствие импеданса, создавая тем самым безопасные протоколы.

Проблема, которая возникает при работе со стандартом обмена данными, следующая: при поддержке стандарта приложения не могут упрощать работу с протоколом. Обычно существуют некоторые базовые черты поведения приложения, которые должны соответствовать стандарту. Для того, чтобы создавать безопасные стандарты, мы должны убедиться в том, что получаем только те данные, которые можем обработать и ничего более.

Каков лисп путь обмена данными? Механизм получения данных в лиспе называется *лисп считывателем (lisp reader)*, а механизм выдачи называется *лисп принтером (lisp printer)*. Если вы уже дочитали эту книгу до этого места, то вы уже знаете более чем достаточно для проектирования и использования лисп протоколов. При программировании на лиспе вы используете подобный протокол. Вы взаимодействуете с лиспом путём передачи ему лисп форм и довольно часто этот способ оказывается лучшим для взаимодействия с остальным миром. Конечно, нельзя доверять всему остальному миру и поэтому нужно принять меры предосторожности. Помните, что для того, чтобы думать о безопасности вы должны думать об атаках. Проектировщики Common Lisp думали об атаках на считыватель. Ранее в этой главе мы описали считывающий макрос `#.` позволяющий считывателю выполнять лисп выражения, таким образом, мы можем кодировать не-сериализуемые структуры данных. Для смягчения очевидных атак на лисп считыватель Common Lisp предоставляет `*read-eval*`. Из CLtL2:

> Привязка `*read-eval*` к `nil` полезна в случае считывания данных из таких ненадёжных источников, как сеть или пользовательские файлы данных; она предотвращает использование считывающего макроса `#.` в роли “Троянского Коня”, позволяющего вычислять произвольные формы.

В июне 1989-го комитет Common Lisp проголосовал за `*read-eval*`, в этот момент они думали как злоумышленники. Какую разновидность троянского коня могли бы использовать злоумышленники? Корректный ответ, с точки зрения автора безопасного программного обеспечения, следующий: самое худшее о чём вы только можете подумать — и даже хуже. Всегда думайте о том, что атакующие могут получить полный контроль над вашей системой. Традиционно, это обозначает, что троянский

конь будет представлять из себя нечто под названием *код оболочки (shell code)*. Кодом оболочки обычно является хитро собранный кусок машинного кода, предоставляющего нечто вроде unix оболочки злоумышленнику, которая в последующем используется для получения r00t доступа. Создание такого кода оболочки — это настоящее произведение искусства, особенно с учётом необычных обстоятельств, задействованных в подобных атаках. Для примера: большинство кодов оболочки не может содержать null байты, поскольку в строках в C-стиле эти байты обозначают конец строки, предотвращая включение дальнейшего кода оболочки. Ниже приведён пример лисповского кода оболочки, здесь подразумевается что жертва запустила CMUCL и в системе уже была установлена программа Hobbit'a *netcat* ("nc") [NETCAT]:

```
#.(ext:run-program
    "/bin/nc" '("-e" "/bin/sh" "-l" "-p" "31337"))
```

Этот код слушает входящие соединения на 31337 порту и будет предоставлять unix оболочку для всех, кто подключился. В троянских конях много сил уходит на то, чтобы сделать их как можно более переносимыми и надёжными, так они смогут получать r00t доступ на большинстве целей. Часто это оказывается чрезвычайно трудной задачей. Но в атаках на лисп считыватель это чрезвычайно просто. Ниже представлен обновлённый код оболочки, он является переносимым между CMUCL и SBCL:

```
#.(#+cmu ext:run-program
   #+sbcl sb-ext:run-program
    "/bin/nc" '("-e" "/bin/sh" "-l" "-p" "31337"))
```

Мораль: всегда убеждайтесь в том, что вы привязали `*read-eval*` в *nil* при обработке любых данных, к которым вы хоть чуть-чуть испытываете недоверие. Если вы редко используете считывающий макрос "`#.`", то вы должны быть достаточно мудры, чтобы установить его `setq` в `nil` и разрешать только тогда, когда вам это нужно.

Отключить считывающий макрос `#.` достаточно легко. Но достаточно ли этого? Это зависит от вашего приложения и того, что вы считаете эффективной атакой. Для интерактивных программ эта мера может быть достаточна. Если мы получим некорректные данные, то будем оповещены об этом настолько громко, насколько это возможно. Однако, для интернет серверов этого скорее всего недостаточно. Рассмотрим этот код оболочки:

```
)
```

Или это:

```
no-such-package:rewt3d
```

Обычно лисп будет выдавать ошибку, поскольку мы пытаемся считать не сбалансированную форму или запрашиваем символ в несуществующем пакете. Скорее всего работа всего нашего приложения будет остановлена. Эта ситуация известна как атака *"отказ в обслуживании" (denial of service attack)*. Наиболее трудно уловимой и более сложной в отладке атакой "отказа в обслуживании" является передача повторяющейся формы с помощью использования считывающих макросов "`##`" и "`#=`". Если наш код, обрабатывающий эти данные, был написан без учёта подобных форм, то результатом будет несоответствие импеданса и, вполне возможно, угроза безопасности. С другой стороны, наше приложение может зависеть от возможности передавать циклические и общие структуры данных. Требования безопасности данных полностью зависят от приложения. К счастью вне зависимости от ваших требований, лисп считыватель и принтер соответствуют своим задачам.

`Safe-read-from-string` — это частичный ответ на проблему безопасности считывателя. Эта функция менее пригодна к использованию в промышленном коде, по сравнению с большинством остального кода в этой книге. Поэтому мы предупреждаем вас: внимательно подумайте о требованиях безопасности в вашем приложении и адаптируйте (или даже перепишите) этот код для вашего приложения. `Safe-read-from-string` — это очень ограниченная версия `read-from-string`. Она имеет свою собственную копию в стандартной лисповской *таблице считывания*. В этой копии удалены большинство интересных макросов, включая диспетчеризующий макрос "`#`". Это означает что нам недоступны векторы, битвекторы, gensym'ы, цикличные ссылки и "`#.`". `Safe-read-from-string` не позволяет использовать ключевые слова или сторонние символы пакетов. Однако, нам по прежнему доступны *cons* структуры, а не только хорошо сформированные списки. Кроме того, нам доступны числа[16] и строки.

`Safe-read-from-string` использует лисповскую систему исключений для перехвата всех ошибок лисп функции `read-from-string`. Если будет обнаружена хоть какая-нибудь проблема при считывании строки, включая обнаружение несбалансированных скобок или обнаружение любого из считывающих макросов, помещённых в чёрный список в перемен-

[16]Упражнение: Какой класс чисел является не доступным?

Листинг 4.7: SAFE-READ-FROM-STRING

```
(defvar safe-read-from-string-blacklist
  '(#\# #\: #\|))

(let ((rt (copy-readtable nil)))
  (defun safe-reader-error (stream closech)
    (declare (ignore stream closech))
    (error "safe-read-from-string failure"))
  (dolist (c safe-read-from-string-blacklist)
    (set-macro-character
     c #'safe-reader-error nil rt))
  (defun safe-read-from-string (s &optional fail)
    (if (stringp s)
        (let ((*readtable* rt) *read-eval*)
          (handler-bind
              ((error (lambda (condition)
                        (declare (ignore condition))
                        (return-from
                         safe-read-from-string fail))))
            (read-from-string s)))
        fail)))
```

ной `safe-read-from-string-blacklist`, то `safe-read-from-string` вернёт значение, переданное в качестве второго аргумента, или `nil` в случае когда ничего не было передано (помните, что вам нужно считывать `nil`). Вот как это обычно используется[17]:

```
(let* ((g (gensym))
       (v (safe-read-from-string
           user-supplied-string g)))
  (if (eq g v)
      (log-bad-data ; осторожно с протоколированием!
       user-supplied-string)
    (process v)))
```

Конечно, эта версия `safe-read-from-string` очень ограничена и, возможно, потребуется модифицировать её для использования в вашем приложении. В частности, вам могут понадобится ключевые символы. Разрешить их очень просто: достаточно привязать список без символа "`:`" к `safe-read-from-string-blacklist` при использовании **safe-read-from-string**, и учтите, что ваши символы могут находиться в нескольких пакетах (включая пакет `keyword`). Даже если вы удалите символ "`:`" наш вышеприведённый пакет кода оболочки будет сорван, поскольку мы отлавливаем все ошибки в процессе чтения, включая ошибки обозначающие несуществующие пакеты. `*Read-eval*` всегда привязан к `nil` на случай если вы решите удалить символ "`#`" из чёрного списка. Если вы захотите поступить таким образом, то вам понадобится создать ещё один чёрный список для диспетчеризующего макроса "`#`" (чёрный список может быть довольно большим). Символ вертикальной черты помещён в чёрный список, поэтому мы не будем считывать различные дурно выглядящие символы.

Таким образом мы можем ограничить работу считывателя так сильно насколько это нам будет нужно и насколько это позволит наше приложение. Мы убедились, что защищены от *векторов атак* исходящих со стороны программного обеспечения, используемого для считывания форм. Теперь нужно перейти к вопросу минимизации импеданса между тем, что мы считаем структурой лисп форм и тем, чем оно на самом деле является. Мы должны убедиться что это соответствует тому, что мы ожидаем. Некоторые структуры данных называют эту процедуру *проверкой* через *схему* (*validation* against *schema*), но лисп называет

[17]Конечно, если мы будем использовать это в макросе, то нам будет нужно использовать `defmacro!` и его автоматические `gensym`'ы.

это `destructuring-bind` (“деструктурирующая-привязка”) через *расширенную лямбда форму* (`destructuring-bind` against an *extended lambda form*). Все эти термины представляют простые концепции, но звучат очень серьёзно. Идея заключается в том, что вы хотите убедиться, что данные — форма или структура — является именно тем, что должно быть обработано. `Destructuring-bind` проверяет эти структуры для нас, предоставляя очень удобный схематичный язык, включающий ключевые слова или необязательные параметры, кроме того, есть бонус в виде возможности именования отдельных частей структуры.

Я могу дать несколько примеров использования `destructuring-bind`, но, это не обязательно: мы ещё будем использовать все виды деструктуризации. Аргумент или список параметров вставляемый сразу после имени макроса при использовании `defmacro`, `defmacro!` или **destructuring-bind** называется расширенным лямбда списком подчёркивая тот факт, что это более мощно чем деструктуризация, применяемая для одинарного лямбда списка. Мы можем вложить расширенные лямбда списки в деструктуризацию списковой структуры на произвольную глубину. Деструктуризация подробно рассмотрена в книга Пола Грэма *On Lisp*. После прочтения *раздела Пандорические Макросы* я рекомендую уделить внимание макросу Грэма `with-places`.

Таким образом, каждый раз, когда вы пишете макрос или функцию, вы в некотором роде, рассматриваете аргументы, принимаемые этим макросом или функцией, как данные, а расширенный или обычный лямбда список — как схему. С этой точки зрения проверка данных выглядит несложной. Лисп может проверять являются ли наши данные должным образом структурированными, и в противном случае будет порождать состояния ошибки. Также, как и в случае со считывателем, при обработке данных, мы должны очень внимательно отнестись к возможным атакам и затем использовать лисповскую системы исключений и макросов для конструирования схемы проверки, пропускающую только необходимый минимум, требуемый приложением, уменьшая или устраняя любые несоответствия импеданса. Для решения этой задачи необходимо использовать регулярные выражения CL-PPCRE. Никакой другой язык не может сравниться с Лиспом в плане создания безопасного программного обеспечения, со временем вы всё больше будете убеждаться в этом.

Глава 5

Программирующие Программы

5.1 Лисп — Не Функциональный Язык

Одно из наиболее распространённых заблуждений о лиспе гласит: лисп — это функциональный язык программирования. Лисп — это не функциональный язык программирования. На самом деле можно утверждать, что лисп — это один из наименее функциональных языков программирования которые когда либо были созданы. Это ошибочное мнение имеет интересные корни и служит хорошим примером того, как маленькое недоразумение может иметь долгоиграющие последствия, приводящие к путанице уже после того, как сами причины уже давно канули в лету. Что такое функциональный язык программирования? Единственное, имеющее смысл определение следующее:

> Функциональный язык — это язык программирования, состоящий из функций.

Итак, что же такое функция? Функция — это *математическая* концепция существующая на протяжении веков:

> Функция — это статичное, точно определённое преобразование исходных значений в результирующие значения.

Похоже на `defun`, который мы используем для определения новых функций в лиспе. Например, следующее выражение выглядит как функция, использующая сложение для преобразования набора всех чисел в новый набор, один из которых также включает все числа:

```
(defun adder (x)
  (+ x 1))
```

Очевидно мы можем применить этот объект к любому числу и получить результат, но является ли `adder` настоящей функцией? Лисп утверждает что `adder` — это функция[1]:

```
CL-USER> (describe #'adder)
#<FUNCTION ADDER>
  [compiled function]

Lambda-list: (X)
Derived type: (FUNCTION (T) (VALUES NUMBER &OPTIONAL))
Source form:
  (SB-INT:NAMED-LAMBDA ADDER
      (X)
    (BLOCK ADDER (+ X 1)))
; No value
```

Но, именование этих объектов функциями является некорректным использованием термина с глубокими корнями в истории лиспа. `Defun` и лямбда формы на самом деле создают *процедуры*, или если говорить более точно, *экземпляры, вызываемые как функции (funcallable instances)* [AMOP]. В чём различие? Процедуры не обязательно должны содержать в себе работу с преобразованием значений, но процедуры являются участками кода, возможно содержащим сохраняемую среду, которую можно выполнить (funcall). Когда лисп программисты пишут программы в определённом стиле, называемом *функциональным стилем*, то получающиеся процедуры можно рассматривать и комбинировать в математическом стиле функциональных преобразований.

Причина, по которой лисп так часто называют функциональным языком связана с историей. Хотите верьте, хотите нет, но было время когда большинство языков не поддерживало концепцию процедур, которую современные программисты считают само собой разумеющимся в любом языке. На заре зарождения языки не предоставляли удобных абстракций для локального именования аргументов в участках повторно используемого кода и программисты были вынуждены вручную управлять регистрами и манипулировать стеком для достижения подобного поведения. Лисп — это язык, который уже тогда поддерживал процедуры, был более функциональным чем остальные языки.

[1]Если вы до сих пор не знакомы с функцией COMMON LISP **describe** то вам нужно немедленно с ней ознакомиться. Примените **describe** к функции, специальной форме, макросу, переменной, символу и замыканию.

После того как процедурным абстракциям было уделено должное внимание и они были включены во все языки программирования, люди медленно начали работать над преодолением барьеров ограниченной природы реализованных процедур. Программисты стали понимать что было бы неплохо иметь возможность возвращать процедуры из других процедур, встраивать их в новые среды, объединять их в структуры данных и, в целом, рассматривать их как самые простые значения. Лозунг, побудивший программистов перейти к подобной абстракции был следующим: общество без классов, *первоклассные процедуры*. В сравнении с языками, которые относили процедуры к предыдущему второму классу, лисп — язык, который уже обладал первоклассными процедурами, выглядел более функциональным.

И наконец, обычно многие языки делают бессмысленное различие между выражением и оператором, для того, чтобы поддерживать ужасный Блаб синтаксис, например: инфиксный. В лиспе всё возвращает что-либо[2] и нет (синтаксических) ограничений для вложения или комбинации. Это простой вопрос с очевидным ответом: что более важно в языке, синтаксис, дружественный к новичкам, или настоящая гибкость? Все языки, которые используют инфиксный синтаксис уменьшают возможности абстракции многими способами. К счастью большинство современных языков доверяют своим пользователям пользователям и дают им возможность комбинировать выражения так, как они считают нужным. В сравнении с языками, которые принимают подобные мозгоубивающие синтаксические решения, лисп выглядит более функциональным.

После того, как программисты привыкли к этой вездесущей и достаточно ошибочной терминологии, пришло понимание что понятие функции, использованной в больших дебатах посвящённым функциональным и не функциональным языкам не только запутывает, но и является фундаментальным шагом назад. Чтобы исправить это программисты и учёные вернулись к меловой доске и математическому определению функции: преобразование входных значений в выходные значения. Если лисп является функциональным языком, то он настолько же функционален, насколько функциональны такие современные языки программирования как Perl и JavaScript.

Очевидно что лисп процедуры не являются функциями. Лисп процедуры могут возвращать не-статичные значения, эти процедуры вы можете вызывать несколько раз с одними и теми же аргументами и каждый раз можете получать разные значения. В наших примерах из преды-

[2]Исключением является ничего не возвращающий **(values)**. Но, это сводится к **nil** и по этой причине может использоваться в выражениях.

дущих глав мы увидели что лисп процедуры могут хранить состояния. Процедуры подобные `rplaca` могут изменять значения не только в памяти, но и в других местах (таких как регистры). Такие лисп процедуры как `terpri` и `format` создают вывод (новые строки в случае `terpri`) направляемые в терминал или файлы[3]. Лисп процедуры подобные `yes-or-no-p` могут читать ввод из терминала и возвращать значения в зависимости от ввода пользователя. Являются ли эти процедуры статичными, точно определёнными преобразованиями?

Поскольку лисп процедуры — это не математические функции, то лисп — это не функциональный язык. На деле сильная аргументация может привести к тому, что лисп может стать менее функциональным чем остальные языки. В большинстве языков выражения, выглядящие как вызовы процедур являются вызовами процедур и изменить это мешает синтаксис языка. В лиспе мы работаем с макросами. Как мы уже увидели ранее, макросы могут невидимо изменить значение некоторых форм от вызова функции и до независимых лисп выражений — техника, которая способна нарушить ссылочную прозрачность многими способами, что попросту невозможно в других языках.

Выяснилось что большинство языков на самом деле не являются функциональными, после этого некоторые разработчики языков решили узнать как будет выглядеть программирование на настоящих функциональных языках. Как вы можете ожидать программирование на функциональных языках в основном раздражает и бывает непрактичным. Почти нет проблем относящихся к реальному миру и которые можно было бы выразить через статичные, чётко определённые преобразования из исходных значений в результирующие значения. Но это не значит что функциональное программирование не имеет плюсов и многие языки были спроектированы так, чтобы получить преимущества от функционального стиля программирования. Это означает нахождение удобного способа изоляции функциональных частей программ от (на самом деле интересных) не-функциональных частей. Такие языки как Haskell и OCaml используют эту изоляцию как средство создания агрессивных оптимизационных допущений.

Но это лисп. Мы очень не-функциональны и очень гордимся этим. По мере того, как увеличивается польза от изоляции побочных эффектов, лисп программисты могут и должны реализовывать это с помощью макросов. Настоящее назначение функционального программирования в том, чтобы разделить функциональное описание того что происходит от механизма происходящих событий. Определённо, лисп не функцио-

[3] **Terpri** и **rplaca** — они были названы не просто так.

нален, но, благодаря макросам не существует более лучшей платформы или материала для реализации более функционального языка чем лисп.

5.2 Программирование Сверху-Вниз

> Вы не можете обучать новичков программированию сверху вниз, поскольку они не знают какой низ является верхом.
>
> — Чарльз Энтони Ричард Хоар

В *разделе Предметно-Ориентированные Языки* при первом знакомстве с предметно-ориентированными языками мы создали простой макрос `unit-of-time`. Этот макрос дал нам удобный способ определения периодов времени в различных величинах с помощью основанного на символах, интуитивно-понятного синтаксиса:

```
CL-USER> (unit-of-time 1 d)
86400
```

`Unit-of-time` — это удобный предметно-ориентированный язык, поскольку программисту не требуется помнить, например, количество секунд, содержащихся в сутках. `Unit-of-time` реализован с помощью простого макроса, использующего оператор case в роли сердцевины для нижележащего расширения.

Важным принципом в проектировании макросов является программирование *сверху-вниз (top-down)*. Проектирование лисп макроса следует начать с создания абстракции. Вам нужно писать программы, использующие макросы ещё до того, как вы написали сам макрос. Что парадоксально, вам нужно знать как писать на языке ещё до того, как вы сможете написать краткое определение/реализацию этого языка.

Таким образом первым шагом в конструировании серьёзного макроса является написание *вариантов использования (use cases)* макроса, даже если нет никакого способа с помощью которых вы смогли бы проверить или использовать эти макросы. Если программы, написанные в этом новом языке окажутся достаточно объемлющими, то идея о том, что лучше реализовать: компилятор или интерпретатор последует сама собой.

Рассмотрим наш макрос `unit-of-time`, существует ли способ перевести этот макрос на ещё больший уровень спецификации и создать язык, предназначенный для создания макросов, создающих удобные единицы измерения? Что же, `unit-of-time` — это макрос, и чтобы выполнить задуманное, нам нужен макрос, определяющий другой макрос...

Cmon! Этот низ является верхом.

Мы начали рассматривать задачу не с реализации языка, а с того, что задали себе вопрос: для чего мы хотим использовать этот язык? Ответ оказался следующим: мы хотим получить простой способ определения вспомогательных программ, преобразующих единицы измерения. В следующем варианте использования мы хотим получить тип единицы измерения, `time`, базовую единицу выраженную в секундах и представленную здесь через `s`, набор пар из единиц и коэффициентов преобразования этих единиц в базовую единицу:

```
(defunits% time s
  m 60
  h 3600
  d 86400
  ms 1/1000
  us 1/ 1000000)
```

`Defunits%` может расшириться в код, определяющий макрос, подобный `unit-of-time`, написанный нами в *разделе Предметно-Ориентированные Языки*, что позволяет нам преобразовывать произвольные единицы времени в секунды. Можно ли ещё более улучшить этот код?

На этом месте мозгового штурма прекращается создание инноваций в архитектуре программы, разрабатываемой на большинстве других языков. Мы всего лишь создали способ применения множителей для различных единиц измерения в код, позволяющий нам преобразовать единицы измерения в удобный нам вид. Но, профессиональный лисп программист определит, что это преобразование — программа, следовательно, может быть расширена через те-же приёмы, что применяются для расширения обычных лисп программ.

При вводе многих разнообразных единиц измерений может быть полезно определять единицы в терминах других единиц. Коэффициент, используемый для умножения единиц может также являться списком со значением, относящемуся к другой единице, например:

```
(defunits%% time s
  m 60
  h (60 m)
  d (24 h)
  ms (1/1000 s)
  us (1/1000 ms))
```

Листинг 5.1: DEFUNITS-1

```
(defmacro! defunits% (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity) (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
                ((,base-unit) 1)
                ,@(mapcar (lambda (x)
                            `((,(car x)) ,(cadr x)))
                          (group units 2))))))
```

Связывание (chaining) единиц измерений выглядит естественной операцией. Минуты определены в терминах наших базовых единиц, секунд, часы — в терминах минут, и дни — в терминах часов. Реализуя наш макрос в итеративном стиле мы, для начала, реализуем несвязываемое поведение с `defunits%`, после этого мы реализуем связывание с `defunits%%` ещё до добавления соответствующих проверок на ошибки и переходу к реализации финальной версии, `defunits`.

Заметьте, что этот новый язык может больше, чем просто предоставление удобного синтаксиса для добавления новых типов единиц. Этот язык также позволяет нам уменьшить воздействие *округления (rounding)* в наших вычислениях и позволяет лиспу использовать настолько точную арифметику насколько это возможно. Например, фарлонг — это $1/8$ мили, и если мы закодируем это значение с использованием очередизации взамен, скажем, метрического приближения, то в конечном итоге мы можем получить более точные результаты или, что ещё более важно, результаты, максимально соответствующие вычислениям, произведёнными с помощью миль. Поскольку нам достаточно всего лишь добавить самый точный коэффициент преобразования не прибегая к каким-либо другим преобразованиям, то этот макрос даёт нам возможность создавать правила на уровне выражений, невозможных в других языках.

Используя поведение автоматических gensym-ов, описанное в *разделе Нежелательный Захват*, написание `defunits%` оказывается чрезвычайно простым делом. Функция Грэма `symb` сгенерирует новое имя для макроса конвертации. Например, если символ `time` — это тип уже существующей единицы, то новый макрос конвертации будет называться `unit-of-time`. `Defunits%` был сконструирован исходя из описания `unit-of-time`, описанном в *разделе Предметно-Ориентированные Языки*, но, этот код окружён `defmacro!` и обратной кавычкой, и замещены части, нуждающиеся в повторном создании при каждом вызове макроса.

`Defunits%` использует *вложенные обратные кавычки* — чрезвычайно сложную для понимания конструкцию. Программирование с обратными кавычками — это тоже самое, что и обладание ещё одним смысловым измерением в коде. В других языках, определённый программный оператор обычно обладает очень простой вычислительной семантикой. Вы знаете когда буден запущен каждый бит кода, поскольку каждый бит кода вынужден выполняться в одно и тоже время: во время выполнения. Но в лиспе с помощью вложенных обратных кавычек мы можем масштабироваться вверх и вниз по *лестнице закавычивания (ladder of quotation)*. Каждая написанная нами обратная кавычка — это один шаг по этой лестнице: наш код — это список, вычисление которого зависит от нашего желания. Но внутри обычного списка каждая встреченная запятая переносит нас на шаг вниз по лестнице закавычивания и исполняет код из соответствующего шага лестницы [CLTL2-P967].

Таким образом это простой алгоритм для определения времени вычисления каждого бита из лисп формы. Просто начните с корня выражения и после встречи с каждой обратной кавычкой пометьте переход на ещё один уровень закавычивания. Для каждой встреченной запятой пометьте переход на уровень ниже. Как заметил Стил [CLTL2-P530], следование этому уровню закавычивания может быть трудной задачей. Эта трудность — потребность отслеживать вашу текущую глубину вложения — это то, что заставляет ощущать использование обратных кавычек другим измерением добавленным в обычное программирование. В других языках вы можете пойти на север, юг, восток и запад, но лисп вдобавок даёт вам возможность перемещаться вверх.

`Defunits%` — это хороший шаг, но в нём по прежнему не реализована очередизация. В данный момент, макро реализация этого языка — это в основном простая замена. Реализация поведения очередизации требует более запутанной программной логики. Простая подстановка уже не будет работать поскольку части макроса зависят от остальных частей макроса, поэтому при постройке нашего расширения нам нужно полностью обрабатывать передаваемые макросу формы, а не просто думать о них в терминах отдельных кусочков, которые мы можем сращивать.

Следует помнить, что макросы — это на самом деле просто функции, мы создали функцию-утилиту для использования её в определении макроса, `defunits-chaining%`. Эта функция-утилита получает единицу, определённую символами, подобными `S`, `M` или `H`, и список определений единиц. Мы разрешаем работу с определениями единиц до получения числа, которое будет интерпретироваться базовой единицей, например `(M 60)` или списком, содержащим неявную отсылку к другой единице в очереди, например `(H (60 M))`.

Листинг 5.2: DEFUNITS-CHAINING-1

```
(defun defunits-chaining% (u units)
  (let ((spec (find u units :key #'car)))
    (if (null spec)
        (error "Unknown unit ~a" u)
        (let ((chain (cadr spec)))
          (if (listp chain)
              (* (car chain)
                 (defunits-chaining%
                     (cadr chain)
                     units))
              chain)))))
```

Это рекурсивная функция — утилита. Для нахождения множителя, применяемого для базовой единицы мы перемножаем каждое звено очереди с другим вызовом функции-утилиты используемой для обработки остальной части очереди. После возврата из стека вызовов мы получим множитель, применяемый для конвертации значения единицы в значение базовой единицы. Например, когда мы конструируем множители для часов, мы находим что в одном часе 60 минут. Мы переходим в рекурсию и находим что в одной минуте 60 секунд. Мы опять переходим в рекурсию и находим что секунда — это конец очереди — минуты были определены непосредственно в терминах базовых единиц. Итак, вернувшись из рекурсии мы вычисляем `(* 60 (* 60 1))`, что равняется 3600: то есть, в одном часе 3600 секунд.

После определения этой функции-утилиты, для работы с множителем для каждой единицы требуется простая модификация `defunits%`, которую мы выполнили в `defunits%%`. Вместо объединения в значение прямо из определения единицы мы передаём каждую единицу и целое определение единицы в утилиту `defunits-chaining%`. Как было описано выше, эта функция рекурсивно преобразует множитель, требуемый для каждой единицы, в базовую единицу. С этим множителем `defunits%%` может объединять значение в case операторы, также, как это выполняет `defunits%`.

Эти макросы всё ещё не завершены. Макрос `defunits%` не поддерживает очередизацию. `Defunits%%` поддерживает очередизацию, но не проверяет на наличие ошибок. Профессиональный писатель макросов всегда заботится об обработке любых возможных ошибочных состояний. Особенно важны возникновения бесконечных циклов или другие труд-

Листинг 5.3: DEFUNITS-2

```
(defmacro! defunits%% (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity) (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
                ((,base-unit) 1)
                ,@(mapcar (lambda (x)
                            `((,(car x))
                              ,(defunits-chaining%
                                 (car x)
                                 (cons `(,base-unit 1)
                                       (group units 2)))))
                          (group units 2))))))
```

ные для отладки в REPL-е случаи.

Проблема связанная с `defunits%%` является частью создаваемого нами языка: в этом языке возможны программы, содержащие циклы. Например:

```
(defunits time s
  m (1/60 h)
  h (60 m))
```

Для того, чтобы добавить соответствующий отладочный вывод, мы должны несколько улучшить нашу реализацию. Наша окончательная версия `defunits` поддерживает очередизацию и печатает удобный отладочный вывод указывающий пользователю на наличие подобных циклических зависимостей. Всё это благодаря `defunits-chaining`, улучшенной версии `defunits-chaining%` работающей со списком ранее встреченных единиц. Таким образом, если мы повторно столкнёмся с уже существующей в списке единицей, то мы можем выдать ошибку, которая лаконично описывает проблему:

```
CL-USER> (defunits time s
           m (1/60 h)
           h (60 m))
 M depends on H depends on M
   [Condition of type SIMPLE-ERROR]
```

Макрос defunits идентичен `defunits%%` за исключением того, что он передаёт дополнительный аргумент `nil` функции `defunits-chaining`,

Листинг 5.4: DEFUNITS

```
(defun defunits-chaining (u units prev)
  (if (member u prev)
      (error "~{ ~a~^ depends on~}"
             (cons u prev)))
  (let ((spec (find u units :key #'car)))
    (if (null spec)
        (error "Unknown unit ~a" u)
        (let ((chain (cadr spec)))
          (if (listp chain)
              (* (car chain)
                 (defunits-chaining
                     (cadr chain)
                     units
                   (cons u prev)))
              chain)))))

(defmacro! defunits (quantity base-unit &rest units)
  `(defmacro ,(symb 'unit-of- quantity)
       (,g!val ,g!un)
     `(* ,,g!val
         ,(case ,g!un
                ((,base-unit) 1)
                ,@(mapcar (lambda (x)
                            `((,(car x))
                              ,(defunits-chaining
                                 (car x)
                                 (cons
                                  `(,base-unit 1)
                                  (group units 2))
                                 nil)))
                          (group units 2))))))
```

являющийся концом списка, отображающего историю уже посещённых единиц. Если при поиске новой единицы мы обнаружим что уже встречали её, то это обозначает обнаружение цикла. Мы можем использовать эту историю посещённых единиц для печати вспомогательных сообщений пользователям макросов, написавших циклы.

Итак, `defunits` — это специфичный язык, предназначенный для введения единиц в процедуру конвертации. На самом деле этот язык предназначен для ещё более тонкой области чем эта; есть множество путей для создания этого языка. Поскольку создавать языки в Блабе трудно, но, легко в Лиспе, лисп программисты обычно не стремятся запихивать всё в один язык. Вместо этого они делают язык всё более и более предметно-ориентированным до тех пор, пока конечная цель не станет тривиальной.

Примером использования `defunits` является `unit-of-distance` (единицы — расстояния). В 1970 году морская сажень сократилась, по крайней мере для британских моряков, на 1/76 часть:

```
CL-USER> (/ (unit-of-distance 1 fathom)
            (unit-of-distance 1 old-brit-fathom))
75/76
```

Что чуть больше 2-х сантиметров:

```
CL-USER> (coerce
          (unit-of-distance 1/76 old-brit-fathom)
          'float)
0.024384
```

5.3 Неявные Контексты

Макросы могут использовать технику под названием *неявный контекст (implicit context)*. Иногда мы решаем неявно добавить лисп код в некоторые выражения, которые мы не хотим часто писать и хотим использовать как абстракцию, нередко это применяется в многократно используемом коде или в коде, который должен быть краток и задействуется в других частях программы. До этого момента мы уже беседовали о неявных контекстах и вам уже должно быть ясно, что неявные контексты даже безотносительно программирования макросов являются фундаментальной частью лисп программирования: формы let и lambda обладают *неявным progn-ом*, поскольку они вычисляют, поочерёдно, формы их тела и

Листинг 5.5: UNIT-OF-DISTANCE

```
(defunits distance m
  km 1000
  cm 1/100
  mm (1/10 cm)
  nm (1/1000 mm)

  yard 9144/10000 ; Принято в 1956
  foot (1/3 yard)
  inch (1/12 foot)
  mile (1760 yard)
  furlong (1/8 mile)

  fathom (2 yard) ; Принято в 1929
  nautical-mile 1852
  cable (1/10 nautical-mile)

  old-brit-nautical-mile ; Отменено в 1970
  (6080/3 yard)
  old-brit-cable
  (1/10 old-brit-nautical-mile)
  old-brit-fathom
  (1/100 old-brit-cable))
```

Листинг 5.6: TREE-LEAVES-1

```
(defun tree-leaves% (tree result)
  (if tree
      (if (listp tree)
          (cons
           (tree-leaves% (car tree)
                         result)
           (tree-leaves% (cdr tree)
                         result))
        result)))
```

возвращают последний результат. `Defun` добавляет *неявную lambda* вокруг форм, и поэтому вам не нужно использовать лямбда формы для именованных функций.

В этом разделе мы изучим макрос `tree-leaves`[4], его создание и конструкцию. Макрос `tree-leaves`, предназначен для *прохода по коду (code-walking)*, который позже будет использоваться в этой книге. Подобно `flatten` этот макрос исследует переданный ему лисп код, рассматривая его как дерево, затем производит некоторые модификации и возвращает новое дерево. Списковая структура исходного выражения не модифицируется: `flatten` и `tree-leaves` оба создают новую структуру. Разница между ними заключается в том, что основная цель `flatten` в удалении вложенных списков и в возвращении плоского списка, уже не являющегося лисп кодом, а `tree-leaves` сохраняет структуру выражения, но изменяет значения отдельных атомов.

Начнём с простого эскиза. `Tree-leaves%` — это функция, рекурсивно проходящая по предоставленному выражению `tree`, cons-я новую списковую структуру с той же формой[5]. Когда она обнаруживает атом, то вместо того, чтобы вернуть этот атом она возвращает значение аргумента `result`:

```
CL-USER> (tree-leaves%
  '(2 (nil t (a . b)))
  'leaf)
(LEAF (NIL LEAF (LEAF . LEAF)))
```

[4]Также стоит посмотреть функцию COMMON LISP `subst`.

[5]Пустой `else` случай в форме `if` возвращает `nil`, также являющийся пустым списком.

Листинг 5.7: PREDICATE-SPLITTER

```
(defun predicate-splitter (orderp splitp)
  (lambda (a b)
    (let ((s (funcall splitp a)))
      (if (eq s (funcall splitp b))
          (funcall orderp a b)
        s))))
```

Итак `tree-leaves%` возвращают новое дерево, в котором все атомы преобразуются в предоставленный нами символ, `leaf`. Заметьте, что атом `nil` в car позиции cons ячейки не изменился поскольку он не изменится при перемещении его в cdr позицию (и представляет из себя пустой список).

Конечно, изменение каждого элемента это весьма бесполезно. То что нам нужно на самом деле — так это возможность выбора определённых атомов и возможность избирательной трансформации их в новые атомы для последующей вставки в новую списковую структуру, оставляя неинтересные нам атомы без изменений. В лиспе самым лёгким способом является написание изменяемой функции-утилиты, дающей возможность применять *плагины (plug-ins)* где пользователь может использовать произвольный код для контроля поведения утилиты. Примером этому может служить функция sort, включённая в COMMON LISP. В этом примере в функцию `sort` подключена в виде плагина функция меньше чем:

```
CL-USER> (sort '(5 1 2 4 3 8 9 6 7) #'<)
(1 2 3 4 5 6 7 8 9)
```

Эта концепция получения функции, предназначенной для контроля поведения особенно удобна, поскольку мы можем создавать анонимные функции, специально предназначенные для наших задач. Или, с целью получения ещё большей мощи, мы можем создать функции, которые будут порождать эти анонимные функции для нас. Также это известно как *комбинирование функций (function composition)*. И хотя комбинирование функций не так интересно как комбинирование макросов[6], оно по прежнему остаётся чрезвычайно полезной техникой, которой должны овладеть все профессиональные лисп программисты.

`Predicate-splitter` — это простой пример комбинирования функции. Эта функция предназначена для комбинирования двух предикатов

[6]Вот почему комбинирование функций занимает только пару параграфов в то время когда комбинирование макросов занимает большую часть этой книги.

Листинг 5.8: TREE-LEAVES-2

```
(defun tree-leaves%% (tree test result)
  (if tree
      (if (listp tree)
          (cons
           (tree-leaves%% (car tree) test result)
           (tree-leaves%% (cdr tree) test result))
        (if (funcall test tree)
            (funcall result tree)
          tree))))
```

в один новый предикат. Первый предикат получает два аргумента предназначенных для упорядочивания элементов. Второй предикат получает один аргумент и определяет относится ли элемент к определённому классу элементов, которых вы хотите разделить с помощью вашего предиката. Например, ниже мы применили `predicate-splitter` для создания нового предиката, работающего также, как и меньше чем, за исключением того, что чётные числа будут меньше чем нечётные числа:

```
CL-USER> (sort '(5 1 2 4 3 8 9 6 7)
               (predicate-splitter #'< #'evenp))
(2 4 6 8 1 3 5 7 9)
```

Итак, как использовать функции в роли плагина для управления работой `tree-leaves%`? В обновлённой версии `tree-leaves%`, мы добавили две отличающихся функции-плагина, из которых одна предназначена для определения изменяемых листьев, а другая определяет как преобразовывать старый лист в новый лист, соответственно, эти плагины названы как `test` и `result`, а новая версия названа как `tree-leaves%%`.

Мы можем использовать `tree-leaves%%` передавая ему два лямбда выражения, каждое из которых должно получать единственный аргумент, `x`. В данном случае нам нужно получить новое дерево: с той же самой списковой структурой что и наш аргумент `tree`, за исключением того, что все чётные числа заменены на символ `even-number`:

```
CL-USER> (tree-leaves%%
          '(1 2 (3 4 (5 6)))
          (lambda (x)
            (and (numberp x) (evenp x)))
          (lambda (x)
```

```
            'even-number))
; in: TREE-LEAVES%% '(1 2 (3 4 (5 6)))
;     (LAMBDA (X) 'EVEN-NUMBER)
; ==>
;   #'(LAMBDA (X) 'EVEN-NUMBER)
;
; caught STYLE-WARNING:
;   The variable X is defined but never used.
;
; compilation unit finished
;   caught 1 STYLE-WARNING condition
(1 EVEN-NUMBER (3 EVEN-NUMBER (5 EVEN-NUMBER)))
```

Вроде бы всё работает за исключением того, что лисп корректно вызывает предупреждение по факту не использования переменной `x` во второй функции — плагине. Если мы не используем переменную, то часто это является признаком наличия проблем в коде. Даже если мы умышленно не используем переменную, как в нашем случае, компилятор выдаст информацию о том, какая переменная будет игнорироваться. Обычно мы используем эту переменную, но есть случаи, подобные нашему, когда нам не нужно использование переменной. Очень плохо если мы получаем аргумент для функции — и после всего этого просто игнорируем этот аргумент. Такая ситуация часто возникает при написании гибких макросов. Решением является передача компилятору информации о том, что переменная `x` может быть проигнорирована. Поскольку нет ничего болезненного в объявлении игнорируемой какую-либо переменную и последующем её использовании[7], то мы можем декларировать информацию о том, что обе переменные `x` должны быть игнорируемыми:

```
CL-USER> (tree-leaves%%
          '(1 2 (3 4 (5 6)))
          (lambda (x)
            (declare (ignorable x))
            (and (numberp x) (evenp x)))
          (lambda (x)
            (declare (ignorable x))
            'even-number))
(1 EVEN-NUMBER (3 EVEN-NUMBER (5 EVEN-NUMBER)))
```

Теперь мы подошли к интересному месту этого урока. Похоже что `tree-leaves%%` прекрасно работает и выполняет то, что нам требуется.

[7] Лисп выяснит что на самом деле не может игнорироваться.

Листинг 5.9: TREE-LEAVES

```
(defmacro tree-leaves (tree test result)
  `(tree-leaves%%
    ,tree
    (lambda (x)
      (declare (ignorable x))
      ,test)
    (lambda (x)
      (declare (ignorable x))
      ,result)))
```

Мы можем изменить любые листья в дереве передав функции-плагины, которые будут выяснять какой лист нужно изменить и на что изменить. На этом месте улучшение утилиты было бы остановлено в языках программирования отличающихся от лиспа. Но не в лиспе. В лиспе мы можем ещё больше улучшить нашу утилиту.

И хотя `tree-leaves%%` предоставляет нам всю требуемую функциональность, его интерфейс остаётся неудобным и избыточным. Чем проще использование утилиты, тем выше вероятность нахождения интересного применения этой функции. С целью уменьшения беспорядка, окружающего нашу утилиту, мы создадим макрос, предоставляющий неявный контекст его пользователям (ну и конечно, нам самим).

Но вместо применения таких простых приёмов как неявный progn или неявную лямбду, мы используем весь неявный лексический контекст, избавляющий нас от накладных расходов при создании этих функций-плагинов и требующий минимального ввода кода при выполнении таких общих задач, как трансляция деревьев. Этот неявный лексический контекст не похож на простые неявности в том смысле, что в нём мы найдём не только другие способы применения общих неявных шаблонов. Вместо этого мы шаг за шагом разработали не-столь-общий шаблон при работе над нашим интерфейсом прохода по коду `tree-leaves%%`.

При конструировании нашего неявного макроса, мы просто скопировали использование `tree-leaves%%` из REPL-а прямо в определение `tree-leaves`, а части, которые должны изменяться в различных применениях макроса были параметризованы при помощи обратных кавычек. Теперь, благодаря этому макросу у нас есть менее многословный интерфейс использования утилиты `tree-leaves%%`. Конечно, этот интерфейс обособлен, поскольку есть множество путей, с помощью которых мы можем добиться подобного поведения. Однако, созданный нами макрос бо-

лее интуитивен и более лёгок в применении, по крайней мере, в наших случаях. Макросы позволили нам создать эффективный программный интерфейс простым и лёгким способом, недоступным в других языках. Вот как мы можем применить наш макрос:

```
CL-USER> (tree-leaves
          '(1 2 (3 4 (5 . 6)))
          (and (numberp x) (evenp x))
          'even-number)
(1 EVEN-NUMBER (3 EVEN-NUMBER (5 . EVEN-NUMBER)))
```

Заметьте, что переменная `x` используется без определения переменной. Причиной этому является *неявная лексическая переменная (implicit lexical variable)*, привязанная к каждому из двух последних переменных. Появление переменной без её объявления называют нарушением *лексической прозрачности (lexical transparency)*. Другими словами можно сказать что *анафора (anaphor)* под названием `x`, введена в эти формы для дальнейшего использования из этих форм. В дальнейшем мы очень, очень сильно разовьём эту идею в *главе 6, Анафорические Макросы.*

5.4 Проход по Коду с Macrolet

> Лисп — это не язык, это строительный материал
>
> — Алан Кэй

Такие редко выговариваемые формы выражений, как компьютерный код, часто получают разнообразные произношения. Большинство программистов выполняют код в голове, размышляя над выражениями и проговаривают операторы чаще неосознанно, а иногда осознанно. Например, наиболее очевидный способ произношения специальной лисп формы `macrolet` — это звуковое объединение двух лисповских компонентов: `macro` и `let`. Но после прочтения Стила [CLTL2-P153] некоторые лисп программисты начинают произносить `macrolet` так, что это слово рифмуется с *Шевроле (Chevrolet)*, такое забавное произношение крайне трудно выкинуть из головы.

В не зависимости от произношения `macrolet` остаётся жизненно важной частью грамотного лисп программирования. `Macrolet` — это специальная форма Common Lisp, которая вводит новые макросы в его закрытую лексическую область видимости. Создание синтаксических трансформаций `macrolet` выполняется в том же стиле, что и определение

глобальных макросов с `defmacro`. Макросы, определённые с помощью `macrolet`, раскрываются лисп системой при *проходе-по-коду (code-walk)* ваших выражений также, как и раскрытие макросов, определённых с помощью `defmacro`.

Но `macrolet` предназначен не только для удобства. У `macrolet` есть свои преимущества перед `defmacro`, используемого для определения макросов. Во-первых, если вы хотите чтобы макрос расширялся по разному в зависимости от лексических контекстов в выражении, то придётся прибегнуть к созданию контекстов с помощью `macrolet`. Здесь `defmacro` попросту не будет работать.

Главное преимущество заключается в том, что с помощью `macrolet` можно решать трудную задачу прохода по коду COMMON LISP выражений. Часто мы получаем какое-либо произвольно взятое дерево лисп кода, например, для его макро-обработки и в этом дереве нам нужно изменить значения переменных или значения различных ветвей дерева. Проход по коду нужен для реализации временных значений некоторых форм и временного переназначения определённых макросов в отдельных частях лексического контекста в выражении. В частности, нам нужно рекурсивно пройти через код, найти место в котором вычисляется нужный макрос или функция и произвести замену на наше выражение.

Просто, не так ли? Сложность заключается в том, что существует множество законных фрагментов лисп кода нарушающих работу наивной реализации прохода-по-коду. Предположим, что нам нужно выполнить замену для вычисления функции в зависимости от какого-либо конкретного символа, например, `blah`. Легко сказать где требуется выполнить замену в следующем выражении:

```
(blah t)
```

В данном списке `blah` находится на месте функции и вычисление этого выражения начнётся с вычисления `blah`. В данном случае весьма очевидно где следует выполнять подстановку. Пока всё хорошо. А что будет если мы передадим вот эту форму:

```
'(blah t)
```

Поскольку выражение закавычено, то этот кусок кода не должен изменяться. В данном случае замена не допустима. Поэтому наша реализация прохода-по-коду должна знать когда ей следует остановиться и не выполнять замены в закавыченных формах. Что же, это достаточно просто реализовать. Задумаемся: существуют ли ещё ситуации, где раскрытие `blah` будет некорректным. Что если нам встретится код, использующий `blah` в качестве имени лексической переменной?

```
(let ((blah t))
  blah)
```

Даже несмотря на то, что `blah` появляется в роли первого элемента списка, он является локальной привязкой для `let` формы, поэтому эта привязка не должна расширяться. Всё не так уж и плохо. Мы можем добавить логику обработки специального случая для нашего прохода-по-коду, благодаря чему он будет знать что делать при встрече `let` формы. К несчастью, у нас по прежнему остаются ещё 23 специальные формы[8] ANSI COMMON LISP которые тоже требуют добавления логики обработки специальных случаев. Усложняет дело ещё то, что многие специальные формы сложны для корректного прохода. Как мы уже видели, `let` может быть хитрым, но могут быть случаи, когда `let` становится ещё более запутанным. Следующая потенциально правильная COMMON LISP форма содержит одно `blah`, которое должно быть раскрыто. Какое именно?

```
(let (blah (blah (blah blah)))
  blah)
```

Проход-по-коду сложен по причине трудностей, возникающих при обработке всех специальных форм (также смотрите [SPECIAL-FORMS] и [USEFUL-LISP-ALGOS2]. Заметьте, что нам не нужна логика обработки специального случая для форм, определённых как макрос. При встрече макроса мы можем его просто расширить до тех пор, пока макрос не перейдёт в вызов функции или в специальную форму. Если это функция, то мы знаем что она следует `once-only` лямбды, семантике вычисления слева-на-право. Это специальные формы, для которых нам нужно разработать логику обработки.

Похоже предстоит выполнить много работы, не так ли? Так и есть. Завершённый проход-по-коду COMMON LISP, особенно с расчётом на переносимость — это большой, сложный код. Так почему же COMMON LISP не предоставляет нам интерфейс для прохода-по-коду? Что же, оказывается, нечто подобное у нас уже есть и называется `macrolet`. Проход-по-коду — это то, что нужно выполнить вашей COMMON LISP системой перед тем как вычислить или скомпилировать выражение. Также, как и наш гипотетический проход-по-коду COMMON LISP-у нужно понимать и обрабатывать специальные семантики `let` и других специальных форм.

[8] В ANSI CL есть 25 специальных форм, 23 без учёта `let` и `quote`.

Поскольку COMMON LISP может проходить по коду для его вычисления, то необходимость в отдельной программе прохода-по-коду возникает очень редко. Если мы хотим выполнить избирательные трансформации в выражения таким умным способом, который будет в состоянии определить что именно следует изменять, то мы можем просто зашифровать нашу трансформацию в виде макроса и обернуть выражение `macrolet` формой. COMMON LISP осуществит проход-по-коду для этого выражения, и при вычислении или компиляции будет применять макро трансформации, определённые `macrolet`-ом. И конечно, поскольку `macrolet` определяет макросы, то не будут возникать никакие дополнительные затраты во время выполнения. `Macrolet` предназначен для коммуникации с COMMON LISP-овским проходом-по-коду и гарантирует только то, что COMMON LISP произведёт раскрытие макроса до того как будет запущена скомпилированная функция [CLTL2-P685][ON-LISP-P25].

Существует один из наиболее распространённых сценариев использования `macrolet`: считать функцию привязанной к определённому лексическому контексту и использовать форму функции не только для вызова функции. Мы не рассматриваем `flet` и `labels` — они могут только определять функции. Таким образом у нас остаются несколько вариантов: написать программу для прохода-по-коду, находящую вызовы этой функции и заменяющую их на что-нибудь ещё, определение глобального макроса, предназначенного для расширения "функции"во что-нибудь ещё или оборачивание формы `macrolet`-ом, используя системный проход-по-коду для наших целей.

Как было замечено выше: написание прохода-по-коду — трудная задача. По возможности мы должны избегать этот способ. Использование глобального `defmacro` временами возможно, но часто довольно проблематично. Главная проблема заключается в том, что в COMMON LISP существуют несколько правил определяющие когда — или как часто — будет расширяться макрос, поэтому мы не можем полагаться на одно и то же имя в различных значениях и в различных лексических контекстах. Если мы переопределим глобальный макрос, то в этом случае мы ничего не знаем о факте расширения COMMON LISP-ом — или о возможном расширении в будущем — старого варианта макроса.

Мы вернёмся к *разделу Управляющие Структуры* и в качестве примера, показывающего пользу от применение прохода-по-коду, мы обратимся к положенной под сукно проблеме. Наша первоначальная версия макроса, реализующего Scheme-овский именованный `let`, `nlet`, использовал специальную форму `labels` для создания новых типов управляющих структур. Использование `labels` позволяет нам временно опреде-

Листинг 5.10: NLET-TAIL

```
(defmacro! nlet-tail (n letargs &rest body)
  (let ((gs (loop for i in letargs
                  collect (gensym))))
    `(macrolet
         ((,n ,gs
            `(progn
               (psetq
                ,@(apply #'nconc
                         (mapcar
                          #'list
                          ',(mapcar #'car letargs)
                          (list ,@gs))))
               (go ,',g!n))))
       (block ,g!b
         (let ,letargs
           (tagbody
              ,g!n (return-from
                    ,g!b (progn ,@body)))))))))
```

лить функцию для внутреннего применения в теле именованного `let`, что позволяет нам применить рекурсию, и мы как-будто запускаем `let` заново с новыми значениями для `let` привязок. Когда мы определяли эту функцию, мы упомянули, что COMMON LISP не гарантирует осуществление оптимизации хвостовых вызовов, это может привести к тому, что каждая итерация именованного `let` может использовать избыточное стековое пространство. Другими словами, в отличие от Scheme, вызов функций в COMMON LISP не гарантируют *оптимизацию хвостовых вызовов*.

И хотя большинство приличных COMMON LISP компиляторов будут выполнять соответствующую оптимизацию хвостовых вызовов нам, иногда, нужно быть уверенными в том что оптимизация будет выполнена. Самый простой и переносимый способ добиться этого — изменить макрос `nlet` таким образом, чтобы генерируемое им расширение потребляло только необходимое стековое пространство.

В `nlet-tail` мы окружили переданное тело макросом и завернули его в несколько форм. Для возвращения значения финального выражения мы использовали операторы `block` и `return-from`, поскольку мы хотим имитировать поведение формы `let` и связанный с этой формой неявный

`progn`. Заметьте, для того, чтобы избежать нежелательный захват мы используем `gensym` для имени этого блока и `gensym` для каждого параметра `let`, а макрос `loop`[9] задействован для сбора этих `gensym`-ов.

`Nlet-tail` используется также, как и исходный `nlet`, за исключением того, что здесь запрещены вызовы именованного `let` в не-хвостовой позиции поскольку они будут расширены в хвостовые вызовы. Ниже представлен всё тот-же прозаический пример, который мы использовали при демонстрации `nlet`, но этот пример гарантированно не будет потреблять избыточного стекового пространства даже если лисп не будет поддерживать оптимизацию хвостовых вызовов:

```
(defun nlet-tail-fact (n)
  (nlet-tail fact ((n n) (acc 1))
             (if (zerop n)
                 acc
                 (fact (- n 1) (* acc n)))))
```

Рассматривая мотивирующий пример из этой секции, заметьте, для того, чтобы найти `fact` мы используем `macrolet` для прохода по коду. Мы хотели бы чтобы наш исходный `nlet`, использующий специальную форму `label` для привязки функции, работал без потребления дополнительного стекового пространства при вызове именованного `let`. Технически мы бы хотели изменить некоторые привязки в нашем лексическом окружении и выполнить прыжок назад в верх именованного `let`. Таким образом `nlet-tail` получает переданное `let` имя, в нашем примере `fact`, и создаёт *локальный макрос (local macro)*, являющийся актуальным только внутри переданного тела. Этот макрос раскрывается в код, который использует `psetq` для создания `let` привязок к новым предоставленным значениям и затем прыгает обратно в верх, не требуя стекового пространства. И что наиболее важно, мы можем использовать имя `fact` для других макросов в нашей программе[10].

Для реализации таких прыжков `nlet-tail` использует комбинацию специальных форм `tagbody` и `go`. Эти две формы могут реализовывать *goto систему*. И хотя проблемы, связанные с goto, широко обсуждаются в *структурном программировании (structured programming)*, COMMON LISP предоставляет эти специальные формы по причинам, с которыми мы уже столкнулись. Контролируя *программный счётчик (program*

[9] `Loop` — это, как ни странно, наиболее спорный вопрос в COMMON LISP. Однако, большинство обвинений выдвигаемых против `loop` беспочвенны. `Loop` — это очень удобный предметно-ориентированный язык для работы с циклами.

[10] Какая книга о программировании не содержит так или иначе несколько реализаций факториала?

counter) — исполняемый в данный момент участок кода — мы можем создавать очень эффективные макро расширения. В то время когда goto не приветствуется в языках *высокого (high-level) уровня*, один взгляд на любой код на ассемблере покажет что goto очень даже жив и здоров в низкоуровневом программном обеспечении. Даже самые непримиримые сторонники анти-goto не предлагают избавлять такие *низкоуровневые языки* как C и ассемблер от инструкций jump и goto. Оказывается, что в низкоуровневом программировании мы всё же испытываем некую потребность в goto, хотя бы для написания эффективного кода.

Однако, как сказал Алан Кэй, лисп — это не язык, а строительный материал. Обсуждения о том является ли лисп высокоуровневым или низкоуровневым языком не имеют смысла. Существуют очень высокоуровневые диалекты лиспа, такие как наши предметно-ориентированные языки. С помощью макросов мы обрабатываем эти языки и конвертируем их в более низкоуровневой лисп. Конечно, эти расширения тоже являются лисп кодом, они просто не сжаты как их исходные версии. После этого мы обычно передаём этот лисп код среднего уровня в компилятор, который последовательно преобразует его в ещё более низкие уровни лиспа. На этом уровне мы приближаемся к концепциям goto, условных ветвлений и операций с битами, но по прежнему это лисп. В конце концов, компилятор кода сконвертирует ваш высокоуровневую лисп программу в язык ассемблера. Но, даже здесь есть шанс что ваша программа по прежнему останется лиспом. Поскольку большинство лисп ассемблеров сами написаны на лиспе, то вполне естественно хранить эти ассемблерные программы в виде лисп объектов, что приводит к тому, что низкоуровневые программы на деле являются лиспом. Программа только тогда перестанет быть лиспом, когда она будет собрана исключительно с помощью машинных кодов. Не так ли?

Понятия высокого и низкого уровня не применимы к лиспу; уровень лисп программы — это вопрос перспективы. Лисп — это не язык, а самый лучший строительный материал из когда-либо открытых, предназначенный для создания наиболее гибких программ.

5.5 Рекурсивные Расширения

При обучении лиспу, к примеру, новичков, неизбежно возникает один вопрос

> `Cadr` — что это за хрень?

Есть два способа объяснить смысл `cadr`. Первый — это объяснить студентам что списки лиспа строятся из `cons` ячеек, каждая из которых состоит из двух указателей: указателя с названием `car` и указателя с названием `cdr`. После усвоения этой концепции, можно легко продемонстрировать каким образом можно скомбинировать функции получения доступа к этим указателям, также названные как `car` и `cdr`, в функцию под названием `cadr`, получающую второй элемент из списка.

Второй подход заключается в ознакомлении студентов с функцией COMMON LISP `second` и в полном игнорировании `cadr`. И `cadr` и `second` выполняют одну и ту же задачу: возвращают второй элемент списка. Разница в том, что название `second` выражает выполняемую операцию, а `cadr` выражает то, как выполняется операция. `Cadr` *определён прозрачно (transparently specified)*. И хотя `second` прост для запоминания, эта функция нежелательным образом скрывает смысл операции[11]. Часто прозрачные определения являются более лучшим выбором, поскольку мы более глубже думаем об использовании функции `cadr`, чем просто о получении второго элемента списка. Например, мы можем прозрачно думать об использовании `cadr` как о концепции получения аргумента из разрушаемого списка лямбда формы. `Cadr` и `second` решают одну и ту же задачу, но концептуально могут представлять различные операции [CLTL2-P530].

Ещё более важно чем философские предпочтения прозрачных спецификаций, это возможность создания большого количества операций для получения элементов списка с помощью `car` и `cdr`, в отличие от получения доступа с помощью кучки английских слов. `Car` и `cdr` удобны тем, что комбинируя их мы создаём новые, независимые функции. Например, `(cadadr x)` — это тоже, что и `(car (cdr (car (cdr x))))`. COMMON LISP определяет, что длина всей комбинации `car` и `cdr` должна быть меньшей либо равной четырём. И хотя функция `second-of-second` получающая второй элемент списка и рассматривающая этот элемент как список и извлекающая из этого списка второй элемент не существует, мы всё же можем использовать для этих целей `cadadr`.

Особенно удобно иметь эти предварительно определённые комбинации `car` и `cdr` доступными для функций, получающих аргумент доступа `:key`, в качестве такой функции можно привести `find`:

```
CL-USER> (find 'a
               '(((a b) (c d)) ((c d) (b a)))
```

[11] Частично поскольку `second` — это то же что и `cadr`: вы не сможете использовать его для получения второго элемента из таких последовательностей, как, например, векторы.

```
               :key #'cadadr)
((C D) (B A))
```

Использование предварительно определённого `cadadr` является более коротким чем создание эквивалентного лямбда выражения из комбинаций англоязычных операторов получения доступа:

```
CL-USER> (find 'a
               '(((a b) (c d)) ((c d) (b a)))
               :key (lambda (e)
                      (second (second e))))
((C D) (B A))
```

Кроме того COMMON LISP предоставляет функции `nth` и `nthcdr` которые могут использоваться как универсальные получатели если, например, мы не не знаем какой именно элемент понадобиться нам во время компиляции. `Nth` определён очень просто: получить `n`-ый `cdr` из списка и после этого применить `car`. Таким образом `(nth 2 list)`[12] это то же самое что и `(caddr list)` и `(third list)`. `Nthcdr` — это то же самое что и `nth` за тем исключением что финальный `car` не выполняется: `(nthcdr 2 list)` — это то же что и `(cddr list)`.

Но, если место в cons структуре не доступно из всех вышеприведённых шаблонов, таких как `nth` или `nthcdr`, то нам придётся их комбинировать. Частое комбинирование неоднородных абстракций является признаком незавершённости. Можем ли мы определить предметно-ориентированный язык решающий задачу получения участков списка и объединяющий в себе функции `car` и `cdr`, англоязычные функции доступа и такие функции как `nth` и `nthcdr`?

Поскольку `car` и `cdr` — фундаментальные операторы, наш язык должен включать в себя комбинирование этих двух методов доступа самым универсальным способом. Поскольку число возможных комбинаций `car` и `cdr` бесконечно, то определение функций для каждого возможного метода доступа просто невозможно. Что нам нужно на самом деле — так это единственный макрос, который может расширяться в эффективный код по работе со списком.

Синтаксис именования функции доступа к элементу списка в котором название начинается с `C`, с последующими одним или более символами `A`

[12] Мы можем без каких-либо проблем использовать `list` в роли имени переменной поскольку в COMMON LISP-е есть второе пространство имени. Подобный пример было бы проблемно использовать в диалектах с единым пространством имён, например Scheme.

Листинг 5.11: CXR-1

```
(defmacro cxr% (x tree)
  (if (null x)
      tree
      `(,(cond
          ((eq 'a (cadr x)) 'car)
          ((eq 'd (cadr x)) 'cdr)
          (t (error "Non A/D symbol")))
        ,(if (= 1 (car x))
             `(cxr% ,(cddr x) ,tree)
             `(cxr% ,(cons (- (car x) 1) (cdr x))
                    ,tree)))))
```

или D и заканчивающееся на R является очень интуитивным и примерно соответствует тому, что мы хотели бы скопировать в наш язык. Макрос cxr% — каламбур, обозначающий что один или более символов A или D замещены символом x[13]. В cxr% символы A и D определены в списке переданному в качестве первого аргумента макроса. В этом списке задаётся комбинация символов A и/или D и их количество.

Например, в COMMON LISP нет англоязычной функции для доступа к одиннадцатому элементу списка, но мы легко можем определить эту функцию:

```
(defun eleventh (x)
  (cxr% (1 a 10 d) x))
```

Цель этого раздела — проиллюстрировать жизненное использование *рекурсивных расширений (recursive expansions)*. Рекурсивное расширение происходит тогда, когда макрос расширяет форму в новую форму, которая в свою очередь содержит использование макроса. Как и все рекурсии этот процесс должен завершиться при достижении *базового случая (base case)*. К счастью в конце концов макрос раскроется в форму, которая не будет содержать использования макроса и раскрытие будет завершено.

Здесь мы применяем macroexpand к примеру с макросом cxr% и получаем форму, которая также использует cxr%:

```
CL-USER> (macroexpand
```

[13]За исключением названия cxr никак не связан с cxr в Maclisp-е. В Maclisp cxr используется для получения доступа к hunk слотам.

```
          '(cxr% (1 a 2 d) some-list))
(CAR (CXR% (2 D) SOME-LIST))
T
```

После применения `macroexpand` к новой рекурсивной форме мы получим ещё одну рекурсию:

```
CL-USER> (macroexpand
          '(cxr% (2 d) some-list))
(CDR (CXR% (1 D) SOME-LIST))
T
```

Результаты следующей рекурсии иллюстрируют другое возможное использование `cxr%`: нулевой список операторов получения доступа[14]:

```
CL-USER> (macroexpand
          '(cxr% (1 d) some-list))
(CDR (CXR% NIL SOME-LIST))
T
```

Оператор получения доступа в виде пустого списка — это наш базовый случай и он расширяется непосредственно в список над которым мы производим наши операции:

```
CL-USER> (macroexpand
          '(cxr% nil some-list))
SOME-LIST
T
```

Используя расширение CMUCL-а `macroexpand-all`, компонент проходчика-по-коду, мы можем увидеть всё расширение нашей исходной `cxr%` формы:

```
* (walker:macroexpand-all
   '(cxr% (1 a 2 d) some-list))
(CAR (CDR (CDR SOME-LIST)))
```

[14]Если когда нибудь этот макрос будет включён в COMMON LISP, то его название должно быть `cr`.

Благодаря нашим замечательным лисп компиляторам, использование `cxr%` будет идентично функциям `caddr` и `third`.

Но, как следует из названия `cxr%` не завершён. Это всего лишь первый набросок нашего завершённого макроса `cxr`. Первая проблема нашего наброска в том, что в качестве счётчика для символов `A` и `D` принимаются только целые числа. Поэтому `nth` и `nthcdr` могут делать то, что не может `cxr%`.

Нам нужно добавить проверку для случая, когда в качестве префикса для символов `A` и `D` передаётся не-целое число. В этом случае наш расширяющийся код должен вычислить то, что ему передано и использовать это значение[15] в качестве количества `car`-ов или `cdr`-ов.

Вторая проблема связанная с `cxr%` заключается во встраивании всех комбинаций `car` и `cdr`. При малых префиксных числах для символов `A` и `D` результат *встраивания* почти не заметен, но при огромных количествах `car` и `cdr` всё становится заметным на глаз; вместо этого мы должны использовать такие циклирующие функции как `nth` или `nthcdr`.

Для исправления этих случаев мы добавим альтернативное расширение. Новое поведение будет применяться в тех случаях, если параметр, предшествующий символам `A` и/или `D`, не является целочисленным и если мы не встраиваем большое количество `car`-ов или `cdr`-ов. *Порог встраивания (inline threshold)* по которому будет выбираться поведение установлен в 10, а новое поведение мы реализуем в макросе `cxr`.

С `cxr` мы можем определить `nthcdr` непосредственно в терминах его прозрачной `car` и `cdr` спецификации:

```
(defun nthcdr% (n list)
  (cxr (n d) list))
```

И, аналогично, `nth`:

```
(defun nth% (n list)
  (cxr (1 a n d) list))
```

Поскольку макросы пишутся в итеративном, послойном процессе, то часто у нас появляется возможность *комбинировать* или *сочетать* ранее созданные макросы. Например, в определении `cxr` наше альтернативное расширение использует макрос `nlet-tail` ранее созданный в предыдущем разделе. `Nlet-tail` удобен тем, что позволяет нам задавать имена

[15] К счастью, это значение должно быть числом. В лиспе мы можем безопасно оставить обработку данной ситуации на совесть лисповской системы исключений, эта система будет сама обрабатывать и выдавать исключения программисту.

Листинг 5.12: CXR

```
(defvar cxr-inline-thresh 10)

(defmacro! cxr (x tree)
  (if (null x)
      tree
      (let ((op (cond
                  ((eq 'a (cadr x)) 'car)
                  ((eq 'd (cadr x)) 'cdr)
                  (t (error "Non A/D symbol")))))
        (if (and (integerp (car x))
                 (<= 1 (car x) cxr-inline-thresh))
            (if (= 1 (car x))
                `(,op (cxr ,(cddr x) ,tree))
                `(,op (cxr ,(cons (- (car x) 1) (cdr x))
                           ,tree)))
            `(nlet-tail
              ,g!name ((,g!count ,(car x))
                       (,g!val (cxr ,(cddr x) ,tree)))
              (if (>= 0 ,g!count)
                  ,g!val
                  ;; Будет хвостом:
                  (,g!name (- ,g!count 1)
                           (,op ,g!val)))))))))
```

итерационным конструкциям, а поскольку мы планируем итерацию с помощью хвостовых вызовов, то мы с уверенностью можем использовать эту итерацию для того, чтобы избежать избыточного потрбеления стека.

Вот как расширяется `cxr` в `nthcdr%`:

```
CL-USER> (macroexpand
          '(cxr (n d) list))
(LET ()
     (NLET-TAIL #:NAME1271 ((#:COUNT1272 N) (#:VAL1273
                                             (CXR NIL LIST)))
                (IF (>= 0 #:COUNT1272)
                    #:VAL1273
                    (#:NAME1271 (- #:COUNT1272 1)
                                (CDR #:VAL1273)))))
T
```

Учтите что расширения сложного макроса часто создают код, который никогда не напишет человек. Особенно обратите внимание на использование `cxr`-ами `nil`-а и применение бесполезного `let`, оба эти элемента оставлены для дальнейших макрорасширений и последующей оптимизации компилятором.

Макросы могут создавать гибкие расширения видимые пользователем макроса, поэтому нам доступны такие способы прозрачных спецификаций, которые не возможны в других языках. Например, из архитектуры `cxr` следует что параметры предшествующие символам `A` и `D`, являющиеся целочисленными и меньшими чем `cxr-inline-thresh` будут встроены как вызовы `car` и `cdr`:

```
CL-USER> (macroexpand '(cxr (9 d) list))
(LET ()
  (CDR (CXR (8 D) LIST)))
T
```

Но, благодаря прозрачной спецификации `cxr` мы можем передавать значения не являющиеся целочисленными, но, после вычисления преобразующиеся в целочисленное значение. При выполнении этой операции мы знаем что в данном случае не происходит никаких встраиваний, поскольку результатом макроса является `nlet-tail` расширение. Простейшая форма, которая вычисляется в целочисленное значение — это закавыченное целочисленное значение:

```
CL-USER> (macroexpand '(cxr ('9 d) list))
(LET ()
  (NLET-TAIL #:NAME1277 ((#:COUNT1278 '9) (#:VAL1279
                                           (CXR NIL LIST)))
             (IF (>= 0 #:COUNT1278)
                 #:VAL1279
                 (#:NAME1277 (- #:COUNT1278 1)
                             (CDR #:VAL1279)))))
T
```

Часто мы можем обнаружить что комбинирование макросов является весьма удобным приёмом: `cxr` может расширяться в ранее написанный макрос `nlet-tail`. Аналогично иногда оказывается полезным комбинировать макрос с самим собой, тем самым прибегая к рекурсивному расширению.

5.6 Рекурсивные Решения

Кажется что макрос `cxr`, определённый в предыдущем разделе, можно отнести к комбинированию функций `car` и `cdr`, также, как и общие функции получения доступа к элементу плоского списка `nth` и `nthcdr`. Ну а что-же насчёт англоязычных методов получения доступа, таких как `first`, `second` и `tenth`? Эти функции бесполезны? Определённо нет. Операция получения четвёртого элемента списка выраженная через `fourth` определённо лучше вычисления трёх `D` в `cadddr` как в плане написания, так и в плане чтения.

На деле самая большая проблема с англоязычными методами доступа к элементам списка в том, что в COMMON LISP-е их только 10, начиная с `first` и заканчивая `tenth`. Но, одна из тем этого раздела, да и в целом всей книги, в том, что каждый слой лисп луковицы может использовать нижележащий слой. В лиспе нет примитивов. Если мы хотим определить больше англоязычных методов доступа, например `eleventh`, то мы легко можем реализовать это как показано выше. Функция `eleventh`, определённая с помощью `defun` не будет отличаться от таких методов доступа определённых в ANSI, как например `first` и `tenth`. Поскольку примитивов не существует и мы можем использовать весь лисп в наших определениях макросов, то в наших макросах нам доступна вся мощь лиспа, например `loop` и `format`[16].

[16] `Format` — это несколько спорная особенность COMMON LISP. Однако неприятие `format`, как и обвинения выдвигаемые против `loop`, основано на не полном понимании

Листинг 5.13: DEF-ENGLISH-LIST-ACCESSORS

```
(defmacro def-english-list-accessors (start end)
  (if (not (<= 1 start end))
      (error "Bad start/end range"))
  `(progn
     ,@(loop for i from start to end collect
             `(defun
                  ,(symb
                    (map 'string
                         (lambda (c)
                           (if (alpha-char-p c)
                               (char-upcase c)
                               #\-))
                         (format nil "~:r" i)))
                (arg)
              (cxr (1 a ,(- i 1) d) arg)))))
```

Макрос `def-english-list-accessors` использует `"~:r"` в качестве формата строки для конвертирования числа, `i`, в строку, содержащую соответствующее английское слово. Как принято в лиспе, мы сменили все не-алфавитные символы на дефисы. Затем мы конвертировали эту строку в символ и использовали её в форме `defun`, которая реализует соответствующую функциональность получения элемента списка с помощью макроса `cxr`.

Например, предположим что нам внезапно понадобилось получить доступ к одиннадцатому элементу списка. Мы можем использовать `nth` или комбинацию `cdr` и англоязычных операторов получения элемента списка, но получившийся результат будет написан в неустойчивом стиле программирования. Мы можем переписать наш код и избежать использования англоязычных операторов, но есть вероятность что причина по которой мы выбрали эту абстракцию по прежнему будет оставаться на первом месте.

И наконец, мы можем самостоятельно определить необходимые отсутствующие операторы. Обычно в других языках это обозначает множество операций копирования-вставки или могут быть некоторые кодогенерирующие сценарии, обрабатывающие некоторые специальные случаи — в любом случае оба варианта не элегантны. Но в лиспе у нас есть

концепции и области использования предметно-ориентированных языков.

Листинг 5.14: CXR-CALCULATOR

```
(defun cxr-calculator (n)
  (loop for i from 1 to n
     sum (expt 2 i)))
```

макросы:

```
CL-USER> (macroexpand '(def-english-list-accessors 11 20))
(PROGN
 (DEFUN ELEVENTH (ARG) (CXR (1 A 10 D) ARG))
 (DEFUN TWELFTH (ARG) (CXR (1 A 11 D) ARG))
 (DEFUN THIRTEENTH (ARG) (CXR (1 A 12 D) ARG))
 (DEFUN FOURTEENTH (ARG) (CXR (1 A 13 D) ARG))
 (DEFUN FIFTEENTH (ARG) (CXR (1 A 14 D) ARG))
 (DEFUN SIXTEENTH (ARG) (CXR (1 A 15 D) ARG))
 (DEFUN SEVENTEENTH (ARG) (CXR (1 A 16 D) ARG))
 (DEFUN EIGHTEENTH (ARG) (CXR (1 A 17 D) ARG))
 (DEFUN NINETEENTH (ARG) (CXR (1 A 18 D) ARG))
 (DEFUN TWENTIETH (ARG) (CXR (1 A 19 D) ARG)))
T
```

Возможность создания этих англоязычных операторов снимает ограничение в десять операторов в ANSI COMMON LISP. Если нам понадобится больше англоязычных операторов доступа, то мы можем создать их с помощью макроса `def-english-list-accessors`.

Ограничение ANSI накладываемое на глубину комбинации `car` и `cdr` равняется четырём. Что можно с этим сделать? Иногда при создании программ обрабатывающих сложные списки, мы можем захотеть изменить функции, получающие доступ к элементу списка и определить их по другому. Например, предположим, что мы используем функцию `cadadr`, `second-of-second`, для получения доступа к списку и мы изменили наше отображение данных и теперь нам нужно использовать `second-of-third`, или `cadaddr`, то здесь мы столкнёмся с ограничениями, налагаемыми COMMON LISP-ом.

Также, как и при работе с англоязычными операторами получения доступа, мы можем написать программу, определяющую дополнительные комбинации `car` и `cdr`. Проблема в том, что в отличие от англоязычных операторов получения доступа, увеличение глубины комбинаций в такой функции как `caddr` приводит к экспоненциальному увеличению

Листинг 5.15: CXR-SYMBOL-P

```
(defun cxr-symbol-p (s)
  (if (symbolp s)
      (let ((chars (coerce
                    (symbol-name s)
                    'list)))
        (and
         (< 6 (length chars))
         (char= #\C (car chars))
         (char= #\R (car (last chars)))
         (null (remove-if
                (lambda (c)
                 (or (char= c #\A)
                     (char= c #\D)))
                (cdr (butlast chars)))))))))
```

числа функций, которые должны быть определены. В частности, число операторов получения доступа, которые должны быть определены для покрытия глубины равной n можно определить с помощью функции `cxr-calculator`.

Мы видим что ANSI определяет 30 комбинаций:

```
CL-USER> (cxr-calculator 4)
30
```

А теперь покажем вам насколько быстро растёт число покрывающих функций:

```
CL-USER> (loop for i from 1 to 16
               collect (cxr-calculator i))
(2 6 14 30 62 126 254 510 1022 2046 4094 8190 16382 32766 65534
   131070)
```

Очевидно, что для покрытия всех комбинаций `car` и `cdr` с помощью `cxr` функций нужен подход, отличающийся от того, что мы применили при решении проблемы с англоязычными операторами. Определение всех комбинаций `car` и `cdr` является недопустимым.

Для начала нам нужно разработать точную спецификацию `cxr` символа. `Cxr-symbol-p` можно описать очень просто: все символы, начинающиеся с `C` и заканчивающиеся на `R`, а между ними 5 и более символов

A и/или D. Нам не нужно рассматривать cxr символы в которых количество A и/или D меньше 5, поскольку эти функции гарантированно определены в COMMON LISP[17].

Поскольку мы планируем использовать cxr для реализации функциональности независимых комбинаций car и cdr, то мы создали функцию cxr-symbol-to-cxr-list конвертирующую cxr символ, как определено в cxr-symbol-p в список, который можно использовать в качестве первого аргумента cxr[18]. Пример его использования:

```
CL-USER> (cxr-symbol-to-cxr-list
          'caddadr)
(1 A 1 D 1 D 1 A 1 D)
```

Заметьте, использование функции list* в cxr-symbol-to-cxr-list. List* — это почти тот же list, за исключением того, что его последний аргумент вставлен в cdr позицию последней cons ячейки в созданном списке. List* удобно применять при написании рекурсивных функций, строящих список, и где каждый фрейм стека может добавлять более одного элемента в список. В нашем случае каждый фрейм будет добавлять два элемента в список: число 1 и один из символов A или D.

Мы решили что единственный способ эффективного предоставления cxr функций с произвольной глубиной — это проход по коду, предоставляющий выражения и определяющий только необходимые функции. Макрос with-all-cxrs использует утилиту Грэма flatten для прохода по коду в переданных выражениях таким же способом, который был использован в макросе defmacro/g! из *раздела Нежелательный Захват*. With-all-cxrs производит поиск всех символов удовлетворяющих предикату cxr-symbol-p, создаёт функции, относящиеся к использованию макроса cxr, и привязывает эти функции вокруг предоставленного кода с помощью формы labels[19].

Теперь мы можем заключить выражения в формах, переданных в with-all-cxrs и считать что эти выражения имеют доступ ко всем возможным cxr функциям. При желании мы можем просто вернуть эти функции и использовать их где угодно:

```
CL-USER> (with-all-cxrs #'cadadadadadr)
#<FUNCTION (LABELS CADADADADADR) {CFCDA45}>
```

[17]Повторная привязка функций, определённых в COMMON LISP запрещена.

[18]Забавно, но нам здесь пригодилась бы устаревшая функция explode, но она не вошла в COMMON LISP, поскольку никто не смог придумать полезное применение этой функции.

[19]Единственная проблема, возникающая при работе с этим подходом, заключается в том, что такие методы получения доступа не setf-абельны.

Листинг 5.16: CXR-SYMBOL-TO-CXR-LIST

```
(defun cxr-symbol-to-cxr-list (s)
  (labels ((collect (l)
             (if l
                 (list*
                  1
                  (if (char= (car l) #\A)
                      'A
                      'D)
                  (collect (cdr l))))))
    (collect
        (cdr ; отсечь C
         (butlast ; отсечь R
          (coerce
           (symbol-name s)
           'list))))))
```

Листинг 5.17: WITH-ALL-CXRS

```
(defmacro with-all-cxrs (&rest forms)
  `(labels
       (,@(mapcar
           (lambda (s)
             `(,s (l)
                  (cxr ,(cxr-symbol-to-cxr-list s)
                       l)))
           (remove-duplicates
            (remove-if-not
             #'cxr-symbol-p
             (flatten forms)))))
     ,@forms))
```

Или, как показано в следующем расширении макроса, мы можем встраивать произвольно-сложный лисп код, использующий этот бесконечный класс:

```
CL-USER> (macroexpand
          '(with-all-cxrs
            (cons
             (cadadadr list)
             (caaaaaaaar list))))
(LABELS ((CADADADR (L)
           (CXR (1 A 1 D 1 A 1 D 1 A 1 D) L))
         (CAAAAAAAAR (L)
           (CXR (1 A 1 A 1 A 1 A 1 A 1 A 1 A 1 A) L)))
  (CONS (CADADADR LIST) (CAAAAAAAAR LIST)))
T
```

Часто сложно выглядящая задача — такая как определение бесконечных классов англоязычных операторов доступа к элементу списка и комбинации `car-cdr` — на самом деле просто набор из более простых задач. В отличие от единой задачи, которая может быть сложной, наборы простых проблем могут быть решены через рекурсивный подход. Думая о способах преобразования сложной задачи в набор из более простых задач, мы используем проверенный подход к решению проблем: *"разделяй и властвуй" ("divide and conquer").*

5.7 Dlambda

В нашей дискуссии о замыканиях мы упомянули как замыкание может быть использовано в виде объекта и как, в целом, неопределённое пространство и лексическая область видимости может заместить усложнённые объектные системы. Но, есть одна особенность, которую предоставляют объекты и которую мы до сих пор игнорировали — это множественные *методы (methods)*. Другими словами, в то время, пока наш простой пример счётчика, реализованного на замыканиях, позволял выполнять только одну операцию, инкрементирование, объекты, обычно, в состоянии отвечать на различные *сообщения (messages)* различным поведением.

Хотя замыкания можно рассматривать как объект с одним — единственным методом — `apply` — но, этот метод можно спроектировать так, что он будет обладать различным поведением в зависимости от переданных ему аргументов. Например, если мы обозначим первый аргумент

символом, обозначающим переданное сообщение, то мы можем реализовать множественные поведения с помощью простого оператора `case`, взаимодействующим с первым аргументом.

Для реализации счётчика с методом инкремента и методом декремента мы должны использовать следующее:

```
(let ((count 0))
  (lambda (msg)
    (case msg
      ((:inc)
       (incf count))
      ((:dec)
       (decf count)))))
```

Заметьте, что для обозначения сообщений мы выбрали *символы ключевых слов (keyword symbols)*, это символы, начинаются с `:` и всегда вычисляются в самих себя. Ключевые слова удобны, поскольку нам не надо их закавычивать или экспортировать их из других пакетов, а ещё они интуитивны, поскольку они спроектированы для выполнения этих и других видов *деструктуризации (destructuring)*. Часто в `lambda` или `defmacro` формах ключевые слова не разрушаются во время выполнения. Но, поскольку мы реализуем систему передачи сообщений, которая является одним из видов *деструктуризации времени выполнения*, то операция обработки ключевых слов будет выполняться во время выполнения. Как мы уже обсуждали, деструктуризация символов — это эффективная операция: простое сравнение указателей. При компиляции нашего примера счётчика он может преобразоваться в следующий машинный код:

```
2FC:      MOV     EAX, [#x582701E4]      ; :INC
302:      CMP     [EBP-12], EAX
305:      JEQ     L3
307:      MOV     EAX, [#x582701E8]      ; :DEC
30D:      CMP     [EBP-12], EAX
310:      JEQ     L2
```

Но, для того, чтобы создать удобные условия нам нужно избежать необходимости писать `case` оператор для каждого создаваемого объекта или класса. Похоже что в этой ситуации нужно прибегнуть к макросам. Макрос, который я хочу использовать, называется `dlambda`, и этот макрос расширяется в `lambda` форму. Это расширение включает в себя код

Листинг 5.18: DLAMBDA

```
(defmacro! dlambda (&rest ds)
  `(lambda (&rest ,g!args)
     (case (car ,g!args)
       ,@(mapcar
          (lambda (d)
            `(,(if (eq t (car d))
                   t
                   (list (car d)))
              (apply (lambda ,@(cdr d))
                     ,(if (eq t (car d))
                          g!args
                          `(cdr ,g!args)))))
          ds))))
```

с большим количеством ветвлений, которые выполняются в зависимости от применённых аргументов. Название `dlambda` произошло из данной разновидности деструктуризации во время исполнения: это версия `lambda` с поддержкой *деструктуризации* или, выражаясь по-другому, *диспетчеризации*.

`Dlambda` спроектирована так, чтобы получать в качестве первого аргумента символ ключевого слова. В зависимости от использованного символа ключевого слова `dlambda` будет выполнять соответствующий кусок кода. Например, наш любимый пример замыканий — простой счётчик — может быть расширен так, что инкрементирование и декрементирование счётчика будет происходить на основе первого элемента с использованием `dlambda`. Этот шаблон известен как *let, окружающий dlambda (let over dlambda)*:

```
CL-USER> (setf (symbol-function 'count-test)
               (let ((count 0))
                 (dlambda
                  (:inc () (incf count))
                  (:dec () (decf count)))))
#<CLOSURE (LAMBDA (&REST #:ARGS0)) {D270B9D}>
```

Мы можем инкрементировать

```
CL-USER> (count-test :inc)
1
```

и декрементировать

```
CL-USER> (count-test :dec)
0
```

замыкание зависит от первого переданного аргумента. И хотя в вышеприведённом счётчике `let`, окружающий `dlambda`, оставлен пустым, списки следующие за символами ключевых слов на самом деле являются списками *лямбда деструктуризации (lambda destructuring)*. Каждый диспетчеризующий вариант, или другими словами каждый аргумент-ключевое-слово, могут обладать собственным лямбда деструктирующим списком, как это показано в следующем усовершенствовании замыкания — счётчика:

```
CL-USER> (setf (symbol-function 'count-test)
               (let ((count 0))
                 (dlambda
                  (:reset () (setf count 0))
                  (:inc (n) (incf count n))
                  (:dec (n) (decf count n))
                  (:bound (lo hi)
                          (setf count
                                (min hi
                                     (max lo
                                          count)))))))
#<CLOSURE (LAMBDA (&REST #:ARGS0)) {D604B45}>
```

Теперь у нас есть несколько возможных лямбда деструктурирующих списков, которые могут использоваться в зависимости от нашего первого аргумента. `:reset` не требует аргументов и восстанавливает начальное значение `count` сбрасывая его в 0:

```
CL-USER> (count-test :reset)
0
```

`:inc` и `:dec` оба получают числовой аргумент, `n`:

```
CL-USER> (count-test :inc 100)
100
```

`:bound` проверяет находится ли значение `count` между двумя *граничными значениями*, `lo` и `hi`. Если же значение `count` окажется за указанными пределами, то значение `count` будет установлено в ближайшее граничное значение:

```
CL-USER> (count-test :bound -10 10)
10
```

Важная особенность `dlambda` заключается в использовании лямбды для всех деструктуризаций так, чтобы сохранить полноценную проверку ошибок и поддержку отладки, предоставляемую нашей COMMON LISP средой. Например, если мы зададим `count-test` только один аргумент, то мы получим ошибку непосредственно относящуюся к неверному *числу аргументов (arity)* лямбда приложения:

```
CL-USER> (count-test :bound -10)
invalid number of arguments: 1
```

Особенно когда `dlambda` встроена в лексическое окружение, формирующее замыкание, `dlambda` позволяет нам программировать — на объектно — ориентированном жаргоне — как бы создание объектов с несколькими *методами (methods)*. `Dlambda` предназначена для простого создания такой функциональности без необходимости отступления от синтаксиса и применения лямбды. `Dlambda` по прежнему расширяется в единственную лямбда форму и, таким образом, результат вычисления `dlambda` является тем же, что и результат вычисления `lambda`: *анонимная функция (anonymous function)*, которая может быть сохранена, применена и что самое главное — использована как лямбда компонент лексического замыкания.

Но, `dlambda` ещё более синхронизирована с лямбдой. Для того чтобы обеспечить как можно более бесшовную работу с кодом, содержащим `lambda` макрос, `dlambda` позволяет нам выполнять вызовы анонимной функции, которая не передаёт аргумент ключевое слово в качестве первого символа. При работе с `dlambda` у нас есть возможность добавлять методы без необходимости изменения остального кода, использующего единый интерфейс, а это немаловажно при работе с большими участками кода.

Если последний возможный метод задан символом `t` вместо аргумента ключевое слово, то этот предоставленный метод будет вызываться тогда, когда не обнаружено ни одного метода для какого-либо аргумента в виде ключевого слова. Вот один искусственный пример:

```
CL-USER> (setf (symbol-function 'dlambda-test)
               (dlambda
                (:something-special ()
                                    (format t "SPECIAL~%"))
```

```
                (t (&rest args)
                   (format t "DEFAULT: ~a~%" args))))
#<FUNCTION (LAMBDA (&REST #:ARGS0)) {B37D805}>
```

С таким определением в большинстве случаев при вызове этой функции будет происходить срабатывание метода по-умолчанию. Наш метод по-умолчанию использует аргумент лямбда деструктуризации `&rest` для получения всех доступных аргументов; в наших силах ограничить число полученных аргументов через использование более специфичных лямбда деструктурирующих параметров.

```
CL-USER> (dlambda-test 1 2 3)
DEFAULT: (1 2 3)
NIL
CL-USER> (dlambda-test)
DEFAULT: NIL
NIL
```

Однако, несмотря на то, что эта анонимная функция в основном работает также, как и обычная лямбда форма с методом по умолчанию, мы можем передавать анонимной функции аргумент в виде ключевого слова для вызова специального метода.

```
CL-USER> (dlambda-test :something-special)
SPECIAL
NIL
```

Ключевая особенность, которая будет много использоваться в следующей главе, заключается в том, что метод по-умолчанию и все специальные методы, конечно, вызываются в лексическом контексте охватывающей `dlambda`. Поскольку `dlambda` очень тесно интегрирована с лямбда нотацией, то у нас возникает возможность ввести мульти-методные техники в область, посвящённую созданию и расширению лексических замыканий.

Глава 6

Анафорические Макросы

6.1 Больше Фор?

Анафорические макросы (Anaphoric macros) — это одни из наиболее интересных макросов из книги Пола Грэма *On Lisp*. Анафорический макрос — это макрос осуществляющий преднамеренный захват переменной из форм, переданных в макрос. Благодаря наличию *прозрачных спецификаций (transparent specifications)* эти преднамеренно захваченные переменные порождают окна, позволяющие осуществлять контроль над расширением макроса. Через эти окна мы можем манипулировать расширением с помощью *комбинаций (combinations)*.

Классическая анафора, такая как в *On Lisp*, названа также, как и дословное[1] *анафор (anaphor)* и во множественном числе, *анафора (anaphora)*. Анафор обозначает захват *свободного слова U-Языка (free U-Language word)* для использования в последующем U-Языке. В терминах программирования, реализация классической анафоры означает нахождение места в вашем коде — или в коде, который вы хотите написать — где выражения могли бы извлечь выгоду, будучи способными сослаться на результаты предыдущих, связанных выражений. Настоятельно рекомендуется к изучению анафора Грэма и код, связанный с этой анафорой. Особенно обратите внимание на макрос `defanaph` [ON-LISP-P223], позволяющий использовать некоторые интересные разновидности программирования *автоматических анафор (automatic anaphor)*.

По истечении некоторого периода было обнаружено, что `alambda` — это наиболее полезный анафорический макрос в *On Lisp*. К тому же `alambda` — это один из наиболее простых и элегантных примеров, демонстрирующих анафорический макрос и преднамеренный захват перемен-

[1] Цитирование U-Языка.

Листинг 6.1: ALAMBDA

```
;; alambda Грэма
(defmacro alambda (parms &body body)
  `(labels ((self ,parms ,@body))
     #'self))
```

ной.

С `alambda` мы захватываем имя `self`, таким образом, мы можем использовать его для построения очень анонимной функции. Другими словами рекурсия становится такой же простой как и вызов `self`. Например, следующая функция возвращает список[2] чисел с `n` до `1`:

```
CL-USER> (alambda (n)
           (if (> n 0)
               (cons
                n
                (self (- n 1)))))
#<FUNCTION (LABELS SELF) {B28725D}>
CL-USER> (funcall
          (alambda (n)
            (if (> n 0)
                (cons
                 n
                 (self (- n 1))))) 3)
(3 2 1)
```

`Alambda` делает наш код более интуитивным и простым для чтения, позволяя нам думать об анонимных функциях как о функциях способных вызывать саму себя с той же лёгкостью, как и добавление единственной буквы[3]. Поскольку `alambda` прозрачно специфицирован для `self` привязки — и тот факт что единственная причина использования `alambda` — это применение этой привязки — то нежелательный захват переменной уже не проблема.

Другой удобный анафорический макрос из *On Lisp* — это `aif`, макрос, привязывающий результат проверки к `it` при положительном (вторич-

[2]Если условие возвращает ложное значение, то отсутствующий третий элемент `if` формы вернёт `nil`, что в свою очередь является списком.

[3]Это другая причина не шарп-закавычивать лямбда формы. Изменение шарп-закавыченных лямбда форм в alambda формы также потребует удаления двух символов.

Листинг 6.2: AIF

```
;; aif Грэма
(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))
```

ном) случае для последующего применения[4]. `Aif` использует удобную COMMON LISP особенность: *обобщённые булевы значения (generalised booleans)*. В COMMON LISP, все не-`nil` значения являются истинными значениями, поэтому COMMON LISP программисты обычно встраивают интересную информацию в истинные значения. Языки в которых зарезервированы истинные и ложные значения — в частности Scheme — используют *явные булевы значения (explicit booleans)*, которые иногда заставляют вас выводить дополнительную информацию для удовлетворения конструкций с избыточным типом. В Scheme добавлен *костыль (kludge)* для того, чтобы `if`, `cond`, `and`, `or` и `do` работали с не-булевыми[5] значениями [R5RS-P25]. Конечно, COMMON LISP спроектирован правильно — всё является булевым значением.

Также следует отметить что `aif` и `alambda`, как и все анафорические макросы, нарушают *лексическую прозрачность*. Или иначе, выражаясь модными словами, можно сказать что они не *гигиеничные (unhygienic)* макросы. Они, как и большое количество макросов в этой книге, вводят невидимые лексические привязки и поэтому не могут быть созданы в макро системах со строгим соблюдением гигиены. Даже большое количество Scheme систем, платформа в которой много экспериментировали с гигиеной, предоставляют макросы в негигиеничном defmacro стиле — возможно, по той причине, что даже разработчики Scheme не воспринимали всерьёз гигиену. Подобно обучению катания на велосипеде, гигиенические системы в своём большинстве игрушки, которые забрасывают после овладевания минимальным уровнем мастерства.

Да, существует много интересных вещей, которые мы можем делать с помощью преднамеренного захвата переменной. Существует множество фор (phors). Эта книга и книга Грэма *On Lisp* описывает лишь малую часть возможностей, заложенных в эту технику. Из грамотного применения анафорических макросов можно извлечь много новых потрясающих изобретений.

[4]Упражнение: Почему ложный (третичный или альтернативный) вариант никогда не использует эту анафору?

[5]С помощью Scheme предиката `boolean?`.

Листинг 6.3: SHARP-BACKQUOTE

```
(defun |#`-reader| (stream sub-char numarg)
  (declare (ignore sub-char))
  (unless numarg (setq numarg 1))
  `(lambda ,(loop for i from 1 to numarg
                  collect (symb 'a i))
     ,(funcall
       (get-macro-character #\`) stream nil)))

(set-dispatch-macro-character
 #\# #\` #'|#`-reader|)
```

После краткой беседы об анафорах, продемонстрированной считывающими макросами, остаток этой главы описывает узкое приложение анафор к одной из центральных тем этой книги: лексические замыкания — *let, окружающий lambda*. Большая часть этой главы посвящена интересным анафорическим макросам, предназначенным для изменения, адаптации и расширении замыканий. И хотя данная тема очень удобна в практическом применении для написания кода, мы используем её как платформу для обсуждения возможностей и вариантов анафорических макросов. Применение макросов для расширения концепции замыканий — это, на данный момент, интенсивно исследуемая тема [FIRST-CLASS-EXTENTS] [OPENING-CLOSURES].

6.2 Шарп-Обратное Закавычивание

Хотя большинство анафор представлено в виде обычных макросов, у считывающих макросов тоже есть возможность использования кода, создающего для нас невидимые привязки. Такие считывающие макросы называются *считывающими анафорами (read anaphora)*. Этот раздел ознакомит вас с таким макросом, который являясь весьма незатейливым, удивил даже меня и который стал одним из наиболее полезных макросов, используемых в этой книге. Я постарался как можно скорее перейти к этому макросу, поскольку он будет использоваться в оставшемся коде. Более того, несколько рассмотренных нами макросов уже используют его.

Шарп-обратное закавычивание — это считывающий макрос, выполняющий считывание как лямбда форма. По умолчанию, эта лямбда фор-

ма получает только один аргумент: a1. Затем считывающий макрос рекурсивно вызывает функцию read для переданного потока. Ниже есть пример с остановленным вычислением (с помощью quote), и мы можем рассмотреть прозрачное введение считывающей анафоры[6]:

```
CL-USER> '#`((,a1))
(LAMBDA (A1) `((,A1)))
```

Этот считывающий макрос формирует абстракцию общего шаблона макроса. Например, если у нас есть список переменных и мы хотим создать список let привязок, где каждая переменная привязана к символу, скажем, empty, то мы можем использовать mapcar следующим способом:

```
CL-USER> (mapcar (lambda (a)
                   (list a ''empty))
                 '(var-a var-b var-c))
((VAR-A 'EMPTY) (VAR-B 'EMPTY) (VAR-C 'EMPTY))
```

Но особенно для сложных списковых структур, такой подход может оказаться довольно грязным, поэтому лисп программисты предпочитают использовать обратное закавычивание для поднятия на один уровень закавычивания:

```
CL-USER> (mapcar (lambda (a)
                   `(,a 'empty))
                 '(var-a var-b var-c))
((VAR-A 'EMPTY) (VAR-B 'EMPTY) (VAR-C 'EMPTY))
```

Наш новый считывающий макрос вводит анафору и скрывает лямбда форму:

```
CL-USER> (mapcar #`(,a1 'empty)
                 '(var-a var-b var-c))
((VAR-A 'EMPTY) (VAR-B 'EMPTY) (VAR-C 'EMPTY))
```

1 в символе a1 выше говорит о том, что пользователи считывающего макроса могут вводить переменное число анафор, зависящее от числа переданного в параметр numarg считывающего макроса:

```
CL-USER> '#2`(,a1 ,a2)
(LAMBDA (A1 A2) `(,A1 ,A2))
```

[6]Конечно, префикс захваченного символа, “a”, расшифровывается анафорой.

Таким образом мы можем одновременно замapcar-ить шарп-обратно закавыченные выражения с несколькими списками:

```
CL-USER> (let ((vars '(var-a var-b var-c)))
           (mapcar #2` (,a1 ',a2)
                   vars
                   (loop for v in vars
                         collect (gensym
                                  (symbol-name v)))))
((VAR-A '#:VAR-A1213)
 (VAR-B '#:VAR-B1214)
 (VAR-C '#:VAR-C1215))
```

Ещё один способ думать о шарп-обратном закавычивании: шарп-обратное закавычивание выполняет интерполяцию списка, также, как функция `format` выполняет интерполяцию строки. `Format` позволяет нам использовать шаблоны со слотами, заполняемыми значениями отдельных аргументов, аналогично шарп-обратное закавычивание позволяет нам отделить структуру интерполированного списка от значений, с которыми мы хотим их объединить. Благодаря ранее описанному *дуализму синтаксиса* между лямбда формами в позиции функции в списке и лямбда формами, использующими макрос `lambda` для расширения в функцию, мы также можем использовать шарп-обратное закавычивание в роли первого элемента при вызове функции:

```
CL-USER> (#3` (((,a1)) ,@a2 (,a3))
              (gensym)
              '(a b c)
              'hello)
(((#:G1216)) A B C (HELLO))
```

В отличие от `format` шарп-обратное закавычивание не использует последовательное позиционирование. Вместо этого используется числа для наших анафорических привязок. Вследствие этого порядок может перемешиваться и мы можем сращивать привязки более одного раза:

```
CL-USER> (#3` (((,@a2)) ,a3 (,a1 ,a1))
              (gensym)
              '(a b c)
              'hello)
(((A B C)) HELLO (#:G1217 #:G1217))
```

Листинг 6.4: ALET-1

```
(defmacro alet% (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     this))
```

Упражнение: Ссылки на `gensym #:G1217` выглядят так, как будто ссылаются на один и тот-же символ, но, конечно, вы никогда не можете утверждать это глядя на напечатанные имена. Являются ли эти символы `eq`-абельными? Обоснуйте ответ.

6.3 Alet и Машины с Конечным Состоянием

С *lambda* и *if* возможна только одна полезная анафорическая конфигурация. Но большинство интересных типов анафорических макросов используют расширения самыми неожиданными способами. Этот раздел — как и большая часть этой главы — посвящена одному из таких макросов: `alet`. Какие дополнительные привязки могут быть полезны для форм внутри тела `let` формы? `Let` используется для того, чтобы создать привязки к переданным переменным. Однако, макро улучшение `let` даст возможность получения доступа ко всем переданным формам, даже если тела этих выражений должны вычисляться с новыми привязками. Так какая же часть тела наиболее полезна? В большинстве случаев это последняя форма тела, поскольку результаты этой формы будут возвращены из самого оператора `let`[7]. Мы уже видели что когда мы возвращаем лямбда выражение, ссылающееся на привязки, созданные `let`-ом, то результатом является лексическое замыкание — объект, часто сохраняемый и используемый для последующего доступа к переменным в `let` операторе. Таким образом, расширение нашего аналога объекта-замыкания, макрос `alet%` работает также, как и специальная форма `let`, с тем отличием, что захватывает символ `this` из тела и привязывает его к последнему выражению в теле формы — единственное, что будет возвращено как замыкание[8].

`Alet%` может быть полезен в случае когда у нас есть инициализирую-

[7]Поскольку `let` предоставляет неявный `progn`.

[8]`Setq` применяется потому что форма привязанная к `this` определена в лексической области видимости других аргументов заданных через `letargs`.

щий код в лямбда форме, который мы не хотим дублировать. Поскольку `this` привязан к возвращаемой лямбда форме, то у нас есть возможность выполнить его перед выходом из окружающего `let`. Следующий пример — это замыкание, конструкция которого показывает простой пример использования `alet%` с целью избежания дублирования инициализирующего и сбрасывающего кода:

```
CL-USER> (alet% ((sum) (mul) (expt))
                (funcall this :reset)
                (dlambda
                 (:reset ()
                         (psetq sum 0
                                mul 1
                                expt 2))
                 (t (n)
                    (psetq sum (+ sum n)
                           mul (* mul n)
                           expt (expt expt n))
                    (list sum mul expt))))
#<CLOSURE (LAMBDA (&REST #:ARGS0)) {AB979AD}>
```

С помощью этого кода мы можем последовательно изменять значения `sum`, `mul` и `expt`:

```
CL-USER> (loop for i from 1 to 5 collect (funcall * 2))
((2 2 4) (4 4 16) (6 8 256) (8 16 65536) (10 32 4294967296))
```

Мы можем сбросить замыкание вызвав метод `:reset`. Заметьте, что благодаря `alet%`, для сброса в базовые состояния (0 для `sum`, 1 для `mul` и 2 для `expt`) достаточно использовать `:reset` в одном месте:

```
CL-USER> (funcall ** :reset)
NIL
```

Теперь, когда переменные замыкания были сброшены, мы можем увидеть новую последовательность:

```
CL-USER> (loop for i from 1 to 5 collect (funcall *** 0.5))
((0.5 0.5 1.4142135) (1.0 0.25 1.1892071) (1.5 0.125 1.0905077)
 (2.0 0.0625 1.0442737) (2.5 0.03125 1.0218971))
```

Листинг 6.5: ALET

```
(defmacro alet (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (lambda (&rest params)
       (apply this params))))
```

Учтите, что `alet%` меняет порядок вычисления форм в теле `let`. Если вы посмотрите на расширение, вы увидите, что последняя форма тела вычислена первой, её результат привязан к лексической привязке `this`, а после этого вычисляются предыдущие формы. И пока последний аргумент является константой, изменение порядка вычислений не играет никакой роли. Помните что лямбда выражение[9] является константой и, таким образом, идеально подходит для использования в `alet%`.

Нам доступны множество уровней свободы, поэтому улучшение этого макроса, как и многие другие улучшения макросов, кажутся нелогичными. И хотя существуют множество возможностей, этот раздел посвящён рассмотрению одного такого специфического улучшения. `Alet%` можно модифицировать таким образом, чтобы он не возвращал последнюю форму своего тела — про которое мы уже заранее знаем что оно будет лямбда формой — взамен функции, находящей другую функцию внутри лексической области видимости `let` формы, и вызывающей эту функцию. Иногда это называется *косвенностью (indirection)*, поскольку вместо возвращения функции, выполняющей что либо, мы возвращаем функцию, которая ищет функцию с помощью разыменовывания указателя и после этого использует эту функцию. Косвенность — это понятие, применимое ко всем языкам программирования. Оно позволяет нам менять объекты во время выполнения в то время когда без косвенности, эти объекты являются неизменными во время компиляции. Лисп позволяет нам использовать косвенность в более кратком и удобном стиле чем во многих других языках программирования. `Alet` — версия `alet%` с косвенностью, позволяет получить доступ к функции, возвращённой нами как замыкание или заместить эту функцию кодом внутри тела `alet` или, если мы используем `dlambda`, как это будет показано позже, даже кодом, находящимся за пределом тела `alet`.

Теперь, когда мы можем изменить функцию, которая запускается при

[9] `Dlambda` расширяется в лямбда формы.

вызове замыкания с нашим макросом `alet`, у нас есть возможность создания пары взаимно ссылающихся функций с помощью шаблона под названием *alet, окружающий alambda*. Если все состояния возвращаются в исходное состояние — вместо того, чтобы переходить в состояния друг в друга — то, `alet`, окружающий `alambda` — это удобный способ определения безымянных машин с состоянием.

Следующее типичное замыкание-счётчик получает аргумент `n` и может менять направление счётчика между инкрементированием и декрементированием на `n` при передаче символа `invert` в качестве аргумента взамен числа `n`:

```
CL-USER> (alet ((acc 0))
               (alambda (n)
                 (if (eq n 'invert)
                     (setq this
                           (lambda (n)
                             (if (eq n 'invert)
                                 (setq this #'self)
                                 (decf acc n))))
                     (incf acc n))))
#<CLOSURE (LAMBDA (&REST PARAMS)) {C43A6CD}>
```

Сохраним это замыкание для того, чтобы мы могли использовать его так часто, насколько это нам будет нужно:

```
CL-USER> (setf (symbol-function 'alet-test) *)
#<CLOSURE (LAMBDA (&REST PARAMS)) {C43A6CD}>
```

При первом запуске счётчик идёт на увеличение:

```
CL-USER> (alet-test 10)
10
```

Но, мы можем изменять функцию, на вызов внутреннего лямбда выражения в нашем определении через передачу символа `invert` замыканию:

```
CL-USER> (alet-test 'invert)
#<CLOSURE (LAMBDA (N) :IN SELF) {C7CFA7D}>
```

И теперь мы идём вниз:

```
CL-USER> (alet-test 3)
7
```

И наконец благодаря привязке `self` предоставляемой `alambda` мы можем снова сменить вызываемую функцию передав символ `invert`:

```
CL-USER> (alet-test 'invert)
#<CLOSURE (LABELS SELF) {C43A6B5}>
```

Мы вернулись к предыдущему направлению и идём вверх:

```
CL-USER> (alet-test 5)
12
```

Это замыкание будет привязано к символу `alet-test` относящемуся к пространству функции . Но это замыкание слегка отличается от обычного замыкания. Оба этих замыкания (обычное замыкание и наше замыкание) являются ссылками к одной среде, которое может иметь любое количество указателей, это замыкание использует косвенность для изменения кода, который будет выполнен при вызове. И хотя установленным может быть любой код, получить доступ к лексическим привязкам анафоре `this` в лексической области `alet` может только один код. Но, нам по прежнему ничто не мешает установить новое замыкание, с его собственными лексическими привязками и, возможно, с изменённым поведением в *косвенной среде*, установленной с помощью `alet`. Большая часть этой главы посвящена полезным приёмам, которые мы можем выполнять с косвенной средой, созданной с помощью `alet`.

Общая техника работы с макросами неформально известна как *выворачивание макроса наизнанку (turning a macro inside out)*. Когда вы выворачиваете макрос наизнанку вы получаете типичную форму, использующую макрос, подобный макросу который вы хотите создать, и расширяете её. Дальше вы используете расширение как шаблон для вашего желаемого макроса. Например, мы хотели бы получить более универсальный метод создания замыканий с множественными состояниями чем способ с вышеприведённым счётчиком, основанном на `alet`, окружающем `alambda`. Вот расширение изнанки в случае с инвертируемым `alambda` счётчиком:

```
CL-USER> (macroexpand
          '(alambda (n)
            (if (eq n 'invert)
                (setq this
```

```
                      (lambda (n)
                        (if (eq n 'invert)
                            (setq this #'self)
                            (decf acc n))))
                 (incf acc n))))
(LABELS ((SELF (N)
           (IF (EQ N 'INVERT)
               (SETQ THIS
                       (LAMBDA (N)
                         (IF (EQ N 'INVERT)
                             (SETQ THIS #'SELF)
                             (DECF ACC N))))
               (INCF ACC N))))
  #'SELF)
T
```

Если мы выполним маленький рефакторинг вышеприведённого расширения с учётом того, что `labels` позволяет нам создавать множественные привязки функций[10], то получится следующее:

```
CL-USER> (alet ((acc 0))
               (labels ((going-up (n)
                          (if (eq n 'invert)
                              (setq this #'going-down)
                              (incf acc n)))
                        (going-down (n)
                          (if (eq n 'invert)
                              (setq this #'going-up)
                              (incf acc (- n)))))
                 #'going-up))
#<CLOSURE (LAMBDA (&REST PARAMS)) {AC359AD}>
```

Здесь нужно учесть что `alambda` может сделать доступными все тела специальной формы `labels` для всех функций. И что ещё более важно, теперь у нас есть довольно неплохой шаблон для нашего событийного макроса.

`Alet-fsm` даёт нам возможность использовать удобный синтаксис для выражения нескольких возможных *состояний (states)* для нашего замыкания. Это очень тонкий слой сахара из макросов поверх `labels`, скомбинированный с *проходом-по-коду* `macrolet` трансформации, позволяющей нам сымитировать функцию `state`, изменяющую текущее состояние

[10]Используется множественность labels.

Листинг 6.6: ALET-FSM

```
(defmacro alet-fsm (&rest states)
  `(macrolet ((state (s)
                `(setq this #',s)))
     (labels (,@states) #',(caar states))))
```

замыкания через анафору `this` предоставляемую `alet`-ом. В качестве примера приведена более чистая версия счётчика с изменяемым направлением роста:

```
CL-USER> (alet ((acc 0))
               (alet-fsm
                (going-up (n)
                          (if (eq n 'invert)
                              (state going-down)
                              (incf acc n)))
                (going-down (n)
                            (if (eq n 'invert)
                                (state going-up)
                                (decf acc n)))))
#<CLOSURE (LAMBDA (&REST PARAMS)) {B4BE13D}>
```

`Alet-fsm` представляет экземпляр техники, которую мы ещё не видели: *инъекция анафоры (anaphor injection)*. Использование этой анафоры нарушает лексическую прозрачность многими способами, что так или иначе, становится *лексически невидимым (lexically invisible)*. Кроме невидимой привязки `this` `alet`-ом, также невидимым является использование `this` макросом `alet-fsm`. `Alet-fsm` инъецирует свободную переменную в наш лексический контекст так, что мы не можем её увидеть.

В данном случае стилистические проблемы не строго очерчены[11], поскольку программирование макросов это конечно не вопрос стиля. Это вопрос мощи. Иногда инъекция свободной переменной может создавать симбиоз между двумя макросами — вместе они смогут создать расширение с более лучшей программной конструкцией чем два изолированных расширения. Поскольку этот тип макро программирования сложен, то

[11]Как и все стилистические вопросы. Если вы что-либо поймёте в полном объёме, то вопросы стиля становятся неуместными. Инъекция свободной переменной понята не до конца.

Листинг 6.7: ICHAIN-BEFORE

```
(defmacro! ichain-before (&rest body)
  `(let ((,g!indir-env this))
     (setq this
           (lambda (&rest ,g!temp-args)
             ,@body
             (apply ,g!indir-env
                    ,g!temp-args)))))
```

опять можно провести параллели с указателями C. Как изучение указателей C порождает сомнительные стилистические приёмы, также происходит и с инъекцией свободной переменной.

Наиболее правдоподобная гипотеза объясняющая трудность понимания инъекции свободной переменной связана с вопросом *отказобезопасного* поведения[12]. С анафорой, в случае когда код переданный пользователем не использует привязку, возможно дальнейшее функционирование кода. Но, есть вероятность, что такой код может отказать без каких-либо предупреждений, а это небезопасно. Однако, когда вы вставите свободную переменную и нет окружения в котором эта переменная может быть захвачена, то всё выражение будет свободным. При возникновении такой ситуации вам нужно решить что делать до того, как вы будете вычислять это выражение. Оно может быть отказобезопасным.

Отставим в сторону стиль. Иногда инъекция свободной переменной — это то, что нам нужно для того, чтобы создать коммуникацию между двумя связанными макросами. Инъекция выполняет то же, что и анафора, но в другом направлении. Поскольку вы открываете новый канал коммуникации между вашими макросами, то проблемы сложности масштабируются очень быстро. Рассмотрим нахождение в доме полном хрупкого стекла. Вы можете безопасно швырять предметы в людей снаружи дома, даже если они не озаботятся их ловлей, но, вам лучше убедиться в том, что вы сможете поймать любой объект брошенный в вас.

6.4 Косвенные Цепи

Существуют много способов с помощью которых мы можем извлечь пользу из анафоры `this`, предоставляемой макросом `alet`. Поскольку

[12]Здесь под понятием безопасности, в отличие от реального мира, понимается максимально быстрый и громкий отказ.

доступ к окружению осуществляется через замыкание-пустышку, перенаправляющее все вызовы к настоящему замыканию, на которое указывает `this`, то мы можем где угодно передавать ссылку на замыкание-пустышку копируя его так часто, сколько нам это будет нужно. Удобство подобной *косвенности* в том, что мы можем менять происходящие события при вызове замыкания-пустышки без изменения ссылок на само замыкание-пустышку.

`Ichain-before` предназначен для расширения в форме `alet`. Этот макрос добавляет исполнение нового кода до вызова главного замыкания. Вернёмся к нашему примеру со счётчиком, `ichain-before` позволяет нам добавить новое замыкание, которое печатает предыдущее значение переменной `acc` до того как это значение будет увеличено:

```
CL-USER> (alet ((acc 0))
               (ichain-before
                (format t "Changing from ~a~%" acc))
               (lambda (n)
                 (incf acc n)))
#<CLOSURE (LAMBDA (&REST PARAMS)) {AEC6015}>
```

Результат такой, какой и ожидался:

```
CL-USER> (funcall * 2)
Changing from 0
2
CL-USER> (funcall ** 2)
Changing from 2
4
```

Есть причина по которой в названии `ichain-before` содержится упоминание цепи (chain). Мы можем вставить сколько угодно исполняемых замыканий:

```
CL-USER> (alet ((acc 0))
          (ichain-before
           (format t "A~%"))
          (ichain-before
           (format t "B~%"))
          (ichain-before
           (format t "C~%"))
          (lambda (n)
            (incf acc n)))
#<CLOSURE (LAMBDA (&REST PARAMS)) {B1B330D}>
```

Каждое новое звено будет добавляться в начало цепи, в результате появление звеньев цепи будет происходить в обратном порядке:

```
CL-USER> (funcall * 2)
C
B
A
2
```

Иногда статическое добавление косвенных цепей оказывается полезным для того, чтобы избежать реструктуризацию макросов просто добавив вместо этого новый окружающий код. Но наиболее интересные возможности косвенных цепей возникают при их динамическом добавлении. Поскольку мы можем создавать новые функции во время исполнения, а также можем получать доступ ко внутренностям замыкания через анафору, то мы можем переписать работу функций во время исполнения. Ниже представлен простой пример в котором каждый вызов замыкания добавляет код, печатающий “Hello world” при запуске:

```
CL-USER> (alet ((acc 0))
               (lambda (n)
                 (ichain-before
                  (format t "Hello world~%"))
                 (incf acc n)))
#<CLOSURE (LAMBDA (&REST PARAMS)) {C3EFE15}>
```

Каждый вызов добавляет новое замыкание к косвенной цепи:

```
CL-USER> (loop for i from 1 to 4
              do
              (format t "~:r invocation:~%" i)
              (funcall * i))
first invocation:
second invocation:
Hello world
third invocation:
Hello world
Hello world
fourth invocation:
Hello world
Hello world
Hello world
NIL
```

Листинг 6.8: ICHAIN-AFTER

```
(defmacro! ichain-after (&rest body)
  `(let ((,g!indir-env this))
     (setq this
           (lambda (&rest ,g!temp-args)
             (prog1
                 (apply ,g!indir-env
                        ,g!temp-args)
               ,@body)))))
```

Макрос `ichain-after` подобен макросу `ichain-before` за исключением того, что он добавляет замыкания в другой конец исполняемой цепи: после вызова главного замыкания. `Ichain-after` использует `prog1`, который последовательно исполняет переданные формы и затем возвращает результат вычисления первой формы.

`Ichain-before` и `ichain-after` могут комбинироваться, пример:

```
CL-USER> (alet ((acc 0))
               (ichain-before
                (format t "Changing from ~a~%" acc))
               (ichain-after
                (format t "Changed to ~a~%" acc))
               (lambda (n)
                 (incf acc n)))
#<CLOSURE (LAMBDA (&REST PARAMS)) {D26585D}>
CL-USER> (funcall * 7)
Changing from 0
Changed to 7
7
```

`Ichain-before` и `ichain-after` — это макросы, которые инъецируют свободные переменные в свои расширения. Они инъецируют символ `this`, который предназначен для захвата расширением макроса `alet`. Такая разновидность инъекции символов может показаться плохим стилем программирования, который способен порождать ошибки, но на самом деле — это общая техника макросов. На деле почти все макросы инъецируют символы в расширение. К примеру: помимо `this`, макрос `ichain-before` также инъецируют такие символы как `let`, `setq` и `lambda` предназначенные для сращивания с окружением, в котором расширяется

Листинг 6.9: ICHAIN-INTERCEPT-1

```
(defmacro! ichain-intercept% (&rest body)
  `(let ((,g!indir-env this))
     (setq this
           (lambda (&rest ,g!temp-args)
             (block intercept
               (prog1
                   (apply ,g!indir-env
                          ,g!temp-args)
                 ,@body))))))
```

макрос. Разница между такими символами как `this` и такими предварительно определёнными символами как `setq` в том, что `lambda` всегда ссылается к единственному хорошо описанному ANSI макросу, а такие символы как `this` могут ссылаться к различным объектам в зависимости от сред, в которых они были расширены.

`Ichain-before` и `ichain-after` полезны для того, чтобы метить код, выполняемый до или после исполнения исходного перекрытого выражения. Но это ни в коем случае не единственная операция, которую мы можем выполнять с помощью анафоры `this`. Другой распространённой операцией является проверка данных замыкания после вызова замыкания.

`Ichain-intercept%` — это другой макрос, предназначенный для использования внутри формы `alet`. Идея заключается в том, что мы хотим перехватывать вызовы замыкания и проверять результаты действия замыкания с целью предотвращения некорректных состояний в замыкании.

Так мы можем добавить перехват в наше обычное замыкание-счётчик:

```
CL-USER> (alet ((acc 0))
               (ichain-intercept%
                (when (< acc 0)
                  (format t "Acc went negative~%")
                  (setq acc 0)
                  (return-from intercept acc)))
               (lambda (n)
                 (incf acc n)))
#<CLOSURE (LAMBDA (&REST PARAMS)) {B0903CD}>
```

Код, установленный при помощи `ichain-intercept%`, предупредит

Листинг 6.10: ICHAIN-INTERCEPT

```
(defmacro! ichain-intercept (&rest body)
  `(let ((,g!indir-env this))
     (setq this
           (lambda (&rest ,g!temp-args)
             (block ,g!intercept
               (macrolet ((intercept (v)
                            `(return-from
                               ,',g!intercept
                               ,v)))
                 (prog1
                     (apply ,g!indir-env
                            ,g!temp-args)
                   ,@body)))))))
```

нас о падении значения счётчика ниже 0:

```
CL-USER> (funcall * -8)
Acc went negative
0
```

Счётчик будет сброшен в 0:

```
CL-USER> (funcall ** 3)
3
```

Наиболее интересным моментом в `ichain-intercept%` является введение *блокирующей анафоры (block anaphor)* под названием `intercept`. Для использования этой анафоры мы применяем `return-from`. Блок будет возвращать это значение из вызова замыкания перехватывая исходное значение.

Вместо захвата блокирующей анафоры `intercept`, `ichain-intercept` создаёт локальный макрос, позволяющий коду внутри `ichain-intercept` использовать `intercept` для расширения в `return-from` где блок определён как `gensym`.

```
CL-USER> (alet ((acc 0))
               (ichain-intercept
                (when (< acc 0)
                  (format t "Acc went negative~%")
```

```
                  (setq acc 0)
                  (intercept acc)))
               (lambda (n)
                 (incf acc n)))
#<CLOSURE (LAMBDA (&REST PARAMS)) {CC4F815}>
```

Результат тот же, что и в `ichain-intercept%`:

```
CL-USER> (funcall * -8)
Acc went negative
0
CL-USER> (funcall ** 3)
3
```

Конечно, прозрачное введение всех этих замыканий в операции может сказаться на производительности во время выполнения. К счастью, современные лисп компиляторы очень хорошо оптимизируют замыкания. Если ваше приложение может разыменовывать несколько указателей — а часто оно это может — косвенные цепи могут быть лучшим способом для их структуризации. Смотрите раздел Область Видимости Указателя, там описан другой взгляд на косвенные цепи. Кроме того, рассмотрите CLOS-овскую функциональность до, после и вокруг.

6.5 Замыкания, Поддерживающие Горячую Замену

В этом разделе было запланировано три цели. Первая, было рассмотрено другое интересное использование анафоры `this` из `alet`. Второе, обсудили шаблон *alet, окружающий dlambda*. И наконец, введена полезная макро техника под названием *закрытие анафоры*.

Для того, чтобы ясно проиллюстрировать закрытие анафоры мы не будем работать с макросом `alet` и переключимся на изнанку расширения. `Alet-hotpatch%` — это расширение `alet` со специальной лямбда формой. Эта лямбда форма проверяет первый аргумент[13] на соответствие с ключевым символом `:hotpatch` и при обнаружении этого символа заменяет косвенное замыкание на переданный аргумент.

Возможность замены замыкания использованного в другом выполняющемся замыкании во время работы программы называется *горячей заменой (hotpatching)*. Например, мы создали замыкание с поддержкой

[13] Со сравнением указателя.

Листинг 6.11: ALET-HOTPATCH-1

```
(defmacro alet-hotpatch% (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (lambda (&rest args)
       (if (eq (car args) ':hotpatch)
           (setq this (cadr args))
           (apply this args)))))
```

горячей замены и, для дальнейшего применения, сохранили его в ячейку символа-функции символа `hotpatch-test`:

```
CL-USER> (setf (symbol-function 'hotpatch-test)
               (alet-hotpatch% ((acc 0))
                               (lambda (n)
                                 (incf acc n))))
#<CLOSURE (LAMBDA (&REST ARGS)) {AE37ED5}>
```

Возможное применение:

```
CL-USER> (hotpatch-test 3)
3
CL-USER> (hotpatch-test 4)
7
```

Мы можем заменить лямбда форму — вместе со связанной с ней средой — вызвав это замыкание с символом `:hotpatch` и заменяющей функцией или замыканием:

```
CL-USER> (hotpatch-test
          :hotpatch
          (let ((acc 0))
            (lambda (n)
              (incf acc (* 2 n)))))
#<CLOSURE (LAMBDA (N)) {B0FF48D}>
```

Теперь замыкание обладает новым, исправленным на лету поведением:

Листинг 6.12: ALET-HOTPATCH

```
(defmacro alet-hotpatch (letargs &rest body)
  `(let ((this) ,@letargs)
     (setq this ,@(last body))
     ,@(butlast body)
     (dlambda
      (:hotpatch (closure)
                 (setq this closure))
      (t (&rest args)
         (apply this args)))))
```

```
CL-USER> (hotpatch-test 2)
4
CL-USER> (hotpatch-test 5)
14
```

Заметьте как значение счётчика сбросилось в 0 после горячей замены среды замыкания на новое значение для аккумулятора счётчика `acc`.

Кажется мы уже видели такую разновидность *деструктуризации во время исполнения*, основанную на символах ключевых слов? Да, на деле мы написали макрос выполняющий то же самое что и в *разделе Dlambda*. `Alet-hotpatch` — это версия `alet-hotpatch%` основанная на `dlambda`. Иногда, даже не осознавая этого, при написании новых макросов мы применяем *комбинирование макросов (macro combination)*. У хорошо спроектированного макроса расширение полностью понятно и хотя возможны разнообразные нарушения лексической прозрачности, проблем с комбинированием не возникает, поскольку все компоненты подходят друг к другу.

`Alet-hotpatch` создаёт замыкание с поддержкой горячей замены, но в этом макросе есть один концептуальный недостаток. Поскольку макрос `alet-hotpatch` используется для создания замыканий с горячей заменой, то мы можем забыть что этот макрос вводит анафору `this` в область видимости переданных форм. Если мы забудем о созданной анафоре, то мы рискуем столкнуться с проблемой нежелательного захвата переменной. Для того, чтобы избежать подобных проблем, мы должны применить технику известную как *закрытие анафоры (anaphor closing)*. При закрытии анафоры мы не меняем способ функционирования анафорического макроса, а просто запрещаем некоторые способы комбинирования этого макроса.

Листинг 6.13: LET-HOTPATCH

```
(defmacro! let-hotpatch (letargs &rest body)
  `(let ((,g!this) ,@letargs)
     (setq ,g!this ,@(last body))
     ,@(butlast body)
     (dlambda
      (:hotpatch (closure)
                 (setq ,g!this closure))
      (t (&rest args)
         (apply ,g!this args)))))
```

Поскольку мы вывернули расширение `alet` наизнанку, то мы можем лексически увидеть создание анафоры `this` в определении макроса `alet-hotpatch`. И поскольку `alet-hotpatch` также содержит код, использующий анафору `this` для реализации горячей замены, то мы можем закрыть анафору так, что символ `this` больше не будет захватываться макросом. Каким стандартным способом мы избегаем введения нежелательных привязок? Конечно мы именуем привязки с помощью `gensym`-ов.

`Let-hotpatch` — это пример закрытия анафоры `this` в более замкнутую версию — более безопасная версия после реализации всех необходимых нам горячих замен. "`A`", удалённое из названия макроса, обозначает что этот новый макрос больше не вносит анафору в переданный код. Конечно если нам необходимо по некоторым причинам, отличающимся от простой горячей замены, сослаться на `this`, то мы должны оставить анафору открытой.

Техника открытия и закрытия анафоры становится второй натурой программиста после того, как он напишет достаточное количество подобных макросов. Также, как мы можем писать макросы, инъецирующие свободные переменные в расширение не задумываясь о том как мы будем их захватывать пока мы пишем лексический контекст в котором они будут расширяться, мы иногда оставляем открытым анафору при разработке макроса для экспериментирования с комбинациями анафорических макросов и инъекции свободных переменных. После того, как будут найдены наиболее полезные комбинации, мы можем произвести слияние макросов и заменить все использованные во время разработки анафоры на `gensym`-ы. Также, как это делает `let-hotpatch`, эта техника может использовать `defmacro!` для перемещения области видимости анафоры с макро расширения в макро определение. Вместо лексического

введения анафоры мы ввели другой тип анафоры — такой, который не может работать в целой лексической области видимости расширения, но работает только в другой, более ограниченной области видимости. Эта область видимости будет описываться в следующем разделе.

6.6 Суб-Лексическая Область Видимости

Наш макро-определяющий макрос `defmacro!` определённый в *разделе Нежелательный Захват* ищет присутствие автоматических `gensym`-ов в переданном коде с помощью утилиты Грэма `flatten`. А теперь пришло время признаться в маленькой лжи. До сих пор, пока мы не рассмотрели инъекцию свободной переменной и анафору, мы считали что символьные G-bang имена в `defmacro!` определениях применимы в лексической области видимости макро определения. На самом деле это не правда — `defmacro!` предоставляет эти привязки под несколько отличающимся типом области видимости которая называется *суб-лексической областью видимости (sub-lexical scope)*.

Помните, что область видимости обозначает корректность ссылок на переменную, а лексическая область видимости означает что имя применимо к коду в текстовом теле таких создающих привязки конструкций, как `let`. Важное различие между лексической областью видимости и суб-лексической областью видимости в том, что лексическая область видимости включает в себя все макрорасширения кода в теле `let`. Таким образом, высказывание о том, что лексическая область видимости — это создание переменных, доступных только из кода в текстовом теле создающих привязки конструкций является ложным — макросы могут *инъецировать* ссылки на переменные. Такие переменные инъецируются извне текстового тела конструкций, создающих привязки.

Реализация уникальной текстовой области видимости с ограничением на возможные способы доступа к лексическим переменным равняется суб-лексической области видимости. Ссылки на переменные в суб-лексической области видимости действительны только если представляющие их символы присутствуют в сырых списках, переданных лиспу до макро-расширения.

Поскольку `defmacro!` производит предварительную обработку полученного кода и создаёт списки всех G-bang символов до расширения кода, то отсюда следует что G-bang символы суб-лексически привязаны. Мы не можем написать макросы, инъецирующие G-bang символы в `defmacro!` поскольку лексические привязки для G-bang символов никогда не создавались. Вот типичное применение суб-лексического G-bang символа:

```
CL-USER> (defmacro! junk ()
           `(let ((,g!var))
              ,g!var))
JUNK
```

Оба G-bang символа были созданы в суб-лексической области видимости `defmacro!`, таким образом, расширение будет таким, каким мы его ожидали увидеть:

```
CL-USER> (macroexpand '(junk))
(LET ()
  (LET ((#:VAR1196))
    #:VAR1196))
T
```

Однако, для того, чтобы изучить концепцию суб-лексической области видимости мы определим макрос, инъецирующий G-bang символ:

```
CL-USER> (defmacro injector-for-g!var ()
           ''g!var)
INJECTOR-FOR-G!VAR
```

Теперь мы можем написать `junk2`. `Junk2` идентичен `junk` с тем исключением, что мы заменили наши G-bang символы макросом, который расширяется в G-bang символ:

```
CL-USER> (defmacro! junk2 ()
           `(let ((,(injector-for-g!var)))
              ,(injector-for-g!var)))
JUNK2
```

Но поскольку G-band символы привязаны суб-лексически — и поэтому не ищутся в макро расширениях форм — то `defmacro!` не конвертирует символы в автоматические `gensym`-ы:

```
CL-USER> (macroexpand '(junk2))
(LET ()
  (LET ((G!VAR))
    G!VAR))
T
```

И хотя вышеприведённый код по-прежнему функционален, указатели на переменные, находящиеся в суб-лексической области видимости могут вывести из строя выражения, в которых некоторые указатели ссылаются на переменные существующие в суб-лексической области видимости а другие — нет:

```
CL-USER> (defmacro! junk3 ()
           `(let ((,g!var))
              ,(injector-for-g!var)))
JUNK3
CL-USER> (macroexpand '(junk3))
(LET ()
  (LET ((#:VAR1209))
    G!VAR))
T
```

Суб-лексическая область видимости удивительно часто появляется в сложных макросах. Кроме `defmacro!` можно привести по крайней мере один пример: макрос `with-all-cxrs` из *раздела Рекурсивные Решения* суб-лексически привязывает функции получения элементов списка. В результате работы суб-лексических привязок оказывается что к подобным привязкам невозможно сослаться из макро расширения. Иногда подобное ограничение полезно, иногда нет. В случае с `with-all-cxrs` суб-лексичность может рассматриваться как нежелательный эффект. Когда наш метод получения доступа к элементу списка находится в суб-лексической области видимости `with-all-cxrs` то проблем не возникает:

```
CL-USER> (with-all-cxrs
             (cadadadr nil))
NIL
```

Кроме этого мы можем даже написать макросы, расширяющиеся в эти методы получения доступа, поскольку макро определения находятся в суб-лексической области видимости `with-all-cxrs`:

```
CL-USER> (with-all-cxrs
             (macrolet ((accessor (l)
                          `(cadadadr ,l)))
               (accessor nil)))
NIL
```

Листинг 6.14: LET-BINDING-TRANSFORM

```
(defun let-binding-transform (bs)
  (if bs
      (cons
       (cond ((symbolp (car bs))
              (list (car bs)))
             ((consp (car bs))
              (car bs))
             (t
              (error "Bad let bindings")))
       (let-binding-transform (cdr bs)))))
```

Но, помните, что `with-all-cxrs` привязывает функции получения доступа суб-лексически, поэтому мы не можем инъецировать макрос, в функцию получения доступа:

```
CL-USER> (macrolet ((accessor (l)
                      `(cadadadr ,l)))
           (with-all-cxrs
               (accessor nil)))
The function COMMON-LISP-USER::CADADADR is undefined.
```

Теперь, когда мы знакомы с понятием *анафор* и увидели большое количество примеров сложных макросов — включая и те, которые используют суб-лексическую область видимости — мы можем обсудить интересный теоретический макрос: `sublet`. Этот макрос создаёт суб-лексические привязки кода, используя синтаксис, подобный синтаксису обычной `let` формы. Обсуждение `sublet` мы начнём также, как и обсуждения многих лисп макросов — с утилиты.

`Let-binding-transform` — это простая утилита, обрабатывающая случай когда `let` форма привязывает единственный символ. В следующем примере `a` нормализуется до `(a)`:

```
CL-USER> (let-binding-transform
          '(a (b) (c nil)))
((A) (B) (C NIL))
```

Кроме того, `sublet` использует утилиту `tree-leaves` определённую нами в *разделе Неявные Контексты*. Напомним, что макрос `tree-leaves`

Листинг 6.15: SUBLET

```
(defmacro sublet (bindings% &rest body)
  (let ((bindings (let-binding-transform
                   bindings%)))
    (setq bindings
          (mapcar
           (lambda (x)
             (cons (gensym (symbol-name (car x))) x))
           bindings))
    `(let (,@(mapcar #'list
                     (mapcar #'car bindings)
                     (mapcar #'caddr bindings)))
       ,@(tree-leaves
          body
          #1=(member x bindings :key #'cadr)
          (caar #1#)))))
```

получает три аргумента: произвольную списковую структуру, выражение, которое может использовать переменную `x` для определения изменяемого листа и другое выражение, которое может использовать другую `x` и определяющее на что должны быть изменены нужные листья.

Выбрав неявственность привязок `x` с одним и тем же именем мы получили преимущества *дуализма синтаксиса*. Если у нас не получается обычными способами выделить общий код в выражении, то иногда мы можем добиться сокращения кода воспользовавшись синтаксическим дуализмом. Определение `sublet` использует считывающий макрос ссылающийся сам на себя, эти макросы описаны в *разделе Циклические Выражения*. Особенно для таких концепций как функции получения доступа к элементам списка, которые могут меняться много раз в процессе написания кода, считывающие макросы позволяют нам работать только с одним-единственным отображением оператора получения доступа к элементу списка. Благодаря использованию *неявственности (implicitisation)* с макросом `tree-leaves`, легко найти и понять дублирование кода.

`Sublet` получает форму, представляющую `let` привязки и применяет нашу утилиту `let-binding-transform`, генерирующую в процессе новую списковую структуру. Далее происходит присоединение[14] `gensym`-а к

[14]Присоединяются, а не добавляются, поскольку в этом случае мы можем продолжать поддерживать привязки без значений по умолчанию, например `(a)`.

каждой привязке с печатью соответствующего имени для имени привязки. `Sublet` расширяется в `let` форму, привязывающую `gensym` символы к значениям, переданным в привязываемой форме, затем применяется `tree-leaves` для замещения всех вхождений имён символов привязок в переданном коде на их соответствующие `gensym`-ы. `Sublet` не раскрывает никаких макросов и не разбирает никаких специальных форм в теле на предмет поиска вхождений этих имён символов привязок поскольку `sublet` создаёт суб-лексические привязки. Например, если все ссылки на `a` являются суб-лексическими, то они будут замещены на `gensym`-ы:

```
CL-USER> (macroexpand
          '(sublet ((a 0))
            (list a)))
(LET ((#:A1214 0))
  (LIST #:A1214))
T
```

Однако, поскольку суб-лексическая область видимости не предполагает расширения макросов, и само собой разумеется отсутствие интерпретации таких специальных форм как `quote`, то экземпляры символа `a` не являющиеся ссылками на переменную также будут меняться:

```
CL-USER> (macroexpand
          '(sublet ((a 0))
            (list 'a)))
(LET ((#:A1215 0))
  (LIST '#:A1215))
T
```

Суб-лексическая область видимости вступает в силу до того как списковая структура будет интерпретирована как лисп код вашим системным проходчиком-по-коду. Это важное наблюдение — поскольку не все последствия этого явления исследованы до конца. Интерпретация кода `sublet`-ом отличается от интерпретации проходчика-по-коду, осуществляемого COMMON LISP-ом.

Теперь мы стоим на одной из многих граней понимания макросов. Сколько разновидностей интересных типов областей видимости лежит между нераскрытой суб-лексической областью видимости и полностью раскрытой лексической областью видимости? Поскольку более лучшего названия я не подобрал, то мы будем называть эту бесконечно большую

Листинг 6.16: SUBLET*

```
(defmacro sublet* (bindings &rest body)
  `(sublet ,bindings
           ,@(mapcar #'macroexpand-1 body)))
```

категорию областей видимости *супер суб-лексическими областями видимости (super sub-lexical scopes)*[15].

Весьма очевидно, что супер суб-лексическая область видимости использует `sublet*`. Этот макрос основан на `sublet` но изменяет каждую форму в теле осуществляя макрорасширение этих форм с помощью функции `macroexpand-1`. Теперь вместо появления в сырой списковой структуре ссылки на символы должны появляться после первого шага макро расширения. Этот тип супер суб-лексической области видимости позволяет макросам инъецировать или удалять ссылки из области видимости внутри тела `let` формы. Если макросы не выполняют эти операции — или если формы вовсе не являются макросами — то этот тип супер суб-лексической области видимости работает также, как и суб-лексическая область видимости:

```
CL-USER> (macroexpand
          '(sublet* ((a 0))
            (list a)))
(LET ((#:A1216 0))
  (LIST #:A1216))
T
```

Но мы можем определить другой макрос-инъектор для проверки этой супер суб-лексической области видимости:

```
CL-USER> (defmacro injector-for-a ()
           'a)
INJECTOR-FOR-A
```

`Sublet*` раскроет этот макрос-инъектор:

```
CL-USER> (macroexpand-1
          '(sublet* ((a 0))
```

[15] Я дал это глупое название по тому что ожидаю что по мере изучения этой концепции появятся более лучшие имена, глубже отображающие суть явления.

```
            (injector-for-a)))
(SUBLET ((A 0)) A)
T
```

Что в свою очередь будет суб-лексически интерпретироваться `sublet`-ом, обозначая присутствие инъецированной переменной внутри супер суб-лексической области видимости предоставляемой `sublet*`-ом:

```
CL-USER> (macroexpand-1 *)
(LET ((#:A1221 0))
  #:A1221)
T
```

Но вложенные макросы в выражении не раскрываются через **macroexpand-1** поэтому `sublet*` не вставляет их в суб-лексическую область видимости `sublet`:

```
CL-USER> (macroexpand-1
          '(sublet* ((a 0))
            (list (injector-for-a))))
(SUBLET ((A 0)) (LIST (INJECTOR-FOR-A)))
T
```

Поэтому `a` не захватывается суб-лексически[16]:

```
* (walker:macroexpand-all *)
(LET ((#:A1666 0))
     (LIST A))
```

С помощью `sublet` и `sublet*` мы можем, используя суб-лексические или супер суб-лексические области видимости, произвольно контролировать на каком уровне макро расширения переменная `a` будет рассматриваться как корректная. Как замечено выше супер суб-лексическая область видимости это почти полностью не исследованный бесконечный класс областей видимости. Способов прохода по коду — большое множество, существует такое же количество множеств супер суб-лексических областей видимости. Этот класс областей видимости подводит нас к другой категории почти неисследованных макросов: макросы, изменяющие работу лисп макросов. Эти макросы способны определять когда следует раскрывать макрос, где ссылки на переменные являются корректными, как интерпретировать специальные формы и т.д. То есть, мы получаем макро-программируемый раскрыватель макросов.

[16] `Walker:macroexpand-all` — это компонент CMUCL и представляет собой завершённый проходчик-по-коду.

6.7 Пандорические Макросы

Ящик Пандоры (Pandora's box) — это греческий миф о первой в мире женщине: Пандоре. Пандора, символ U-Языка переводится с греческого как "всем одарённая". Снедаемая любопытством Пандора открыла маленький ящик и тем самым безвозвратно выпустила на весь мир всё человеческое зло и грехи. Макросы, описанные в этом разделе являются очень мощными и они должны научить вас такому способу программирования, который вы никогда не забудете, и будьте уверены, что получившийся результат будет куда более лучшим чем то, что получила бедная Пандора. Откроем ящик.

Для начала мы отойдём от нашего повествования и ознакомимся с другой знаменитой книгой, посвящённой лиспу: *Lisp in Small Pieces* [SMALL-PIECES] за авторством Кристиана Кеннека (Christian Queinnec). Кеннек — это широко известный и уважаемый лисп эксперт и он внёс большой вклад в понимание лиспа. Книга Кеннека рассказывает о реализации различной сложности компиляторов и интерпретаторов в и для языка программирования Scheme[17].

В *Lisp in Small Pieces* есть короткая, но очень интересная дискуссия о макросах. Большая часть посвящена описанию различных вариаций макро системы возможной благодаря неоднозначности макро спецификации Scheme[18], но, кроме того, есть несколько интересных заметок о причинах использования макросов и способах их использования. Если вы прочитали и поняли *главу Основы Макросов*, то большинство макросов описанных в главе о макросах в *Lisp in Small Pieces*, для вас будут тривиальными, но, существует один интересный макрос, который нам стоит рассмотреть подробнее.

Как и многие книги, посвящённые программированию, *Lisp in Small Pieces* переносит и оставляет нас на реализации системы *объектно-ориентированного* программирования. Обычно эти реализации служат примером подмножества CLOS, Объектной Системы COMMON LISP. Кеннек называет это подмножество *MEROONET*. Кеннек отмечает, что при определении методов для MEROONET класса было бы хорошо иметь возможность прямо ссылаться к полям определяемого объекта вместо использования получателей доступа. Слова Кеннека (переведено) [SMALL-PIECES-P340-341]:

> Возьмём, к примеру, макрос `with-slots` из CLOS; мы адаптируем его к контексту MEROONET. Поля объекта — пусть

[17] Иногда через него описываются другие лиспы и их особенности.

[18] Спасибо, не надо.

> это будут поля экземпляра `Point` — обрабатываются с помощью чтения и записи через такие функции, как `Point-x` или `set-Point-y!`. Будет гораздо проще взаимодействовать с ними по имени их полей, `x` или `y`, например в контексте определения метода.

Ниже представлен пример желаемого Кеннеком интерфейса (который он назвал `define-handy-method`), определяющий новый метод **double**:

```
(define-handy-method (double (o Point))
  (set! x (* 2 x))
  (set! y (* 2 y))
  o )
```

Этот подход более удобен программистам нежели MEROONET синтаксис:

```
(define-method (double (o Point))
  (set-Point-x! o (* 2 (Point-x o)))
  (set-Point-y! o (* 2 (Point-y o)))
  o )
```

Другими словами: было бы неплохо использовать макросы для получения доступа к сторонним привязкам — в данном случае к слотам объекта — как если бы они были лексическими привязками. И хотя такой подход чрезвычайно удобен в целях сокращения кода, его главное следствие — это способность дать нам *дуализм синтаксиса* для уже существующих и будущих макросов.

Как замечает Кеннек, COMMON LISP реализует эту функциональность для CLOS с помощью макроса под названием `with-slots`. Это COMMON LISP-овский пример выполнения того, что было спроектировано: создание абстракций на базе чистой, стандартизированной макро системы. В то время, когда большинство языков спроектированы для того, чтобы быть элегантными, COMMON LISP спроектирован для того, чтобы быть мощным для программирования. Кеннек пришёл к выводу, что ограничения языка делают невозможным подобные ухищрения в Scheme, особенно там, где необходима переносимость:

> Из-за отсутствия информации о языке и его реализациях мы не можем написать переносимый проходчик-по-коду в Scheme, поэтому нам пришлось отказаться от написания **define-handy-method**.

Листинг 6.17: PANDORICLET

```
(defmacro pandoriclet (letargs &rest body)
  (let ((letargs (cons
                  '(this)
                  (let-binding-transform
                   letargs))))
    `(let (,@letargs)
       (setq this ,@(last body))
       ,@(butlast body)
       (dlambda
        (:pandoric-get (sym)
                       ,(pandoriclet-get letargs))
        (:pandoric-set (sym val)
                       ,(pandoriclet-set letargs))
        (t (&rest args)
           (apply this args))))))
```

Хотя COMMON LISP предоставляет большое количество корректных способов реализации макро систем, он спроектирован с учётом поддержки универсальных инструментов мета-программирования, которые способны работать как в стандартном, так и в переносимом режимах. Существует две мощных макро особенности COMMON LISP-а, позволяющих реализовывать такую функциональность как CLOS-овское `with-slots`, — это *обобщённые переменные (generalised variables)* и *символьные макросы (symbol macros)*. В этом разделе мы покажем вам слияние возможностей COMMON LISP-а и к какому замечательному результату это приведёт, кроме того, мы объединим воедино всё что мы уже узнали в отношении анафорических макросов и в процессе мы откроем интересный класс макросов под названием *пандорические макросы (pandoric macros)*.

Идея лежащая в основе `pandoriclet` — это *открытые замыкания*, дающие возможность доступа извне к их перекрытым лексическим переменным. Также как и предыдущие макросы, такие как `alet-hotpatch`, `pandoriclet` компилируется в косвенную среду, которая по разному реагирует на различные переданные аргументы.

Мы опять начали с *изнанки расширения* `alet`, не забывая о присутствии анафоры под названием `this`. `Pandoriclet` похож на уже виденные нами остальные макросы. Как и все наши анафорические варианты `let`, мы предполагаем что финальной формой тела `pandoriclet` будет лямбда форма. Как и `alet-hotpatch` `pandoriclet` использует макрос `dlambda`

для осуществления диспетчеризации между различными возможными участками исполняемого кода при возвращении замыкания из вызова `pandoriclet`. `Pandoriclet` использует утилиту `let-binding-transform`, представленную в предыдущем разделе для работы с `null` привязками — такими как `(let (a) ...)`. Эта функция утилита необходима для `pandoriclet` по той же причине, по которой она была необходима для `sublet`: эти макросы осуществляют проход-по-коду переданных привязок `let`, в то время как предыдущие макросы слепо сращивались с привязками в другой `let`.

У нас присутствует два вызова списко-создающих функций утилит, которые ещё не были определены: `pandoriclet-get` и `pandoriclet-set`, каждая из которых получает список `let` привязок. Мы можем ссылаться на ещё не определённые функции до расширения макроса что, очевидно, не произойдёт до тех пор, пока мы не используем макрос. Использование вспомогательных функций, облегчающих определение макросов — это хорошая привычка и эту привычку можно взять на вооружение. Она может не только улучшить читабельность ваших определений, но также может помочь при тестировании компонентов макроса и доказать пригодность использования в будущих макросах. Наиболее примечательная черта такой разновидности абстракций в том, что при комбинировании макросов мы продолжаем поддерживать доступность лексического контекста для остальных используемых утилит.

Поэтому мы написали `pandoriclet-get` и `pandoriclet-set` с учётом лексического контекста. `Pandoriclet-get` мы написали с учётом того, что `dlambda` привязывает переменную `sym` к сращиванию нашего списка. Дальше `case` форма производит сравнение `sym` с символами, переданными в `pandoriclet`[19]. Если мы найдём символ, то будет возвращено значение на которое ссылается привязка. Если нет, то будет сгенерирована ошибка. `Pandoriclet-set` похож на `pandoriclet-get`, с той разницой, что `dlambda` привязывает ещё один используемый символ: `val`. `Pandoriclet-set` использует `setq` для изменения привязок в зависимости от `sym` на `val`.

`Pandoriclet` предоставляет тот-же интерфейс, что и все наши анафорические реализации `let`, поэтому мы можем использовать его для создания простого замыкания-счётчика:

```
CL-USER> (setf (symbol-function 'pantest)
       (pandoriclet ((acc 0))
    (lambda (n) (incf acc n))))
```

[19]Напомним, что `case` с символами компилируется в сравнение с одним указателем для каждого `case` случая.

Листинг 6.18: PANDORICLET-ACCESSORS

```
(defun pandoriclet-get (letargs)
  `(case sym
     ,@(mapcar #`((,(car a1)) ,(car a1))
               letargs)
     (t (error
         "Unknown pandoric get: ~a"
         sym))))

(defun pandoriclet-set (letargs)
  `(case sym
     ,@(mapcar #`((,(car a1))
                  (setq ,(car a1) val))
               letargs)
     (t (error
         "Unknown pandoric set: ~a"
         sym val))))
```

```
#<CLOSURE (LAMBDA (&REST #:ARGS2)) {C452955}>
```

[20]

Что работает так, как и задумывалось:

```
CL-USER> (pantest 3)
3
CL-USER> (pantest 5)
8
```

Однако, сейчас у нас есть прямой доступ к привязке под названием `acc`, с помощью которого было создано замыкание:

[20]Примечание переводчика: При вычислении этого выражения выводится такое предупреждение о стиле:

```
;
; caught STYLE-WARNING:
; Too many arguments (2) to ERROR Unknown pandoric set:  a": uses at most 1."
; See also:
; The ANSI Standard, Section 22.3.10.2
;
; compilation unit finished
; caught 1 STYLE-WARNING condition
```

```
CL-USER> (pantest :pandoric-get 'acc)
8
```

И мы можем также просто поменять значение этой привязки:

```
CL-USER> (pantest :pandoric-set 'acc 100)
100
CL-USER> (pantest 3)
103
```

Здесь доступно даже значение анафоры `this`, поскольку мы преднамеренно оставили анафору открытой и добавили символ `this` в список привязок `letargs` при раскрытии макроса:

```
CL-USER> (pantest :pandoric-get 'this)
#<CLOSURE (LAMBDA (N)) {C452945}>
```

Поэтому это замыкание, которое мы создали с помощью `pandoriclet`, — больше не является закрытым. Среда используемая этим замыканием — даже если все лексические символы значений были удалены компилятором — по прежнему доступны через нашу анонимную функцию, возвращённую от `pandoriclet`. Как это работает? В случае с пандорическими макросами дополнительный код компилируется так, чтобы предоставить способ получения доступа к замыканию снаружи. Но мощь пандорических макросов невозможно понять изучая низкоуровневую картину происходящего. То что мы создали называется *протоколом взаимодействия замыканий (inter-closure protocol)* или системой передачи сообщений, предназначенной для коммуникации между замыканиями.

Прежде чем мы продолжим изучение пандорических макросов, мы должны уделить внимание одной из наиболее важных примеров *двойственности синтаксиса* в COMMON LISP: *обобщённым переменным*. Детали обобщённых переменных сложны и я бы не хотел описывать их все здесь. Для этого я рекомендую книгу Грэма *On Lisp* — лучшую книгу, посвящённую разбору этой темы. В конечном счёте всё сводится к простой идее: доступ к обобщённой переменной синтаксически двойственен установлению её значения. У вас есть только одна, устанавливающая значения форма, `setf`, которая может устанавливать все типы переменных используя тот же синтаксис, который вы используете для получения доступа к ним.

Например, при работе с обычной переменной вы традиционно получаете её значение через её символ, к примеру, `x`. Для присваивания какого либо значения вы можете использовать `(setf x 5)`. Аналогично, для

Листинг 6.19: GET-PANDORIC

```
(declaim (inline get-pandoric))

(defun get-pandoric (box sym)
  (funcall box :pandoric-get sym))

(defsetf get-pandoric (box sym) (val)
  `(progn
     (funcall ,box :pandoric-set ,sym ,val)
     ,val))
```

получения доступа к `car` слоту `cons` ячейки с названием, скажем, `x` вы используете `(car x)`. Для присвоения значения вы можете использовать форму `(setf (car x) 5)`. Такой способ скрывает тот факт, что правильный способ изменить данные `cons` ячейки — это использование функции `rplaca`. Реализовав подобную двойственность синтаксиса мы сократили более чем половину операторов получения доступа/присваивания значений и теперь нет нужды их всех запоминать, и что ещё более важно, у нас появляются новые способы использования макросов.

Функция `get-pandoric` — это обёртка вокруг получения синтаксиса протокола взаимодействия замыканий. Она объявлена встраиваемой для того, чтобы устранить любые влияния на производительность, которые может оказывать эта обёртка.

`Defsetf` — это интересный COMMON LISP макрос, в отличие от нашего `defmacro!`, который расширяется в `defmacro` и неявно привязывает `gensym`-ы вокруг переданной формы. `Defsetf` прекрасно работает для определения присваивающей стороны обобщённой переменной если присваиватель можно выразить как функцию или макрос, который вычисляет все свои аргументы только один раз. `Get-pandoric` можно было бы определить как макрос, но мы этого не сделали из-за встраивания. Макросы не для встраивания, компиляторы для встраивания.

Вернёмся к нашему пандорическому счётчику, сохранённому в символ-функцию `pantest`, мы можем использовать эту новую функцию-получатель для получения текущего значения `pantest`-овской привязки `acc`:

```
CL-USER> (get-pandoric #'pantest 'acc)
103
```

Теперь благодаря обобщённым переменным и `defsetf` мы можем использовать синтаксический дуализм и поменять значение переменной:

```
CL-USER> (setf (get-pandoric #'pantest 'acc) -10)
-10
CL-USER> (pantest 3)
-7
```

Среды над функциями — вот что подразумевает `let` в *let, окружающий lambda*, — начинают выглядеть как просто обобщённые переменные с несложным доступом, такие как `cons` ячейка или элемент хэш таблицы. Замыкания стали ещё более *первоклассными* структурами данных чем в начале. Привязки которые были в начале закрыты для внешнего кода теперь широко доступны для разнообразных операций, даже если эти привязки были скомпилированы для чего-то другого и имеют забытые символы доступа.

Но, любая дискуссия о обобщённой переменной будет неполной без упоминания её ближайшего родственника: *символьного макроса (symbol macros)*. `Symbol-macrolet`, как следует из названия, позволяет нам расширять символы в общие лисп формы. Поскольку интуитивной и наиболее гибкой в использовании формой, представляющей макро трансформации[21], является рассмотрение формы как функции, то **symbol-macrolet** главным образом предназначен для одной цели: символьные макросы дают нам возможность скрывать обобщённые переменные таким образом, что пользователи макроса будут считать что получают доступ к обычным лексическим переменным.

Введение символьных макросов привело к появлению одного из наиболее странных *костылей (kludges)* в языке COMMON LISP: обычно при присваивании значения переменной через обычный символ, например `(setf x t)`, `setf` будет расширяться в форму `setq`, поскольку это именно то, для чего и был предназначен `setq`: присваивание значений лексическим и динамическим переменным (на которые всегда будут ссылаться по символам). Поскольку специальная форма `setq` не может присваивать значения обобщённым переменным, но, при введении символьных макросов стало возможным выражать с помощью символов не только лексические/динамические привязки, но и любые обобщённые переменные, и тут появилась необходимость в конвертации `setq` форм обратно в `setf` формы. Как это ни странно, но это было *правильным* решением, поскольку такой подход позволяет макросам полностью скрывать присутствие обобщённых переменных от пользователя макроса, даже если он решит использовать `setq`. По *настоящему правильным* решением было бы удаление избыточной `setq` из языка в пользу более универсального `setf`, но,

[21] Символьные макросы не получают аргументов, поэтому определение символьного макроса всегда расширяется в самого себя.

Листинг 6.20: WITH-PANDORIC

```
(defmacro! with-pandoric (syms o!box &rest body)
  `(symbol-macrolet
       (,@(mapcar #`(,a1 (get-pandoric ,g!box ',a1))
                  syms))
     ,@body))
```

это не произойдёт во-первых, по очевидным причинам совместимости, а во-вторых, поскольку при создании макроса `setq` можно использовать в целях безопасности — `setf` плюс проверка на то что символ был сращен и не представляет списковую форму. При использовании `setq` помните что он полезен только для обеспечения безопасности сращивания; как мы уже видели, благодаря `symbol-macrolet` символ может ссылаться на любую обобщённую переменную.

Макрос `with-pandoric` расширяется в `symbol-macrolet`, определяющий символьные макросы для каждого символа содержащегося в `syms`. Каждый символьный макрос будет расширять ссылки на его символ в лексическую область видимости `symbol-macrolet` в ссылки обобщённой переменной используя `get-pandoric` для получения доступа к результатам вычисления второго аргумента макроса: `o!box` (сохраняется в `g!box`).

Таким образом `with-pandoric` позволяет нам заглянуть в привязку переменных, через закрывающее их замыкание:

```
CL-USER> (with-pandoric (acc) #'pantest
                        (format t "Value of acc: ~a~%" acc))
Value of acc: -7
NIL
```

Поскольку в нашей архитектуре используются обобщённые переменные для формирования синтаксической двойственности, дающее нам возможность получения и присваивания значений, то мы можем считать что имеем дело с обычной лексической переменной и присваивать ей значения с помощью `setq`:

```
CL-USER> (with-pandoric (acc) #'pantest
                        (setq acc 5))
5
CL-USER> (pantest 1)
6
```

Листинг 6.21: PANDORIC-HOTPATCH

```
(defun pandoric-hotpatch (box new)
  (with-pandoric (this) box
                 (setq this new)))
```

Мы только что познакомились с основными составными частями пандорических макросов. Первое — макрос, создающий замыкания: **pandoriclet**, захватывающий анафору `this`, ссылающуюся на функцию, использованную при вызове замыкания. Этот макрос компилируется в некоторый специальный код, который перехватывает определённые вызовы этого замыкания и получает доступ или модифицирует его перекрытые лексические переменные. Второе — единый синтаксис и для получения доступа и для присваивания нового значения реализованный с помощью `get-pandoric` и `defsetf`. И, наконец, макрос `with-pandoric` использующий `symbol-macrolet` для закладки обобщённых переменных выглядящих как новые лексические переменные с теми же именами что и перекрытые переменные. Эти переменные ссылаются на исходную среду, созданную с помощью `pandoriclet`, но из отдельных лексических контекстов.

В качестве примера мы используем эту способность открывать замыкания и сравним их с макросами *горячей замены* из *раздела Замыкания, Поддерживающие Горячую Замену*. Напомним, что `alet-hotpatch` и его близкий анафорический двоюродный брат `let-hotpatch` создают замыкания с косвенной средой, таким образом функции, вызываемые при вызове замыкания могут изменяться на лету. Главное ограничение связанное с этими макросами заключается в том, что они принуждают вас выбрасывать все лексические привязки, перекрывающиеся предыдущей анонимной функцией при её горячей замене. Это было неизбежно поскольку во время написания этих макросов замыкания были закрыты для нас.

При работе с `alet-hotpatch` и `let-hotpatch` мы должны были компилировать код специального назначения для каждого замыкания, которое было способно присвоить анафорической лексической привязке `this` новое значение. Но поскольку мы можем открыть замыкание определённое с `pandoriclet` и запустить внешний присваивающий код, то теперь возможно определить функцию горячей замены `pandoric-hotpatch`, которая способная работать с любыми пандорическими замыканиями.

Иногда абстракции лишь кажутся правильными и трудно сказать почему. Возможно, поскольку в большинстве случаев программирование —

Листинг 6.22: PANDORIC-RECODE

```
(defmacro pandoric-recode (vars box new)
  `(with-pandoric (this ,@vars) ,box
                  (setq this ,new)))
```

это негармоничная комбинация несовместимых частей, и удивительно и приятно обнаружить абстракции, которые — как бы случайно — замечательно комбинируются вместе. `Pandoric-hotpatch` работает также, как и читается: открывает пандорический интерфейс, получает переменную `this` из лексической области замыкания `box` и затем использует `setq` для присваивания `this` нового заменяющего замыкания, `new`.

Мы можем применить `pandoric-hotpatch` на пандорических замыканиях которые мы создали ещё до того как задумались о горячей замене. Помните замыкание-счётчик с которым мы упражнялись на протяжении этого раздела? Это замыкание должно быть по прежнему привязано к `symbol-function` символа `pantest`. Счётчик был установлен в 6:

```
CL-USER> (pantest 0)
6
```

Давайте внедрим новое замыкание — такое, которое имеет новую привязку к `acc` начинающуюся со 100 и является уменьшающимся:

```
CL-USER> (pandoric-hotpatch #'pantest
                            (let ((acc 100))
                              (lambda (n) (decf acc n))))
#<CLOSURE (LAMBDA (N)) {C67F6ED}>
```

Для того, чтобы убедиться, что горячая замена прошла хорошо:

```
CL-USER> (pantest 3)
97
```

Итак, наше замыкание-счётчик обладает новым значением, привязанным к `this`, которое используется для осуществления подсчёта. Однако, изменяет ли данная горячая замена пандорическое значение привязки `acc`?

```
CL-USER> (with-pandoric (acc) #'pantest
                        acc)
6
```

Листинг 6.23: PLAMBDA

```
(defmacro plambda (largs pargs &rest body)
  (let ((pargs (mapcar #'list pargs)))
    `(let (this self)
       (setq
        this (lambda ,largs ,@body)
        self (dlambda
              (:pandoric-get (sym)
                             ,(pandoriclet-get pargs))
              (:pandoric-set (sym val)
                             ,(pandoriclet-set pargs))
              (t (&rest args)
                 (apply this args)))))))
```

Нет. `Acc` по прежнему равен предыдущему значению, 6, происходит это по той причине, что мы меняем только одну привязку `this` в пандорическом окружении, а `this` меняется только на новое замыкание в его собственной привязке `acc`.

Макрос `pandoric-recode` применяет несколько другой подход реализации горячей замены. Он сохраняет исходную лексическую среду кода и в то же время продолжает управлять изменением функции, исполняемой при вызове замыкания и компилируется извне. Звучит слишком хорошо, чтобы быть правдой? Вспомните, что текущее значение `acc` равняется 6 в исходной пандорической среде, и мы можем использовать `pandoric-recode` для размещения новой функции, использующей исходное значение и, скажем, уменьшать счётчик на половину от переданного значения `n`:

```
CL-USER> (pandoric-recode (acc) #'pantest
                          (lambda (n)
                            (decf acc (/ n 2))))
#<CLOSURE (LAMBDA (N)) {B186EB5}>
```

Убедимся в том, что у нас есть новое поведение, которое уменьшит `acc` на `(* 1/2 2)` с 6 до 5:

```
CL-USER> (pantest 2)
5
```

И связано ли это значение с исходной пандорической привязкой?

```
CL-USER> (with-pandoric (acc) #'pantest
                        acc)
5
```

Да. Как работает `pandoric-recode`? Этот макрос перекрывает переданную лямбда форму пандорически открытыми привязками исходного замыкания.

Макрос, который мы до сих пор использовали для создания пандорических замыканий называется `pandoriclet`. `Plambda` — это переписанная изнанка `pandoriclet` добавляющая несколько важных особенностей. Первое и самое главное — `plambda` не создаёт `let` среды используемые в наших пандорических методах доступа. Вместо этого `plambda` получает список символов ссылающихся на переменные, наличие которых подразумевается в лексической среде вызывателя макроса. `Plambda` может *экспортировать* любые переменные в вашу лексическую среду, делая их прозрачно доступными для других лексических областей видимости — даже если они были написаны и скомпилированы до или после формы `plambda`.

Это следующее улучшение нашей системы замыканий *let, окружающий lambda*, спроектированное с упором на максимальное усиление синтаксической двойственности. Благодаря пандорическим макросам, наиболее важными из которых являются `plambda` и `with-pandoric`, мы можем при желании легко и просто выйти за границы лексической области видимости. Замыкания больше не закрыты; мы можем открыть замыкания также легко, как и переписать наши лямбда формы в `plambda` формы. Мы используем `plambda` для экспорта лексических переменных и `with-pandoric` для их импорта, в результате чего они становятся полностью эквивалентными лексическим переменным. На самом деле эти новые переменные настолько эквиваленты, что их уже нельзя рассматривать как новые переменные. Лучший способ думать о пандорических переменных — это просто считать их расширением исходной лексической области видимости. Простой пример использования `plambda` — это пандорический счётчик, экспортирующий переменные из двух потенциально различных лексических сред:

```
CL-USER> (setf (symbol-function 'pantest)
               (let ((a 0))
                 (let ((b 1))
                   (plambda (n) (a b)
                            (incf a n)
                            (setq b (* b n))))))
#<CLOSURE (LAMBDA (&REST #:ARGS2)) {B529715}>
```

Листинг 6.24: MAKE-STATS-COUNTER

```
(defun make-stats-counter
    (&key (count 0)
     (sum 0)
     (sum-of-squares 0))
  (plambda (n) (sum count sum-of-squares)
           (incf sum-of-squares (expt n 2))
           (incf sum n)
           (incf count)))
```

Заметьте, насколько легко было экспортировать эти лексические указатели. Создание пандорического замыкания также легко как и добавление символа `p` перед `lambda` и добавление списка экспортируемых переменных после лямбда аргументов. Мы можем открыть это замыкание — и любое пандорическое замыкание, которое экспортирует символы `a` и `b` — используя `with-pandoric`:

```
CL-USER> (defun pantest-peek ()
           (with-pandoric (a b) #'pantest
                          (format t "a=~a, b=~a~%" a b)))
PANTEST-PEEK
CL-USER> (pantest-peek)
a=0, b=1
NIL
```

`Plambda` является примером того, насколько полезным может быть исключение общих компонентов макро расширений. Помните, когда мы писали `pandoriclet` мы решили переместить создание `case` условий для кода выполняющего получение доступа на функцию `pandoriclet-get`, а код, отвечающий за присваивание на `pandoriclet-set`? `Plambda` использует эти же функции. Даже если эти макросы сращивают результаты этих функций в абсолютно других лексических контекстах, но поскольку макросы написаны с соблюдением того же соглашения именования и протокола взаимодействия замыканий, то этот код пригоден к комбинированию с другим кодом.

Итак, пандорические макросы разрушают лексические границы. Они позволяют вам открывать замыкания по первому желанию и представляют прекрасное слияние различных особенностей языка COMMON LISP: анафорические макросы, обобщённые переменные и символьные макросы. Но какую пользу можно извлечь из них?

Листинг 6.25: DEFPAN

```
(defmacro defpan (name args &rest body)
  `(defun ,name (self)
     ,(if args
          `(with-pandoric ,args self
                          ,@body)
          `(progn ,@body))))
```

Пандорические макросы важны по той причине, что они дают нам все основные преимущества таких объектных систем как CLOS не требуя от нас отказа от более естественного стиля программирования, основанного на комбинациях `let-lambda`. В частности мы можем добавлять функциональность или методы для замыканий без необходимости создавать экземпляры экземпляров уже созданных объектов.

`Make-stats-counter` — это *lambda, окружающая let, окружающий p-lambda*, предназначенная для создания счётчиков, содержащих три единицы информации. В дополнение к сумме (`sum`), есть сумма квадратов (`sum-of-squares`) и число обработанных шагов (`count`). Если бы мы использовали `lambda` вместо `plambda` в определении `make-stats-counter`, то большинство этой информации не было бы нам доступно. Мы находились бы снаружи и эти переменные были бы закрыты для нас.

Как мы напишем пандорические методы? Мы можем просто получить доступ к переменным используя `with-pandoric` так, как это было показано выше, или, поскольку это лисп, спроектировать более узкоспециализированный интерфейс.

`Defpan` — это *комбинация* `defun` и макроса `with-pandoric`. Главная цель `defpan` — это реализация *двойственности синтаксиса* между функцией, написанной с помощью `defun` и доступом к сторонней лексической области видимости с помощью `with-pandoric`. И хотя мы передаём аргументы `defpan` в том же синтаксисе что и в лямбда формах — список символов — аргументы `defpan` обозначают нечто другое. Вместо создания новой лексической среды эти *пандорические функции* расширяют лексическую среду пандорических замыканий к которым они применяются. В случае с `defun` и обычными лямбда формами имя (символ) которое вы даёте переменной не играет важной роли. В случае с пандорическими функциями имя — это всё. Кроме того, в случае с пандорическими функциями порядок аргументов не имеет значения и количество используемых экспортируемых лексических переменных зависит от вашего желания.

Листинг 6.26: STATS-COUNTER-METHODS

```
(defpan stats-counter-mean (sum count)
  (/ sum count))

(defpan stats-counter-variance
    (sum-of-squares sum count)
  (if (< count 2)
      0
      (/ (- sum-of-squares
            (* sum
               (stats-counter-mean self)))
         (- count 1))))

(defpan stats-counter-stddev ()
  (sqrt (stats-counter-variance self)))
```

Кроме того `defpan` предоставляет анафору под названием `self`, которая позволяет нам применять технику под названием анафорическая очередизация. Невидимо передавая значение `self` между пандорическими функциями мы можем управлять значением этой анафоры с помощью очередей из вызовов функций. Как и со всеми конструкциями очередей, убедитесь что очередь не заканчивается бесконечным циклом.

Мы написали три метода, которые могут быть использованы в наших замыканиях, созданных с помощью `make-stats-counter` или любом другом пандорическом замыкании, экспортирующем необходимые имена переменных. `Stats-counter-mean` просто возвращает среднее значение всех значений, переданных замыканию. `Stats-counter-variance` вычисляет дисперсию этих значений следуя ссылкам в очереди, а **stats-counter-stddev** применяется для того, чтобы вычислить стандартное отклонение. Обратите внимание что каждая ссылка в очереди нужна для того, чтобы передать анафору `self` для того, чтобы сослаться на полный лексический контекст замыкания. Мы видим, что отдельной пандорической функции нужно только сослаться на используемые переменные и на эти переменные можно ссылаться в том порядке, в каком нам заблагорассудится.

Итак, `plambda` создаёт другую анафору — `self`. В то время когда анафора `this` ссылается на замыкание, которое будет вызвано, `self` ссылается на косвенную среду, вызвавшую это замыкание. И хотя это прозвучит несколько странно, код внутри нашей `plambda` может использовать

Листинг 6.27: MAKE-NOISY-STATS-COUNTER

```
(defun make-noisy-stats-counter
    (&key (count 0)
     (sum 0)
     (sum-of-squares 0))
  (plambda (n) (sum count sum-of-squares)
           (incf sum-of-squares (expt n 2))
           (incf sum n)
           (incf count)

           (format t
                   "~&MEAN=~a~%VAR=~a~%STDDEV=~a~%"
                   (stats-counter-mean self)
                   (stats-counter-variance self)
                   (stats-counter-stddev self))))
```

`self` для пандорического доступа к своей собственной лексической среде, вместо использования прямого доступа к ней. Пока что может показаться что такой трюк полезен для `defpan` методов, написанных для работы внутри нашей лексической области видимости.

`Make-noisy-stats-counter` идентичен `make-stats-counter` с тем исключением, что использует анафору `self` для вызова наших `defpan` функций `stats-counter-mean`, `stats-counter-variance` и **stats-counter-stddev**.

`Plambda` и `with-pandoric` могут перезаписать лексическую область видимости так, как мы этого захотим. Мы завершим эту главу таким примером. К одним из ограничений лексической области видимости иногда относят тот факт, что функция COMMON LISP `eval` отбрасывает нашу текущую лексическую среду при вычислении переданной ей формы. Другими словами, `eval` вычисляет форму в *null лексической среде*. В COMMON LISP иначе и не может быть: `eval` — это функция. Вот такая проблема:

```
CL-USER> (let ((x 1))
           (eval
            '(+ x 1)))
The variable X is unbound.
```

Иногда бывает желательно расширить вашу лексическую среду на `eval`. Но будьте осторожны. Весьма часто использование `eval` говорит о

Листинг 6.28: PANDORIC-EVAL

```
(defvar pandoric-eval-tunnel)

(defmacro pandoric-eval (vars expr)
  `(let ((pandoric-eval-tunnel
          (plambda () ,vars t)))
     (eval `(with-pandoric
                ,',vars pandoric-eval-tunnel
                ,,expr))))
```

том, что вы, возможно, делаете что-то не так. Злоупотребление функцией `eval` может привести к замедлению программ поскольку `eval` может быть очень ресурсоёмкой операцией — в основном, это происходит из необходимости расширять макросы присутствующие в переданной форме. Если вы вдруг обнаружили необходимость в `eval` при программировании, спросите себя почему вы не сделали то, что вам нужно сделать гораздо раньше. Если ответом будет: “Раньше я не мог сделать это”, скажем, поскольку вы только что считали форму, то, примите поздравления, вы обнаружили один из редких правомерных причин использования `eval`. Все другие ответы ведут к тому способу, который вы должны были применить в первую очередь: к макросу.

Но, предположим, вам действительно нужно использовать `eval`, если бы только можно было взять с собой этот надоедливый лексический контекст. Макрос `pandoric-eval` — это пример интересного применения `plambda` и `with-pandoric`. `Pandoric-eval` использует специальную переменную под названием `pandoric-eval-tunnel` для того, чтобы сделать пандорическое замыкание доступным для функции `eval` через динамическую среду. Выбор лексических переменных предназначенных для *тунелирования* через динамическую среду осуществляется через передачу списка с их символами в качестве первого аргумента `pandoric-eval`. Вот вышеприведённый пример, но уже с использованием `pandoric-eval`:

```
CL-USER> (let ((x 1))
           (pandoric-eval (x)
                          '(+ 1 x)))
2
```

И выражение вычисленное с помощью `pandoric-eval` может модифицировать исходную лексическую среду; `pandoric-eval` — это двухсторонний туннель:

```
CL-USER> (let ((x 1))
           (pandoric-eval (x)
                          '(incf x))
           x)
2
```

В этом разделе, хотя он был очень длинный, мы всего лишь вскользь прошлись о возможностях пандорических макросов и различных их вариациях. И я ожидаю большого количества различных интересных разработок, которые разовьются из них.

Упражнение: Может ли `pandoric-eval` вызываться вложенным? То есть, можно ли `pandoric-eval` применять для вычисления форм, вычисляющих `pandoric-eval`? Объясните ответ.

Упражнение: Хотя реализация пандорических макросов в этом разделе весьма эффективна, она может быть улучшена. Попробуйте заменить `pandoriclet-get` и `pandoriclet-set` на генерацию кода, использующего хэш-таблицу вместо `case` и замерьте производительность этих двух реализаций на малых и больших числах пандорических переменных. Рассмотрите вашу любимую реализацию CLOS, сымитируйте диспетчеризацию CLOS и замерьте производительность.

Глава 7

Темы эффективности макросов

7.1 Лисп быстр

> Если вы облицовываете пол плиткой, размер которой с ноготь,
> вы не тратите излишних усилий.
>
> — Пол Грем

Некоторые люди думают что лисп — это медленный язык. Это высказывание может быть правдивым для некоторых ранних реализаций лиспа, но явно ложное в наши дни. На самом деле такие современные лиспы, как COMMON LISP, спроектированы так, чтобы мы могли использовать макросы и тем самым ускорять лисп. По-настоящему ускорять. Основная цель этой главы должна удивить вас, если вы верите в *миф о производительности*, который гласит, что низкоуровневые языки более эффективны, чем лисп. Эта глава демонстрирует, что лисп может быть более быстрым чем любые другие языки программирования, и что такие низкоуровневые языки как C на самом деле — по причине отсутствия макросов — уступают лиспу в производительности. Лисп позволяет нам писать код более эффективный чем код, написанный на других языках. Особенно при разработке больших и сложных программ, макросы позволяют создавать код, превосходящий в производительности Блаб языки. Иногда языки проектируются с упором на эффективные реализации, но чаще они проектируются с расчётом предоставления максимальной выразительности для программиста. Когда мы выбираем выразительность, то здесь лисп оказывается более быстрым. По-настоящему быстрым.

В то время когда остальные языки дают вам маленькие, квадратные плитки, лисп даёт вам плитки любого размера и любых форм. В C программисты всегда используют язык, напрямую привязанный к железу.

Помимо процедур и структур в C возможны некоторые небольшие абстракции. В отличие от C лисп никогда не был привязан к возможностям и ограничениям машин.

Но конечно другие языки могут быть написаны в менее эффективном, но более удобном способе. Например Perl позволяет программисту творить чудеса с помощью одного короткого выражения, но, помимо этого, содержит в себе много способов улучшения быстродействия кода. Так о чём мы говорим, когда рассуждаем о том, что лисп позволяет нам контролировать эффективность наших абстракций так, как это не в состоянии сделать ни один язык? Как вы можете ожидать, ответ заключается в теме нашей книги: макросы.

Вместо того, чтобы спрашивать "что позволяет ускорить программы?", лучше спросить "что замедляет программы?". Это один из наиболее исследуемых тем в программировании. Причины можно условно разделить на три широких категорий: плохие алгоритмы, плохие структуры данных и код в целом.

Все реализации языков нуждаются в хороших алгоритмах. Алгоритм — это предположительно хорошо исследованное процедурное описание исполнения программной задачи. На разработку алгоритма расходуется гораздо больше усилий, чем на их реализацию, поэтому алгоритмы используются повсеместно во всех компьютерных науках. Кто-то уже описал как, почему и как быстро работают алгоритмы; всё, что вам нужно — это взять алгоритм и перевести его *псевдо-код* на что-то, что понимает ваша система. Поскольку реализации COMMON LISP, как правило, хорошо реализовывались умными людьми и последовательно улучшались на протяжении десятилетий, то, в целом, в таких реализациях используются самые лучшие и самые быстрые алгоритмы для наиболее часто решаемых задач. Например CMUCL использует специальную реализацию сортировки кучи для сортирования списков и векторов и алгоритм Вихря Мерсенна 19937 и его огромный период[1] для генерирования случайных чисел[2].

Хорошие структуры данных также необходимы для любого приличного языка программирования. Структуры данных очень важны, и их игнорирование может привести к тому, что любой язык, созданный без должных структур данных, будет еле ползти. Оптимизация структур данных, по сути, сводится к концепции под названием *локальности (locality)*. Описание концепции очень простое — доступ к наиболее часто ис-

[1] `(1- (expt 2 19937))`

[2] Последовательность MT19937 порождается линейной рекурсией и, поэтому, не пригодна для криптографии.

пользуемым данным должен осуществляться как можно быстрее. Структуры данных и локальность настолько важны, что их можно обнаружить почти во всех вычислениях, где важна производительность: большие наборы CPU регистров, кэши памяти, базы данных и кэширующие сетевые прокси — это всего лишь некоторые примеры. Лисп предлагает большой набор стандартных структур данных с очень хорошей реализацией. Хэш таблицы, связанные списки (очевидно), векторы с заполняемыми указателями, пакеты с *добавляемыми (internable)* символами и другие более специфицированные структуры данных и их преимущества доступны для COMMON LISP программистов.

Если лисп предоставляет такие хорошие алгоритмы и структуры данных, то по какой причине лисп код может быть медленнее, чем код в других языках? Объяснение основывается на наиболее важном архитектурном решении лиспа: *обобщённом коде (general code)*, концепции так или иначе знакомой нам как *дуализм синтаксиса*. При написании лисп кода мы используем столько дуализма, насколько это возможно. Сама структура языка поощряет нас к этому. Одной из причин, по которой лисп программы обычно гораздо короче, чем Блаб программы, является возможность гораздо большего повторного использования лисп кода по сравнению с соответствующим Блаб кодом. Исходя из точки зрения Блаб языка, мы сталкиваемся с довольно необычным явлением: *писать больше, получать меньше (to write more to get less)*, но мы говорим о важном решении в лисп архитектуре — дуализме синтаксиса. Чем больше дуализма содержится в каждом выражении, тем короче будет программа. Означает ли это, что для достижения и превышения производительности C мы должны сделать наши программы такими же длинными и опасными, как и соответствующие C программы? Нет. В лиспе есть макросы.

7.2 Макросы Ускоряют Лисп

Этот раздел показывает примеры использования трёх типов макросов, помогающих создавать эффективные программы: обычные макросы, считывающие макросы и новый тип макросов, вводимых в этой главе: *компилирующие макросы (compiler macros)*.

Макросы могут использоваться для контроля алгоритмов, структур данных, проверки типов, проверки безопасности, уровней оптимизации кода или участков кода и многого другого. Мы можем получить безопасный и обобщённый код программы — или даже функции — работающий со скоростью быстрого и опасного кода. Если кратко: ни один язык не предоставляет такого открытого интерфейса для контролирова-

ния компиляторов так, как это делает лисп, и всё это благодаря (чему же ещё?) макросам. Из беглого чтения стандарта ANSI можно узнать, что макросы и *декларации*, наиболее короткий способ взаимодействия с компилятором, не совсем хорошо работают вместе:

> *Макро формы не могут расширяться в декларации; выражения деклараций должны появляться только как действительные подвыражения форм, к которым они относятся.*

Из ANSI следует, что следующий макрос не будет работать так, как было задумано:

```
(defmacro go-fast () ; Не исправный код!
  '(declare (optimize (speed 3) (safety 0))))
```

Мы не можем вставить вызов макроса туда, где ожидаются декларации. Можно думать и так: системный проходчик-по-коду не требует расширения макросов в телах специальных форм перед их проверкой на декларации. Потребность ускориться возникает очень часто, поэтому мы можем попробовать изобрести что-нибудь получше чем вышеприведённый макрос **go-fast**. Если мы хотим сжать какой-либо смысл как можно сильнее, то довольно часто мы можем прибегнуть к *считывающим макросам*. Считывающие макросы также удобны для расширения в декларации, поскольку они расширяются задолго до того, как проходчик-по-коду приступит к проходу по коду. Они считываются как действительные подвыражения.

Sharp-f — это считывающий макрос, который можно использовать для контроля наиболее важных компромиссов производительности в программах на COMMON LISP: баланс между декларациями **speed (скорость)** и **safety (безопасность)**. Например, сам sharp-f считывается в то, во что должно было бы расшириться **go-fast**:

```
* '#f

(DECLARE (OPTIMIZE (SPEED 3) (SAFETY 0)))
```

Но мы можем изменить это поведение и объявить значение безопасности, передав число меньшее, чем 3, в качестве аргумента считывателя. Все диспетчеризующие считывающие макросы могут принимать подобные числовые аргументы. В функцию считывающего макроса этот аргумент передаётся как третий аргумент, который часто называется как **numarg**. В примере ниже мы ставим безопасность превыше скорости передав 0:

Листинг 7.1: SHARP-F

```
(set-dispatch-macro-character #\# #\f
   (lambda (stream sub-char numarg)
      (declare (ignore stream sub-char))
      (setq numarg (or numarg 3))
      (unless (<= numarg 3)
(error "Bad value for #f: ~a" numarg))
      `(declare (optimize (speed ,numarg)
  (safety ,(- 3 numarg))))))
```

Листинг 7.2: FAST-PROGN

```
(defmacro fast-progn (&rest body)
  `(locally #f ,@body))
```

```
* '#0f

(DECLARE (OPTIMIZE (SPEED 0) (SAFETY 3)))
```

Также для следующих деклараций можно передавать значения 1 и 2. Результат этих различных деклараций сильно зависит от компилятора, поэтому вы, скорее всего, никогда не будете их использовать:

```
* '(#1f #2f)

((DECLARE (OPTIMIZE (SPEED 1) (SAFETY 2)))
 (DECLARE (OPTIMIZE (SPEED 2) (SAFETY 1))))
```

И, хотя макросы не могут прямо расширяться в декларации, мы по прежнему можем использовать обычные макросы для управления декларациями. Поскольку проходчик-по-коду не может пройти по макросу в поисках деклараций до тех пор, пока он не расширил макрос, нет никакого способа сказать, что декларация является действительным подвыражением написанной вами формы, или что макрос добавил декларацию при расширении.

Fast-progn и **safe-progn** — это простые примеры макросов, которые расширяются в формы, содержащие декларации. Заметьте, что мы использовали неявный progn от **locally** вместо обычного **progn**, поскольку обычный **progn** не принимает декларации[3]. Эти два макроса используют

[3]Поскольку не устанавливает привязки.

Листинг 7.3: SAFE-PROGN

```
(defmacro safe-progn (&rest body)
  `(locally #0f ,@body))
```

ранее определённый считывающий макрос sharp-f. Мы можем использовать эти формы как разновидности **progn**, содержащие в себе выражения для оптимизации по скорости (но потенциально опасные) и оптимизацию по безопасности (но потенциально медленные).

```
* (macroexpand
   '(fast-progn
     (+ 1 2)))

(LOCALLY
  (DECLARE (OPTIMIZE (SPEED 3) (SAFETY 0))) (+ 1 2))
T
```

Мы можем передавать другие декларации в аргументах макроса, поскольку их расположение не проверяется и не будет проверено, до тех пор, пока макрос не будет расширен:

```
* (macroexpand
   '(fast-progn
     (declare (type fixnum a))
     (the fixnum (+ a 1))))

(LOCALLY
 (DECLARE (OPTIMIZE (SPEED 3) (SAFETY 0)))
 (DECLARE (TYPE FIXNUM A))
 (THE FIXNUM (+ A 1)))
T
```

При экспериментировании с расширениями макросов нам может понадобиться выяснить результат встраивания их в различные лексические контексты. Комбинирование макросов, вычисляющихся во время чтения, описано в *разделе Время-Исполнения и Время-Считывания* с помощью переменной *, сохраняющей последние результаты REPL и позволяющей убедиться в том, что наша форма вычисляется так, как и было задумано:

```
* (let ((a 0)) #.*)
```

```
1
```

И, хотя вышеприведённый код вычисляется корректно, декларации иногда применимы только для скомпилированного кода. Например, поскольку наше вышеприведённое вычисление рассматривается как код[4], то он, возможно, будет игнорировать декларации безопасности и вернёт переполненный результат в виде bignum. Посмотрим, что получится у нас:

```
* (let ((a most-positive-fixnum)) #.**)

536870912
```

Так и есть. CMUCL игнорирует декларации для интерпретируемого кода. Теперь нам нужно продолжать экспериментировать с нашим выражением в ***, но поскольку мы не уверены в том, что нужный нам результат будет получен в этот раз, и для того, чтобы не потерять его, мы вернёмся к *:

```
* ***

(LOCALLY
  (DECLARE (OPTIMIZE (SPEED 3) (SAFETY 0)))
  (DECLARE (TYPE FIXNUM A))
  (THE FIXNUM (+ A 1)))
```

Всё на месте. Теперь у нас есть три попытки запустить этот код. Попробуем скомпилировать этот код и посмотреть, как выполняется обёртка fixnum:

```
* (funcall
    (compile nil
      `(lambda ()
         (let ((a most-positive-fixnum))
           ,*))))

; Warning: This is not a (VALUES FIXNUM &REST T):
;   536870912

536870912
```

[4]В большинстве реализаций.

Хм, что у нас получилось? Разве мы не указали лиспу не выполнять проверку? Рассуждения о декларациях усложняются при проведении таких оптимизаций во время компилирования, как *свёртка константы (constant folding)*. Получилось так, что при компиляции формы лисп смог выполнить сложение во время компилирования, поскольку мы складываем константы, и поэтому он знает, что результат также будет константой, и его не требуется вычислять во время исполнения. Когда лисп выполнил эти операции, то обнаружил, что наша декларация о fixnum определённо неверна. Этим предупреждением лисп говорит нам "Вы дурак, я проигнорирую ваше определение, поскольку вам нельзя доверять." Если мы немного перетасуем наше выражение так, чтобы лисп не смог сворачивать константы, то мы, наконец, увидим эффект от обёртки вокруг fixnum:

```
* (funcall
    (compile nil
      `(lambda (a)
         ,**))
    most-positive-fixnum)

-536870912
```

Другим важным свойством деклараций является способность *затенять (shadow)* другие декларации примерно также, как одни лексические переменные затеняют другие лексические переменные. Например, мы можем захотеть написать макрос, выполняющий проверку безопасности, даже будучи встроенным в код с декларированным низким уровнем безопасности:

```
(defmacro error-checker ()
  `(safe-progn
    (declare (type integer var))
    do-whatever-other-error-checking))
```

Обернув ещё один слой, мы можем использовать эти макросы для добавления кода, проверяющего на наличие ошибок некоторый код, для которого важна скорость выполнения, а не безопасность, воспользовавшись *вложенным* применением другого макроса, **fast-progn**:

```
(defun wrapped-operation ()
  (safe-progn
    do-whatever-error-checking
```

```
(fast-progn
  but-this-needs-to-go-fast)))
```

Безопасная проверка параметров с кодом проверки на ошибки, окружающий быструю реализацию некоторой функциональности — это общий шаблон в создании высокопроизводительного лисп кода. Особенно для таких итеративных процедур, как проход по массиву, значительное улучшение производительности может быть достигнуто через выполнение проверок на ошибки в таких случаях, как тип и границы в начале операции, и в последующем пропуске проверки в тех местах, где это возможно.

COMMON LISP, в первую очередь, спроектирован с упором на мощь в программировании; эффективность находится на втором плане. Однако эти характеристики, мощь и эффективность, не обязательно должны находиться по разную сторону. С помощью макросов мы можем использовать мощь лиспа для решения проблем эффективности. В дополнение к обычным макросам и считывающим макросам — которые сами по себе представляют серьёзную мощь — COMMON LISP позволяет использовать *макросы компилятора (compiler macros)*. Макросы компилятора — это макросы, решающие почти ту же задачу, что и обычные макросы: программы, создающие программы. Макросы компилятора не широко описываются в большинстве лисп учебников, что в свою очередь показывает приоритетность производительности для программистов (почти нулевая). Однако, макросы компилятора представляют из себя элегантные решения для определённого класса проблем эффективности и заслуживают своего места в наборе инструментов у каждого лисп профессионала.

Макросы компилятора определяют трансформации, которые будут применены вашим лисп компилятором к вызовам (именованных) функций. Это означает что вы можете взять функцию, созданную с помощью **defun**, и сказать лиспу, что вместо компилирования вызовов этой функции он должен скомпилировать другой, определённый макросом компилятора, код. Зачем нам нужно использовать функцию в сочетании с макросом компилятора? Ведь проще было бы просто написать макрос с тем же именем? Первая, менее важная, причина заключается в том, что такой подход позволяет нам осуществлять больший контроль над устранением накладных расходов компиляции. В частности, COMMON LISP не определяет, когда или как часто должен быть расширен макрос. В случае с интерпретируемым кодом вполне возможен вариант, когда макрос будет раскрываться при каждом вызове[5]. При выполнении опти-

[5]Другими словами, кэширование макро расширений не гарантируется при интер-

мизаций во время компиляции нам нужно выполнять (возможно долгие и ресурсоёмкие) вычисления перед выполнением функции, это делается с целью сократить непосредственные вычисления самой функции. Как это и должно быть — компилирующие макросы дают нам возможность выполнять ресурсоёмкие вычисления только единожды, при компиляции кода.

Но, есть и другая, более важная, чем просто однократное вычисление в нужное время, причина — макросы компилятора полезны ещё и тем, что они вводят *дуализм синтаксиса* в язык. Макросы компилятора позволяют нам добавить второе значение в любую форму, представляющую вызов (именованной) функции. В дополнение к обычному значению слова "макрос" компилирующие макросы добавляют в это значение ещё и компиляцию. Настоятельно рекомендуется убедиться, что ваше скомпилированное значение реализует ту же задачу, что и изначальное значение, но вы вольны сами определять как выполнять это действие (вот в чём изюминка). Преимущество в использовании двойственного синтаксиса заключается в том, что мы можем вносить изменения в эффективность кода без необходимости изменения всего кода. Мы можем взять существующую кодовую базу — которая предположительно использует большое количество вызовов функций — и изменить процесс компиляции кода, введя двойственность синтаксиса. Всё, что нам нужно — это найти недопустимо дорогостоящие вызовы функций и затем реализовать макросы компилятора, преобразующие их в более дешёвые расширения.

Вызовы каких функций являются наиболее затратными? В качестве первого примера вспомним *раздел Безопасность Считывателя*. В нём упоминается, что функции могут выполнять *лямбда деструктуризацию* и что лямбда деструктуризация — это подмножество более общего множества *defmacro деструктуризации*[6]. Если функции принимают в виде аргументов ключевые слова, то мы передаём их в виде сгруппированных пар символов ключевого слова и их соответствующих значений. Ключевые слова, используемые в качестве аргументов — это очень удобно, но функции, использующие подобные аргументы, подвержены накладным расходам от большего числа вызовов, нежели функции, не использующие подобные аргументы. Деструктуризация не даётся бесплатно. Компилятору приходится компилировать код в функцию, перебирающую список аргументов переменной длины, получающую значения в правильном порядке (включая вставку значений по-умолчанию) и только потом выпол-

претации.

[6]Лямбда деструктуризация не может деструктурировать списки, переданные как аргументы, и не поддерживает таких возможностей, как whole из defmacro.

Листинг 7.4: FAST-KEYWORDS-STRIP

```
(defun fast-keywords-strip (args)
  (if args
    (cond
      ((eq (car args) '&key)
        (fast-keywords-strip (cdr args)))
      ((consp (car args))
        (cons (caar args)
      #1=(fast-keywords-strip
   (cdr args))))
      (t
        (cons (car args) #1#)))))
```

няющую функцию. В целом лисп компилирует очень быстрый код для деструктуризации подобных ключевых слов в роли аргументов, и мы почти никогда не заметим (или озадачимся) малым падением эффективности. Однако бывают случаи, когда приходится обратить внимание на падение эффективности, в частности, когда мы вызываем подобную функцию из цикла, в котором критически важна производительность.

Fast-keys-strip — это утилита, получающая лямбда деструктурируемый список, состоящий только из обычных аргументов и аргументов ключевых-слов, и возвращает список символов, использованных в качестве ссылки на эти аргументы. Другими словами, она вернёт **(a b c)**, если ей передано **(a b c)** или **(a &key b (c 0))**, но передача утилите **(a &optional b c)** запрещена.

Defun-with-fast-keywords используется схожим с **defun** способом. Как и **defun**, его первый аргумент — символьное наименование функции, второе — список аргументов, а остальные формы — это формы, исполняемые в теле функции. Однако в отличие от **defun**, формы, предназначенные для **defun-with-fast-keywords**, могут содержать в себе только обычные аргументы и аргументы ключевые-слова (нельзя использовать optional, rest и т.д.). Упражнение: Расширьте **fast-keywords-strip** и добавьте обработку всех лямбда деструктурирующих списков[7].

Расширение **defun-with-fast-keywords** сложно. **Defun-with-fast--keywords** расширяется в три формы[8]. Первая определяет функцию так,

[7]Но при работе с лиспом не забывайте золотое правило Норвига: никогда не смешивайте аргументы ключевые-слова и optional аргументы.

[8]Которые рассматриваются как высокоуровневые, поскольку высокоуровневые progn формы рассматриваются по-особому — ценное свойство COMMON LISP.

Листинг 7.5: DEFUN-WITH-FAST-KEYWORDS

```
(defmacro! defun-with-fast-keywords
           (name args &rest body)
  `(progn
     (defun ,name ,args ,@body)
     (defun ,g!fast-fun
    ,(fast-keywords-strip args)
            ,@body)
     (compile ',g!fast-fun)
     (define-compiler-macro ,name (&rest ,g!rest)
       (destructuring-bind ,args ,g!rest
 (list ',g!fast-fun
,@(fast-keywords-strip args))))))
```

Листинг 7.6: KEYWORDS-TESTS

```
(defun
  slow-keywords-test (a b &key (c 0) (d 0))
  (+ a b c d))

(compile 'slow-keywords-test)

(defun-with-fast-keywords
  fast-keywords-test (a b &key (c 0) (d 0))
  (+ a b c d))
```

Листинг 7.7: KEYWORDS-BENCHMARK

```
(defun keywords-benchmark (n)
  (format t "Slow keys:~%")
  (time
    (loop for i from 1 to n do
      (slow-keywords-test 1 2 :d 3 :c n)))
  (format t "Fast keys:~%")
  (time
   (loop for i from 1 to n do
     (fast-keywords-test 1 2 :d 3 :c n))))

(compile 'keywords-benchmark)
```

как будто мы работаем с обычным **defun**. Вторая определяет функцию с автоматическим gensym'ом в качестве имени, **g!fast-fun**. Эта функция аналогична первой с тем исключением, что просто получает в качестве аргументов не-ключевые-слова для каждого аргумента (являющегося ключевым словом или нет). Далее определяется макрос компилятора, преобразующий вызовы нашей первой функции в вызовы второй функции. Таким образом, вместо деструктуризации ключевого слова, осуществляемого первой функцией, мы воспользуемся сведениями из момента компиляции об использованной форме для вызова функции и вставим ключевые слова с правильным расположением совместно с деструктурирующими привязками.

Теперь у нас есть (почти) двойственный синтаксис с **defun**. Обычное определение функции с аргументами, содержащими ключевые слова, выглядит как **slow-keywords-test**. Ниже мы будем компилировать его с целью проведения замеров производительности. **Fast-keywords-test** написан идентично **slow-keywords-test**, с тем исключением, что вместо **defun** используется **defun-with-fast-keywords**. Благодаря **defun-with-fast-keywords**, нам не нужно компилировать эту функцию, поскольку **defun-with-fast-keywords** расширяется в вызов компиляции только одного, нужного нам определения — то, которое было названо с помощью автоматического gensym'а **g!fast-fun**.

Keywords-benchmark — это простая функция, использующая макрос **time**, для того, чтобы замерять время исполнения эквивалентных серий обеих функций. Следует учесть, что мы также компилируем и **keywords-benchmark**. Тема о замерах производительности будет раскрыта в *разделе Написание и Замеры Производительности Компилято-*

ров.

```
* (keywords-benchmark 100000000)
Slow keys:
; Evaluation took:
;   17.68 seconds of real time

Fast keys:
; Evaluation took:
;   10.03 seconds of real time
```

Достаточно 100 миллионов вызовов этой функции для того, чтобы увидеть что хотя обе функции компилируются, но функция, определённая с помощью **defun-with-fast-keywords**, исполняется примерно на 40% быстрее благодаря макросу компилятора. Также заметим, что производительность нашего макроса компилятора не зависит от того, являются ли аргументы ключевые-слова постоянным значением, известным во время компиляции. Мы передаём **n**, произвольную лисп форму, как аргумент ключевого слова **:c**. Таким образом, макрос компилятора расширяет быструю версию в тот же код, что и медленная версия, с той разницей, что в быстрой версии нет накладных расходов в виде деструктуризации ключевого слова.

Так почему же COMMON LISP не может выполнять подобную оптимизацию для каждой функции, которая принимает ключевые слова и, таким образом, избежать накладных расходов? Макросы компилятора применяются только во время компиляции, а мы хотим сохранить возможность деструктурировать аргументы во время выполнения. В этом заключается суть макросов компилятора: макросы компилятора предназначены для оптимизации вызовов функции, а не для самих функций. В случае с ключевыми словами макросы компилятора позволяют нам устранить накладные расходы для скомпилированных вызовов функций, но оставляют без изменения исходную функцию — и её код, деструктурирующий ключевые слова — доступным для использования во время исполнения. Макросы компилятора дают нам двойственный синтаксис для двух различных операций, различимых только по контексту. Другой способ, с помощью которого можно избежать накладных расходов, связанных с ключевыми словами, описан в книге PAIP Норвига [PAIP-P323].

Какую ещё пользу можно извлечь из макросов компилятора для вызовов функций? Мы можем не только уменьшить накладные расходы, связанные с деструктуризацией, но и сократить издержки связанные с

самими функциями, предварительно обработав постоянные аргументы. Макрос компилятора может выполнить некоторую подготовку во время компиляции, вследствие чего сокращается время, расходуемое во время выполнения. Одно из наиболее очевидных примеров — это функция **format**. Рассмотрим как работает **format** (или, в C, **printf**). Это функция, которой вы передаёте управляющую строку во время выполнения программы. **Format** обрабатывает управляющую строку и печатает форматированный вывод в поток (или возвращает вывод в виде строки). В сущности, при использовании **format** вы выполняете вызов функции интерпретатора, который осуществляет форматирование строки, с управляющей строкой в роли программы. Воспользовавшись макросами компилятора, мы можем устранить вызов функции, предварительно обработав управляющую строку, и изменить вызов функции на специализированный код, сращённый с вызывающей стороной, благодаря которому компилятор может выполнять будущие оптимизации. Звучит сложновато, не так-ли? Мы должны знать, как конвертировать форматируемые контрольные строки в эквивалентный лисп код. К счастью, как и со многими другими нюансами, COMMON LISP уже позаботился об этом. COMMON LISP *правильно* выполняет форматирование. Это означает, что предметно-ориентированный язык, предназначенный для создания форматированного вывода, может макро-компилировать себя в лисп код. Это часть философии лиспа — всё должно компилироваться в лисп. **Formatter** — это макрос, который компилирует управляющие строки в лисп. Когда вы передаёте управляющую строку **formatter**'у, то он расширяется в лямбда форму, которая выполняет желаемое форматирование. Например, расширение для простой управляющей строки может выглядеть так[9]:

```
* (macroexpand '(formatter "Hello ~a~%"))

#'(LAMBDA
   (STREAM
    &OPTIONAL
    (#:FORMAT-ARG1347
     (ERROR 'SB-FORMAT:FORMAT-ERROR
            :COMPLAINT "required argument missing"
            :CONTROL-STRING "Hello ~a~%" :OFFSET 7))
    &REST SB-FORMAT::ARGS)
   (DECLARE (IGNORABLE STREAM))
```

[9] **Terpi** печатает символ новой строки в поток.

```
   (BLOCK NIL
          (WRITE-STRING "Hello " STREAM)
          (PRINC #:FORMAT-ARG1347 STREAM)
          (TERPRI STREAM))
   SB-FORMAT::ARGS)
T
```

Таким образом, **formatter** расширяется в лямбда форму[10]. Она компилирует управляющую строку в лисп форму, пригодную для вычисления или для макро встраивания в другой лисп код, где она будет представлять из себя скомпилированную функцию или будет встроена в скомпилированный код на стороне вызывающего элемента. Также надо учесть, что расширению **formatter** нужно передавать поток, и что это расширение не может принимать **nil** так, как это делает **format**. Всё это происходит потому, что функции в которые расширяется **formatter** (такие как **write-string** и **terpi**) требуют потоков. Для того, чтобы избежать эту проблему, используйте макрос **with-output-to-string**.

Fformat — это замечательная невидимая обёртка вокруг **format**. Благодаря этой обёртке мы можем определить макрос компилятора для форматирования. Нам нужно новое имя, отличающееся от **format**, поскольку определение макросов компилятора поверх функций, определённых в COMMON LISP, запрещено [CLTL2-P260]. Наш компилирующий макрос использует свойство defmacro деструктуризации, &whole. Мы применяем его для привязки **form** к списковой структуре вызова макроса. Сделано это для того, чтобы воспользоваться ещё одной особенностью макросов компилятора: макросы компилятора могут избирательно не раскрывать формы. Если мы вернём **form**, то лисп **увидит**, что мы просто вернули переданную форму (произведя проверку с помощью **eq**) и попросит макрос компилятора не выполнять последующие расширения формы — даже если мы раскрываем её для того, чтобы применить функцию совместно с макросом компилятора. При компиляции мы можем решить, что нужно использовать другое значение формы. Это фундаментальное различие между макросом компилятора и обычным макросом. Макрос компилятора может разделить дуализм синтаксиса с функцией, но этого не может сделать обычный макрос. В **fformat** макрос компилятора не будет производить раскрытие в более эффективный код в случае, когда аргумент, управляющая строка, не является константой. В случае с **fformat** нам нужно, чтобы работали вызовы **fformat** для нестроковых управляющих строк (как вызовы функции, возвращающей

[10]Шарп-закавыченная лямбда форма — если быть более точным.

Листинг 7.8: FFORMAT

```
(defun fformat (&rest all)
  (apply #'format all))

(compile 'fformat)

(define-compiler-macro fformat
                       (&whole form
stream fmt &rest args)
  (if (constantp fmt)
    (if stream
      `(funcall (formatter ,fmt)
 ,stream ,@args)
      (let ((g!stream (gensym "stream")))
`(with-output-to-string (,g!stream)
   (funcall (formatter ,fmt)
     ,g!stream ,@args))))
    form))
```

строки). Другими словами, нам нужно сохранить возможность генерировать управляющие строки во время исполнения. Очевидно, что подобные вызовы не могут применять оптимизацию во время компиляции для управляющих строк.

Fformat-benchmark — это функция почти идентичная функции **keywords-benchmark**, описанной выше. Она использует **time** для сравнения времени, затрачиваемого для обработки большого количества операций форматирования с обычным **format** и новым **fformat**. Вот результаты, полученные для миллиона итераций:

```
* (fformat-benchmark 1000000)
Format:
; Evaluation took:
;   37.74 seconds of real time
;   [Run times include 4.08 seconds GC run time]
;   1,672,008,896 bytes consed.
Fformat:
; Evaluation took:
;   26.79 seconds of real time
;   [Run times include 3.47 seconds GC run time]
;   1,408,007,552 bytes consed.
```

Листинг 7.9: FFORMAT-BENCHMARK

```
(defun fformat-benchmark (n)
  (format t "Format:~%")
  (time
    (loop for i from 1 to n do
      ( format nil "Hello ~a ~a~%" 'world n)))
  (format t "Fformat:~%")
  (time
    (loop for i from 1 to n do
      (fformat nil "Hello ~a ~a~%" 'world n))))

(compile 'fformat-benchmark)
```

Увеличение ускорения примерно на 30%. Макрос компилятора не только уменьшил время, затрачиваемое на форматирование, но и уменьшил количество cons'ов (что в свою очередь уменьшило время, затрачиваемое на сборку мусора). Макрос компилятора позволил избежать интерпретации форматируемой строки во время исполнения, выполняя вместо этого большинство вычислений только единожды, во время компиляции функции — как это и должно быть. К несчастью замеры производительности часто скрывают или устраняют некоторые важные детали. Предварительная компиляция форматируемых строк с помощью **fformat** устраняет накладные расходы интерпретирования, но за это приходится платить большим размером скомпилированной программы. И, даже если памяти в достатке, большой код может выполняться медленнее за счёт уменьшения производительности кэша инструкции.

В этом разделе мы рассмотрели способы изменения производительности кода с помощью обычных макросов, считывающих макросов и специальных типов макросов, предназначенных именно для этой задачи: макросов компилятора. К счастью, этот раздел и оставшаяся часть этой главы убедят вас в том, что если вы хотите писать по-настоящему эффективный код, то вам нужен COMMON LISP. А причина, по которой вам нужен COMMON LISP заключается в макросах.

Упражнение: Скачайте CL-PPCRE Эди Вейтза (описано в *разделе CL-PPCRE*) и изучите, как **api.lisp** использует макросы компилятора. Посетите web-сайт Эди и скачайте несколько лисп пакетов, которые покажутся вам интересными.

Упражнение: При написании макроса компилятора для **fformat** мы были вынуждены явно использовать **gensym**, поскольку у нас нет мак-

роса **define-compiler-macro!**. Исправьте это положение. Трудное Упражнение: Определите **define-compiler-macro!** так, чтобы он использовал функциональность **defmacro!**, и сам не вызывал **gensym**. Подсказка: Думайте за пределами ящика.

7.3 Знакомство с Вашим Дизассемблером

Трудно получить представление о происходящих событиях в лиспе без исследования сырых инструкций, выполняемых вашим процессором в различных лисп формах. Изучение расширения макросов помогает нам в деле создания макросов, аналогично исследование *скомпилированных расширений* лисп программ — обычно ассемблерных инструкций — также оказывается весьма полезным. Поскольку лисп компиляторы могут и часто являются макро расширителями, то машинный код, генерируемый ими, можно рассматривать с некоторой интересной точки зрения, как разновидность лисп кода. Поскольку лисп — это не столько язык, сколько строительный материал и фабрика для создания языков, то можно считать, что лисп используется для определения и компиляции языков в язык, состоящий из набора инструкций вашего процессора.

В COMMON LISP есть функция **disassemble**, предназначенная для просмотра скомпилированных расширений. **Disassemble** — это аналог CMUCL-овского расширения **macroexpand-all**, описанного в [USEFUL-LISP-ALGOS2]. Передав **disassemble** функцию или символ, для которого существует привязка **symbol-function**, мы можем увидеть сырые инструкции машинного кода, которые будут выполнены при вызове функции.

Проблема в том, что все эти инструкции сырого машинного кода не выглядят как лисп. Вместо привычных лисповских вложенных скобок эти инструкции представляют из себя странные, мелкие шаги некоторой весьма произвольной машины. Изучение скомпилированного расширения лисп кода подобно рассматриванию плаката с помощью увеличительного стекла. Вы можете разглядеть любой участок картинки с большой детализацией, но представить всю *большую картину* из малых участков — это сложная, а зачастую, и невозможная задача. Но всё становится ещё более сложным, когда дело доходит до машинного кода. При такой детализации, порою невозможно, взглянув на участок машинного кода, сказать наверняка, почему компилятор разместил его здесь.

К сожалению, никто по-настоящему не знает, как лучше всего реализовать в лиспе функцию **compile**. Существуют множество макрорасширений, которые можно применить к коду, и, можно сказать наверняка,

что некоторые из них достойны стандартизирования, но поиск наилучшего способа использования таких аппаратных ресурсов, как циклы CPU и память, остаются (и скорее всего останутся) актуальной темой исследований. Кроме того, улучшения в архитектуре компиляторов зависят от улучшений аппаратного обеспечения. Первоначальные оптимизации впоследствии могут потерять свою актуальность или корректность. Не надо идти далеко за примерами того, как изменение мира влияет на восприятие самого понятия эффективности.

Учёные[11] избегали использование кода, применяющего вычисления с плавающей запятой, и отдавали предпочтение вычислениям, основанным на машинном-слове с фиксированной запятой. Причина заключалась в том, что в компьютерах не было устройства для работы с плавающей запятой, и приходилось использовать целочисленные инструкции процессора для симуляции плавающей запятой. А поскольку тогда процессоры не были оптимизированы для работы с плавающей запятой, то вполне естественно, что операции с плавающей запятой уступали по производительности операциям с фиксированной запятой. Однако, с течением времени, появились сопроцессоры, предназначенные для работы с плавающей запятой со скоростью света. В течении буквально одной ночи, учёные поставили под сомнение предположение о том, что операции с фиксированной запятой всегда будут быстрее операций с плавающей запятой, и начали производить замеры и исследования своего аппаратного обеспечения. В процессе разработки аппаратного обеспечения изменилась производительность операций с плавающей запятой. Позже в компьютерах появились 2, 4 и более сопроцессоров для работы с плавающей запятой, и тут учёные обнаружили, что, если поддерживать заполненность *конвейера (pipeline)* обработки операций с плавающей запятой, то производительность операций с плавающей запятой довольно часто оказывается большей, чем производительность операций с фиксированной запятой. И если раньше программисты выбирали фиксированную запятую из соображений производительности, то теперь — спустя десятилетие — они отказались от *правильной* реализации в пользу *неправильной реализации*.

При разработке макросов полезно изучать вывод **macroexpand** и **macroexpand-all**, так же полезно изучать вывод **disassemble**, причём не только со стороны функционирования реализации, но и для того, чтобы убедиться в том, что мы передали лиспу всю требуемую информацию для генерации эффективных расширений. **Dis** — это макрос, облегчаю-

[11] Одни из немногих пользователей компьютеров, для которых требуется эффективный код.

Листинг 7.10: DIS

```
(defmacro dis (args &rest body)
  `(disassemble
     (compile nil
       (lambda ,(mapcar (lambda (a)
                          (if (consp a)
                            (cadr a)
                            a))
                        args)
         (declare
           ,@(mapcar
               #`(type ,(car a1) ,(cadr a1))
               (remove-if-not #'consp args)))
         ,@body))))
```

щий проверку вывода **disassemble** для некоторого участка лисп кода. Его первый аргумент — это список символов или списки типа и символа. Для того, чтобы увидеть, как работает **dis**, достаточно просто расширить его. Вот как **dis** расширяется для простого бинарного сложения:

```
* (macroexpand
    '(dis (a b) (+ a b)))

(DISASSEMBLE
  (COMPILE NIL
    (LAMBDA (A B)
      (DECLARE)
      (+ A B))))
T
```

Что здесь делает пустая форма **declare**? Это место, в которое **dis** может вставить декларации типов, которые могут быть определены в параметрах, например:

```
* (macroexpand
    '(dis ((fixnum a) (integer b))
       (+ a b)))

(DISASSEMBLE
 (COMPILE NIL
```

```
          (LAMBDA (A B)
                  (DECLARE (TYPE FIXNUM A)
                           (TYPE INTEGER B))
                  (+ A B))))
T
```

Dis во многом работает как лямбда форма, поскольку расширяется в (обёрнутую) лямбда форму. При желании вы можете добавить дополнительные декларации, и будет возвращено значение (поскольку лямбда формы создают невидимый progn). Загрузите код этой книги и введите следующую форму в вашу лисп среду:

```
(dis (a b)
  (+ a b))
```

Машинный код должен быть очень коротким, а всё потому, что большинство сложных элементов будут скрыты вызовом предварительно скомпилированной функции — достаточно умной, чтобы предоставить такие возможности лиспа в обработке чисел, как автоматическое приведение числовых типов (type contagion), упрощение рациональных чисел (rational simplification) и т.д.. Такой приём называется *косвенностью (indirection)*, что в свою очередь можно увидеть в выводе вашего дизассемблера:

```
CALL #x1000148 ; GENERIC-+
```

А теперь попробуйте применить **dis** к трём аргументам:

```
(dis (a b c)
  (+ a b c))
```

Упражнение: Сколько косвенностей вы видите в общей функции сложения? А как насчёт **N** где **(<= 0 N)** аргументов?

Теперь попробуем заблокировать тип одной переменной. А теперь сравните этот пример с примером выше, в котором мы не декларировали типы:

```
(dis ((fixnum a) b)
  (+ a b))
```

Здесь должно появиться некое предупреждение об ошибке **OBJECT-NOT-FIXNUM-ERROR**. Лисп компилируется с некоторым дополнительным кодом, осуществляющим проверку типа, при косвенном управлении общей функцией сложения поскольку тип **b** неизвестен во время компиляции, а это требует наличия всех таких лисповских обработок чисел, как автоматическое приведение числового типа и т.д..

Здесь мы не говорим о том, как получить быстрый код. На самом деле этот код может быть чуть менее эффективным чем предыдущий код. Для быстрого кода мы должны воспользоваться процессом под названием *встраивание (inlining)*. Для некоторых специальных операций, где доступно достаточно информации о типе, лисп компиляторы знают как избежать косвенности и прямо добавляют машинный код в компилируемую функцию для выполнения желаемой операции. В следующем примере не должно быть никаких косвенностей в общей функции сложения:

```
(dis ((fixnum a) (fixnum b))
  (+ a b))
```

Процесс встраивания может привести к более машинному коду, чем какой-либо из примеров, в котором использовалась косвенность. К этому приводит копирование некоторой (но не всей) функциональности, реализованной в общей функции сложения, в компилируемую функцию. И хотя данный пример может показаться более длинным, этот код будет более производительным за счёт меньшей косвенности.

Но эта мешанина из машинного кода по прежнему менее эффективна чем реализация на C. Сюда прикомпилированы разнообразные функции подсчёта аргументов, типов и проверок на переполнение — их так много, что выгода, получаемая от них оказывается ниже, чем накладные расходы. Если же мы используем эту функцию в цикле, то возникающие накладные расходы будут попросту неприемлемыми.

В таких языках как C вы везде определяете типы и нигде не применяете безопасность, поэтому код всегда эффективен, всегда опасен и всегда надоедлив при написании. В большинстве динамических Блаб языках вы нигде не определяете типы и везде применяете безопасность, таким образом, код всегда безопасен, никогда не назойлив, но и в то же время всегда неэффективен. В большинстве строгих, статических Блаб языках вы везде определяете типы и везде применяете безопасность, таким образом код всегда эффективен и всегда безопасен, но всегда надоедлив. Лисп даёт вам право выбора. Поскольку лисп по-умолчанию включён в режим безопасность-и-не-назойливость, то лисп программы часто оказываются

немного менее быстрыми, чем их C эквиваленты, но, лисп программы почти всегда безопасны. Поскольку лисп предоставляет программистам замечательную систему декларации типов, и превосходную реализацию компиляторов, то лисп программы почти всегда безопасны также, как и динамические Блаб эквиваленты и, в целом, гораздо быстрее их. Что ещё более важно, в лиспе есть макросы, поэтому, если вам что-то не нравится, то просто измените его!

Что же, пойдём дальше и попросим лисп ускорить нашу операцию сложения. Напомним, что шарп-f — это аббревиатура считывающего макроса, предназначенного для декларации высокоскоростного и небезопасного кода.

```
(dis ((fixnum a) (fixnum b))
  #f
  (+ a b))
```

Эта последовательность инструкций машинного кода должна быть немного короче, чем предыдущая. В этой версии удалены проверки типов и подсчёт аргументов. Но этот код — по-прежнему далеко не та короткая опасная инструкция, выполняющая сложение, к которой мы стремились. Для того, чтобы понять что происходит, мы должны прочитать *замечания* компилятора. Посредством замечаний компилятор сообщает нам свои наблюдения: “Похоже, что вы собираетесь сделать что-то эффективное и вы почти у цели, но я хочу внести некоторую ясность в ваши намерения. Вот подсказка, которая поможет прояснить ситуацию...”

Замечания компилятора — это бесценный источник информации. Если вы хотите создать эффективный лисп код, то следует обратить особое внимание на эти замечания. Лисп компиляторы используют системы *логического вывода типов (type inference)* для обнаружения различных нюансов кода, которые могут остаться незамеченными программистом. В нашем случае компилятор должен дать примерно такое замечание:

```
; Note: Doing signed word to integer coercion
;       (cost 20) to "<return value>".
```

Лисп никогда не совершит таких глупых поступков, как игнорирование переполнения fixnum, конечно, если мы его сами об этом не попросим[12]. Таким образом, для того, чтобы лисп выкинул свои предупреждения на ветер и написал нам очень быструю, но потенциально опасную,

[12]В C программах переполнение fixnum — это класс нарушений безопасности, который часто эксплуатируют злоумышленники.

функцию, нам нужно заставить лисп игнорировать проверку преобразования знаковых слов (fixnum) в целые (bignum) и связанные с этим ограничения. Нужно сообщить лиспу о том, что переполнение для нас приемлемо, и что мы действительно хотим от него тихого возвращения fixnum:

```
(dis ((fixnum a) (fixnum b))
  #f
  (the fixnum (+ a b)))
```

Код жжёт напалмом. Примерно эквивалентный C функции, складывающей fixnum-ы: несколько машинных инструкций, складывающих вместе два регистра и возвращающих управление вызвавшему коду. Ваш дизассемблер может послужить толчком для многих идей в области эффективности лиспа, но, он также может научить вас двум навыкам. В этом разделе мы покрыли довольно большую часть первого навыка: как использовать декларации для получения эффективной обработки чисел, особенно внутри циклов. Второй навык — это эффективное использование таких структур данных как массивы/векторы. Этой теме посвящён *раздел 7.4, Область Видимости Указателя.*

Аналогично техническому прогрессу, изменившему эффективность арифметики с плавающей запятой с техники, которую следует избегать в технику, которую следует использовать, улучшения в технологии лисп компиляторов — в сочетании с *правильными* системами типов и декларирования безопасности COMMON LISP-а — изменяет способ мышления об эффективности. Благодаря этим инструментам и росту сложности систем программного обеспечения[13], вопрос, посвящённый теме улучшения эффективности лиспа до уровня низко-уровневых языков, меняется на тему, посвящённую улучшению эффективности других языков до уровня лиспа. И конечно, ответ заключён в макросах, используемых для реализации других языков в лиспе.

7.4 Область Видимости Указателя

Уменьшится ли мощь языка при удалении из него указателей? В частности, мешает ли отсутствие явной *области видимости указателей* в лиспе созданию эффективных алгоритмов, определённых в терминах арифметики указателей? Оказывается, что нет, отсутствие прямой поддержки

[13]Особенно ярко лисповский оптимизационный потенциал проявляется при использовании макро техник для улучшения производительности больших и сложных приложений.

указателей не является ни теоретической и ни практической преградой. Любые алгоритмы или структуры данных, реализованные с помощью указателей в таком языке, как C можно реализовать также или ещё лучше в лиспе.

И всё же, что на самом деле представляет из себя область видимости указателей, и почему нам может потребоваться использование области видимости указателей? Область видимости указателей позволяет рассматривать память компьютера (или виртуальную память) как большой, индексируемый массив, из которого мы можем загружать, и в который мы можем сохранять fixnum значения. Выглядит опасно? Так и есть, на сегодняшний день область видимости указателей — это источник многих сложных ошибок и непосредственная причина возникновения огромных классов проблем безопасности программного обеспечения.

Область видимости указателя — это на самом деле способ определения косвенностей, то есть получение доступа через различные среды, что, собственно, и происходит в случае с fixnum арифметикой. Как мы обычно программируем через среды? Мы используем либо динамическую, либо лексическую области видимости, поддерживаемые COMMON LISP-ом, или комбинируем эти две области видимости, или используем новые типы областей видимости, создаваемые с помощью макросов. Макрос **pointer-&** и функция **pointer-*** — это примеры-наброски, создающие для нас иллюзию области видимости указателя, что в свою очередь показывает, что потребность в указателях на самом деле может являться простой потребностью в замыкании. Первое и единственное мнение об аналогии между указателями и замыканиями я прочитал в сообщении в новостной группе **comp.lang.scheme** от Олега Киселёва (Oleg Kiselyov) [POINTERS-AS-CLOSURES]. Он предложил использовать замыкания для эмуляции указателей и написал реализацию для Scheme[14].

Pointer-& и **pointer-*** показывают возможный способ имитирования косвенности указателя через замыкания. Когда мы используем макрос **pointer-&**, то он расширяется в довольно умную лямбда форму, которая определяет, хотите ли вы получить или установить значение, и выполняет соответствующее действие. Для этого **pointer-&** использует *gensym-ы*. Вместо того, чтобы использовать их в качестве имён для привязок с целью избежать нежелательного захвата переменной во время компиляции, **pointer-&** использует их для того, чтобы убедиться в отсутствии возможных *захватов во время запуска (run-time capture)*, где мы уже

[14]На web-сайте Олега содержится много материалов посвящённых подобным исследованиям. Настоятельно рекомендую ознакомиться с ними.

Листинг 7.11: POINTER-SCOPING-SKETCH

```
(defmacro! pointer-& (obj)
  `(lambda (&optional (,g!set ',g!temp))
     (if (eq ,g!set ',g!temp)
       ,obj
       (setf ,obj ,g!set))))

(defun pointer-* (addr)
  (funcall addr))

(defsetf pointer-* (addr) (val)
  `(funcall ,addr ,val))

(defsetf pointer-& (addr) (val)
  `(setf (pointer-* ,addr) ,val))
```

не сможем присвоить значению замыкания некоторое определённое значение, поскольку оно может создавать коллизии с нашей реализацией. Например, мы можем выбрать в качестве значения по-умолчанию лисповский nil, и такой подход, обычно, будет работать, за исключением случая когда мы попытаемся передать nil в качестве аргумента. Gensym-ы удобны для использования во время выполнения, поскольку мы знаем, что данному gensym-у не будет никакого **eq**'абельного значения. В этом их raison-d'etre (смысл жизни).

Pointer-* и его **defsetf** — это каркас для получения доступа к этим косвенным значениям через обобщённые переменные. **Defsetf**, относящийся к **pointer-&**, предназначен для того, чтобы расширение **pointer-&** знало как присваивать вложенные косвенности. В качестве простого примера мы можем создать замыкание, имитирующее шаблон *указатель на указатель (pointer to a pointer)* характерный для C, создав ссылку, привязывающуюся к let среде:

```
* (let ((x 0))
    (pointer-& (pointer-& x)))

#<Interpreted Function>
```

Сохраним это замыкание для дальнейшего использования и переместим его в специальную переменную * (сохраним всё, что содержал этот астериск):

```
* (defvar temp-pointer *)

#<Interpreted Function>
```

Теперь мы можем *разыменовать (dereference)* это замыкание:

```
* (pointer-* temp-pointer)

#<Interpreted Function>
```

Похоже что у нас есть ещё одно замыкание. Мы разыменовали только одно звено в цепи указателей. Используем специальную переменную * для того, чтобы сослаться на последний результат и продолжим разыменование:

```
* (pointer-* *)

0
```

0 — это исходный объект, на который мы ссылались. Кроме того, мы можем использовать этот разыменовывающий синтаксис — который является иллюзией из замыканий — и присвоить ему значение через цепь из указателей:

```
* (setf (pointer-* (pointer-* temp-pointer)) 5)

5
```

Конечно же здесь меняется исходная let среда, на которую ссылается указатель, на новое значение 5:

```
* (pointer-* (pointer-* temp-pointer))

5
```

При желании мы можем добавить другой слой косвенности:

```
* (pointer-& temp-pointer)

#<Interpreted Function>
```

И теперь потребуются три разыменования:

```
* (pointer-* (pointer-* (pointer-* *)))

5
```

И по-прежнему можно получать доступ как к обобщённой переменной:

```
* (setf (pointer-* (pointer-* (pointer-* **))) 9)

9
```

И, хотя они могут находиться в разных уровнях косвенности, все эти замыкания в данной разыменовывающей цепи указывают на исходную let среду:

```
* (pointer-* (pointer-* temp-pointer))

9
```

Но, возможно это не то о чём вы подумали, когда мы разговаривали о области видимости указателя. Поскольку большинство компьютерных процессоров рассматривают память как большой массив fixnum-ов, и поскольку C был спроектирован с учётом возможностей существующих процессоров, то область видимости указателей в C неразрывно связана с fixnum арифметикой. В C, при разыменовании указателя вы всегда знаете что происходит: компилятор компилирует в коде индекс в памяти с fixnum-ом и получает или присваивает fixnum значение. Самая большая разница между областью видимости указателя в C и нашей техникой разыменования замыкания в том, что C позволяет нам изменять направление указателя добавив к нему или вычтя из него fixnum-ы, в отличие от статических замыканий, скомпилированных с помощью **pointer-&**, и получения доступа с помощью **pointer-***. Код получения доступа и присваивания значения — вне зависимости чего-либо — добавлен в косвенную среду во время компиляции. Даже в нашем простом примере мы использовали по крайней мере две различных разновидности замыканий, доступ к которым, благодаря обобщённым переменным, осуществляется через унифицированный синтаксис разыменования. Исходная **x**, к которой мы ссылались, была лексической переменной, а temp-pointer — *туннельная* переменная, используемая для ссылки, — являлась динамической переменной. С помощью *раздела 6.7, Пандорические Макросы* мы можем изменять замыкания, а следовательно, можем изменять косвенность так, как только захотим.

Таким образом, оказывается, что замыкания более гибки и менее опасны, нежели указатели в стиле C. Если вы думаете, что вам нужен указатель, то возможно, что на самом деле вам нужно замыкание. И если fixnum можно использовать как адрес, то замыкания — это код, скомпилированный для того, чтобы получать и изменять любые разновидности данных в любых разновидностях сред. И хотя для большинства задач замыкания — это наилучшая конструкция для достижения косвенности, иногда нам может понадобиться функциональность процессора в области адресации с помощью fixnum для достижения чрезвычайно эффективного кода. C позволяет нам достичь этого; COMMON LISP ещё лучше выполняет ту же операцию.

Использовать в лиспе указатели в C — стиле на самом деле очень просто и не требует от нас отказа от обычной лисп техники. Мы просто обеспечим поддержку fixnum массива и используем числовые индексы для индексации в этом массиве — то есть, думаем именно в C стиле. Затем используем декларации для того, чтобы лисп отбросил проверки типов и безопасности, в результате скомпилированный код будет похож на C код. И, наконец, мы используем макросы для того, чтобы сделать процесс в целом удобным и безопасным.

В целом индексация в массиве — это сложная, медленная процедура. Компилятору нужно проверить, что ваш индекс — это числовое значение, что вы пытаетесь индексировать массив, и что индекс находится в пределах массива. Кроме того, массивы различных типов могут иметь различный код для доступа к элементам. Загрузив код из этой книги, попробуйте вычислить следующую форму (dis в деталях описан в *разделе 7.3, Знакомство с Вашим Дизассемблером*):

```
(dis (arr ind)
  (aref arr ind))
```

Поскольку при неизвестных типах aref может означать большое множество чего-либо, то, скорее всего, ваш компилятор не будет встраивать код доступа к массиву. В вышеприведённом выводе дизассемблера вы должны обнаружить вызов функции похожей на CMUCL-овский **data-vector-ref**. Упражнение: Скачайте исходный код вашей лисп среды и изучите эту функцию. В CMUCL этим файлом будет **array.lisp**. Кроме того изучите другие функции, содержащиеся в этом файле, включая **data-vector-set**. Если ваша лисп среда не поставляется с полным исходным кодом, или вы не можете выполнить упражнение с имеющимся исходным кодом, то обновите вашу COMMON LISP среду как можно скорее.

COMMON LISP может встраивать функцию + при достаточном объёме информации, аналогично возможно встраивание и **aref**. Воспользуйтесь следующей формой:

```
(dis (((simple-array fixnum) arr)
      (fixnum ind))
  (aref arr ind))
```

Вышеприведённый код должен переместить косвенность в функцию ссылки общего массива. Простые массивы — это массивы с одним измерением, где расположение элементов подобно памяти в C стиле. В этом коде мы определили в качестве элемента массива **fixnum**, но в вашей COMMON LISP среде возможно представление типа fixnum в другом размере, например: в байтах, в без знаковых байтах, с плавающей запятой, с плавающей запятой двойной точности и многих других. И хотя вышеприведённый код не содержит косвенности, он по прежнему содержит код, реализующий проверку типа и безопасности, на которые мы обычно полагаемся при программировании в лиспе. Однако, мы уже использовали считывающий макрос sharp-f из *раздела 7.2, Макросы Ускоряют Лисп* для того, чтобы сообщить лиспу о необходимости быстрой арифметики, то же самое можно сделать и для ссылок на массив:

```
(dis (((simple-array fixnum) arr)
      (fixnum ind))
  #f
  (aref arr ind))
```

В отличие от предыдущих **aref-ов** в этом коде на первый план выходит не производительность, а проверки типов и безопасности. Этот код предназначен для использования внутри циклов, в которых важна производительность. Заметьте, что здесь мы удалили из кода почти всю безопасность и теперь этот код почти также опасен, как и его C эквивалент. В частности, он подвержен проблемам *переполнения буфера (buffer overflow)*[15]. В C подобный стиль программирования применяется везде. В лиспе везде применяется безопасное программирование, за исключением *горячих точек (hotspots)*, где вы модифицируете код с целью ускорения программы в целом. Благодаря макросам, количество подобных горячих точек может быть очень мало. Отсутствует нужда в компиляции, например, всех функций в быстром/опасном режиме. Макросы позволяют нам оптимизировать узкие, специфичные части выражения. Быстрый

[15]"Переполнение буфера" — общий термин, характеризующий различные возможные проблемы безопасности в C (и иногда даже лисп) программах.

код прозрачно сосуществует с безопасным кодом, и макросы позволяют сохранить минимальную безопасность вкупе с достижением требуемой производительности.

Поскольку вы уже сильно продвинулись в чтении этой книги, то к этому моменту у вас уже должны сформироваться хорошие представления о создании макросов и деклараций, и в отношении области видимости указателей у нас осталось не так уж много тем. Вкратце: C предоставляет из себя очень специфичный предметно-ориентированный язык для контроля ЦПУ на базе целочисленной арифметики, но вы можете написать более лучшие языки если воспользуетесь макросами. Эффективная область видимости указателя (теперь мы можем признать, что это означает доступ к массиву — вопреки примерам с замыканиями) — это, в основном, вопрос знания работы макросов, знание работы деклараций и умение читать ваш дизассемблер.

Пример макроса, реализующий эффективное получение доступа к массивам — **with-fast-stack**. Этот макрос выбран не случайно — он даёт возможность обсудить *амортизацию (amortisation)*. **With-fast-stack** реализует стековую структуру данных под названием **sym**. В отличие от COMMON LISP-овских **push** и **pop** стеков использующих cons ячейки для хранения элементов любых типов, эти стеки используют простой массив для хранения элементов фиксированного типа, определённых с помощью ключевого слова **:type**. Кроме того, размер этого массива фиксирован и определяется с помощью ключевого слова **:size**. Доступ к стеку осуществляется с помощью нескольких локальных макросов, определённых с помощью **macrolet**. Если вы назовёте ваш стек как **input**, то макросы будут привязаны к **fast-push-input**, **fast-pop-input** и **check-stacks-input**. Изучите скомпилированное расширение с помощью **dis**:

```
* (dis ((fixnum a))
    (with-fast-stack (input :size 2000)
      (loop for i from 1 to 1000000 do
        (fast-push-input a))))
```

Операция **fast-push-input** компилируется в очень короткий (и очень небезопасный) машинный код:

```
;;; [8] (FAST-PUSH-INPUT A)
MOV ECX, [EBP-20]
MOV EDX, [EBP-16]
MOV EAX, [EBP-12]
MOV [ECX+EDX+1], EAX
```

Листинг 7.12: WITH-FAST-STACK

```
(defmacro! with-fast-stack
           ((sym &key (type 'fixnum) (size 1000)
                      (safe-zone 100))
            &rest body)
  `(let ((,g!index ,safe-zone)
         (,g!mem (make-array ,(+ size (* 2 safe-zone))
                             :element-type ',type)))
     (declare (type (simple-array ,type) ,g!mem)
              (type fixnum ,g!index))
     (macrolet
       ((,(symb 'fast-push- sym) (val)
            `(locally #f
               (setf (aref ,',g!mem ,',g!index) ,val)
               (incf ,',g!index)))
        (,(symb 'fast-pop- sym) ()
           `(locally #f
              (decf ,',g!index)
              (aref ,',g!mem ,',g!index)))
        (,(symb 'check-stack- sym) ()
           `(progn
              (if (<= ,',g!index ,,safe-zone)
                (error "Stack underflow: ~a"
                       ',',sym))
              (if (<= ,,(- size safe-zone)
                      ,',g!index)
                (error "Stack overflow: ~a"
                       ',',sym)))))
        ,@body)))
```

```
MOV EAX, [EBP-16]
ADD EAX, 4
MOV [EBP-16], EAX
```

Но цикл был реализован как обычно, с реализацией проверок на ошибки и косвенность в арифметических функциях, даже несмотря на то, что он находится внутри макроса **with-fast-stack**.

```
;;; [7] (LOOP FOR I FROM 1 ...)
...
CALL    #x1000O1D0 ; #x1000O1D0: GENERIC-+
...
CALL    #x10000468 ; #x10000468: GENERIC->
```

Очевидно, что этот цикл будет исполняться не так быстро, как могло бы быть. Его производительность будет преобладать над накладными расходами выполнения цикла, а не операциями стека. Если нам нужна скорость, то мы должны явно декларировать **i** как целочисленный тип и добавить отдельные декларации скорости для цикла так, как мы это делали до сих пор. Безопасный код может сосуществовать с быстрым кодом. Конечно, код, который мы только что дизассемблировали — чрезвычайно опасен. Он даже не проверяет размер стека для контроля перехода за нижнюю или верхнюю границы стека. Этим мы расплатились за эффективность. Решение, реализованное в **with-fast-stack** вдохновлено словом **stacks (стеки)** из языка программирования *forth (форт)*. С помощью локального макроса **check-stacks-input** наш код может проверить, находится ли стек в границах, и выдаст ошибку в противном случае. Поскольку форт спроектирован с учётом существования на крайне ограниченных платформах, то форт *амортизирует* цену, которую приходится платить за проверку границ. Вместо того, чтобы выполнять проверку после каждой операции, как это по-умолчанию делает лисп, форт выполняет проверку после каждых N операций. Часто в форте это слово вызывается только после вычисления формы в REPL (мы сосредоточимся на форте в *главе Лисп, Переходящий в Форт, Переходящий в Лисп*). Таким образом, вместо того, чтобы проверять границы после каждой операции, мы запускаем проверку через каждые 10 операций, что позволяет уменьшить цену проверки границ на, примерно, 90%[16]. При проверке стека мы знаем, что в худшем случае за пределами

[16]Хотя подсчёт каждой выполненной операции вносит свою долю в понижение быстродействия.

границ окажутся 10 элементов. Или что это участок кода с не-критичной-производительностью, допускающей выполнение макроса проверки.

Другая особенность **with-fast-stack** в том, что созданные массивы обладают *зонами безопасности (safe zones)*. То есть, выделяется дополнительная память по обе стороны стека, которая будет использоваться в качестве *тормозного пути (run-away lane)* при выходе за границы стека. Это не означает, что работа в зонах безопасности будет хорошей идеей (особенно при переходе за нижнюю границу), но данный вариант всё же лучше, чем выход в участки памяти, не предназначенные для работы со стеком.

Как было замечено выше — код, который мы только что собрали, — очень опасен и может записать целочисленные значения в не предназначенную для этого память. Никогда не делайте этого. Упражнение: Сделайте это. Вот что у меня получилось:

```
* (compile nil
    '(lambda (a)
       (declare (type fixnum a))
       (with-fast-stack (input :size 2000)
         (loop for i from 1 to 1000000 do
           (fast-push-input a)))))

#<Function>
NIL
NIL
```

Опасный код удачно скомпилирован. Попробуем запустить его:

```
* (funcall * 31337)

NIL
```

Ну что же, катастрофы, которой мы боялись, не произошло. Случится ли что-то плохое?

```
* (compile nil '(lambda () t))

; Compilation unit aborted.
```

Хм, выглядит нехорошо.

```
* (gc)
Help! 12 nested errors.
KERNEL:*MAXIMUM-ERROR-DEPTH* exceeded.
** Closed the Terminal

NIL
```

Это определённо выглядит нехорошо. Поскольку лисп — это unix процесс, то, возможно, что вы получите сигнал, обозначающий запись за пределами выделенной вам виртуальной памяти (называется *segfault (ошибка сегментации)*). CMUCL обрабатывает эти сигналы как исправляемые состояния (хотя в таких случаях вы должны перезагрузить ваш лисп образ):

```
Error in function UNIX::SIGSEGV-HANDLER:
   Segmentation Violation at #x58AB5061.
   [Condition of type SIMPLE-ERROR]
```

Лисп образ в таком состоянии называется *повреждённым (hosed)*. Программы, которые могут стать повреждёнными — это катастрофа безопасности, ждущая своего часа. Разница между C и лиспом выражается в том, что риск повреждения программ на C присутствует практически везде, а риск повреждения лисп программ — практически нигде. Если вы согласны принять подобные риски безопасности для массива, созданного на области видимости указателя, то лисп макросы — это наименее навязчивый и безопасный способ реализации. Конечно, вы практически никогда не согласитесь на такие риски — просто придерживайтесь замыканий.

7.5 Tlist-ы и Cons Пулы

Этот раздел рассказывает об управлении памятью, но, здесь вы можете не найти то, что хотели бы узнать. Я был вынужден включить в книгу этот раздел, поскольку боялся увековечивания одного из неверных мифов о лиспе, утверждающего что cons — это медленная операция. Извините, но это просто неправда; cons на самом деле быстр [GC-IS-FAST]. Конечно, алгоритмы, минимизирующие неопределённо расширяемое хранилище, обычно идеальны, но, большинство алгоритмов можно написать более просто и прямо, если воспользоваться cons-ом. Не бойтесь использовать cons-ы, если вам нужна память. На самом деле, иногда самая лучшая оптимизация, которая может быть выполнена в лис-

Листинг 7.13: TLISTS

```
(declaim (inline make-tlist tlist-left
                 tlist-right tlist-empty-p))

(defun make-tlist () (cons nil nil))
(defun tlist-left (tl) (caar tl))
(defun tlist-right (tl) (cadr tl))
(defun tlist-empty-p (tl) (null (car tl)))
```

пе, — это приведение алгоритма в форму, выраженную через cons ячейки, после этого мы можем воспользоваться возможностями настроенного лисповского сборщика мусора. Также, как написание своей собственной хэш-таблицы — это, скорее всего, плохая идея, также и хакание своих собственных процедур распределения памяти — это, скорее всего, тупая идея. Тем не менее в этом разделе описываются некоторые способы распределения памяти. Сюрприз, мы сделаем это с помощью макросов.

Прежде чем вернуться к распределению памяти, мы поговорим о другой теме. Несмотря на то, что COMMON LISP — это инструмент, который выбирают профессиональные лисп программисты, многие лучшие учебники-о-введении-в-лисп написаны с использованием *Scheme*. Как правило, наиболее почитаемой является *Структура и Интерпретация Компьютерных Программ* [SICP] за авторством Харольда Абельсона, Джеральда Сассмана и Джули Сассман. SICP[17] изучался первокурсниками МТИ (MIT) и боготворился ими на протяжении десятилетий. Привлекательность Scheme для академических кругов была глубокой и всеобъемлющей. Большинство макро профессионалов начали постигать лисп с помощью Scheme — только тогда, когда они были готовы начать программировать серьёзные макросы, они перешли на язык для хакеров макросов: COMMON LISP.

Но при переходе вы всегда берёте с собой некоторые инструменты. Вы не убежите от вашего прошлого — ваших корней. Если ваши корни — это Scheme и вы читали SICP, то, скорее всего, вы помните *очереди (queues)* (также почитайте [USEFUL-LISP-ALGOS1-CHAPTER3]). Другое описание, которое мы будем использовать, дано в другой книге, посвящённой Scheme — *Схемы Вычислений (Schematics of Computation* [SCHEMATICS]), в ней очередь называется *tlist*. Tlist — это структура данных, названная в честь её открывателя, Interlisp хакера Уоррена

[17]Произносится как сик-пи (sick-pea).

Листинг 7.14: TLIST-ADD

```
(declaim (inline tlist-add-left
                 tlist-add-right))

(defun tlist-add-left (tl it)
  (let ((x (cons it (car tl))))
    (if (tlist-empty-p tl)
      (setf (cdr tl) x))
    (setf (car tl) x)))

(defun tlist-add-right (tl it)
  (let ((x (cons it nil)))
    (if (tlist-empty-p tl)
      (setf (car tl) x)
      (setf (cddr tl) x))
    (setf (cdr tl) x)))
```

Тейтелмана (Warren Teitelman). И хотя tlists дан в виде Scheme кода в *"Схемы Вычислений"*, мы будем использовать порт на COMMON LISP.

Как мы можем выяснить из конструктора, **make-tlist**, tlist — это всего лишь cons ячейка. Но, вместо того, чтобы использовать car в роли элемента, а cdr в роли следующего cons как в обычном списке, tlist использует car для того, чтобы сослаться на первый cons настоящего списка, а cdr — для последнего. Если car tlist-а **nil**, то tlist считается *пустым*. В отличие от обычных списков, пустые tlist-ы разнообразны (не **eq**-абельны). Car tlist-а указывает на cons ячейку, хранящую *левый* элемент tlist-а. Cdr указывает на cons, хранящий *правый* элемент.

Функции **tlist-left** и **tlist-right** возвращают левый и правый элементы tlist-а без модификации самого tlist. Если tlist будет пустым, то эти функции вернут **nil**. Если вы будете использовать только эти функции, то вы не сохраните **nil** в ваш tlist. К счастью, вы можете проверить пустоту tlist-а перед тем как использовать его с предикатом **tlist-empty-p** и, тем самым, сохранить **nil** в tlist.

Поскольку эти операции выполняются очень просто, то мы решили сообщить компилятору что все эти функции должны быть *встроены*[18]. Это позволит лисп компилятору генерировать более эффективные расширения для функций tlist. В некоторых языках не предоставляющих

[18] **Declaim** — это разновидность глобальной версии **declare**.

Листинг 7.15: TLIST-REM-LEFT

```
(declaim (inline tlist-rem-left))

(defun tlist-rem-left (tl)
  (if (tlist-empty-p tl)
    (error "Remove from empty tlist")
    (let ((x (car tl)))
      (setf (car tl) (cdar tl))
      (if (tlist-empty-p tl)
        (setf (cdr tl) nil)) ;; For gc
      (car x))))
```

достаточный контроль компилятора — например C — примитивные макро системы применяются для того, чтобы быть уверенными в том, что такие функции, как наша tlist утилита будут встроенными. В лиспе, где мы полностью контролируем компилятор, нет потребности в использовании макросов для таких нужд. Макросы, о которых идёт речь в этой главе, — это гораздо больше чем простое встраивание.

Мы можем добавлять элементы с левой стороны tlist с помощью функции **tlist-add-left**, и с правой стороны с помощью **tlist-add-right**. Поскольку указатель на конец списка всегда под рукой, то добавление элементов в конец tlist — это операция с *постоянной временной сложностью*, не зависящей от длины tlist. Однако, в целом, операция добавления к tlist не в полной мере представляет из себя постоянную временную сложность, поскольку сюда добавляются накладные расходы привнесённые cons-ом и связанным с ним выделением памяти. Обычно использование **cons** обозначает, что tlist добавление включает в себя агрегированную побочную нагрузку от сборки мусора.

Данными функциями поддерживается только удаление элемента с левой стороны tlist. Поскольку мы храним указатели только к первому и последнему элементам tlist-а, то единственный способ найти предпоследний элемент — это пройти через весь список, начиная с левой стороны tlist.

Tlist — это абстракция очереди, построенная на основе cons ячеек, особенно удобная из-за *прозрачных* структур данных. В то время, когда другие структуры данных реализуют tlist функциональность — такие как очереди — предоставляя вам только ограниченный интерфейс к структуре данных, tlist-ы же прямо определены как cons ячейки. Вместо создания некоторого API, которое бы удовлетворяло потребности

Листинг 7.16: TLIST-UPDATE

```
(declaim (inline tlist-update))

(defun tlist-update (tl)
  (setf (cdr tl) (last (car tl))))
```

Листинг 7.17: COUNTING-CONS

```
(defvar number-of-conses 0)

(declaim (inline counting-cons))

(defun counting-cons (a b)
  (incf number-of-conses)
  (cons a b))
```

всех, Тейтелман решил определить спецификацию tlist непосредственно на лисповской cons ячейке. Это архитектурное решение отделяет tlist от других реализаций очереди. При программировании с использованием прозрачных спецификаций, мы не создаём специальные API функции, выполняющие что-либо, нет, сам код *и есть* API.

Если мы решим получить доступ к car tlist-a и модифицировать его содержимое, то нам нужно убедиться в том, что tlist по-прежнему остаётся последовательным. Предполагая, что после наших манипуляций желаемый список будет сохранён в car tlist-a, мы можем использовать **tlist-update** для присваивания соответствующего значения для cdr[19].

Таким образом, главный плюс tlist — это максимально близкая эмуля-

[19]Часто существует более эффективный способ сохранения последнего элемента списка в cdr tlist-a. Так мы можем избежать использования tlist-update, операции со сложностью, линейно-зависимой-от-длины-списка. Поскольку спецификация tlist прозрачна, то оба способа корректны.

Листинг 7.18: WITH-CONSES-COUNTED

```
(defmacro! with-conses-counted (&rest body)
  `(let ((,g!orig number-of-conses))
     ,@body
     (- number-of-conses ,g!orig)))
```

Листинг 7.19: COUNTING-PUSH

```
(defmacro counting-push (obj stack)
  `(setq ,stack (counting-cons ,obj ,stack)))
```

ция лисп списков, с поддержкой операции добавления элементов в концы списка с постоянной сложностью по времени. Поскольку tlist использует cons также, как и обычные списки, то накладные расходы на память остаются теми же самыми.

В COMMON LISP нет достаточной функциональности для отслеживания и контроля выделения памяти. Давайте напишем эту функциональность. Первым делом, вспомним, что в *разделе 3.5, Нежелательный Захват*, говорится, что не разрешается переопределять или заново привязывать функции, определённые в COMMON LISP. Мы не можем прямо перехватывать **cons** вызовы, поэтому мы воспользуемся *обёрткой (wrapper)*. **Counting-cons** — это функция, идентичная **cons** с тем исключением, что она при каждом вызове инкрементирует переменную **number-of-conses**.

With-conses-counted — это наш главный интерфейс, предназначенный для проверки значения **number-of-conses**. Его расширение будет записывать начальное значение, выполнять операции, указанные в теле макроса, и затем возвращать количество вызовов **counting-cons**.

К несчастью, наша стратегия переименования **cons** в **counting-cons** приводит к тому, что для проверки любого кода на потребление памяти нам приходится переписывать применение **counting-cons** на **counting-push**. Ниже мы можем увидеть, что при каждом вызове **counting-push** **counting-cons** вызывается только единожды:

```
* (let (stack)
    (with-conses-counted
      (loop for i from 1 to 100 do
        (counting-push nil stack)
        (pop stack))))
100
```

Вышеприведённый оператор **pop** удаляет использованные элементы и cons ячейки предназначенные для хранения в стеке. Что происходит с этими cons ячейками? Они становятся мусором. Обычно лисп отбрасывает этот мусор и в дальнейшем никто об этом мусоре не заботится, поскольку в среде COMMON LISP существуют превосходные очищаю-

Листинг 7.20: WITH-CONS-POOL

```
(defmacro with-cons-pool (&rest body)
  `(let ((cons-pool)
         (cons-pool-count 0)
         (cons-pool-limit 100))
     (declare (ignorable cons-pool
                         cons-pool-count
                         cons-pool-1imit))
     ,@body))
```

щие программы, известные как *сборщики мусора (garbage collectors)*, занимающиеся очисткой хранилища. Однако, сборка мусора не даётся бесплатно — кто-то должен платить за сбор мусора, его транспортировку и переработку в повторно используемый продукт. А что, если мы создадим здесь программу мини-переработки? Например, вышеприведённый loop вызывает **counting-cons** 100 раз, генерируя 100 кусочков мусора, требующих сборки. Однако, беглое чтение кода покажет что в **stack'e** никогда не находится более одного элемента одновременно. Если мы утилизируем эту cons ячейку, то она снова станет доступна для **counting-push**, мы можем избежать вызова **counting-cons** для получения другой cons ячейки. Эта концепция известна под названием *cons пула (cons pool)*. В дополнение к понижению нагрузки на сборщик мусора, cons пулы могут улучшить *локализацию* часто выделяемых в памяти структур данных.

With-cons-pool — это способ с помощью которого мы можем создавать cons пулы. Заметьте, что этот макрос расширяется в let форму, создающую привязку для **cons-pool**, **cons-pool-count** и **cons-pool-limit**. Эти переменные будут использоваться для хранения cons ячеек, подлежащих переработке. Поскольку такой подход привносит невидимость переменных, то **with-cons-pool** будет *анафорическим макросом*. Заметьте, что поскольку COMMON LISP поддерживает *двойственный синтаксис* для лексических и динамических переменных, то анафорические привязки, созданные расширением этого макроса, могут создавать как динамические, так и лексические привязки, в зависимости от места объявления анафоры: на стороне макроса или на другой стороне.

Cons-pool-cons расширяется в некоторый код, извлекающий cons ячейки из cons пула. Подразумевается что работа происходит внутри лексической области видимости **with-cons-pool**, или, если анафора объявлена специальной, то существуют динамические привязки к ним. При пустом пуле **cons-pool-cons** только вызывает **counting-cons**. При до-

Листинг 7.21: CONS-POOL-CONS

```
(defmacro! cons-pool-cons (o!car o!cdr)
  `(if (= cons-pool-count 0)
     (counting-cons ,g!car ,g!cdr)
     (let ((,g!cell cons-pool))
       (decf cons-pool-count)
       (setf cons-pool (cdr cons-pool))
       (setf (car ,g!cell) ,g!car
             (cdr ,g!cell) ,g!cdr)
       ,g!cell)))
```

Листинг 7.22: CONS-POOL-FREE

```
(defmacro! cons-pool-free (o!cell)
  `(when (<= cons-pool-count
             (- cons-pool-limit 1))
     (incf cons-pool-count)
     (setf (car ,g!cell) nil)
     (push ,g!cell cons-pool)))
```

стижении ограничения **cons-pool-limit** этот макрос ничего не будет сохранять.

Если мы обнаружим, что нам больше не нужна cons ячейка, то мы можем переместить её в cons пул, освобождая с помощью **cons-pool-free**. Если произошёл такой случай, то код должен пообещать что больше никогда не попытается получить доступ к освобождённой ячейке. Код, в который расширяется **cons-pool-free**, поместит освобождённую cons ячейку в **cons-pool** и увеличит **cons-pool-count**, если же **cons-pool-count** окажется больше чем **cons-pool-limit**, то в этом случае ячейка будет оставлена для сборки мусора. Заметьте, если вы решили что вам эти cons ячейки больше не нужны, то не требуется прибегать к **cons-pool-free** для очистки cons ячеек поскольку сборщик мусора в состоянии определить, когда их следует очистить. Освобождение cons ячеек — это простая и эффективная оптимизация которую можно осуществить, в том случае, если нам доступна некоторая информация, не доступная лиспу.

Таким образом архитектура cons пулов состоит из двух макросов, один из которых создаёт анафору, невидимо вводящую лексические или специальные привязки, и другой макрос, использующий эту анафору. Обычно для *комбинирования* этих макросов используется другой мак-

Листинг 7.23: MAKE—CONS-POOL-STACK

```
(defmacro make-cons-pool-stack ()
  `(let (stack)
     (dlambda
       (:push (elem)
         (setf stack
               (cons-pool-cons elem stack)))
       (:pop ()
         (if (null stack)
           (error "Tried to pop an empty stack"))
         (let ((cell stack)
               (elem (car stack)))
           (setf stack (cdr stack))
           (cons-pool-free cell)
           elem)))))
```

рос. **Make-cons-pool-stack** — это один из таких примеров. Этот макрос создаёт структуру данных, подобную стеку COMMON LISP, которая, конечно, на самом деле представляет всего лишь список, обновляемый макросами **push** и **pop**. Однако, наша структура данных отличается от **push** и **pop** по причине непрозрачной спецификации. Детали реализации этих стеков отделены от их использования. Это важно, поскольку мы не хотим чтобы пользователи наших стеков применяли свои собственные методы вставки и извлечения данных, вместо этого мы хотим чтобы пользователи использовали наши, оптимизированные по памяти, версии. **Make-cons-pool-stack** использует **dlambda** из *раздела 5.7, Dlambda*. Вот пример, в котором мы создаём лексический cons пул, скрывающий в себе новую стековую структуру данных, и затем вставляем в стек и извлекаем из стека один элемент 100 раз подряд:

```
* (with-cons-pool
    (let ((stack (make-cons-pool-stack)))
      (with-conses-counted
        (loop for i from 1 to 100 do
          (funcall stack :push nil)
          (funcall stack :pop)))))

1
```

Заметьте, что **counting-cons** — функция, применяемая для размеще-

Листинг 7.24: MAKE-SHARED-CONS-POOL-STACK

```
(with-cons-pool
  (defun make-shared-cons-pool-stack ()
     (make-cons-pool-stack)))
```

Листинг 7.25: WITH-DYNAMIC-CONS-POOLS

```
(defmacro with-dynamic-cons-pools (&rest body)
  `(locally (declare (special cons-pool
                              cons-pool-count
                              cons-pool-limit))
     ,@body))
```

ния в памяти — вызвана только единожды. Потребовалась только одна cons ячейка — она использовалась повторно. Если такой цикл будет использоваться в скомпилированном коде и будет повторён достаточное количество раз, то можно ожидать, что версия с cons пулом будет работать быстрее, просто потому, что не будет вызываться сборка мусора. И что гораздо важнее, наш цикл не будет сталкиваться с неожиданными паузами во время работы сборщика мусора. Конечно, мы почти никогда не столкнёмся с подобными паузами, поскольку лисп достаточно умён и не будет выполнять за раз полную сборку мусора, вместо этого он *амортизирует (amortising)* операцию техникой, известной как *инкрементальная сборка (incremental collection)*. Сборщики мусора также реализуют оптимизацию на *принципе поколений (generational collection)*, где недавно выделенная память будет очищаться чаще чем давно выделенная память. Удивительно, но такая сборка мусора оказывается разновидностью подсчёта ссылок [UNIFIED-THEORY-OF-GC].

Но, с помощью cons пулов, мы выполняем меньше (или вовсе не выполняем) cons операций, а это, в свою очередь, уменьшает (или устраняет) индетерминизм времени, затрачиваемого на сборку мусора. Большинство лисп систем поддерживают временное отключение сборки мусора, так вы сможете выполнять что-либо без пауз, и допустить более продолжительную паузу в тот момент времени, когда пауза уже не будет иметь значения. Для этих целей в CMUCL вы можете применить функции **gc-on** и **gc-off**. Также взгляните на код в **signal.lisp**. Упражнение: Отключите сборку мусора, а затем с помощью loop ccons-ите большое количество мусора. Используйте юниксовскую программу **top** для отслеживания потребления памяти.

Листинг 7.26: FILL-CONS-POOL

```
(defmacro fill-cons-pool ()
  `(let (tp)
     (loop for i from cons-pool-count
                 to cons-pool-limit
           do (push
                (cons-pool-cons nil nil)
                tp))
     (loop while tp
           do (cons-pool-free (pop tp)))))
```

И, хотя вышеприведённая реализация стека, требует от вас расположения в одном лексическом контексте с **with-cons-pool** для обозначения стеков, вам может понадобится поделится cons пулом, благодаря прозрачной архитектуре этих макросов, мы можем скомбинировать их с замыканиями для обозначения локальности так, как нам хочется. **Make-shared-cons-pool-stack** работает также, как и **make-cons-pool-stack** с тем исключением, что этот макрос не требует от вас обёртки с **with-cons-pool**. Эти переменные будут захвачены. Таким образом, все стеки, созданные с помощью **make-shared-cons-pool-stack**, будут разделять между собой один cons пул.

Благодаря двойственности синтаксиса между лексической и специальной переменными мы можем использовать динамическую среду для хранения наших cons пулов. Макрос **with-dynamic-cons-pools** преобразовывает ссылки cons пула к лексической области видимости в динамические привязки анафоры. Одна из стратегий заключается в оборачивании всего кода, использующего cons пулы, в **with-dynamic-cons-pools**, после чего во время исполнения вашей программы получатся динамические привязки, созданные для cons пула. Поскольку вы можете затенить динамические привязки новыми динамическими привязками, то вы можете сохранить размещение в любой динамической детализации. Для создания динамических привязок, просто оберните **with-dynamic-cons-pools** вокруг **with-cons-pool**.

Особенно для того, чтобы понизить индетерминизм времени выполнения сборки мусора, может возникнуть необходимость убедиться в том, что cons пул содержит доступные ячейки в пуле, и что программа не будет cons-ить всё (предполагаем, что мы не исчерпаем весь пул). Для этого нужно при инициализации ccons-ить необходимое количество ячеек — когда cons ещё допустим — затем добавить их в пул через **fill-cons-**

pool, заполняя cons пул до достижения **cons-pool-limit**.

Память — это очень сложная тема и эффективность работы с памятью зависит от вашего оборудования, вашей реализации лиспа и особенностей технологий. Если вы не до конца понимаете причины и последствия ваших действий, то попытки улучшить процедуры работы с памятью могут привести к гораздо большим проблемам, чем пользе. Системные программисты постоянно производят тонкие настройки с памятью на протяжении существования всей системы. И, безусловно, эта работа будет продолжаться ещё долгое время. Управление памятью — это тяжёлая работа, и можно быть уверенным в том, что макросы — это лучший инструмент для такой работы.

7.6 Сортирующие Сети

Нет лучшего инструмента для экспериментов с эффективностью или непосредственно реализацией эффективных программ чем лисп. Лисп уникален не только потому, что позволяет нам сконцентрироваться на умных алгоритмах и архитектурах, но и тем, что позволяет нам использовать эти алгоритмы и архитектуры с максимальной эффективностью заложенного потенциала используя мощь компиляторов машинного кода. Этот раздел описывает, с точки зрения лиспа, один из интенсивно изучаемых, но ещё не исчерпанный до конца раздел программирования: сортировку. Большинство людей считают сортировку решённой проблемой, и вы можете удивиться, когда узнаете, что до сих пор существует много важных открытых проблем сортировки.

Нам известно много замечательных алгоритмов сортировки общего назначения. Наиболее распространёнными являются такие алгоритмы, как quick sort (быстрая сортировка), причина этому — эффективная сортировка больших массивов данных. Но, если вместо больших массивов данных нам понадобится сортировать множество маленьких наборов данных, то такие алгоритмы сортировки общего назначения, как *quick sort*, могут оказаться излишними. Этот раздел посвящён решению подобных проблем, на эти проблемы люди тратят многие десятилетия, а простор для исследований всё ещё очень большой. И что ещё более важно для нас — эти решения дают возможность продемонстрировать расширенные техники оптимизации, простые в лиспе, но достаточно сложные в большинстве других языках программирования. В этом и следующем разделе мы повторно реализуем макрос, описанный Грэмом в *On Lisp* под названием **sortf** [ON-LISP-P176]. И если **sortf** Грэма спроектирован для иллюстрации создания макросов, применяющих обобщённые

переменные, наш **sortf** будет создан с упором на производительность. С учётом некоторых обстоятельств, наш **sortf** сможет достичь производительности, превышающей даже хорошо настроенную системную функцию **sort**.

Этот раздел посвящён моему учителю и другу Алану Паету (Alan Paeth), обучившему меня многому, и показавшему, что даже сортировка может быть интересной. Также я выражаю признательность Джону Гэмблу (John Gamble) за его замечательную Perl программу, **Algorithm-Networksort** [ALGORITHM-NETWORKSORT]. Эта программа была использована для экспериментов с различными алгоритмами и для генерации ASCII рисунков сетей приводимых в этом разделе.

Сортирующая сеть — это алгоритм, предназначенный, *очевидно*, для сортировки наборов данных различного фиксированного размера. То есть, в отличие от большинства таких алгоритмов, как quick sort, действие сортирующей сети не зависит от сортируемых наборов данных. Каждый шаг такой сортировки был заранее предрешён ещё на этапе разработки сети. Сортирующая сеть представляет из себя простой список пар, с индексами набора данных. Каждая из этих пар соответствует индексам, которые должны использоваться в операции сравнивания-замены. После выполнения всей последовательности операций сравнивания-замены элементы будут находиться в упорядоченном порядке.

Такие алгоритмы, как quick sort, отлично работающие с большими наборами данных, могут работать с недопустимыми накладными расходами для некоторых разновидностей задач сортировки. Во-первых, обычно реализации quick sort позволяют вам выбрать произвольный оператор сравнения с целью универсализации кода сортировки. Это означает, что при каждом сравнении будет происходить вызов функции сравнения, что можно было бы, скажем, реализовать как встраиваемый машинный код. Во-вторых, поскольку реализации quick sort являются универсальными, то часто они не могут воспользоваться оптимизациями, которые можно осуществить при работе с наборами данных малых, фиксированных размеров. В-третьих, довольно часто нам не нужно полностью сортировать весь набор данных, а достаточно отсортировать только некоторую часть данных — например, найти серединный элемент. Сортирующие сети, не выполняющие полную сортировку иногда называют *выбирающими сетями (selection networks)*.

Для объяснения того, что такое сортирующая сеть, и демонстрации их сложного устройства и неинтуитивности темы мы рассмотрим одну из простейших возможных сетей: сортировка трёх элементов. Большинство программистов знают, что для сортировки трёх элементов можно просто воспользоваться тремя сравнениями и не прибегать к использо-

Листинг 7.27: BAD-3-SN

```
(defvar bad-3-sn
  '((0 1) (0 2) (1 2)))
```

Листинг 7.28: BAD-3-SN-PIC

```
o--^--^-----o
   |  |
o--v--|--^--o
      |  |
o-----v--v--o
```

ванию quick sort для трёх элементов. Легко убедиться в том, что эти операции сравнения-замены могут быть выполнены в любом порядке, и это не влияет на конечный результат. Но не так легко увидеть, что одни способы сортировок менее эффективны чем другие.

Сеть **bad-3-sn** может выглядеть как наиболее очевидная реализация трёхэлементной сети, но — как следует из названия — это не самый оптимальный вариант. ASCII рисунок позволяет визуализировать алгоритм сети, описанный с помощью списковой записи в **bad-3-sn**. Алгоритм производит сравнение элементов под индексами 0 и 1 из набора данных и, если они не расположены по порядку, меняет их местами в правильном порядке. Далее следует выполнение тех же операций для пары индексов **(0 2)** и наконец для **(1 2)**. После этого процесса элементы будут отсортированы. Если мы реализуем эту сортирующую сеть в виде кода для массива из трёх элементов, назовём этот массив как **a**, то программа может выглядеть так[20]:

```
(progn
  (if (> (aref a 0) (aref a 1))
    (rotatef (aref a 0) (aref a 1)))
  (if (> (aref a 0) (aref a 2))
    (rotatef (aref a 0) (aref a 2)))
  (if (> (aref a 1) (aref a 2))
    (rotatef (aref a 1) (aref a 2))))
```

Bad-3-sn — корректная программа, но не так эффективна как **good-3-sn**. Изменив порядок первых двух операций сравнения-замены, мы получаем более эффективную сеть. В среднем эта сеть будет производить

[20] **Rotatef** — это лисповский оператор замены.

Листинг 7.29: GOOD-3-SN

```
(defvar good-3-sn
  '((0 2) (0 1) (1 2)))
```

Листинг 7.30: GOOD-3-SN-PIC

```
o--^--^-----o
   |  |
o--|--v--^--o
   |     |
o--v-----v--o
```

меньше операций замены, чем **bad-3-sn**. Для доказательства этого следует воспользоваться *условной вероятностью (conditional probability)*, но, поскольку эта книга повествует о лиспе, а не о сортирующих сетях, то мы уклонимся от этой темы. Вместо этого мы покажем, что **good-3-sn** лучше, чем **bad-3-sn**, перебирая все изменения и подсчитывая количество замен, происходящих при интерпретации этих двух сетей. Сейчас мы можем привести следующее интуитивное объяснение: если первой операцией будет произведена длинная связь, то после первой операции по крайней мере один минимальный или максимальный элемент будет находиться на своём окончательном месте. Таким образом, по крайней мере одна последующая операция не будет выполнять замену. Если же первым действием была осуществлена короткая связь, то есть вероятность, что ни один из этих элементов не будет находиться на окончательной позиции, и потребуются последующие замены.

Для изучения этого феномена мы реализуем интерпретатор для сортирующих сетей: **interpret-sn**. Этот интерпретатор будет применять сортирующую сеть **sn** к набору данных, представленных в виде списка. В качестве первого значения он будет возвращать число выполненных перемещений, а в качестве второго значения — результирующий набор отсортированных данных. Заметьте, что ссылающиеся сами-на-себя считывающие макросы #= и ## используются для предотвращения повторного набора форм доступа. Также следует отметить применение отслеживающей переменной, и если мы хотим отследить по-шаговое выполнение процесса сортировки, то эту переменную нужно привязать к не-null значению. Для начала рассмотрим уже отсортированный набор данных. Очевидно, что и **bad-3-sn** и **good-3-sn** не произведут ни одной операции замены:

Листинг 7.31: INTERPRET-SN

```
(defvar tracing-interpret-sn nil)

(defun interpret-sn (data sn)
  (let ((step 0) (swaps 0))
    (dolist (i sn)
      (if tracing-interpret-sn
        (format t "Step ~a: ~a~%" step data))
      (if (> #1=(nth (car i) data)
             #2=(nth (cadr i) data))
        (progn
          (rotatef #1# #2#)
          (incf swaps)))
      (incf step))
    (values swaps data)))
```

```
* (let ((tracing-interpret-sn t))
    (interpret-sn '(1 2 3) bad-3-sn))
Step 0: (1 2 3)
Step 1: (1 2 3)
Step 2: (1 2 3)
0
(1 2 3)
* (let ((tracing-interpret-sn t))
    (interpret-sn '(1 2 3) good-3-sn))
Step 0: (1 2 3)
Step 1: (1 2 3)
Step 2: (1 2 3)
0
(1 2 3)
```

Далее рассмотрим случай, когда каждый элемент расположен не на своём месте последовательности. И опять, обе сортирующие сети работают одинаково, выполняют необходимые две замены:

```
* (let ((tracing-interpret-sn t))
    (interpret-sn '(3 1 2) bad-3-sn))
Step 0: (3 1 2)
Step 1: (1 3 2)
```

```
Step 2: (1 3 2)
2
(1 2 3)

* (let ((tracing-interpret-sn t))
    (interpret-sn '(3 1 2) good-3-sn))
Step 0: (3 1 2)
Step 1: (2 1 3)
Step 2: (1 2 3)
2
(1 2 3)
```

Однако, в следующем примере результаты **bad-3-sn** оказываются *хуже*, что выражается в трёх шагах:

```
* (let ((tracing-interpret-sn t))
    (interpret-sn '(3 2 1) bad-3-sn))
Step 0: (3 2 1)
Step 1: (2 3 1)
Step 2: (1 3 2)
3
(1 2 3)
* (let ((tracing-interpret-sn t))
    (interpret-sn '(3 2 1) good-3-sn))
Step 0: (3 2 1)
Step 1: (1 2 3)
Step 2: (1 2 3)
1
(1 2 3)
```

Здесь **bad-3-sn** выполняет три замены, в то время когда оптимальный **good-3-sn** выполняет только одну замену. Может ли существовать такой симметричный случай, когда **good-3-sn** будет проигрывать **bad-3-sn**? Оказывается, такого случая не существует, **good-3-sn** лучше чем **bad-3-sn**. Если вы всё ещё не верите в это, то изучите *парадокс Монти Холла*, и тогда вы поймёте насколько не интуитивными могут быть проблемы такого рода. Интуитивно кажется, что для достижения результата с меньшим количество шагов элементы нужно менять местами сразу, как только появляется такая возможность.

Для количественной оценки превосходства **good-3-sn** перед **bad-3-sn** мы представляем утилиту **all-sn-perms**, генерирующую все возможные

Листинг 7.32: ALL-SN-PERMS

```
(defun all-sn-perms (n)
  (let (perms curr)
    (funcall
      (alambda (left)
        (if left
          (loop for i from 0 to (1- (length left)) do
            (push (nth i left) curr)
            (self (append (subseq left 0 i)
                          (subseq left (1+ i))))
            (pop curr))
           (push curr perms)))
       (loop for i from 1 to n collect i))
     perms))
```

перестановки чисел от 1 до **n**. **All-sn-perms** включает в себя большое количество лисп шаблонов, включая рекурсивный cons сетей из связанных, временных списков и использует анафорический макрос Грэма **alambda**. С помощью **all-sn-perms** мы можем сгенерировать все 6 (факториал 3) перестановок чисел от 1 до 3:

```
* (all-sn-perms 3)
((1 2 3) (2 1 3) (1 3 2)
 (3 1 2) (2 3 1) (3 2 1))
```

Поскольку из-за особенности написания **all-sn-perms** вышеприведённые списки разделяют свою структуру с другими списками, то при использовании их для интерпретации сортирующих сетей (деструктивная операция) нам нужно убедиться в том, что мы будем сортировать их копии, как это реализовано в **average-swaps-calc**. Для задач такого рода подобное структурирование с разделением структуры — это, в большинстве случаев, хорошая программистская техника, поскольку она позволяет уменьшить общее потребление памяти для ваших структур данных[21].

Используя наш интерпретатор сортирующей сети **interpret-sn**, мы можем выяснить действительное количество перемещений, потребовавшихся для всех возможных комбинаций с помощью **average-swaps-calc**.

[21] Хотя данный код появился здесь по той причине, что это наиболее простой способ получить требуемый результат.

Листинг 7.33: AVERAGE-SWAPS-CALC

```
(defun average-swaps-calc (n sn)
  (/ (loop for i in (all-sn-perms n) sum
       (interpret-sn (copy-list i) sn))
     (fact n)))
```

Эта функция просто проходит циклом через все комбинации, применяет интерпретатор с заданной сортирующей сетью, суммирует произведённые перемещения и возвращает результат деления этой суммы на число возможных комбинаций. Если мы сделаем предположение о том, что каждая возможная комбинация равнозначна, то данное вычисление покажет среднее число перемещений, для каждой комбинации. Таким образом мы можем увидеть, что результат **bad-3-sn** в среднем даёт 1.5 перемещения для одной комбинации:

```
* (average-swaps-calc 3 bad-3-sn)

3/2
```

Когда результат **good-3-sn**, в среднем, всего 1.166... перемещений:

```
* (average-swaps-calc 3 good-3-sn)

7/6
```

На текущий момент наши сортирующие сети производят сортировку наборов данных размерностью в три элемента. Существуют ли алгоритмы для генерации сортирующих сетей произвольного размера? Да, и эти алгоритмы уже известны. В 1968 году Кен Батчер (Ken Batcher) описал свой гениальный алгоритм [SN-APPLICATIONS], названный Дональдом Кнутом (Donald Knuth) как сортировка *перемещением и заменой (merge exchange sort)* или *алгоритм 5.2.2M* из [TAOCP-VOL3-P111]. Алгоритм Батчера — это разновидность комбинации *сортировки Шелла (Shell sort)* и *сортировки перемещением (merge sort)*, с тем исключением, что известен входной размер, операции сравнения-замены, создаваемые этим алгоритмом, абсолютно не зависят от входных данных — именно то, что нам нужно для сортирующих сетей. Таким образом, для создания сортирующей сети мы запускаем алгоритм Батчера и записываем полученные операции сравнения-замены. Позже мы сможем встроить эти операции в

Листинг 7.34: BUILD-BATCHER-SN

```
(defun build-batcher-sn (n)
  (let* (network
         (tee (ceiling (log n 2)))
         (p (ash 1 (- tee 1))))
    (loop while (> p 0) do
      (let ((q (ash 1 (- tee 1)))
            (r 0)
            (d p))
        (loop while (> d 0) do
          (loop for i from 0 to (- n d 1) do
            (if (= (logand i p) r)
              (push (list i (+ i d))
                    network)))
          (setf d (- q p)
                q (ash q -1)
                r p)))
      (setf p (ash p -1)))
    (nreverse network)))
```

функцию, предназначенную для работы с определённым входным размером. Этот процесс, в отличие от *развёртки цикла (loop unrolling)*, не выполняется полностью, и мы можем в дальнейшем воспользоваться этим.

Build-batcher-sn — это лисп реализация алгоритма Батчера, переведённая с описания Кнута. Благодаря лисповской точности побитовых целочисленных операций эта реализация не подвержена никаким искусственным ограничениям **n**, скажем, 32 или 64. Мы можем использовать **build-batcher-sn** для конструирования эффективных сортирующих сетей любого размера. Вот конструкция сети, предназначенной для сортировки массива из трёх элементов — то-же самое, что и вышеприведённый **good-3-sn**:

```
* (build-batcher-sn 3)

((0 2) (0 1) (1 2))
```

А вот конструкция сети, сортирующей массив из 7 элементов:

```
* (build-batcher-sn 7)
```

Листинг 7.35: BATCHER-7-SN-PIC

```
o--^--------^-----^-----------------o
   |        |     |
o--|--^-----|--^--v--------^--^-----o
   |  |     |  |           |  |
o--|--|--^--v--|--^-----^--|--v-----o
   |  |  |     |  |     |  |
o--|--|--|-----v--|--^--v--|--^--^--o
   |  |  |        |  |     |  |  |
o--v--|--|--^-----v--|--^--v--|--v--o
      |  |  |        |  |     |
o-----v--|--|--------v--v-----|--^--o
         |  |                 |  |
o--------v--v-----------------v--v--o
```

```
((0 4) (1 5) (2 6) (0 2) (1 3) (4 6) (2 4)
 (3 5) (0 1) (2 3) (4 5) (1 4) (3 6) (1 2)
 (3 4) (5 6))
```

Сети Батчера хороши, но известно, что они не совсем оптимальны для большинства размеров. Нахождение более лучших сетей для различных размеров, процесс их нахождения, а также выяснение их оптимальности — это важная нерешённая задача. В этой области исследований достигнуты значительные успехи с помощью *эволюционных алгоритмов (evolutionary algorithms)*, использующих новые техники искусственного интеллекта для эффективного поиска в супер-экспоненциальных областях проблем сортирующих сетей. Например, известная на данный момент наилучшая сеть размерностью тринадцать была найдена с использованием *Развивающегося Не-Детерминированного (Evolving Non-Determinism)* алгоритма [END].

Использованные здесь отображения сортирующих сетей в виде ASCII картинок созданы с помощью замечательной Perl программы Джона Гэмбла **Algorithm-Networksort**. Обратите внимание: в картинке есть несколько связей, которые можно выполнять одновременно, такие связи изображены в одной вертикали. Отсюда можно сделать вывод, что алгоритмы сортирующих сетей можно использовать по крайней мере в специализированных устройствах, с целью извлечения выгоды из параллелизма в операциях сравнения-замены. Изучение создания хороших параллельных сортирующих сетей наряду с распараллеливанием сетей — это также важная и не до конца изученная задача.

Листинг 7.36: PRUNE-SN-FOR-MEDIAN

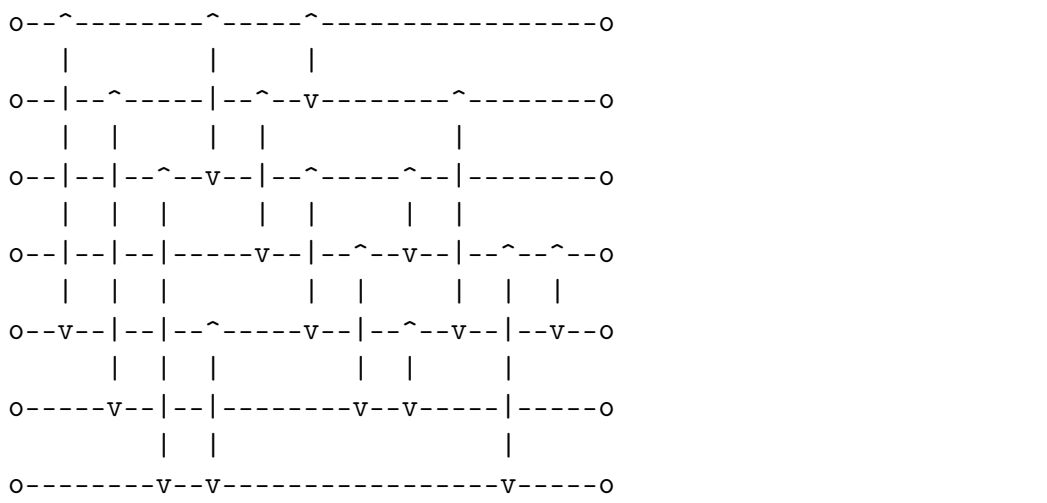
```
(defun prune-sn-for-median (elems network)
  (let ((mid (floor elems 2)))
    (nreverse
      (if (evenp elems)
        (prune-sn-for-median-aux
          (reverse network)
          (list (1- mid) mid))
        (prune-sn-for-median-aux
          (reverse network)
          (list mid))))))

(defun prune-sn-for-median-aux (network contam)
  (if network
    (if (intersection (car network) contam)
      (cons (car network)
            (prune-sn-for-median-aux
              (cdr network)
              (remove-duplicates
                 (append (car network) contam))))
      (prune-sn-for-median-aux
        (cdr network) contam))))
```

Выше мы уже упоминали что одним из недостатков сортирующих функций общего-назначения является то, что они жёстко завязаны на сортировку всего массива. Например, нам может понадобится узнать элемент, находящийся на определённом месте. Обычно нам нужно определить элемент, находящийся посередине, или *медианный (median)* элемент. Функции **prune-sn-for-median** и **prune-sn-for-median-aux** используют экономный, и, в основном, очевидный алгоритм, открытый мною, который может устранить множество ненужных операций сравнивания-замены и может послужить основой для конструирования сетей произвольного выбора.

Алгоритм начинается с сети Батчера, затем работает в обратном направлении, отслеживая *заражённые (contaminated)* элементы — элементы, существующие связи которых нельзя удалять, поскольку подобное удаление изменит исход сети для этого элемента. Любые связи, связывающие не заражённые элементы могут быть удалены, поскольку они не играют роли для заражённых элементов. Связь, соединяющая зара-

Листинг 7.37: HOYTE-7-MEDIAN-SN-PIC



Листинг 7.38: PRUNE-SN-FOR-MEDIAN-CALC

```
(defun prune-sn-for-median-calc (n)
  (loop for i from 2 to n collect
    (let* ((sn (build-batcher-sn i))
           (snp (prune-sn-for-median i sn)))
      (list i
            (length sn)
            (length snp)))))
```

жённый элемент с не заражённой связью, приводит к заражению не заражённого элемента. Если мы заражаем только средний элемент (или два средних элемента в случае с чётным входным размером), то в этом случае у нас получается сеть для выбора медианы.

Вывод алгоритма для седьмого размера (показано на рисунке) — это модифицированная сеть Батчера, в которой удалены две связи. После запуска этой сети, серединный элемент будет расположен на своём месте, но не гарантируется отсортированный порядок остальных элементов. Ниже, в качестве примера, мы отсортировали список ровно настолько, насколько это требовалось для выяснения серединного элемента (в данном случае 4):

```
* (interpret-sn
    '(4 2 3 7 6 1 5)
```

```
    (prune-sn-for-median
      7 (build-batcher-sn 7)))
6
(1 3 2 4 5 7 6)
```

Для сети с размером семь элементов наша модифицированная серединная сеть Батчера выполняет 12 операций сравнения-замены, в то время, когда обычная сеть Батчера выполнит 14 операций. **Prune-sn-for-median-calc** даст нам материал, позволяющий сравнить данные классы сетей для различных длин. Он вычисляет сеть Батчера для размера **n** и группирует его размер с размером связанной серединной сети, созданной нашим алгоритмом.

Вычислены сети с размерами до 49. Заметьте, что для малых размеров, экономится очень мало операций, если они вообще будут экономиться. Но для более значительных величин, мы видим, что экономим около 20% операций сравнения-замены. Если нам нужны только серединные значения — то эти сети будут хорошим выбором. Однако, конструирование оптимальных сетей, предназначенных для нахождения серединных элементов, по-прежнему, являются неисследованной областью. Модифицированные сети Батчера, разработанные в этой главе, показывают неплохие результаты, но, они всё ещё не оптимальны. Лучшие сети для нахождения серединного элемента размерностями 9 и 25 (размеры изображений 3x3 и 5x5) были открыты Паетом (Paeth) [GRAPHICS-GEMS-P171-175], они демонстрируются здесь и включены в код книги. Вот длины серединных сетей Паета:

```
* (length paeth-9-median-sn)

20
* (length paeth-25-median-sn)

99
```

Для сетей размером 9 полностью сортирующая сеть Батчера выполняет 26 операций. Лучшая, известная на данный момент, сортирующая сеть для девятого размера была открыта Флойдом (Floyd) и выполняет сортировку за 25 операций. Наша обрезанная серединная версия сети Батчера выполняет 22 операции, а серединная сеть Паета — 20 операций. Для сетей размером 25 показатели будут следующими: Батчер: 138, обрезанная версия: 113, Пает: 99. Таким образом наша серединная сеть

Листинг 7.39: PRUNED-MEDIAN-DATA

```
* (prune-sn-for-median-calc 49)

((2 1 1) (3 3 3) (4 5 5) (5 9 8) (6 12 12)
 (7 16 14) (8 19 17) (9 26 22) (10 31 29)
 (11 37 31) (12 41 35) (13 48 40) (14 53 47)
 (15 59 49) (16 63 53) (17 74 61) (18 82 72)
 (19 91 75) (20 97 81) (21 107 88) (22 114 98)
 (23 122 100) (24 127 105) (25 138 113) (26 146 98)
 (27 155 127) (28 161 133) (29 171 130) (30 178 150)
 (31 186 152) (32 191 157) (33 207 169) (38 265 223)
 (39 276 226) (40 283 233) (41 298 244) (42 309 259)
 (43 321 263) (44 329 271) (45 342 280) (46 351 293)
 (47 361 295) (48 367 301) (49 383 313))
```

Листинг 7.40: PAETH-9-MEDIAN-SN

```
(defvar paeth-9-median-sn
  '((0 3) (1 4) (2 5) (0 1) (0 2) (4 5) (3 5) (1 2)
    (3 4) (1 3) (1 6) (4 6) (2 6) (2 3) (4 7) (2 4)
    (3 7) (4 8) (3 8) (3 4)))
```

Листинг 7.41: PAETH-25-MEDIAN-SN

```
(defvar paeth-25-median-sn
  '((0 1) (3 4) (2 4) (2 3) (6 7) (5 7) (5 6) (9 10)
    (8 10) (8 9) (12 13) (11 13) (11 12) (15 16)
    (14 16) (14 15) (18 19) (17 19) (17 18) (21 22)
    (20 22) (20 21) (23 24) (2 5) (3 6) (0 6) (0 3)
    (4 7) (1 7) (1 4) (11 14) (8 14) (8 11) (12 15)
    (9 15) (9 12) (13 16) (10 16) (10 13) (20 23)
    (17 23) (17 20) (21 24) (18 24) (18 21) (19 22)
    (8 17) (9 18) (0 18) (0 9) (10 19) (1 19) (1 10)
    (11 20) (2 20) (2 11) (12 21) (3 21) (3 12)
    (13 22) (4 22) (4 13) (14 23) (5 23) (5 14)
    (15 24) (6 24) (6 15) (7 16) (7 19) (13 21)
    (15 23) (7 13) (7 15) (1 9) (3 11) (5 17) (11 17)
    (9 17) (4 10) (6 12) (7 14) (4 6) (4 7) (12 14)
    (10 14) (6 7) (10 12) (6 10) (6 17) (12 17)
    (7 17) (7 10) (12 18) (7 12) (10 18) (12 20)
    (10 20) (10 12)))
```

больше сети Паета (лучшей, на данный момент, сети для этих размеров) примерно на 10%. Как и следовало ожидать, мы не можем обрезать некоторые операции из сетей Паета:

```
* (length (prune-sn-for-median
            9 paeth-9-median-sn))

20
* (length (prune-sn-for-median
            25 paeth-25-median-sn))

99
```

В теории, всё это очень интересно. Но на практике теория достаточно скучна. Мы разработали все эти сортирующие сети для данных в виде списков, некоторые выполняют полную сортировку, используя алгоритм Батчера, некоторые применяют заражающую оптимизацию для алгоритма Батчера с целью нахождения медиан. Затем мы разработали игрушечный интерпретатор для этих сетей, без сомнений работа этого интерпретатора ужасает по сравнению с настоящими сортирующими

программами. Что общего между этими программами и эффективностью? Ведь в результате этих экспериментов мы получили теоретические результаты вместо полезного кода[22]? В большинстве языков результаты этих экспериментов — наши сортирующие сети — будут отображены в виде некоторых высокоуровневых структур данных и могут быть недостаточно хороши. Но в лиспе эти сети уже чрезвычайно эффективные сортирующие программы; мы не сделали только одно дело: не написали для них компилятор.

Упражнение: модифицируйте алгоритм обрезки (и его подход заражения) таким образом, чтобы он создавал выбирающие сети для квартилей сортируемого массива. Эти сети определяют не только серединное значение всего массива, но и находят серединные значения меньшей и большей половин упорядоченных элементов.

7.7 Написание и Замеры Производительности Компиляторов

Компилятор (compiler) — это страшная концепция для большинства программистов, поскольку большинство языков неудачны в плане написания для них компиляторов. Вот аналогия: разбор сложного log файла — это пугающая и создающая множество ошибок перспектива для программистов, знающих только C или Ассемблер, но благодаря Perl'у и регулярным выражениям такая задача не трудна для программистов, знающих несколько языков. Аналогично разработка мощного, выразительного языка программирования с последующим созданием компилятора для преобразования программ на этом языке в эффективный машинный код будет устрашающей задачей, если мы не знаем лисп. Преимущества лиспа, применённые для написания компиляторов, не только делают его лучше по сравнению с остальными языками, — но создают новый уровень выразительности. В целом, это преимущество заключается в разнице между наличием и отсутствием способности к выполнению чего-либо. Лисп программисты используют компиляторы повсюду, такими способами и для таких задач, в существование которых не-лисп программисты иногда даже не верят. Сколько C программистов рассматривали накладные расходы интерпретации функции **printf**, описанные (и решённые) в *разделе 7.2, Макросы Ускоряют Лисп*? Сколько C про-

[22]Нет ничего неправильного в каких-либо теоретических результатах. Некоторые наиболее важные открытия на свете были совершены из-за теории — возможно, что даже и лисп.

Листинг 7.42: SN-TO-LAMBDA-FORM-1

```
(defun sn-to-lambda-form% (sn)
  `(lambda (arr)
     #f
     (declare (type (simple-array fixnum) arr))
     ,@(mapcar
         #`(if (> #1=(aref arr ,(car a1))
                  #2=(aref arr ,(cadr a1)))
             (rotatef #1# #2#))
         sn)
     arr))
```

граммистов пытались написать компилятор для **printf**? В лиспе — это в порядке вещей. Всё должно компилироваться до лиспа.

Что такое компилятор? Если вы пришли из Блаба, то, возможно, ответ будет заключаться в большой стопке книг, посвящённых разбору, трансляции на основе синтаксиса, контекстно-свободных грамматик и т.д. Но не стоит переживать, это лисп и компиляторы просты. Настолько прост, что если вы имели дело с достаточно серьёзным лисп программированием, то вы уже писали его, возможно, даже не осознавая этого. У компилятора есть другое имя — "макрос". Макрос компилирует программы из одного языка в другой. На самом деле Лисп предназначен для написания таких компиляторов — всё остальное вторично. В лиспе единственным нетривиальным моментом в архитектуре программы является сохранение корректности целевой программы с одновременным обнаружением эффективного расширения этого кода. Другими словами — это сущность вашей проблемы компиляции. До сих пор мы рассматривали применение макросов в создании специализированных языков, предназначенных для решения насущных задач, кроме того, мы изучили вопросы касающиеся эффективности лисп кода и применяли определения для удаления двойственностей и проверок безопасности. Создание эффективного компилятора — всего лишь вопрос комбинации этих двух умений.

Во что расширялся компилятор **formatter** в *разделе 7.2, Макросы Ускоряют Лисп*, где мы создали компилирующий макрос для обработки **format**? Это была лямбда форма[23]. Иногда имеет смысл производить компилирование в лямбда формы, поскольку в дальнейшем мы можем

[23]Вернее шарп-закавыченная лямбда форма.

использовать функцию compile для непосредственной конвертации в машинный код. Возвращаясь к нашим сортирующим сетям из предыдущей главы, **sn-to-lambda-form%** — это функция, возвращающая лямбда форму. Эта лямбда форма будет содержать инструкции для каждой операции сравнения-замены в списковой сортирующей сети. Каждая инструкция будет (небезопасно) индексироваться в целочисленный массив для сравнивания и, возможно, применения **rotatef** для замены элементов. Целочисленный массив будет передан как аргумент (**arr**) функции, созданной этой лямбда формой. Это всё, что нужно для создания хорошего компилятора машинного кода. Как и со всеми лямбда формами, благодаря макросу **lambda**, мы в состоянии вычислить их для получения функций:

```
* (eval
    (sn-to-lambda-form%
      (build-batcher-sn 3)))

#<Interpreted Function>
```

Результат можно преобразовать в скомпилированные функции, просто применив к ним **compile**:

```
* (compile nil *)

#<Function>
```

Взглянем на вывод **disassemble** (скомпилированное расширение):

```
* (disassemble *)
...
;;; (> (AREF ARR 0) (AREF ARR 2))
      9E:       MOV     EAX, [EDX+1]
      A1:       MOV     ECX, [EDX+9]
      A4:       CMP     EAX, ECX
      A6:       JLE     L0
;;; (ROTATEF (AREF ARR 0) (AREF ARR 2))
      A8:       MOV     EAX, [EDX+9]
      AB:       MOV     ECX, [EDX+1]
      AE:       MOV     [EDX+1], EAX
      B1:       MOV     [EDX+9], ECX
;;; (> (AREF ARR 0) (AREF ARR 1))
      B4: L0:   MOV     EAX, [EDX+1]
```

```
      B7:       MOV     ECX, [EDX+5]
      BA:       CMP     BAX, ECX
      BC:       JLE     L1
;;; (ROTATEF (AREF ARR 0) (AREF ARR 1))
      BE:       MOV     EAX, [EDX+5]
      C1:       MOV     ECX, [EDX+1]
      C4:       MOV     [EDX+1], EAX
      C7:       MOV     [EDX+5], ECX
;;; (> (AREF ARR 1) (AREF ARR 2))
      CA: L1:   MOV     EAX, [EDX+5]
      CD:       MOV     ECX, [EDX+9]
      D0:       CMP     EAX, ECX
      D2:       JLE     L2
;;; (ROTATEF (AREF ARR 1) (AREF ARR 2))
      D4:       MOV     EAX, [EDX+9]
      D7:       MOV     ECX, [EDX+5]
      DA:       MOV     [EDX+5], EAX
      DD:       MOV     [EDX+9], ECX
      E0: L2:   ...
```

Этот машинный код быстр, но он может быть ещё быстрее. Лисп компиляторы умны — одни из самых умнейших — но они всегда могут быть ещё умнее. В тех редких случаях, когда нам нужно позаботиться о производительности, изучение скомпилированных расширений имеет важное значение, поскольку трудно узнать насколько умна ваша лисп реализация. Если мы внимательно рассмотрим последний дизассемблированный код, мы увидим что он выполняет ненужную операцию чтения перед тем, как выполнить замену. Проблема заключается в том, что **rotatef** расширяется в избыточный доступ. *Более умный компилятор* мог бы обнаружить, что нужное значение уже содержится в регистре и, тем самым, не допустил бы выполнения доступа к массиву. Но такого у нас нет, поэтому, я реструктуризовал код так, чтобы он выдавал более эффективное расширение.

Sn-to-lambda-form — это улучшенная версия **sn-to-lambda-form%**. Она создаёт временные привязки для считанных переменных, поэтому инструкции считывания массива не выполняются для операции замены. Вот улучшенное скомпилированное расширение:

```
* (disassemble
    (compile nil
      (sn-to-lambda-form%
```

```
        (build-batcher-sn 3))))
...
;;; (LET ((A (AREF ARR 0)) (B (AREF ARR 2))) ...)
      2E:       MOV     EAX, [EDX+1]
      31:       MOV     ECX, [EDX+9]
      34:       CMP     EAX, ECX
      36:       JLE     L0
;;; (SETF (AREF ARR 0) B (AREF ARR 2) A)
      38:       MOV     [EDX+1], ECX
      3B:       MOV     [EDX+9], EAX
;;; (LET ((A (AREF ARR 0)) (B (AREF ARR 1))) ...)
      3E: L0:   MOV     EAX, [EDX+1]
      41:       MOV     ECX, [EDX+5]
      44:       CMP     EAX, ECX
      46:       JLE     L1
;;; (SETF (AREF ARR 0) B (AREF ARR 1) A)
      48:       MOV     [EDX+1], ECX
      4B:       MOV     [EDX+5], EAX
;;; (LET ((A (AREF ARR 1)) (B (AREF ARR 2))) ...)
      4E: L1:   MOV     EAX, [EDX+5]
      51:       MOV     ECX, [EDX+9]
      54:       CMP     EAX, ECX
      56:       JLE     L2
;;; (SETF (AREF ARR 1) B (AREF ARR 2) A)
      58:       MOV     [EDX+5], ECX
      5B:       MOV     [EDX+9], EAX
      5E: L2:   ...
```

Изучая ваш лисп компилятор, вы узнаёте в насколько эффективное расширение будет раскрываться ваш макрос, а это очень важно для создания эффективных лисп программ. К сожалению, единственный способ по настоящему развить навыки создания быстрого лисп кода — это **disassemble**, исходные коды вашей лисп системы и такие инструменты замера производительности, как макрос **time**.

Расширение в лямбда форму, как это делалось в нашем макросе **sn-to-lambda-form**, — это наиболее очевидный способ реализовать компилятор в том случае, если вы пришли в лисп из Блаб языков. Цикл "исходный код → лямбда форма → **compile** → **disassemble**" очень похож на Блаб цикл "редактирование → компиляция → дизассемблирование". На вход вы подаёте исходный код, а на выходе получаете машинный код. Однако, этот подход может быть более лисповым. В лиспе мы чаще

Листинг 7.43: SN-TO-LAMBDA-FORM

```
(defun sn-to-lambda-form (sn)
  `(lambda (arr)
     #f
     (declare (type (simple-array fixnum) arr))
     ,@(mapcar
         #`(let ((a #1=(aref arr ,(car a1)))
                 (b #2=(aref arr ,(cadr a1))))
             (if (> a b)
               (setf #1# b
                     #2# a)))
         sn)
     arr))
```

всего делаем наши компиляторы невидимыми — встроенными непосредственно в другие лисп программы. В идеале функция **compile** должна запускаться только тогда, когда мы хотим осуществить быстрое исполнение кода. Макрос не должен постоянно компилировать всё в машинный код, вместо этого достаточно создать хорошее расширение, такое, чтобы компилятор, вне зависимости от момента запуска, обладал достаточной информацией для создания эффективной программы.

Отдельно отмечу, что нам нет нужды вызывать **compile** во время-выполнения. **Compile** — это дорогостоящая операция, поскольку не исключено, что для компиляции одного кода придётся расширить множество уровней макросов. Вместо вызова compile во время-выполнения, вспомним что лисп уже обладает всеми скомпилированными лямбда формами внутри скомпилированной функции. Учитывая это свойство конструирования замыканий в уже скомпилированном во время-выполнения коде, легко убедиться в том, что большинство вычислений во время-компиляции уже выполнено при создании произвольных функций (замыканий) во время-выполнения.

В этой книге мне больше всего нравится макрос **sortf**. И не только потому, что он краткий, элегантный и замечательно демонстрирует большинство приёмов использования макросов, но и потому, что этот макрос — полезный код осуществляющий чрезвычайно быстрые операции сортировки, этот код можно использовать на промышленном уровне. Но что самое лучшее — этот макрос очень просто использовать в других лисп программах, по причине отличной сочетаемости. Поэтому мы не можем пройти мимо такой продвинутой лисп оптимизации. **Sortf** пред-

Листинг 7.44: SORTF

```
(defmacro! sortf (comparator &rest places)
  (if places
    `(tagbody
       ,@(mapcar
       #`(let ((,g!a #1=,(nth (car a1) places))
               (,g!b #2=,(nth (cadr a1) places)))
           (if (,comparator ,g!b ,g!a)
             (setf #1# ,g!b
                   #2# ,g!a)))
       (build-batcher-sn (length places))))))
```

назначен для сортирования маленьких наборов данных фиксированного размера. Этот макрос прост для встраивания, иногда даже проще чем функция **sort**. Вместо расширения в лямбда форму, **sortf** расширяется в tagbody форму, поскольку **tagbody** — это канонический **progn**, возвращающий nil. Вот расширение **sortf**:

```
* (macroexpand
    '(sortf < a b c))

(LET ()
  (TAGBODY
    (LET ((#:A1824 A) (#:B1823 C))
      (IF (< #:B1823 #:A1824)
        (SETF A #:B1823 C #:A1824)))
     (LET ((#:A1824 A) (#:B1823 3))
       (IF (< #:B1823 #:A1824)
         (SETF A #:B1823 B #:A1824)))
     (LET ((#:A1824 B) (#:B1823 C))
       (IF (< #:B1823 #:A1824)
         (SETF B #:B1823 C #:A1824)))))
T
```

Архитектура интерфейса **sortf** пришла из *On Lisp*, эта архитектура настолько естественна, что почти любой лисп программист реализует её таким способом. Первый аргумент обычно представляет из себя символ, обозначающий оператор сравнения — чаще всего что-то вроде $<$. Этот аргумент чаще всего функция, но, как отмечает Грэм, первый аргумент

может быть и макросом, и специальной формой, поскольку аргумент помещается непосредственно на позицию функции в списке. А поскольку первый аргумент будет на месте функции, то мы можем передавать в качестве первого оператора лямбда форму[24]:

```
* (let ((a -3) (b 2))
    (sortf (lambda (a b) (< (abs a) (abs b)))
      a b)
    (list a b))

(2 -3)
```

Также, как и макрос Грэма, сортируемые аргументы являются обобщёнными переменными. Это означает, что мы можем использовать **sortf** для сортировки любых типов переменных, не только представляемых символами, но и вообще всё, к чему можно применить **setf**. Вот пример:

```
* (let ((a 2) (b '(4)) (c #(3 1)))
    (sortf < a (car b) (aref c 0) (aref c 1))
    (format t "a=~a b=~a c=~a~%" a b c))
a=1 b=(2) c=#(3 4)
NIL
```

Поскольку **sortf** Грэма и наш **sortf** компилирует один исходный язык, то их расширения не будут сильно отличаться. Макрос Грэма, пожалуй, более корректен чем наш макрос, по той причине, что в его макросе код доступа к данным исполняется только один раз. Мы можем передавать макросу Грэма значения с побочными эффектами и, как ожидалось, получим их вычисленными только один раз. Например, **sortf** Грэма только один раз инкрементирует **i** при передаче **(aref arr (incf i))**. **Sortf** Грэма работает так: копирует каждое сортируемое значение во временную привязку, применяет пузырьковую сортировку для сортирования этих временных привязок, затем использует **setf** расширения[25] для записи временных переменных в изначальное место, но уже в отсортированном порядке. Вместо этого наш **sortf** будет вычислять каждое место формы по нескольку раз на протяжении всей сортировки, поэтому вам совет: не используйте места с побочными эффектами. Если вы заботитесь об эффективности, то в следствии этой архитектуры, вам следует убедиться в эффективности доступа к переменным. В частности не используйте

[24]Заметьте, что мы не можем передавать шарп-закавыченные лямбда формы. Не используйте шарп-закавычивание для ваших лямбда форм.

[25]Подробности в мельчайших деталях описаны в *On Lisp*.

методы получения доступа перебирающие весь список, например **caddr**, поскольку они будут проходить по списку множество раз. В нашей реализации мы сортируем аргументы *на месте*, без использования временных привязок. Вместо пузырьковой сортировки, с *большой "О" (Big-O)* сложностью **(O (expt N 2))**[26], мы используем гораздо более быструю сортировку Батчера со сложностью **(O (* N (expt (log N) 2)))**. Есть методы для конструирования сортирующих сетей со сложностью **(O (* N (log N)))** — то же, что и быстрая сортировка (quick sort) — но большинство из них выполняют больше операций для сортировки малых размеров, чем сети Батчера.

Поскольку **sortf** не содержит деклараций типов, то, возможно, вам захочется использовать sortf совместно с ними. Если вам нужна действительно быстрая сортировка, то убедитесь, что компилятор знает типы всех сортируемых обобщённых переменных. Если вы определите типы, всегда убедитесь в том, что все обобщённые переменные определены одним типом. Это обязательное условие, поскольку любой элемент может оказаться в любом месте.

Но как нам узнать, выигрывает или проигрывает в производительности наш макрос функции **sort**? Нам нужно *замерить производительность (benchmark)*. Замеры производительности обсуждались множество раз, особенно среди программистов, поскольку бесконечно *страдать ерундой* — это очень приятно. К несчастью, результаты почти всех замеров производительности — бесполезны. Более того, я советую вам относиться с недоверием к результатам замеров производительности опубликованных в этой книге. Но, тщательно спланированный, аккуратно проведённый, с точными замерами эксперимент по сравнению быстродействия двух версий кода, запущенных на одном лисп образе, на одной машине, может оказаться очень полезным для понимания и устранения узких мест производительности. Эти замеры полезны не только потому, что могут показать какая техника более эффективна, но и на сколько эффективна. Поскольку макросы пишут код для нас, то макросы — это лучший инструмент для проведения таких экспериментов.

Макрос **sort-benchmark-time** — это компонент нашего эксперимента. Он расширяется в код, который предполагает, что любая лямбда форма или функция является сортировщиком **sorter**, и эта функция будет использоваться для сортировки целочисленного массива размерности **n**. После чего компилирует эту функцию и с помощью неё сортирует случайным образом сгенерированный массив в нескольких итерациях **iters**.

[26]Мы не любим инфиксную запись (*примечание переводчика:* по-моему, Даг перебарщивает).

Листинг 7.45: SORT-BENCHMARK-TIME

```
(defmacro sort-benchmark-time ()
  `(progn
     (setq sorter (compile nil sorter))
     (let ((arr (make-array
                 n :element-type 'fixnum)))
       (time
         (loop for i from 1 to iters do
           (loop for j from 0 to (1- n) do
             (setf (aref arr j) (random n)))
           (funcall sorter arr))))))
```

Макрос **time** используется для сбора статистики времени, потраченного на процедуру сортировки.

Do-sort-benchmark — это, непосредственно, интерфейс для осуществления замеров производительности. Заданный набор данных размерности **n** и число итераций **iters** будут использоваться как для замера функции COMMON LISP **sort**, так и для нашего макроса **sortf**. Он сохраняет состояние генератора случайных чисел и сбрасывает его после выполнения замера **sort**, но до запуска **sortf**, таким образом сортируемые массивы будут идентичными. Важно, чтобы **do-sort-benchmark** был скомпилирован при запуске, таким образом мы как можно больше сокращаем возможные помехи замеров.

После запуска **do-sort-benchmark** покажет нам не только то, что **sortf**, мало того, эффективнее, но и что сортирующие алгоритмы общего назначения не то-же самое, что сортирующие сети по отношению к производительности при работе с малыми, фиксированными наборами данных. Также, можно заметить, что **sortf** не потребляет память, что в свою очередь выливается в меньшее время сборки мусора и в повышение производительности. Вот результаты для наборов данных размерностями 2, 3, 6, 9, 25, 49:

```
* (do-sort-benchmark 2 1000000)

CL sort:
; Evaluation took:
;   1.65 seconds of real time
;   8,000,064 bytes consed.
```

Листинг 7.46: DO-SORT-BENCHMARK

```
(defun do-sort-benchmark (n iters)
  (let ((rs (make-random-state *random-state*)))
    (format t "CL sort:~%")
    (let ((sorter
            '(lambda (arr)
               #f
               (declare (type (simple-array fixnum)
                              arr))
               (sort arr #'<))))
       (sort-benchmark-time))

    (setf *random-state* rs)
    (format t "sortf:~%")
    (let ((sorter
            `(lambda (arr)
               #f
               (declare (type (simple-array fixnum)
                              arr))
               (sortf <
                 ,@(loop for i from 0 to (1- n)
                         collect `(aref arr ,i)))
               arr)))
     (sort-benchmark-time))))

(compile 'do-sort-benchmark)
```

```
sortf:
; Evaluation took:
;   0.36 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 3 1000000)

CL sort:
; Evaluation took:
;   3.65 seconds of real time
;   24,000,128 bytes consed.

sortf:
; Evaluation took:
;   0.46 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 6 1000000)

CL sort:
; Evaluation took:
;   10.37 seconds of real time
;   124,186,832 bytes consed.

sortf:
; Evaluation took:
;   0.8 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 9 1000000)

CL sort:
; Evaluation took:
;   19.45 seconds of real time
;   265,748,544 bytes consed.

sortf:
; Evaluation took:
;   1.17 seconds of real time
;   0 bytes consed.
```

```
* (do-sort-benchmark 25 1000000)

CL sort:
; Evaluation took:
;   79.53 seconds of real time
;   1,279,755,832 bytes consed.

sortf:
; Evaluation took:
;   3.41 seconds of real time
;   0 bytes consed.

* (do-sort-benchmark 49 1000000)
CL sort:
; Evaluation took:
;   183.16 seconds of real time
;   3,245,024,984 bytes consed.

sortf:
; Evaluation took:
;   8.11 seconds of real time
;   0 bytes consed.
```

Таким образом, для общих задач, где важно упорядочивание и производительность мы можем использовать сортирующие сети. Эти измерения делались не для дискредитации реализации **sort** (на самом деле это замечательная сортирующая процедура), но, для того, чтобы показать реалистичный пример того, как умное программирование с макросами может дать значительное повышение производительности. Лисп макросы дали нам возможность писать умные программы простым и переносимым способом. Но, для того чтобы писать умные программы на Блаб языках, приходится тратить столько усилий, что в результате Блаб программисты почти всегда довольствуются примитивным программированием. В лиспе всё компилируется в лисп, и поэтому нет никаких барьеров для оптимизаций. Если что-то оказывается неприемлемо медленным, то измените эту вещь и сделайте эту вещь быстрее. Нам почти никогда не требуется быстрое исполнение чего-либо, но когда нам требуется скорость, то решением будет лисп.

Другой макрос похожий на **sortf** — это **medianf**, использующий нашу урезанную сеть нахождения медианы или вручную написанные Паетовские сети нахождения медианы для неполной сортировки. Предназначе-

Листинг 7.47: MEDIANF

```
(defun medianf-get-best-sn (n)
  (case n
    ((0)  (error "Need more places for medianf"))
    ((9)  paeth-9-median-sn)
    ((25) paeth-25-median-sn)
    (t    (prune-sn-for-median n
            (build-batcher-sn n)))))

(defmacro! medianf (&rest places)
  `(progn
     ,@(mapcar
         #`(let ((,g!a #1=,(nth (car a1) places))
                 (,g!b #2=,(nth (cadr a1) places)))
             (if (< ,g!b ,g!a)
               (setf #1# ,g!b
                     #2# ,g!a)))
         (medianf-get-best-sn (length places)))
     ,(nth (floor (1- (length places)) 2) ; lower
           places)))
```

ние **medianf** — убедиться в том, что серединный элемент расположен на правильной позиции. В случае чётных размеров сети и нижняя, и верхняя медиана будут находиться на правильном месте. В отличие от **sortf**, который всегда возвращает **nil**, **medianf** вернёт значение нижней медианы (то же самое, что верхняя медиана для сетей нечётных размеров).

Как мы сказали ранее, **sortf** и **medianf** сортируют любые разновидности объектов, к которым можно применить **setf**. В случае с переменными, сохраняемыми в регистрах, такой подход даёт возможность создавать сортирующий код, которому даже не нужен доступ к памяти. Например, вот скомпилированное расширение для **medianf** на трёх fixnum переменных:

```
* (dis ((fixnum a) (fixnum b) (fixnum c))
    #f
    (medianf a b c))
...
;;; (MEDIANF A B C)
     34:         MOV     EBX, EAX
     36:         CMP     EDX, EAX
     38:         JL      L4
     3A: L0:     MOV     EBX, EAX
     3C:         CMP     ECX, EAX
     3E:         JL      L3
     40: L1:     MOV     EAX, ECX
     42:         CMP     EDX, ECX
     44:         JNL     L2
     46:         MOV     ECX, EDX
     48:         MOV     EDX, EAX
     4A: L2:
...
     5B: L3:     MOV     EAX, ECX
     5D:         MOV     ECX, EBX
     5F:         JMP     L1
     61: L4:     MOV     EAX, EDX
     63:         MOV     EDX, EBX
     65:         JMP     L0
```

Лисп обладает гораздо большим потенциалом для создания эффективного кода, чем другие языки и всё это — благодаря макросам. Макросы очень хороши в плане создания контролируемых экспериментов по некоторым замерам, но, кроме того, макросы — это решение позволяю-

щее определить и сравнить эффективность различных техник. Компиляторы — это программы, которые пишут другие программы, и ничто так хорошо не подходит под эту задачу, как макросы.

Глава 8

Лисп, переходящий в Форт, переходящий в Лисп

8.1 Причудливая Архитектура

Эта глава – кульминация многих, увиденных ранее в этой книге макро техник. Используя абстракции из написанных нами макросов мы создадим реализацию одного из моих самых любимых языков программирования: *forth*. И хотя эта реализация воплощает многие идеи форта, она очень отличается от форта и является лисповой. Несмотря на то, что код приведённый в этой главе можно использовать в других интересных целях, его главным назначением является обучение основам и концепциям метапрограммирования форта для лисп аудитории и использование в роли платформы для дискуссий по центральной теме этой книги – создание и использование дуализма синтаксиса в макросах.

Форт обладает богатой, захватывающей историей развития, в отличие от многих других языков, за исключением лиспа и я рад, что открыл форт для себя. По этой и другим причинам эта глава посвящена с любовью моему отцу Брайану Хойту, познакомившему меня с фортом и с программированием на компьютере. Эта глава частично вдохновлена [THREADING-LISP] и исследованием Генри Бейкера [LINEAR-LISP] [LINEAR-LISP-AND-FORTH].

Форт был первым языком программирования, который был создан и развивался без сильной правительственной, академической или корпоративной помощи – или по-крайней мере первый развившийся подобным образом язык. Вместо того, чтобы появиться в результате нужд большой организации, форт был создан независимо от всего Чаком Муром около 1968 года для решения его собственных вычислительных задач в

астрономии, проектировании устройств и многого другого. После чего форт был распространён, реализован и улучшен силами многочисленного сообщества энтузиастов [EVOLUTION-FORTH-HOPL2]. В отличие от патронажа MIT (и позже DARPA) над ранними лиспами и COMMON LISP, IBM-овского FORTRAN-а и AT&T-шного юникс языка C.

В следствии происхождения и, в целом, другой философии роли компьютерного программного и аппаратного обеспечения, форт стал иным. Даже больше чем лисп форт выглядит странным. Но как и в лиспе, странность форта обладает своей причиной: он был спроектирован больше для ума, чем для стиля. Форт странен своей архитектурой, и эта архитектура имеет отношение к макросам.

Сегодня форт чаще всего встречается в так называемых встраиваемых платформах – компьютерах с сильно ограниченными ресурсами. Это свидетельствует об архитектуре форта, как языке, который можно полностью реализовать на почти всех, когда либо созданных программируемых компьютерных системах. Форт спроектирован так, чтобы быть как можно более простым для реализации и экспериментов с ним. На самом деле создание клона форта настолько тривиально, что реализация форто-подобного стекового языка или нескольких языков является обязательным обрядом посвящения для программистов интересующихся архитектурой языков программирования. Некоторые стековые языки программирования восходят к форту, одними из интересных языков являются PostScript и Joy.

Часто важные решения при реализации форта основываются на непосредственных ресурсах компьютера на котором будет реализовываться форт. Форт программисты разработали набор *абстрактных регистров*, которые должны отображаться либо в настоящие регистры, либо располагаться в памяти или, возможно, реализуются какими-то другими способами. Но что если мы реализуем форт на лиспе, среде с неограниченным потенциалом и малыми ограничениями? Прежде чем начать отображать абстрактные регистры форта в лисп код, мы сделаем отступление. Как бы выглядел форт если бы у Чака была лисп машина? Чем подгонять форт к особенностям отдельной машины, настоящей или виртуальной, мы рассмотрим минимальный набор абстрактных регистров форта, оптимизированных для упрощения реализации при реализации на лиспе.

Вместо того, чтобы искать оптимальный набор абстрактных концепций, воспользуемся опытом Чака, создавшего десятки различных реализаций форта на многих архитектурах. Это то, что делает форт настолько замечательным. Как и лисп, форт предоставляет огромный простор в архитектуре языка и также, как и лисп, форт не столько язык про-

Листинг 8.1: FORTH-REGISTERS

```
(defvar forth-registers
  '(pstack rstack pc
    dict compiling dtable))
```

Листинг 8.2: FORTH-WORD

```
(defstruct forth-word
  name prev immediate thread)
```

граммирования или набор стандартов, сколько строительный материал и набор знаний о том, что работает или не работает.

Переменная **forth-registers** – это список символов, представляющих абстрактные регистры нашей форт машины. Конечно, лисп не думает терминами регистров и целыми числами, а думает переменными и символами. Может показаться странным начинать нашу разработку форт среды всего лишь списком имён переменных, но этот факт всегда является первым шагом в реализации форт системы. Создание форта – это гениальный процесс самопостроения красивей и продуманней которого может быть только лисп. Небольшое описание этого процесса последует позже.

Одной из характерных особенностей форта является его прямой доступ к структуре стековых данных, используемый вашей программой и для передачи параметров подпрограммам и для отслеживания процесса выполнения программы через эти подпрограммы. Особенно интересен форт ещё и тем, что, в отличие от большинства языков программирования, разделяет эти два применения структуры стековых данных на два стека, которыми вы можете управлять непосредственно сами[1]. В типичной C реализации параметры вызова функции и, так называемого *адреса возврата*, сохраняются в единственном *кадре стека* размером в переменную для каждого вызова функции. В форте они представляют из себя два различных стека под названием стек параметра и стек возврата, которые представлены нами как наши абстрактные регистры **pstack** и **rstack**. Мы используем макросы COMMON LISP **push** и **pop** для обозначения этих стеков, реализованных с помощью связанного списка cons ячеек взамен структур из массива данных, используемых в большинстве фортов.

[1]Большинство языков вообще не позволяют управлять стеком.

Листинг 8.3:FORTH-LOOKUP

```
(defun forth-lookup (w last)
  (if last
      (if (eql (forth-word-name last) w)
          last
          (forth-lookup
           w (forth-word-prev last)))))
```

Абстрактный регистр **pc** – это сокращение от *program counter (счётчик программы)*, указатель на исполняемый в данный момент код. Что такое форт код, как мы можем ссылаться на него и для чего служат абстрактные регистры **compiling** и **dtable** будет описано позже.

Другим строительным блоком форта является концепция *словаря*. Словарь форта – это единый связанный список форт *слов*, аналогичных лисп функциям[2]. Слова представлены лисп *структурой*. Структуры – это эффективные слот-ориентированные структуры данных, обычно реализуемые как векторы. Слот **name** предназначен для символа, используемого для поиска слова в словаре. Имейте в виду, что словарь форта составлен не в алфавитном, а в хронологическом порядке. Когда мы добавляем новые слова, то мы добавляем их в конец словаря, таким образом при проходе по словарю последние определённые слова будут рассматриваться первыми. Последний элемент нашего словаря всегда хранится в абстрактном регистре **dict**. Для прохода по словарю мы начинаем с **dict** и следуем по указателю **prev** в структуре слова, которое указывает либо на предыдущее определённое слово, либо на **nil** если текущее слово было последним словом[3].

Дано **w**, искомое слово, и **last**, словарь, по-которому выполняется поиск, **forth-lookup** будет возвращать структуру форт слова или **nil** в зависимости от того, было ли найдено слово **w** в словаре или нет. Вместо **eq** было использована функция поиска **eql**, в отличие от лиспа, форт позволяет в качестве названий слова использовать цифры и другие несимвольные знаки.

Слот **immediate** нашего форт слова – это флаг, обозначающий признак *незамедлительности* слова. Незамедлительность – это концепция метапрограммирования форта, которую мы скоро будем серьёзно изучать. Ну а пока мы сделаем грубую аналогию с лиспом: незамедлитель-

[2]Иначе говоря, они вовсе не функции, но скорее процедуры.

[3]Да, иногда лисп программисты используют альтернативы **cons** ячейкам для связанных списков.

ные слова похожи на лисп макросы тем, что это форт функции, выполняемые во время компиляции, а не во время выполнения. Что? Предполагалось что только в лиспе есть макросы. Правда в том, что макросистема COMMON LISP мощнее всех других макросистем – включая лучшие форт реализации – но форт обладает способностью расширения, превосходящей почти все остальные языки. Как и в лиспе, эта способность является результатом философии архитектуры: если это хорошо для автора реализации языка, то это достаточно хорошо и для разработчика приложений на этом языке. Как и лисп, форт не признаёт понятия примитива. Вместо этого предоставляется набор *мета-примитивов*, которые можно комбинировать для построения того языка, который хотите видеть вы, программист. Как и в лиспе, но не в Блаб языках, расширение языка новыми способами через применение макросов не только возможно, но и поощряется. Как и лисп, форт создан не из соображений стиля, но из соображений мощи.

8.2 Cons Шитый Код

В предыдущем разделе мы сфокусировались на абстрактных регистрах. Эти регистры находятся в центре фокуса, благодаря им философия форта настолько фундаментальна, но на самом деле эти регистры всего лишь компоненты более важной концепции: *абстрактная машина*. Возможно наиболее отличительная черта различных форт систем – это их реализации *шитого кода*. То, что в форте обозначается под шитым кодом – очень отличается от традиционного понимания нитей процессов в разделённой памяти с вытесняющей многозадачностью[4]. Форт нити не имеют ничего общего с параллелизмом. Форт нити – это основа для разговора о компиляции кода и метапрограммировании.

И если лисп даёт доступ к дереву структуры данных[5] символов из которых собрана в памяти и скомпилирована ваша программа, то в форте нет символьных манипуляций. Вместо этого форт даёт доступ к процессу сборки – сшивания – кода в памяти. Для посторонних наиболее примечательными особенностями форта являются стеки и постфиксная нотация, но, на самом деле изюминкой форта являются нити. Форт также завязан на стеки, как и лисп на списки. Они просто созданы для того, чтобы быть наиболее удобными структурами данных для решения задач мета-программирования – то, для чего предназначены форт и лисп.

[4]Эта книга не рекомендует использовать эти типы нитей из-за соображений безопасности и надёжности.

[5]На самом деле – ориентированный ацикличный граф.

Классический стиль сшивания известен как *косвенное сшивание*, но более современные разновидности форта реализованы с *прямым сшиванием*. Разница заключается в уровне косвенности. Низкоуровневая эффективность реализации косвенности зависит от процессора, и мы не будем углубляться в детали. По форт сшиваниям есть немало хороших руководств [STARTING-FORTH] [MOVING-FORTH]. В памяти компьютера эти оба стиля сшивания реализованы как смежные *ячейки*, в которых целочисленные машинные слова представляют указатели. *Внутренний интерпретатор* – маленький, компактный машинный код – обычно предназначается для процессора, выполняющего свою важную работу: следовать по указателям этих форт нитей, с последующей интерпретацией их значений. По-умолчанию, при встрече ячейки используется такое поведение: поместить текущий указатель расположения программы в стек возврата, после чего изменить указатель расположения программы на то, что указывает содержимое ячейки. Когда внутренний интерпретатор достигнет конца нити, он извлечёт значение из стека возврата и продолжит исполнение с прежнего места – там где оно остановилось[6].

Как вы можете себе представить, такой тип программного хранилища предназначен для чрезвычайно маленьких программ. Скомпилированное форт слово – это всего лишь последовательный массив целых чисел, большинство из которых представляют указатели на другие слова. Это всегда было одним из преимуществ форта. Из-за прозрачности сшивания программы в памяти, форт позволяет хорошо контролировать многие компромиссы, на которые приходится идти при программировании, включая один из наиболее важных: Скорость выполнения против размера программы. Шитый код позволяет нам оптимизировать наши абстракции настолько близко к проблеме, насколько это возможно, в результате получая экстремально быстрые, маленькие программы. Но, лисп макросы предназначены для гораздо большего, чем просто эффективность, то же самое можно сказать и по отношению к форт нитям. Как и большинство лисп программистов, форт программисты склонны считать себя реализаторами нового, а не просто пользователями. Форт и лисп – оба предназначены для контроля – пишите свои собственные правила.

Существует, по-крайней мере, два общих типа техник форт сшивания: *шитый код на лексемах* и *шитый код на подпрограммах*. Они представляют противоположные подходы при решении компромисса между скоростью и размером. Иногда эти техники сшивания сосуществуют с косвенным и прямым шитым кодом в одном форте. Сшивание на лексемах

[6]В большинстве фортов, конец нити обозначается форт словом **exit**.

включает в себя добавление другого уровня косвенности через использование целых чисел, даже меньших чем указатели на существующие слова в нити. На другом конце спектра располагается сшивание на подпрограммах. Этот тип шитого кода становится популярным и лучшие, современные форт компиляторы частично используют сшивание на подпрограммах. Вместо следующих подряд указателей на слова, по которым будет следовать внутренний интерпретатор, шитый код на подпрограммах содержит встроенные машинные инструкции, вызывающие эти указатели. В шитом коде на подпрограммах исчезает внутренний интерпретатор – он уже реализован оборудованием (или виртуальной машиной). Шитый код на подпрограммах обычно считается непрозрачным блоком, которым может управлять только специальный, не программируемый компилятор. Если же в этот код внесены различные оптимизации, то эти непрозрачные блоки становятся совсем непохожими на единообразные нити, основанные на ячейках. Почти все не-форт компиляторы в результате работы выдают шитый код на подпрограммах и не считаются с тем, что вы можете захотеть сделать что-то другое, что подводит нас к такому своеобразному определению:

> *Flub (Флаб)* – это язык, работающий только с шитым кодом на подпрограммах или язык, реализация которого предоставляет только шитый код на подпрограммах.

Например, C – это Флаб потому, что единственное что он позволяет программистам – это создавать функции – непрозрачные блоки шитого кода на подпрограммах. Конечно, мы можем реализовать на C внутренний интерпретатор для обработки косвенного шитого кода[7] и, с помощью этой программы, самостоятельно выстроить стековый язык, но, тогда получается, что мы уже не программируем на C. Почти все Блаб языки – Флабы. Форт – как мы это только что выяснили – не Флаб. Позже мы увидим что форт даёт программистам/создателям языков много возможности для контроля компиляции их программ.

Является ли лисп Флабом? Интересно, но лисп, возможно, был первым не Флаб языком программирования, но потом постепенно стал в основном Флабом. Хотя это не строго требуется стандартом, основные COMMON LISP компиляторы только компилируют функции в блоки непрозрачного машинного кода и, таким образом, они являются Флабами. Но в самых первых версиях лиспа функции сохранялись как списки – странная разновидность сшивания кода, не сильно отличающаяся

[7]Впрочем, можно реализовать работу с прямым шитым кодом, но это значительно сложнее.

Листинг 8.4: FORTH-INNER-INTERPRETER

```
(defmacro forth-inner-interpreter ()
  `(loop
      do (cond
           ((functionp (car pc))
            (funcall (car pc)))
           ((consp (car pc))
            (push (cdr pc) rstack)
            (setf pc (car pc)))
           ((null pc)
            (setf pc (pop rstack)))
           (t
            (push (car pc) pstack)
            (setf pc (cdr pc))))
      until (and (null pc) (null rstack))))
```

от форт нити. Такой подход позволял выполнять некоторые очень умные трюки во время выполнения программы, включая создание циклического кода, но, был безнадёжно неэффективным. В отличие от фортовских разновидностей сшивания – эффективно реализованных на почти всех архитектурах – такое внутреннее представление лисп функций было недопустимым и лисп был модифицирован для того, чтобы генерировать (чрезвычайно) эффективный код. Соответственно, для метапрограммистов, большинство реализаций COMMON LISP-а являются Флабами.

Но есть разница между свойствами, которые невозможно добавить в язык и свойствами, которые мы можем добавить с помощью макросов. С помощью макросов мы можем расширять язык любыми способами и он по-прежнему будет оставаться лиспом. В COMMON LISP-е нет форт нитей из-за тех же причин, по которым нет продолжений и первоклассных макросов: они сознательно не реализованы в языке для того, чтобы писатели макросов реализовали их так, как им нужно. Одна из наиболее важных целей самой главы и её кода в том, чтобы показать, что лисп хоть и является Флабом, но может трансформироваться в не-Флаб языки с помощью макросов. Не Блаб подразумевает, не Флаб, или другими словами, если вы не можете преобразовать язык в не Флаб, то он должен быть Блабом. Однако, обратное неверно. Не Флаб языки, такие как форт продолжают оставаться Блабом и самый короткий путь сделать их не Блабом – это реализовать с их помощью лисп среду – после чего вам остаётся программировать на лиспе.

Листинг 8.5: PRIM-FORMS

```
;; Prim-form: (name immediate . forms)
(defvar forth-prim-forms nil)

(defmacro def-forth-naked-prim (&rest code)
  `(push ',code forth-prim-forms))

(defmacro def-forth-prim (&rest code)
  `(def-forth-naked-prim
       ,@code
       (setf pc (cdr pc))))
```

Вместо того, чтобы использовать последовательные ячейки памяти для представления косвенных/непосредственных нитей, наш форт воспользуется преимуществом лисповской динамической типизации и структурой списка из cons ячеек. Мы называем такой код *cons шитым кодом*. Макрос **forth-inner-interpreter** расширяется в код, способный следовать по нитям, состоящим из cons ячеек связанных списков. Может показаться странным начинать программировать логику нашей форт среды именно так – с помощью макросов, которые должны раскрываться в некоторое, ещё недостаточно известное выражение – но, на самом деле – это предпочтительный шаблон лисп программирования. Поскольку макросы позволяют нам начинать программировать везде где мы хотим, почему бы не начать с действительно интересного, основного механизма программы? Этот механизм будет оказывать наибольшее влияние на окончательную архитектуру программы.

Определение **forth-inner-interpreter** представляет собой четкое описание того, что мы подразумеваем под cons шитым кодом. **Car** каждой **cons** ячейки указывает либо на функцию, другую **cons** ячейку, либо на другой лисп атом. Функции исполняются в таком порядке, в каком они встречаются. Заметьте, что изменение **pc** регистра остается на усмотрение самой функции. Если в нити найдена **cons** ячейка, то предполагается указание на вызов подпрограммы – вызов слова. Наш внутренний интерпретатор поместит точку возобновления из **pc** в стек возврата и перейдет к этой новой нити. Если встретится какой-то другой атом, то он будет просто помещен в стек параметров и выполнение продолжится со следующей ячейки нашей нити. Внутренний интерпретатор завершит работу, когда дойдет до конца нити и не останется других нитей для возобновления в его стеке возврата.

Но конечно же функции не могут изменить **pc** переменную, если они не определены в ее лексической области видимости[8], поэтому обратимся к другой технике макросов: вместо использования **defun** мы создаем похожий интерфейс, который делает нечто совершенно иное. **Def-forth-naked-prim** очень похож на создание функций с помощью **defun** за исключением того, что код, в который он расширяется, вставляет формы, определённые пользователем, в список, хранимый в **forth-prim-forth**. Наш итоговый макрос будет использовать эти формы, для того чтобы определять примитивы форта в лексической области видимости. Так как эти формы всегда будут разворачиваться в эту среду, то мы можем без ограничений писать код, использующий все наши абстрактные регистры форта, такие как **pc**, **pstack** и т.д.

Примитивы, определенные с помощью **def-forth-naked-prim** не будут устанавливать переменную **pc** на следующую **cons** ячейку в нити. Для большинства примитивов нам следует использовать **def-forth-prim** для выполнения обычного обновления. Оба этих макроса ожидают, что первым аргументом будет символ для ссылки на данный примитив, вторым будет булево значение, указывающее, будет ли данный примитив незамедлительным. Остаток аргументов – лисп формы, которые будут вычисляться при выполнении примитива.

Восемь простых примитивов – ни один из них не является изолированным (naked) или незамедлительным – будут описаны прямо сейчас. **Nop** – фиктивная инструкция, которая ничего не делает ("нет операции" ("no operaiton")). Примитив ***** является операцией умножения: он достает два верхних элемента из стека параметров, перемножает их, затем помещает результат обратно. **Dup** – краткое название для "дубликат" ("duplicate"), он помещает верхний элемент стека снова в стек, создавая два идентичных значения. **Swap** меняет местами два элемента стека параметров, используя очень полезный COMMON LISP макрос: **rotatef**. Это не случайно, что форт также имеет (оперирующие стеком) механизмы замены. **Print** вынимает элемент стека параметров и печатает его. **>r** перемещает значение из стека параметров в стек возврата, **r>** делает обратную операцию.

Нарушает ли имя ***** наше важное правило захвата переменых из *раздела 3.5 Нежелательный Захват*, которое запрещает переопределять функции, определенные самим COMMON LISP? Нет, потому что, на самом деле, мы не импользуем этот символ для связывания с какой-то функцией — это просто первый элемент одного из списков в **foth-prim-forms**. Мы не сделали ничего неправильного. Символы независимы от

[8]Есть исключения. Смотри Анафорические макросы.

Листинг 8.6: BASIC-PRIM-FORMS

```
(def-forth-prim nop nil)

(def-forth-prim * nil
  (push (* (pop pstack) (pop pstack))
        pstack))

(def-forth-prim drop nil
  (pop pstack))

(def-forth-prim dup nil
  (push (car pstack) pstack))

(def-forth-prim swap nil
  (rotatef (car pstack) (cadr pstack)))

(def-forth-prim print nil
  (print (pop pstack)))

(def-forth-prim >r nil
  (push (pop pstack) rstack))

(def-forth-prim r> nil
  (push (pop rstack) pstack))
```

функций или макросов, которых они иногда обозначают. Мы можем использовать любые символы где угодно, до тех пор, пока мы уверены, что не нарушаем наши важные правила захвата переменных. Они вступают в игру только при написании лиспа; мы же пишем форт.

8.3 Двойственность Синтаксиса, Определённая

Если вы ещё ничего не запомнили из этой книги, то запомните заголовок этого раздела. Здесь мы, наконец, определяем и объясняем, уже затронутую ранее, концепцию: *двойственность синтаксиса*. Подразумевается что ещё до чтения этой главы вы прочитали хотя бы три первых глав, *главу 6, Анафорические Макросы* и предыдущие разделы этой главы.

Большинству программистов очевидно из опыта, что программировать на лиспе более производительно и в итоге более естественно, чем программировать на Блаб языке, но гораздо труднее ответить на вопрос почему это так. Хоть и правда, что лисп черпает удивительную силу выразительности из макросов — и мы видели много интересных примеров в этой книге и в других источниках — все объяснения кажутся неудовлетворительными. В чем настоящее преимущество макросов? Частичное объяснение конечно включает *лаконичность*, делающее ваши программы короче. Вот его определение:

> Пусть L язык программирования, F свойство (feature) этого языка программирования, и A произвольная (arbitrary) программа на L. F обеспечивает лаконичность L, если A короче, чем была бы в версии L без F.

Свойства лаконичности определяют базис и рациональность *теории лаконичности*:

> Трудозатраты, необходимые для написания программы обратно пропорциональны количеству свойств лаконичности, доступных в используемом языке программирования.

Теория лаконичности основана на идее о том, что если ваши программные абстракции делают выражения в программе очень короткими и выразительными, то написание их становится легче, потому что требуется написать меньше кода. Наши CL-PPCRE макросы чтения являются примерами свойств лаконичности: они укорачивают достаточно длинные

CL-PPCRE имена функций до выражений в стиле Perl, которые экономят нажатия на клавиши каждый раз, когда мы их используем. Теория лаконичности хорошо применима для написания маленьких программ, которые понятны для написания с самого начала[9]. К сожалению, большинство программ не такие. Большинство программ — по крайней мере интересных — созданы *итеративно* посредством серии интерактивных пиши-тестируй циклов, результаты которых принимаются в расчет на каждом шаге на всем протяжении процесса. Ваши абстракции могут быть краткими, но если вы всегда вынуждены менять их на другие (возможно такие же короткие) абстракции, вы скорее всего не сократите усилия. Вместо оценки длины итоговой программы, возможно, будет лучше если мы оценим длину процесса её создания.

В каждом языке законченные программы выглядят отличными от их начальной версии. Большинство программ рождаются просто как набросок, который заполняется и детализруется по мере изучения автором задачи. Перед тем как мы вернемся к лаконичности и двойственности, эта глава проведет нас через разработку простой программы, которая подогреет дискуссию: нашей среды форта.

Хм, где мы были? Ах да, мы поболтали немного об *абстрактных регистрах*, *абстрактных машинах* и *шитом коде*, кроме этого определили утилиту для поиска команд под названием **forth-lookup**, внутренний интерпретатор для нашего cons шитого кода и систему для сбора *примитивов*, представленных списками в нашей форт системе. Но каким будет форт на лиспе? Ну, какая самая естественная форма для любой абстракции, которая сочетает в себе поведение и состояние? Замыкания, конечно. Наши старые друзья, let и lambda. Поработав над этой идеей можно получить следующий макрос:

```
(defmacro new-forth ()
  `(let ,forth-registers
     (forth-install-prims)
     (lambda (v)
       (let ((word (forth-lookup v dict)))
 (if word
     (forth-handle-found)
   (forth-handle-not-found))))))
```

Наш список абстрактных регистров форта, **forth-registers**, будет объединён непосредственно с расширением, где абстрактные регистры изначально привязанны к **nil**. Обратите внимание что мы оставили прилично

[9]Часто называемые как Perl-однострочники, даже если они написаны не на Perl.

Листинг 8.7: GO-FORTH

```
(defmacro! go-forth (o!forth &rest words)
  `(dolist (w ',words)
     (funcall ,g!forth w)))
```

пробелов в функциональности этого макроса. Мы обнаружили, что нам ещё предстоит определить макрос **forth-install-prims**, устанавливающий наши формы-примитивы, а также макросы **forth-handle-found** и **forth-handle-not-found**. Но самое важное, что мы вынесли из этого наброска: да, архитектура этого замыкания выглядит жизнеспособной и вполне может работать. Идея, являющаяся логическим следствием простому следованию архитектуре лиспа, приводит к тому, что форт будет замыканием, вызываемым единожды для каждого слова, переданного нами в форт. Наш набросок описывает реализацию для *последующего использования (use case)*. Так мы представляем создание новой форт среды:

```
(defvar my-forth (new-forth))
```

Некоторый форт код, возводящий в квадрат число 3 и печатающий результат:

```
3 dup * print
```

Вот так мы можем исполнить его в нашей форт среде:

```
(progn
  (funcall my-forth 3)
  (funcall my-forth 'dup)
  (funcall my-forth '*)
  (funcall my-forth 'print))
```

Довольно корявый в использовании интерфейс, но мы программируем в лиспе, и мы знаем что всегда можем создать макрос, скрывающий эти детали, именно это мы сделаем в макросе **go-forth**. Заметьте, что **go-forth** использует автоматическую **once-only** (только единожды) функциональность **defmacro!**, поскольку первый аргумент переданный в **go-forth** попадает во внутренний цикл определённый через **dolist** и, возможно, будет вычислен не только единожды, как это могло бы быть задумано пользователем макроса. С помощью **go-forth** ввод форт кода в нашу форт среду становится гораздо более ясным:

Листинг 8.8: FORTH-STDLIB

```
(defvar forth-stdlib nil)

(defmacro forth-stdlib-add (&rest all)
  `(setf forth-stdlib
         (nconc forth-stdlib
                ',all)))
```

```
(go-forth my-forth
  3 dup * print)
```

В этот момент обстоятельства могут сложиться так, что нам периодически может потребоваться выполнение некоторого самозагружаемого форт кода при создании новых форт сред. Мы должны быть в состоянии вызвать замыкание при создании сред. Что в свою очередь может потребовать изменить архитектуру приложения "Let, окружающий Lambda"или, возможно, создать некоторую разновидность функции-обёртки поверх нашего макроса **new-forth** использующую макрос **new-forth**, загружающую стандартную библиотеку и возвращающую получившийся форт.

Поскольку форт код — это всего лишь список символов и других атомов, то наша *стандартная библиотека*, обеспечивающая всю необходимую нам самозагрузку (за исключением некоторых примитивов), может содержаться в списке. Переменная **forth-stdlib** содержит этот список форт кода и это содержимое будет запускаться при создании новых фортов, а макрос **forth-stdlib-add** будет расширяться в лисп код, объединяющий новый форт код со списком **forth-stdlib**.

С помощью какого самого лёгкого способа можно изменить **new-forth** так, чтобы он поддерживал загрузку этой стандартной библиотеки? Вы помните макрос **alet**, который мы написали в разделе 6.3, Alet и Машины с Конечным Состоянием? Целью этого макроса было создание двойственности синтаксиса с помощью **let** COMMON LISP'а одновременно привязывая анафорическую переменную **this** к переданному коду. **This** ссылается на результат, который будет возвращен из **alet** — замыкание форта.

Изменить наш набросок будет даже легче, чем ожидалось. Все, что нам нужно сделать — поменять первый **let** символ в нашем наброске на **alet** и затем добавить немного кода для загрузки стандартной среды в

this, замыкание форта[10]. Мы ничего не должны перестраивать, потому что синтаксис **alet** предусмотрительно связан с синтаксисом **let**. Ниже представлен вид этой следующей итерации:

```
(defmacro new-forth ()
  `(alet ,forth-registers
         (forth-install-prims)
         (dolist (v forth-stdlib)
           (funcall this v))
         (lambda (v)
           (let ((word (forth-lookup v dict)))
             (if word
                 (forth-handle-found)
               (forth-handle-not-found))))))
```

Помните, что **alet** вводит слой косвенности, используя замыкание и таким образом делает нашу среду форта слегка менее эффективной. Однако, так как мы не знаем будет ли эта неэффективность слишком критичной, мы также не знаем стоит ли избавляться от этой косвенности. Чтобы исключить косвенность, используйте макрос **alet%**, определенный до **alet**.

Возможно теперь, а возможно позже, когда мы попытаемся собрать или отладить нашу форт среду, может оказаться, что было бы полезно иметь доступ к абстрактным регистрам извне среды форта. К несчастью эти переменные закрыты **let**-ом, окружающим **lambda**. Мы должны будем снова изменить нашу программу, чтобы сделать их доступными. Существует, конечно же, много способов сделать это. Мы можем встроить и возвращать много замыканий в нашей среде форта, часть из которых сохраняет и обращается к абстрактным регистрам, или мы можем пересмотреть нашу стратегию **let**, окружающую **lambda** полностью. Но перед тем как начать действовать посмотрим есть ли какая-нибудь двойственность, помогающая нам? Вспомним **plambda** из разела 6.7, Пандорические Макросы? Их цель была создать двойственность синтаксиса с помощью **lambda**, а также создать замыкание, на самом деле открытое для остального мира. Изменение нашего наброска для поддержания этого лишь вопрос добавления к префиксу символа p к **lambda**, которое мы возвращаем как наше замыкание и добавление списка переменных, который мы хотим экспортировать. Наш список удобно доступен нам в

[10]Выполняется после установки примитивов, таким образом, ими может воспользоваться и стандартная библиотека.

Листинг 8.9: NEW-FORTH

```
(defmacro new-forth ()
  `(alet ,forth-registers
         (setq dtable (make-hash-table))
         (forth-install-prims)
         (dolist (v forth-stdlib)
           (funcall this v))
         (plambda (v) ,forth-registers
                  (let ((word (forth-lookup v dict)))
                    (if word
                        (forth-handle-found)
                      (forth-handle-not-found))))))
```

forth-registers[11]. Наш набросок приобретает такой вид:

```
(defmacro new-forth ()
  `(alet ,forth-registers
         (forth-install-prims)
         (dolist (v forth-stdlib)
           (funcall this v))
         (plambda (v) ,forth-registers
                  (let ((word (forth-lookup v dict)))
                    (if word
                        (forth-handle-found)
                      (forth-handle-not-found))))))
```

С открытым замыканием форта нам становится доступен следующий сценарий использования. Вставим пять элементов в стек форта:

```
* (go-forth my-forth
            1 2.0 "three" 'four '(f i v e))
NIL
```

И мы можем пандорически открыть **my-forth**, чтобы посмотреть его стек параметров:

[11]Хотя вклеивание этих переменных с помощью раскавычивания будет работать только при написании макросов, макросы чтения позволяют нам сделать похожие вещи при написании функций.

```
* (with-pandoric (pstack) my-forth
                 pstack)
((F I V E) FOUR "three" 2.0 1)
```

Это был путь, который привел к нашей окончательной версии макроса **new-forth**. Окончательное определение идентично последнему наброску за исключением того, что устанавливается абстрактный регистр **dtable**, указывающий на хэш таблицу (будет рассмотрено позже).

Программирование, по крайней мере интересная его часть, это не написание программ, а, на самом деле, их изменение. С точки зрения эффективности лаконичность только подводит нас к этому. Мы можем переименовать **lambda** в, скажем **fn**, но такое свойство лаконичности не сэкономит много за исключением нескольких нажатий кавиш там и сям[12]. Что может сэкономить усилия, так это наличие абстракций, похожих на **lambda**, которые мы можем использовать для изменения работы кода, не меняя значительно сам код. Двойственность синтаксиса сэкономит усилия.

Также как выделение *"наушниками"* имен ваших специальных переменных может раздражать вас необходимостью добавлять или убирать символы астериска (*****), когда вы меняете свое решение о том должна быть эта переменная специальной или лексической[13], ненужное разделение синтаксиса и избегание двойственности может стать причиной многих необоснованных усилий во время программирования. Другой пример: шарп-закавычивать ваши лямбда формы — это плохая идея, потому что в конечном счёте вам придётся вносить намного больше изменений, когда вы решите, что функция на самом деле должна быть **alambda** или когда вы решите использовать **lambda** форму на позиции функции в списке. Обобщенные переменные также обеспечивают очень важную двойственность: когда пишем макросы одна и та же форма может быть вклеена в раскрытие как для считывания, так и для изменения переменной. Другой пример — COMMON LISP-овксая двойственность в обозначении пустого списка и значения **false** булевой переменной — на самом деле нет никаких причин для того, чтобы эти две сущности объединялись в одну за исключением двойственности синтаксиса. Также из-за двойственности эта книга пропагандирует замыкания вместо других свойств CLOS[14] таких как **defclass** и **defmethod**. Обычно бывает меньше *труд-*

[12]Многие из нас считают, что **lambda** и так хороша, спасибо.

[13]Или возможно оставляя некорректную документацию в коде.

[14]Здесь слово "других" подразумевает тот факт, что даже программирование с замыканиями в COMMON LISP является свойством CLOS. CLOS настолько фундаментален, что вы не можете избежать его (и не должны этого хотеть).

ностей при модификации программ, которые используют замыкания, чем при модификации программ, использующих классы и объекты, так как у нас есть так много хороших двойственностей синтаксиса для замыканий и потому что программирование макросов, которые строят замыкания, более унифицировано[15]. Учитывая эти и другие примеры мы можем наконец дать ясное определение того, что мы подразумеваем под двойственностью синтаксиса:

> Пусть L — язык программирования, F — свойство в этом языке программирования, и A и B — произвольные программы на L. F является свойством двойственности синтаксиса, если требуется меньше модификаций для преобразования A в B чем в версии языка L без F.

Что приводит к *гипотезе двойственности*:

> Усилия, необходимые для написания программы обратно пропроциональны объему двойственного синтаксиса, доступного в используемом языке программирования.

В то время как концепция двойственности синтаксиса и влияния их преимуществ совершенно понятны, гораздо менее понятно как на деле придумать хорошие двойственности. Какие двойственности наиболее полезны в конкретном языке? Как можно определить какой из двух различных языков предлагает лучшие двойственности синтаксиса для некоторой данной проблемы?

Поскольку в лиспе мы контролируем язык программирования полностью, то у нас есть возможность спроектировать язык с таким уровнем двойственности синтаксиса, какой только мы пожелаем. На мой взгляд такая цепь размышлений, на данный момент, наиболее плодородная область в исследовании языков программирования. Используя лисп макросы мы можем делать похожими друг на друга наши разрозненные программы, упростится ли изменение этих программ в новые программы[16]?

В обоих определениях лаконичности и двойственности, является ли свойство F эффективным или нет, зависит от программы, которую пишут или изменяют. Иногда свойства, которые обеспечивают лаконичность или двойственность, могут на деле увеличить количество затрачиваемых усилий. Лучшим подходом может стать предоставление стольких

[15]Стоит сказать, что обобщенные функции очень важны, так как они реализуют двойственность мульти-методов.

[16]Иногда превращение программ в новые программы называется разработкой, особенно когда итоговая программа больше чем исходная.

полезных свойств лаконичности и двойственности сколько является возможным, так как удаление ставших ненужными свойств может привести больше к вреду, чем к пользе.

8.4 Работающий Форт

В этом разделе мы получим работающий форт, заполнив *пробелы*, оставленные в макросе **new-forth** из предыдущего раздела. После проверки работы механизма нити форта, мы запустим среду программирования форта, и, попутно, рассмотрим что же такое **незамедлительность (immediacy)** форта, и как оно относится к макросам лиспа.

В определении **new-forth** мы оставили пробелы в макросе, которые будут заполнены с помощью **forth-install-prims**. Мы хотели бы использовать именованную абстракцию без отбрасывания лексической среды, поэтому нам нужен макрос. Смысл этого макроса заключается в том, чтобы компилировать и устанавливать примитивы в форт словарь при создании нового экземпляра форта. **Forth-install-prims** расширяется в **progn** форму, где каждая внутренняя форма является инструкцией, добавляющей слово-примитив в связанный список **dict**, оборачивает переданный код в **lambda**, и устанавливает **name** (*имя*) слова и слоты **immediate**. В дополнение, функция, созданная для каждого слова с помощью **lambda**, называется как **thread** (*нить*) и добавляется к нашей хэш-таблице **dtable** (рассмотрим позже). Поскольку все эти функции будут созданы в области видимости нашего исходного макроса **new-forth**, то у них будет полный доступ к форт среде, определённой нашими абстрактными регистрами. Заметьте, что привязка **thread** не захватывает **thread** из переданного пользователем кода, поэтому нам не нужно прибегать к именованию через **gensym**.

Мы сказали, что форт обеспечивает систему мета программирования, не совсем отличную от лисповой и эта система базируется на концепции так называемой *незамедлительности*. В традиционных фортах есть переменная под названием *состояние* (**state**), принимающая значения ноль или не-ноль. Если значение ноль, то считается, что форт находится в обычном интерпретирующем (исполняющем) состоянии. Если мы передадим слово в форт в этом состоянии, то оно будет найдено и выполнено. Если же переменная **base** не-ноль, то говорится, что форт находится в состоянии компиляции. Если мы передаем слово в форт в этом состоянии, то адрес переданного слова будет добавлен к текущей компилируемой нити — обычно самому последнему созданному слову в словаре. Правда существует одно исключение, и это важное свойство

Listing 8.10: FORTH-INSTALL-PRIMS

```
(defmacro forth-install-prims ()
  `(progn
     ,@(mapcar
         #`(let ((thread (lambda ()
                           ,@(cddr a1))))
             (setf dict
                   (make-forth-word
                      :name ',(car a1)
                      :prev dict
                      :immediate ,(cadr a1)
                      :thread thread))
             (setf (gethash thread dtable)
                   ',(cddr a1)))
         forth-prim-forms)))
```

Listing 8.11: FORTH-PRlMS-COMPILATION-CONTROL

```
(def-forth-prim [ t ; <- t означает незамедлительность.
  (setf compiling nil))

(def-forth-prim ] nil ; <- не незамедлительность
  (setf compiling t))
```

незамедлительности. Если в состоянии компиляции передать незамедлительное слово, то это слово будет выполнено, а не скомпилировано. Таким образом, также как в лиспе, форт позволяет нам выполнять произвольный код во время компиляции.

Так как мы строим наш форт как абстрактную машину на лиспе, нам не запрещено произвольное сопоставление целочисленных значений значениям **true** и **false**. В лиспе динамическая система типов, которая позволяет наслаждаться произвольным сопоставлением любых значений в **true** и **false**. Вместо переменной **state** состояния форта наша система форта использует абстрактный регистр компиляции **compiling** для хранения нашего состояния компиляции как *обобщенной булевой переменной*. Традиционные слова форта для изменения состояния компиляции — **[** и **]**, открывающая и закрывающая квадратные скобки. **[** выводит нас из состояния компиляции и поэтому это слово должно быть незамедлитель-

Listing 8.12: FORTH-COMPILE-IN

```
(defmacro forth-compile-in (v)
  `(setf (forth-word-thread dict)
         (nconc (forth-word-thread dict)
                (list ,v))))
```

ным словом.] возвращает нас обратно в режим компиляции, выполнятся только если мы в режиме интерпретации и не должно быть незамедлительным. Такой выбор символов может сейчас показаться странным, но станет более понятным в высокоуровневом форт коде. Эти квадратные скобки позволяют нам пометить для выполнения блок кода во время компиляции форт нити. В определенном смысле эти скобки похожи на лисповские операторы обратной кавычки и раскавычивания. Ниже пример типичного использования этих слов в форт коде:

```
... компилированные слова ...
[ interpret these words ]
... ещё компилированные слова ...
```

Как и большинство деталей форта эти слова определены прозрачно, что позволяет нам использовать их нетрадиционным способом. Например, эти слова не сбалансированы в том же смысле, что и скобки лиспа. При желании мы можем использовать их в противоположном направлении:

```
. интерпретировать эти слова ...
] компилировать эти слова [
. интерпретировать эти слова ...
```

У нас даже возникает вложенность, но это не настоящая вложенность, поскольку нам доступно единственное состояние: Компилировать или не компилировать.

```
. компилированные слова ...
[ интерпретировать эти слова
] компилировать эти слова [
интерпретировать эти слова
]
. ещё компилированные слова . . .
```

Листинг 8.13: FORTH-HANDLE-FOUND

```
(defmacro forth-handle-found ()
  `(if (and compiling
            (not (forth-word-immediate word)))
       (forth-compile-in (forth-word-thread word))
       (progn
         (setf pc (list (forth-word-thread word)))
         (forth-inner-interpreter))))
```

Наш форт использует макрос **forth-compile-in** как макрос-аббревиатуру. Этот макрос компилирует форт слово в нашу текущую нить, нить последнего созданного слова. Поскольку наши нити представлены в виде **cons** ячеек, то мы можем использовать лисп функцию **nconc** для простого добавления указателя на нить нужного слова в нашу текущую нить.

Другой пробел оставленный в макросе **new-forth** заключается в том, что должен делать форт если удалось найти указанное слово в словаре. Этот пробел заполнится макросом **forth-handle-found**. Этот макрос реализует описанную выше незамедлительность форта. Если мы выполняем компиляцию и найденное слово не является незамедлительным, то мы компилируем его в текущую нить. В противном случае мы помещаем в наш программный счётчик **pc** указатель на нить найденного слова и запускаем внутренний интерпретатор на исполнение слова. Не забудьте, что этот макрос будет расширен в лексическую среду, в которой **word** привязан к найденному форт слову.

Наш последний пробел в **new-forth** заключается в том, что форт должен сделать если не может найти слово в своём словаре. **Forth-handle-not-found** заполняет этот пробел и реализует некоторые специальные случаи. Напомним, что **forth-handle-not-found** будет расширен в лексическую среду, содержащую привязку **v**, ссылающуюся на значение, переданное форту. Также мы знаем, что если будет вызван этот код, **v** не будет ссылаться ни на какое слово в словаре. Если **v** будет символом, **forth-handle-not-found** вызовет ошибку. Если значение не будет символом, то поведение будет заключаться во вставке **v** в стек параметров или, если мы выполняем компиляцию, скомпилировать его в текущую нить. Однако, будет выполняться проверка на два специальных случая. Если **v** будет списком с первым элементом **quote**, то мы вставим закавыченное значение в стек параметров. Это объясняется тем, что мы можем вставлять символы в стек параметров без интерпретирования их

Листинг 8.14: FORTH-HANDLE-NOT-FOUND

```
(defmacro forth-handle-not-found ()
  `(cond
     ((and (consp v) (eq (car v) 'quote))
      (if compiling
          (forth-compile-in (cadr v))
          (push (cadr v) pstack)))
     ((and (consp v) (eq (car v) 'postpone))
      (let ((word (forth-lookup (cadr v) dict)))
        (if (not word)
            (error "Postpone failed: ~a" (cadr v)))
        (forth-compile-in (forth-word-thread word))))
     ((symbolp v)
      (error "Word ~a not found" v))
     (t
      (if compiling
          (forth-compile-in v)
          (push v pstack)))))
```

как слов. Второй специальный случай возникает тогда, когда **v** является списком с первым элементом **postpone** (*отложить*). **Postpone** — это слово из ANSI Forth, объединяющее и разъясняющее пару традиционных форт слов. **Postpone** используется для того, чтобы всегда компилировать слово даже если это слово будет незамедлительным. Таким образом, если мы находимся в режиме компиляции, отложенное незамедлительное слово будет скомпилировано в наш текущий тред даже если это слово незамедлительно. Пример откладывания слова **[**:

```
... компилирование ...
(postpone [)
... компилирование продолжается ...
```

Все пробелы в макросе **new-forth** заполнены, и теперь мы можем создать новый экземпляр форта с помощью макроса **new-forth**. Раньше мы создали специальную переменную под названием **my-forth** с помощью **defvar**. Даже если бы мы этого не сделали, мы можем неявно декларировать её специальной присвоив ей значение с помощью высокоуровневого **setq**[17]:

[17]Некоторые реализации не дадут вам выполнить это. Решением будет перейти на

Листинг 8.15: FORTH-PRIMS-DEFINING-WORDS

```
(def-forth-prim create nil
  (setf dict (make-forth-word :prev dict)))

(def-forth-prim name nil
  (setf (forth-word-name dict) (pop pstack)))

(def-forth-prim immediate nil
  (setf (forth-word-immediate dict) t))
```

```
* (setq my-forth (new-forth))
#<Interpreted function>
```

Теперь, с помощью **go-forth** мы можем использовать форт:

```
* (go-forth my-forth 2 3 * print)

6
NIL
```

Но до сих пор мы определили только слова **dup**, ***** и **print**. Для того, чтобы сделать что-то полезное нам нужно больше примитивов. Как и лисп, реализации форта промышленного уровня обладают большим количеством слов, определённых для удобства программиста. В процессе десятилетий использования форта, были определены многие общие шаблоны программирования, абстрагированы в слова и затем введены в общеиспользуемый форт. Как и в лиспе наличие возможности расширять язык, определённый как часть языка, стала результатом очень ценных экспериментов. Поскольку мы изучаем этот процесс и философию, то мы не будем определять многие форт слова, которыми пользуются опытные форт программисты. Вместо этого нашей целью будет минимальный набор примитивов, требуемых для изучения системы метапрограммирования форта и её сравнение с лисп макросами.

Определены ещё три примитива, ни один из них не является незамедлительным или незащищённым: **create**, **name** и **immediate**. Слово **create** добавляет безымянное слово в словарь. **Name** извлекает значение из стека параметров и устанавливает название последнего слова словаря

реализацию, позволяющую выполнить эту операцию (например CMUCL) или использовать в таких случая **defparameter** вместо **setq**.

в это значение. **Immediate** просто делает незамедлительным последнее определённое слово. По умолчанию слова не являются незамедлительными.

Напомним что мы можем исполнять код с помощью нашего макроса **go-forth** в нашей форт среде **my-forth**. Так мы возведём в квадрат число 3 и напечатаем результат:

```
* (go-forth my-forth 3 dup * print)

9
NIL
```

Достаточно ли у нас форта для запуска самопостроения форта с помощью форт слов? И хотя мы ещё не определили достаточно слов, но благодаря прозрачной спецификации поточного кода мы можем написать форт слова, используя форт. Например, вот так можно использовать **create** для добавления новых пустых слов в словарь:

```
* (go-forth my-forth create)

CREATE
NIL
```

Теперь мы можем использовать **]** для начала компиляции, добавить слова **dup** и ***** в тред, затем использовать **[**, стобы выйти из режима компиляции:

```
* (go-forth my-forth ] dup * [)

NIL
```

Теперь у нас есть новое слово в нашем словаре — одно с законченной форт нитью, которое, исполненное нашим внутренним интерпретатором, возведет в квадрат первое число в стеке. Но это слово не очень полезно, если у нас не будет способа вызвать его. Мы можем дать ему имя, используя слово **name**. Присвоенное имя будет ключом доступа к нашей новой нити.

```
* (go-forth my-forth 'square name)

NIL
```

Листинг 8.16: FORTH-START-DEFINING

```
(forth-stdlib-add
 create
 ] create ] [
 '{ name)
```

Листинг 8.17: FORTH-STOP-DEFINING

```
(forth-stdlib-add
 { (postpone []) [
 '} name immediate)
```

Обратите внимание, что первое передаваемое значение закавычено. Вспомните, что мы решили что такое поведение заставит форт поместить символ **square** в стек параметров. Этот символ потом будет использован словом **name**. Теперь наше слово названо, и мы можем вычислять его как любое другое, используя символ **square**:

```
* (go-forth my-forth 3 square print)

9
NIL
```

Итак, следующий шаблон представляет общую технику создания новых слов:

```
create
] ... компилируемые слова ... [
'что-нибудь name
```

Но мы можем использовать кусочек мета-программирования форта, чтобы улучшить этот интерфейс. Определение нового форт слова **{** добавлено в стандартную библиотеку. Эта нить состоит из двух указателей, первый указывает на слово **create**, а второй указывает на слово **]**. Таким образом, когда нить данного слова выполняется, то в словарь добавляется новое слово и мы переключаемся в режим компиляции. Форт обычно использует для этой цели слово **:**, но из-за конфликта с использованием **:** в лиспе мы выбрали слово **{** для определения слова.

Аналогично, мы добавляем сопряженное слово **}** в стандартную библиотеку (заменяя традиционное фортовское **;**). На самом деле нет причин

для определения этого слова — единственное, что оно делает, выводит нас из режима компиляции. Для этого у нас уже есть слово **[**. Несмотря на это, определение **{** полезно, потому что даёт нам *нормально сбалансированные скобки*[18] создавая пару слов **{** и **}**, что делает определение новых слов интуитивным.

Теперь мы можем создать форт для того, чтобы воспользоваться новыми особенностями стандартной библиотеки (отбросив наше предыдущее определение слова **square**):

```
* (setq my-forth (new-forth))

#<Interpreted Function>
```

С новыми словами определения слов **{** и **}** **square** будет выглядеть так:

```
* (go-forth my-forth { dup * } 'square name)

NIL

* (go-forth my-forth 5 square print)

25
```

И новые нити могут ссылаться на наши специально созданные слова так, как если бы они были примитивами. Так мы можем определить слово **quartic** как нить с двумя указателями на наше слово **square**:

```
* (go-forth my-forth { square square } 'quartic name)

NIL
```

Действительно, **(Expt 1/2 4)** равен **1/16**:

```
* (go-forth my-forth 1/2 quartic print)

1/16
NIL
```

[18]Как противоположность к обратно сбалансированным квадратным скобкам форта, данным для определения слова.

Поскольку не-символы напрямую компилируются в форт нить и наш внутренний интерпретатор рассматривает не-функции как элементы данных, при встрече с которыми следует выполнить вставку в стек, то мы можем включать числа в определение слов:

```
* (go-forth my-forth { 3 } 'three name three three * print)

9
NIL
```

Напомним, что мы просматриваем все элементы переданные форту в поисках имени в словаре с помощью функции **eql**. Следствием этого является то, что мы можем использовать любой лисп объект в качестве имени слова. Так мы используем число[19]:

```
* (go-forth my-forth { 4.0 } '4 name 4 4 * print)

16.0
NIL
```

Форт — замечательный язык для изучения использования области видимости указателя. Форт определяет два простых оператора, используемых для чтения/записи значений из/в память: **@** (fetch — получить) и **!** (store — поместить). Поскольку наши форт слова сохранены в **cons** ячейки вместо слов памяти, то переопределение указателя с помощью fetch реализовано через получение **car** указателя. Присвоение значения реализовано через присвоение значения **car**-у используя **setf**. **Fetch** извлечёт значение из стека параметров, предполагая его **cons** ячейкой, выделит его **car** и поместит его в стек. Store извлечёт значение из стека параметров, предполагая его **cons** ячейкой, извлечёт другое значение из стека и поместит его в **car** первого значения. Пример того, как мы можем создать и напечатать циклический список:

```
* (let ((*print-circle* t))
    (go-forth my-forth '(nil) dup dup ! print))

#1=(#1#)
NIL
```

[19]Во многих фортах, такие общие числа, как 0, 1, -1 и так далее, определены как слова поскольку скомпилированный указатель на слово, обычно, занимает меньше памяти чем скомпилированная лексема.

Листинг 8.18: MEMORY-PRIMS

```
(def-forth-prim @ nil
    (push (car (pop pstack))
          pstack))

(def-forth-prim ! nil
  (let ((location (pop pstack)))
    (setf (car location) (pop pstack))))
```

Listing 8.19: FORTH-UNARY-WORD-DEFINER

```
(defmacro forth-unary-word-definer (&rest words)
  `(progn
     ,@(mapcar
        #`(def-forth-prim ,a1 nil
            (push (,a1 (pop pstack))
                  pstack))
        words)))
```

Теперь мы программируем в форте используя шитый код. Где же мы? Отошли ли мы отошли от лиспа? Разница между двумя языками настолько размыта, что её едва ли можно заметить. Остаток этой главы будет посвящён ещё большему размытию при дальнейшем изучении метапрограммирования форта.

8.5 Более Фортово

В COMMON LISP есть порядочно функций, которые нам хотелось бы иметь возможность вставить в наши форт нити. **Forth-unary-word-definer** раскрывается в такое же количество форм **def-forth-prim**, сколько элементов передано в его тело макроса. Под элементами подразумеваются символы, обозначающие либо макросы, либо функции, но они также могут быть лямбда формами. Единственным ограничением для примитивов, именованных лямбда формами является то, что для их вызова вам понадобится передать такую же (**eq**) лямбда форму в среду форта. Вот раскрытие, когда передан один символ, **not**:

```
(macroexpand '(forth-unary-word-definer not))
```

Listing 8.20: FORTH-BINARY-WORD-DEFINER

```
(defmacro! forth-binary-word-definer (&rest words)
  `(progn
     ,@(mapcar
        #`(def-forth-prim ,a1 nil
            (let ((,g!top (pop pstack)))
              (push (,a1 (pop pstack)
                         ,g!top)
                    pstack)))
        words)))
```

Листинг 8.21: FORTH-AND-LISP-WORDS

```
(forth-unary-word-definer
 not car cdr cadr caddr cadddr
 oddp evenp)
(forth-binary-word-definer
 eq equal + - / = < > <= >=
 max min and or)
```

```
(PROGN (DEF-FORTH-PRIM NOT NIL
 (PUSH (NOT (POP PSTACK)) PSTACK))) ;
T
```

Мы можем использовать любую функцию COMMON LISP, которая принимает один аргумент, и **forth-unary-word-definer** определит форт примитив, который применит эту функцию к верхнему элементу стека параметров форта.

Расширением этой идеи служит **forth-binary-word-definer**, который делает то же самое, только для операторов, принимающих два значения. Правило форта использования второго от вершины элемента как первого элемента для таких двухэлементных функций, как **-** и / достигается созданием временного let связывания для удержания верхнего элемента из стека параметров. Вот раскрытие для слова **-**:

```
* (macroexpand '(forth-unary-word-definer -))

(LET ()
     (PROGN
```

```
      (DEF-FORTH-PRIM - NIL
(LET ((#:TOP1767 (POP PSTACK)))
     (PUSH (- (POP PSTACK) #:TOP1767)
   PSTACK)))))
T
```

Упражнение: Каким образом достигается возможность рассмотрения таких макросов как **and** и **or** в качестве первоклассных значений при использовании **forth-binary-word-definer**?

Трудное упражнение: Почему необходимо использовать gensym **(g!top)** для избежания нежелательного захвата переменной в **forth-binary-word-definer**? Подсказка: Мы уже говорили об этом в этом разделе.

Итак, эти макросы позволяют нам добавить разные функции лиспа в нашу среду примитивов форта, и мы можем использовать их внутри форта. Вот пример использования унарного примитива **cadr**:

```
* (go-forth my-forth '(a (b) c) cadr print)

(B)
NIL
```

А вот бинарного $<$:

```
* (go-forth my-forth 2 3 < print)

T
NIL
```

Так как наши нити суть *ориентированные ацикличные графы*, это означает, что они состоят из структуры связанных ячеек, которые нигде не указывают на себя (не являются самоотносящимися) и которые в итоге оканчиваются нашими примитивами, листьями дерева. Например, мы можем использовать пандорические макросы, чтобы получить нить, которую мы создали в предыдущем разделе, когда определяли слово **quartic**:

```
* (with-pandoric (dict) my-forth (forth-word-thread
                                   (forth-lookup 'quartic
                                                 dict)))
((#<FUNCTION :LAMBDA NIL (PUSH          ;; square->|->dup
                          (CAR PSTACK)
  PSTACK)
```

```
            (SETF PC (CDR PC))>
  #<FUNCTION :LAMBDA                    ;;          |->*
  NIL (PUSH
       (* (POP PSTACK)
  (POP PSTACK)) PSTACK)
  (SETF PC (CDR PC))>)
 (#<FUNCTION :LAMBDA NIL                ;; square->|->dup
     (PUSH (CAR PSTACK)
   PSTACK)
            (SETF PC (CDR PC))>
  #<FUNCTION :LAMBDA NIL (PUSH          ;;          |->*
  (* (POP
      PSTACK)
     (POP PSTACK))
  PSTACK)
  (SETF PC (CDR PC))>))
```

Вышерасположенные комментарии показаны с точки зрения печати формы в лиспе. Мы не можем увидеть из кода или из комментариев, что это нитевая структура на самом деле общая. Чтобы увидеть это, используем **eq**:

```
* (eq (car *) (cadr *))
T
```

Или посмотрим на это с установленным ***print-circle***:

```
* (let ((*print-circle* t)) (print **) t)

(#1=(#<FUNCTION :LAMBDA NIL (PUSH (CAR PSTACK) PSTACK)
                (SETF PC (CDR PC))>        ;; square->|->dup
                #<FUNCTION :LAMBDA NIL
                (PUSH (* (POP PSTACK)
                         (POP PSTACK))
      PSTACK)
                (SETF PC (CDR PC))>)       ;;          |->*
    #1#)                                   ;; --------|
T
```

Шитый код форта позволяет достичь удивительных преимуществ в памяти и размере. В целом, системы форта — это скомпилированный код, сшитый вместе так же, как и наш код — начиная с сетевых драйверов

Листинг 8.22: BRANCH-IF

```
(def-forth-naked-prim branch-if nil
  (setf pc (if (pop pstack)
               (cadr pc)
               (cddr pc))))
```

и заканчивая высокоуровневыми пользовательскими программами. Что более важно, обратите внимание, как мы смогли безболезненно извлечь нить из **quartic** не имея дело со многими другими нитями. Например, в нашем языке есть много таких примитивов, как + и **cadddr**, но они не появляются в вышеприведённой нити. Как будто у нас есть сборка мусора использующая алгоритм пометок и извлекающая для нас только нить необходимую для исполнения данного форт слова. В лиспе этот процесс называется *утряской дерева (tree shaking)* и в целом этот процесс не очень эффективен. И наоборот, в форте данный процесс чрезвычайно эффективен.

К несчастью нить **quartic**, пандорически извлекаемая из **my-forth**, на самом деле не так уж и полезна для нас. Она по прежнему постоянно расположена в замыкании **my-forth**. Таким образом, лямбда выражения, представляющие примитивы **dup** и ***** обладают своими ссылками на абстрактные регистры форта, захваченные расширением нашего макроса **new-forth**. Можем ли мы сохранить этот код в плоскости лисп макросов так, чтобы в последующем встроить его в новые программы? Позже мы вернёмся к этому вопросу, но сейчас мы немного углубимся в мета-программирование форта.

На некотором уровне любого языка — обычно уровне скрытом от программиста — обязательным условием для кода является возможность сослаться на самого себя. Наиболее удобным примером этой обязательности является наблюдение, что коду нужна возможность сослаться на самое себя для того, чтобы реализовать циклы, рекурсии и такие условные выражения, как определения **if**. Разница между *Флаб* и не-Флаб языками в том, что Флаб не даёт вам возможности прямой модификации того как, где и каким образом вставлять ссылку на самого себя. Но, лисповский не-Блаб статус обозначает что мы можем сконструировать не-Флаб и мы сейчас это сделаем.

Наша форт система в своём текущем состоянии (без возможности вставки ссылок на саму себя) — почти *чистый Флаб*. Подобно тому, как чистые функциональные языки сознательно определены как язык лишённый побочных эффектов и не статических отображений, так же и

чистые Флаб языки определены так, чтобы не содержать конструкции кода, не содержащие ссылок на самих себя, как например, циклы и рекурсии. Следствием этого является то, что интерпретация Флаб нитей всегда конечна. Наша форт среда не полностью чиста поскольку мы можем — и будем — нарушать Флаб чистоту, но, чиста в том смысле, что если мы будем продолжать использовать уже определённые конструкции, то результатом будут чистые Флаб нити. Чистый Флаб не очень полезен, поэтому давайте разрушим Флаб чистоту нашей форт среды. Вместо того, чтобы идти в направлении Флаба — к таким Флаб языкам, как COMMON LISP где шитьё кода непрозрачно и недоступно — давайте двигаться в направлении форта и сделаем этот атрибут кода макро изменяемым.

Примитив **branch-if** — это первый *изолированный примитив*. Напомним, что изолированные примитивы — это примитивы не обновляющие автоматически абстрактный регистр счётчика программы (**pc**). Вместо автоматического обновления счётчика эти примитивы должны произвести самостоятельное обновление счётчика. **Branch-if** извлекает значение из стека параметров. Если значение не-null, то **pc** устанавливается в содержимое следующей ячейки интерпретируемой нити. Если значение **nil**, то **pc** обрабатывается как обычно, с тем исключением, что пропускается следующая ячейка в интерпретируемой нити.

Например, следующий пример создаёт форт среду, в которой мы можем воспользоваться возможностями нашего нового примитива **branch-if** и определяются два слова: **double** и **if-then-double**.

```
* (go-forth (setq my-forth (new-forth))
            { 2 * } 'double name
            { branch-if double "Not doubling" print }
            'if-then-double name)
NIL
```

Double просто умножает верхний элемент стека параметров на два, удваивает значение. **If-then-double** требует двух элементов в стеке параметров. Верхний элемент нужен для **if-then-double**, а второй элемент будет удваиваться только тогда, когда значение верхнего элемента будет не-null. Заметьте, что поскольку следующее значение в нити после **branch-if** является указателем на другую нить (**double**), то контроль исполнения перемещается на эту другую нить без вставки места возобновления в стек возврата. В лиспе это называется *хвостовым вызовом (tail-call)*. Поэтому если мы передадим **nil**[20] в **if-then-double** то будет

[20]Нам нужно закавычить **nil** поскольку мы не хотим чтобы выполнялся поиск в словаре форта.

Листинг 8.23: EXIT

```
(forth-stdlib-add
 { r> drop } 'exit name)
```

выбрана не удваивающая ветвь и напечатана строка:

```
* (go-forth my-forth
    4 'nil if-then-double print)

"Not doubling"
4
NIL
```

Но если значение не-null, то ветвь не будет выбрана, произойдёт удваивание и строка не напечатается:

```
* (go-forth my-forth
    4 't if-then-double print)

8
NIL
```

Но, существует более простой способ выйти из слова, и этот способ реализован с помощью нового слова под названием **exit**. Интересной особенностью форта является то, что будучи вызванным слово может решить быть ему хвостовым вызовом или нет. **Exit** — это обычное форт слово и процедура вызова осуществляется так же, как и всегда: форт вставляет текущее расположение нити в стек возврата, затем присваивает счётчику программы указатель на начало слова **exit**. После вызова **exit**, поскольку у этого слова есть прямой доступ к стеку возврата с помощью примитивов **r>** и **>r**, мы можем сделать так, чтобы вызванное слово никогда не возвращало контроль исполнения просто переместив место возобновления из стека возврата и отбросив это значение. Вот пример использования **exit**:

```
* (go-forth my-forth
    { "hello" print
    exit
    ;; Никогда не дойдёт до этого места
    "world" print } 'exit-test name
```

Листинг 8.24: COMPILER-PRIMS

```
(def-forth-naked-prim compile nil
  (setf (forth-word-thread dict)
        (nconc (forth-word-thread dict)
               (list (cadr pc))))
  (setf pc (cddr pc)))

(def-forth-prim here nil
  (push (last (forth-word-thread dict))
        pstack))
```

```
    exit-test)

"hello"
NIL
```

Так как **branch-if** реализует прыжок или *goto* инструкцию, с помощью которой возможно перейти к значению, хранящемуся в последующей ячейке нити, которую мы выполняем. Получение значения из нити, которую вы в данный момент выполняете — является обычным шаблоном и требует изолированных примитивов. Другой примитив — **compile** также использует этот шаблон. **Compile** — это изолированный примитив, который будет брать значение следующей ячейки, в нити, выполняющейся в данный момент, и затем компилирует это значение в нить последнего слова, добавленного в словарь, — обычно слова, которое мы в данный момент компилируем. **Here** простой примитив, который вставляет последнюю ячейку компилируемой нити в стек параметров. Наш **here** слегка отличается от обычного фортового слова **here**. В форте **here** обычно вставляет следующее место, которое будет компилироваться вместо последнего скомпилированного места. Это потому что, в традиционном форте, следующее место в памяти для компиляции известно сразу — это будет следующая примыкающая ячейка памяти. Для cons шитого кода мы не можем этого знать, так как мы еще не присоединили эту память.

С доступными **compile** и **here** мы можем теперь начать писать форт макросы. Вспомним, что когда форт слово является *незамедлительным*, то во время компиляции оно будет выполнено, а не скомпилировано в нить последнего определенного слова. Такими же способами, которыми мы можем адаптировать и расширять лисп с помощью макросов, мы

можем использовать незамедлительные слова для адаптации и расширения форта. В лиспе основными структурами данных, используемых для метапрограммирования являются списки. В форте основные структуры данных — стеки.

Вы могли заметить, что наша среда форта даже не имеет выражения **if**. У нас есть условно ветвящийся (branching) примитив, названный **branch-if**, но пока он может быть полезен только для выполнения хвостовых вызовов других слов форта. Вспомним, что форт слова представлены нитями и мы можем поместить значение любой нити в ячейку, на которую совершит переход **branch-if**. Что если мы положим значение, которое ведет к части нити, компилируемой в данный момент? Мы сможем в известном смысле сделать хвостовой вызов к другой части нашего текущего форт слова. Ну что ж, выражение **if** делает то же самое — условный хвостовой вызов к окончанию **if** выражения, который выполняется только когда условие соответствует null-у.

Так как мы сейчас программируем на форте, то нет необходимости добавлять новые примитивы. Чтобы добавить **if** выражение в форт, мы просто добавим некоторый форт код к нашей стандартной библиотеке макросом **forth-stdlib-add**. Заметьте, что **if** определен как незамедлительное слово, это значит, что оно должно использоваться во время компляции. Но так как оно незамедлительное, то оно будет выполнено, а не скомпилировано. Когда встречаются неазмедлительные слова, в заданную нить ничего автоматически не компилируется. Так **if** сам компилируется в три слова в заданную нить: **not**, **branch-if** и **nop**. Затем выполняется слово **here** которая вставляет адрес последнего скомпилированного слова (**nop**) в стек. Оно вставляется в стек? Странное поведение слова во время компиляции. В какой стек вставляем? Строго говоря стек, используемый во время компиляции называется *контрольным стеком*. В большинстве фортов контрольный стек это то же самое, что и стек параметров. Отличие необходимо из-за различных способов реализации форта. Иногда, особенно в кросс-компиляционных средах, контрольный стек совершенно отличается от того что будет в итоге стеком параметров. Но здесь — как с большинством интерактивных форт сред — мы используем стек параметров для контрольного стека.

Итак, **if** помещает значение, соответствующее месту, где был скомпилирован **nop**. Какая от этого польза? **Nop** сам по себе не очень важен, важно то, что ему предшествует. В ячейке, непосредственно предшествующей **nop**, находится скомпилированная инструкция **branch-if**. Что бы мы не меняли значение **nop** будет местом, куда перейдет наш интерпретатор в случае, если условие окажется null.

Листинг 8.25: IF

```
(forth-stdlib-add
 { compile not
 compile branch-if
 compile nop
 here } 'if name immediate)
```

Листинг 8.26: THEN

```
(forth-stdlib-add
 { compile nop
 here swap ! } 'then name immediate)
```

Но почему мы поместили туда **nop**, а не адрес ячейки памяти? Потому что мы еще не знаем этот адрес. Мы должны дождаться, пока программист выполнит другое незамедлительное слово — **then** — которое потребит значение, оставленное **if** в контрольном стеке. **Then** скомпилирует **nop** и запишет расположение этого **nop** поверх **nop**, скомпилированного **if**. Таким образом, все слова между **if** и **then** будут пропущены, если условие равняется null.

Abs — это слово, использующее **if** и **then** для вычисления абсолютного значения верхнего элемента стека. Оно просто проверяет, что значение меньше 0 и, если это так, вызывает другое слово, **negate**, для перевода отрицательного значения в его абсолютное значение.

Самой важной причиной использования контрольного стека в этом процессе компиляции является то,что при использовании стека появляется возможность для *вложенных* структур типа **if**. Таким образом мы можем включать **if** выражения внутрь других **if** выражений до тех пор, пока мы уверены в балансе всех **if** слов с соответствующими **then**.

Так как форт не Флаб язык, то, для нас, для адаптации и расширения, создание и сшивание нитей с контролирующими структурами типа **if** выражений, *определено прозрачно* и открыто. Большинство языков имеет

Листинг 8.27: ABS

```
(forth-stdlib-add
 { 0 swap - } 'negate name
 { dup 0 < if negate then } 'abs name)
```

Листинг 8.28: ELSE

```
(forth-stdlib-add
 { compile 't
 compile branch-if
 compile nop
 here swap
 compile nop
 here swap ! } 'else name immediate)
```

Листинг 8.29: MOD2

```
(forth-stdlib-add
 { evenp if 0 else 1 then } 'mod2 name)
```

else часть, связанную с **if** выражением; может и нам стоит ее добавить. Еще одно незамедлительное слово **else** добавлено к стандартной библиотеке. **Else** компилируется в безусловную ветку, чтобы завершить **then**, так что если мы пошли по истинной (второй или последующей) ветке, то мы пропустим ложную (третью или альтернативную) ветку. **Else** таким образом использует значение, оставленное **if**-ом в стеке, чтобы заменить этот **nop** ссылкой на начало **else** части. Затем оно оставляет в стеке ссылку на свой **nop** для использования **then**-ом. Так как мы хотим, чтобы **then** работал одинаково независимо от того, что оставило ссылку **if** или **else**, **then** будет работать даже если **if** не имеет **else** части.

Слово **mod2** использует **if**, **else** и **then** чтобы уменьшить целое до его натурального остатка по модулю 2. Оно помещает 0, если вершина стека четная и 1, если нечетное.

Листинг 8.30: BEGIN-AGAIN

```
(forth-stdlib-add
 { compile nop
 here } 'begin name immediate
 { compile 't
 compile branch-if
 compile nop
 here ! } 'again name immediate)
```

Так как наши условные операторы выполняют хвостовые вызовы к другим частям компилируемой нити, то нет причины, по которой мы не можем применить ту же технику для создания таких итеративных конструкций как циклы. Самый базовый форт цикл определён с помощью незамедлительных слов **begin** и **again**. Эти два слова обеспечивают простой бесконечный цикл и реализованы очень похоже на **if** и **then**, кроме того, что адрес, который сохраняется в стеке между этими словами соответствует адресу,который должен быть скомпилирован в выражение ветвления, а не в место для компиляции адреса. Вот простой цикл для обратного отсчета до 1, начиная от числа, хранящегося в стеке, по достижению 1 производится выход из слова:

```
* (go-forth my-forth
    { begin
    dup 1 < if drop exit then
    dup print
    1 -
    again } 'countdown name
    5 countdown)

5
4
3
2
1
NIL
```

В вышеприведенном примере видно, что **if** и **then** конструкции вложены внутрь **begin-again** цикла. Благодаря фортовскому контрольному стеку, он идеально приспособлен для хранения любой контролирующей структуры, соблюдающей стек. Для того, чтобы соблюдать стек, контролирующая структура должна избегать вынимания из стека значений, которые она туда не клала, и не оставлять значений по завершению. Но также как мы, при конструировании лисп макросов, выбираем нарушение ссылочной прозрачности, так и в форте мы часто выбираем несоблюдение стека при компиляции. Следующий пример идентичен предыдущему, кроме использования слова **exit** при выходе из цикла. Вместо этого мы входим в режим компиляции, используя слова **[** и **]** и переставляем указатели, помещенные туда словами **begin** и **if** так что мы можем использовать соответствующие им слова **then** и **again** в другом порядке:

```
* (go-forth my-forth
```

```
    { begin
    dup 1 >= if
    dup print
    1 -
    [ swap ] again
    then
    drop } 'countdown-for-teh-hax0rz name
    5 countdown-for-teh-hax0rz )

5
4
3
2
1
NIL
```

Выше код, скомпилированный словом **again**, переносит нас обратно к **begin** и выполняется только в **if** выражении. Очень немногие из других языков позволяют вам получать доступ к компилятору таким способом — если быть точным такое позволяют сделать только не-Флаб языки. Из-за этой свободы форт программисты иногда более привычны к *комбинациям* макросов чем лисп программисты. Хотя лисп код в этой книге регулярно использует техники комбинаций макросов, эти техники и выигрыш, который они могут предоставить, не используются в полную возможную силу большинством существующего лисп кода. Однако, как это пытается проиллюстрировать книга, лисп исключительно хорошо подходит для комбинаций макросов. Место таких комбинационных техник там, где, как я думаю, находятся одни из наибольших достижений в продуктивности программиста которые будут найдены в, примерно, следующее десятилетие исследований языков.

8.6 Переходим в Лисп

В этой главе до этого момента определялась минимальная форт среда и показаны некоторые из наиболее важных фортовских концепций метапрограммирования с точки зрения лиспа. Надеюсь вам стало видно, как мало усилий требуется для проектирования и реализации форта, когда у вас есть правильный инструмент (COMMON LISP). Мы можем писать фортовские программы, которые пишут фортовские программы — но мы и так это знали. Для этого и существует форт. Более того, имея

Листинг 8.31: PRINT-FORTH-THREAD

```
(defun get-forth-thread (forth word)
  (with-pandoric (dict) forth
                 (forth-word-thread
                  (forth-lookup word dict))))

(defun print-forth-thread (forth word)
  (let ((*print-circle* t))
    (print (get-forth-thread forth word))
    t))
```

систему макросов лиспа мы можем писать лисповские программы, которые пишут фортовские программы. Но можем ли мы писать фортовские программы, которые пишут лисповские программы?

Вспомним, что наши форт нити это cons ячейки, связанные вместе и что листья этих деревьев это либо функции (представлящющие примитивы), либо атомы (представляющие данные для помещения в стек параметров). Так как мы решили сделать абсртактные регистры форта доступными через пандорические макросы, легко написать утилиты для получения и печати форт нитей. **Get-forth-thread** пандорически открывает форт замыкание **forth**, переданное ему, затем получает и возвращает нить слова, переданного в **word**. **Print-to-forth-thread** печатает эту результирующую нить с ***print-circle***, установленным в **t** на случай если она содержит циклы.

Для демонстрации предположим что мы определим два форт слова **square** и **square3**:

```
* (go-forth my-forth
    { dup * } 'square name
    { 3 square print } 'square3 name)
NIL
```

В скомпилированной форт нити, все символы и другая информация, относящаяся к слову будут удалены. Всё что у нас есть — это кусок списковой структуры, извлечённой из форт словаря **my-forth**:

```
* (print-forth-thread my-forth 'square3)

(3
 (#<FUNCTION :LAMBDA NIL (PUSH (CAR PSTACK) PSTACK)
```

```
     (SETF PC (CDR PC))>
     #<FUNCTION :LAMBDA NIL
     (PUSH (* (POP PSTACK)
      (POP PSTACK))
   PSTACK) (SETF PC (CDR PC))>)
 #<FUNCTION :LAMBDA NIL (PRINT (POP PSTACK))
 (SETF PC (CDR PC))>)
T
```

Вышеприведённый код не имеет циклов и поэтому является чистой Флаб программой. Как было сказано ранее, почти все интересные программы содержат циклы. Для создания условий и циклов мы можем использовать изолированный форт примитив **branch-if** который позволяет нам изменять абстрактный регистр **pc** и сослаться с помощью некоторого значения в последующей ячейке на некоторое место в исполняемой форт нити. Также мы в состоянии реализовать хвостовые вызовы через прямой доступ к стеку возврата с помощью $>$**r** и **r**$>$. В отличие от большинства других языков, мы можем непосредственно модифицировать вызовы в хвостовые вызовы — даже если слово было вызвано извне.

Но, похоже что мы упустили конструкцию, занимающую одно из центральных мест в лиспе: рекурсия. Может ли слово вызвать само себя? Мы увидели как мы можем использовать **branch-if** для прыжка назад в начало слова — хвостовую рекурсию. Однако, нам по-настоящему хотелось бы получить слово, способное вызвать себя через обычный механизм нити. Для того чтобы сделать это, нужно сохранить начало нити как ячейку в нашей нити, так текущее местоположение будет сохранено в стеке возврата и нужно установить **pc** в начало слова. До сих пор ни одно из наших слов не было способно использовать *полную рекурсию*, однако, поскольку мы не называли слово до тех пор, пока не скомпилировали его — оно недоступно для нас пока мы ищем его в словаре, пытаясь скомпилировать рекурсивный вызов. К счастью, есть простой трюк с помощью которого мы можем обойти это ограничение. Мы можем просто выйти из режима компиляции и назвать компилируемое слово до того, как мы скомпилируем рекурсивный вызов. Вот пример определения полностью рекурсивного вычисления факториала:

```
* (go-forth (setq my-forth (new-forth))
    { [ 'fact name ]
    dup 1 -
    dup 1 > if fact then
```

```
    * })
NIL
```

Убедимся, что **(fact 5)** равно 120:

```
* (go-forth my-forth 5 fact print)

120
NIL
```

Упражнение. Некоторые реализации форта используют слово **recurse**, которое просто ищет начало нити слова, компилируемого в данный момент, и вставляет ее в компилируемую нить. Это называется *анонимной рекурсией*. Напишите незамедлительное слово, которое делает это как альтернативу *именованной рекурсии*.

Нить **fact** более сложна чем нить **square3**, описанная выше. Она содержит код, ссылающийся на себя:

```
* (print-forth-thread my-forth 'fact)

#1=(#2=#<FUNCTION :LAMBDA NIL (PUSH (CAR PSTACK) PSTACK)
  (SETF PC (CDR PC))> 1
       #<FUNCTION :LAMBDA NIL (LET ((#:TOP5608
      (POP PSTACK)))
   (PUSH (- (POP PSTACK)
    #:TOP5608) PSTACK))
       (SETF PC (CDR PC))> #2# 1
       #<FUNCTION :LAMBDA NIL (LET ((#:TOP5608
      (POP PSTACK)))
   (PUSH (> (POP PSTACK)
    #:TOP5608)
 PSTACK))
       (SETF PC (CDR PC))>
       #<FUNCTION :LAMBDA NIL (PUSH (NOT (POP PSTACK))
    PSTACK)
       (SETF PC (CDR PC))>
       #<FUNCTION :LAMBDA NIL (SETF PC (IF (POP PSTACK)
   (CADR PC)
   (CDDR PC)))>
       #3=(#<FUNCTION :LAMBDA NIL (SETF PC (CDR PC))>
           #<FUNCTION :LAMBDA NIL (PUSH (* (POP PSTACK)
```

```
   (POP PSTACK))
PSTACK)
   (SETF PC (CDR PC))>) #1# .
    #3#)
T
```

В вышеприведенном примере `#2#` ссылается на примитив **dup** и был скомпилирован дважды. `#1#` ссылается на нить **fact**, реализуя рекурсию.

Эти структуры очень похожи на списковые структуры лиспа, которые мы использовали для написания лисп программ, не так ли? Поскольку мы понимаем абстрактную машину, которая будет исполнять эти нити, то мы можем, с помощью нескольких коротко описанных ограничений, скомпилировать эти форт нити обратно в списковую структуру лиспа, которая может быть вставлена в выражение с помощью макроса и скомпилирована с помощью нашего лисп компилятора. Этот процесс известен как *флабирование (flubifying)* кода, поскольку мы конвертируем скомпилированную программу из однообразной, программируемой структуры данных (нить) в непрозрачный, недоступный блок кода (скомпилированную COMMON LISP функцию).

Конечно, есть большая разница между нитями форта и лисповскими списковыми структурами, которые мы можем вычислить или вставить в макросы. Во-первых, примитивы форта — это простые указатели на функции (показанные здесь как `#<FUNCTION ...>`), но нам нужна лисповская списковая структура, с которыми были созданы эти функции. И наконец, пришло время разобраться с созданным нами абстрактным регистром **dtable**. **Dtable** — это хэш-таблица, которая даёт отображение с этих функций в создавшую их списковую структуру.

Большая разница между форт нитями и нашими лисп программами заключается в том, что форт нити предполагают возможность использования стека возврата — концепции, не существующей на самом деле в таких Флабах, как COMMON LISP. Мы хотим избавиться от зависимости кода в **inner-interpreter**-е и, вместо этого, позволить лисп компилятору обрабатывать это с помощью обычных лисповских контрольных структур, как вызов функции и формы **tagbody**/**go**.

Описание оставшейся части кода в этой главе отличается от описания другого кода книги в том, что реализация не будет освещаться в деталях, но упор будет сделан на высокоуровневую точку зрения. Причина в том, что механика реализации сложна, грязна и, если честно, не настолько интересна. Достаточно сказать, что я полагаю, что большинство лисп программистов напишут эту реализацию подобным образом.

Flubify-aux — это макрос, расширяющийся в функцию, получаю-

щую форт нить и конвертирующую её в кусок флаб лисп кода пользуясь тем фактом, что каждое не примитивное слово окружено **tagbody** и таким образом, можно воспользоваться gensym-ами в роли тегов для **goto**.

Assemble-flub активно используется в **flubify-aux** как аббревиатура, проверяющая хэш-таблицу **go-ht** на присутствие любых переходов обнаруженных в предыдущем проходе, которые ссылаются на место, компилируемое нами в текущий момент. Если это так, то добавляется ранее выбранный gensym тег для этой формы тела тега.

Flubify — это функция, использующая **flubify-aux**. Она выполняет первый проход, проверяет на инструкции **branch-if** и строит хэш-таблицу **go-ht**. Также рекурсивно выполняется флабирование всех нитей, присоединённых к нашей текущей нити. На деле **flubify** может быть *двойным рекурсивным* — вы просто не видите этого пока не расширите использование **flubify-aux**. Вы не видите этого, но лисп может. Если ссылочная прозрачность — это прозрачный лист стекла, то здесь мы смотрим в дом зеркал.

Compile-flubified получает хэш-таблицу, построенную с помощью **flubify** и конвертирует её в форму, использующую **labels** для привязки каждой из этих флабированных нитей в функцию, именованную с помощью gensym-а в пространстве имён функций. Внутри этой области видимости, его расширение выполняет **funcall**-ы исходной нити (также флабированной).

Flubify-thread-shaker — это функция, которая в действительности обходит нашу форт нить. Она рекурсивно перетряхивает все соединенные форт нити. Это значит, что она изолирует только соответствующую структуру форт нити, необходимую для выполнения данной нити с помощью утилиты **get-forth-thread**, затем переводит каждую функцию в соответствующий лисп код, пропуская **if-branch**, и создает ошибку, когда обнаруживает **compile**.

Forth-to-lisp — окончательная функция, которую подготовили предыдущие макросы и функции этой главы. Она принимает среду форта, созданную **new-forth**, ищет нить, на которую указывает символ, переданный как **word**, затем возвращает соответствующий лисп код, чтобы выполнить эту нить. Сначала она перетряхивает нить (рекурсивно перебирая все связанные нити), затем применяет процедуру флабификации. И наконец, она оборачивает маленькое количество лисп кода, реализующего внутренний интерпретатор с помощью обычных управляющих структур лиспа.

Для демонстрации вспомним форт слова **square** и **square3**, определённые нами ранее. И опять, вот как они были определены в среде форта **my-forth**:

Листинг 8.32: FLUBIFY-AUX

```
(defmacro flubify-aux ()
  `(alambda (c)
            (if c
                (cond
                  ((gethash (car c) prim-ht)
                   (assemble-flub
                    `(funcall
                      ,(gethash (car c) prim-ht))
                    (self (cdr c))))
                  ((gethash (car c) thread-ht)
                   (assemble-flub
                    `(funcall #',(car (gethash (car c)
                                               thread-ht)))
                    (self (cdr c))))
                  ((eq (car c) branch-if)
                   (assemble-flub
                    `(if (pop pstack)
                         (go ,(gethash (cadr c) go-ht)))
                    (self (cddr c))))
                  ((consp (car c))
                   (flubify forth (car c) prim-ht
                            thread-ht branch-if)
                   (self c))
                  (t
                   (assemble-flub
                    `(push ',(car c) pstack)
                    (self (cdr c))))))))
```

Листинг 8.33: ASSEMBLE-FLUB

```
(defmacro assemble-flub (form rest)
  `(if (gethash c go-ht)
       (list* (gethash c go-ht)
              ,form
              ,rest)
       (list* ,form
              ,rest)))
```

Листинг 8.34: FLUBIFY

```
(defun flubify (forth thread prim-ht
                thread-ht branch-if)
  (unless #1=(gethash thread thread-ht)
          (setf #1# (list (gensym)))
          (let ((go-ht (make-hash-table)))
            (funcall
             (alambda (c)
                      (when c
                        (cond
                          ((eq (car c) branch-if)
                           (setf (gethash (cadr c) go-ht)
                                 (gensym))
                           (self (cddr c)))
                          ((consp (car c))
                           (flubify forth thread prim-ht
                                    thread-ht branch-if)))
                        (self (cdr c))))
             thread)
            (setf #1# (nconc #1# (funcall
                                  (flubify-aux)
                                  thread))))))
```

Листинг 8.35: COMPILE-FLUBIFIED

```
(defun compile-flubified (thread thread-ht)
  `(labels (,@(let (collect)
                (maphash
                 (lambda (k v)
                   (declare (ignore k))
                   (push
                    `(,(car v) ()
                       (tagbody ,@(cdr v)))
                    collect))
                 thread-ht)
                (nreverse collect)))
     (funcall #',(car (gethash thread thread-ht)))))
```

Листинг 8.36: FLUBIFY-THREAD-SHAKER

```
(defun flubify-thread-shaker
    (forth thread ht tmp-ht branch-if compile)
  (if (gethash thread tmp-ht)
      (return-from flubify-thread-shaker)
      (setf (gethash thread tmp-ht) t))
  (cond
    ((and (consp thread) (eq (car thread) branch-if))
     (if (cddr thread)
         (flubify-thread-shaker
          forth (cddr thread) ht
          tmp-ht branch-if compile)))
    ((and (consp thread) (eq (car thread) compile))
     (error "Can't convert compiling word to lisp"))
    ((consp thread)
     (flubify-thread-shaker
      forth (car thread) ht
      tmp-ht branch-if compile)
     (if (cdr thread)
         (flubify-thread-shaker
          forth (cdr thread) ht
          tmp-ht branch-if compile)))
    ((not (gethash thread ht))
     (if (functionp thread)
         (setf (gethash thread ht)
               (with-pandoric (dtable) forth
                              (gethash thread dtable)))))))
```

Листинг 8.37: FORTH-TO-LISP

```
(defun forth-to-lisp (forth word)
  (let ((thread (get-forth-thread forth word))
        (shaker-ht (make-hash-table))
        (prim-ht (make-hash-table))
        (thread-ht (make-hash-table))
        (branch-if (get-forth-thread forth 'branch-if))
        (compile (get-forth-thread forth 'compile)))
    (flubify-thread-shaker
     forth thread shaker-ht
     (make-hash-table) branch-if compile)
    (maphash (lambda (k v)
               (declare (ignore v))
               (setf (gethash k prim-ht) (gensym)))
             shaker-ht)
    (flubify forth thread prim-ht thread-ht branch-if)
    `(let (pstack)
       (let (,@(let (collect)
                 (maphash
                  (lambda (k v)
                    (push `(,(gethash k prim-ht)
                            (lambda () ,@(butlast v)))
                          collect))
                  shaker-ht)
                 (nreverse collect)))
         ,(compile-flubified
           thread thread-ht)))))
```

```
* (go-forth my-forth
    { dup * } 'square name
    { 3 square print } 'square3 name)
NIL
```

Здесь мы конвертируем нить **square3** в лисп:

```
* (forth-to-lisp my-forth 'square3)
(LET (PSTACK)
 (LET
  ((#:G7403 (LAMBDA NIL (PRINT (POP PSTACK))))
   (#:G7404
    (LAMBDA NIL (PUSH (* (POP PSTACK)
 (POP PSTACK)) PSTACK)))
   (#:G7405 (LAMBDA NIL (PUSH (CAR PSTACK) PSTACK))))
  (LABELS
   ((#:G7407 NIL (TAGBODY (FUNCALL #:G7405)
  (FUNCALL #:G7404)))
    (#:G7406 NIL (TAGBODY (PUSH '3 PSTACK)
  (FUNCALL #'#:G7407)
  (FUNCALL #:G7403))))
   (FUNCALL #'#:G7406))))
```

Достаточно очевидно, что вышеприведённая программа — это исполняемый лисп код. Если он нам понадобится, то мы можем скомпилировать его где-нибудь используя макрос. Или мы можем просто за-**eval**-ить его:

```
* (eval *)

9
NIL
```

Для того, чтобы увидеть как флаббируется форт нить с ветвлением и рекурсией посмотрим на кусок скомпилированного лисп кода с форт слова **fact**:

```
* (forth-to-lisp my-forth 'fact)
  (LABELS
   ((#:G7429 NIL                    ; fact
     (TAGBODY
      (FUNCALL #:G7428)     ; dup
```

```
    (PUSH '1 PSTACK)
    (FUNCALL #:G7427)     ; -
    (FUNCALL #:G7428)     ; dup
    (PUSH '1 PSTACK)
    (FUNCALL #:G7426)     ; >
    (FUNCALL #:G7425)     ; not
    (IF (POP PSTACK) (GO #:G7430))
    (FUNCALL #'#:G7429)   ; fact
    #:G7430
    (FUNCALL #:G7424)     ; nop
    (FUNCALL #:G7423))))  ; *
 (FUNCALL #'#:G7429))             ; fact
```

Итак, мы написали эту программу с помощью форта, но сейчас эта программа на лиспе. Мы использовали незамедлительные форт слова **if** и **then** для компиляции управляющей структуры, работающей с условиями и контролирующей где имеет, а где не имеет место рекурсия. В случае со стеком возврата лисп будет реализовывать для нас эту рекурсию используя традиционную инфраструктуру вызова функции.

Проводя тесты с помощью **eval** имейте в виду, что слово **fact** подразумевает наличие значения в стеке, а мы начинаем работу со свежесозданного стека. Для тестирования этого слова мы должны создать обёртку, добавляющую значение в стек. Например:

```
* (go-forth my-forth
    { 5 fact print } 'fact5 name)
NIL
```

Которую мы можем выполнить:

```
* (eval (forth-to-lisp my-forth 'fact5))

120
NIL
```

Как было замечено, из-за отличий между фортом и лиспом наш **forth-to-lisp** компилятор имеет определенные ограничения. Например, так как мы не можем обеспечить доступ к стеку возврата, то запрещены любые форт слова, использующие **r>** и **>r**. К ним относится **exit**, поэтому наше ранее определенное слово **countdown** не будет работать. Но будет работать **countdown-for-teh-hax0rz**, так как оно не использует **exit**. По причине того, что лисп программы не могут получить доступ к их стекам

возврата, не все из фортовых управляющих структур могут быть реализованы во флаб языках типа COMMON LISP. Упражнение: Добавьте **exit** как слово специального случая, использующее лисп блокировки для возврата из слова.

Другое ограничение заключается в том, что лисп код не может компилировать не-флаб код, поэтому мы не можем транслировать форт слова использующие **compile**, например, **if**, **then**, **begin** и **again**. Конечно, следует помнить, что форт нити созданные с помощью этих слов по прежнему могут быть использованы для компиляции слов, пример: вышеприведённый **fact**. И последнее ограничение, в форте **branch-if** может перепрыгнуть к любой нити, даже если она была создана внутри слова отличающегося от исполняемого в данный момент. В лиспе мы можем использовать **go** для перехода в другое место только в пределах нашего **tagbody**. Форт допускает не локальные ветвления, но, в целом, не локальные ветвления не допустимы в таких Флабах как COMMON LISP.

Секундочку. Разве мы не обошли все эти Флаб ограничения ранее, когда программировали в нашей форт среде? Да. Макросы позволяют нам конвертировать программы с лиспа и на лисп. Благодаря макросам всё возможно запрограммировать в лиспе.

"Но я хочу взбитые!"
Картинка из книги Лео Броди Thinking Forth.
Используется с разрешения автора.

Let Over Lambda - это одна из наиболее трудных книг, посвящённых программированию на компьютере. Повествование начинается с основ, и описываются наиболее мощные особенности наиболее мощного языка: COMMON LISP. Только верхний персентиль программистов использует лисп и если вы можете понять эту книгу, то это значит что вы находитесь в этом персентиле лисп программистов.

Эта книга о макросах – программах, которые создают другие программы. Макросы – это та деталь, которая делает Lisp величайшим языком программирования на свете. При правильном использовании макросы позволяют достичь невиданных доселе уровней абстракции, продуктивности программирования, эффективности кода и безопасности. Задачи, решаемые с помощью макросов, будут решаться проще, чем в других языках.

Если вам нужно просто руководство по программированию, в котором излагаются общие техники, применимые для любых языков – то эта книга не для вас. Эта книга раздвигает границы того, что мы знаем о программировании. Кроме обучения работе с макросами, которые могут вам помочь решить ваши задачи здесь и сейчас, эта книга построена так, чтобы быть интересной и вдохновляющей. Если вы вообще задумывались о том, что представляет из себя Lisp или даже программирование – то эта книга для вас.

```
(defmacro! sortf (comparator &rest places)
  (if places
      `(tagbody
        ,@(mapcar
           #`(let ((,g!a #1=,(nth (car a1) places))
                   (,g!b #2=,(nth (cadr a1) places)))
               (if (,comparator ,g!b ,g!a)
                   (setf #1# ,g!b
                         #2# ,g!a)))
           (build-batcher-sn (length places))))))
```

ISBN 978-1-4357-1275-1
90000
9 781435 712751