HIGH TECH

ПОЛ ГРЭМ

ANSI Common LISP

СИМВОЛ®

ANSI Common Lisp

Paul Graham

PRENTICE HALL

HIGH TECH

ANSI Common Lisp

Пол Грэм

Санкт-Петербург — Москва
2012

Серия «High tech»

Пол Грэм

ANSI Common Lisp

Перевод И. Хохлов

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*В. Демкин*
Редактор	*А. Родин*
Корректор	*С. Беляева*
Верстка	*Д. Орлова*

Грэм П.

ANSI Common Lisp. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 448 с., ил.

ISBN 978-5-93286-206-3

Книга «ANSI Common Lisp» сочетает в себе введение в программирование на Лиспе и актуальный справочный материал по ANSI-стандарту языка. Новички найдут в ней примеры интересных программ с их тщательным объяснением. Профессиональные разработчики оценят всесторонний практический подход.

Автор постарался показать уникальные особенности, которые выделяют Лисп из множества других языков программирования, а также предоставляемые им новые возможности, например макросы, которые позволяют разработчику писать программы, которые будут писать другие программы. Лисп – единственный язык, который позволяет с легкостью осуществлять это, потому что только он предлагает необходимые для этого абстракции.

Книга содержит: детальное рассмотрение объектно-ориентированного программирования – не только описание CLOS, но и пример собственного встроенного объектно-ориентированного языка; более 20 самостоятельных примеров, в том числе трассировщик лучей, генератор случайного текста, сопоставление с образцом, логический вывод, программа для генерации HTML, алгоритмы поиска и сортировки, файлового ввода-вывода, сжатия данных, а также вычислительные задачи. Особое внимание уделяется критически важным концепциям, включая префиксный синтаксис, связь кода и данных, рекурсию, функциональное программирование, типизацию, неявное использование указателей, динамическое выделение памяти, замыкания, макросы, предшествование классов, суть методов обобщенных функций и передачи сообщений. Вы найдете полноценное руководство по оптимизации, примеры различных стилей программирования, включая быстрое прототипирование, разработку снизу-вверх, объектно-ориентированное программирование и применение встраиваемых языков.

ISBN 978-5-93286-206-3
ISBN 0-13-370875-6 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 03.10.2012. Формат 70×100 $^{1}/_{16}$.
Печать офсетная. Объем 28 печ. л.

Оглавление

Half lost on my firmness gains to more glad heart,
Or violent and from forage drives
A glimmering of all sun new begun
Both harp thy discourse they march'd,
Forth my early, is not without delay;
For their soft with whirlwind; and balm.
Undoubtedly he scornful turn'd round ninefold,
Though doubled now what redounds,
And chains these a lower world devote, yet inflicted?
Till body or rare, and best things else enjoy'd in heav'n
To stand divided light at ev'n and poise their eyes,
Or nourish, lik'ning spiritual, I have thou appear. [1]

Henley

Я потерял итог моих трудов, что сердце грели мне,
И было то ожесточенье сердца иль движение вперед,
Но солнце снова озарило нас,
И арфа вновь твоя заговорила,
Вперед, как можно раньше и без промедленья;
Тот ураган для них стал мягок, как бальзам.
Он тень сомнения отверг и обвился вокруг с насмешкой девять раз,
Наперекор тому, что дважды отразилось эхом,
И цепи эти уходили к преисподней, разве не ужасно?
Покуда тело дивное испытывало райские услады
Стоять мне, рассеченным светом, и взглядом воспарить,
Иль вскормленный, подобно духам, я появился пред тобою.

Henley

[1] Эта строфа написана программой Henley на основе поэмы Джона Мильтона «Потерянный рай». Можете попытаться найти тут смысл, но лучше загляните в главу 8. – *Прим. перев.*

Предисловие

Цель данной книги – быстро и основательно научить вас языку Common Lisp. Книга состоит из двух частей: в первой части на множестве примеров объясняются основные концепции программирования на Common Lisp, вторая часть – это современное описание стандарта ANSI Common Lisp, содержащее каждый оператор языка.

Аудитория

Книга «ANSI Common Lisp» предназначена как для студентов, изучающих этот язык, так и для профессиональных программистов. Ее чтение не требует предварительного знания Лиспа. Опыт написания программ на других языках был бы, безусловно, полезен, но не обязателен. Повествование начинается с основных понятий, что позволяет уделить особое внимание моментам, которые обычно приводят в замешательство человека, впервые знакомящегося с Лиспом.

Эта книга может использоваться в качестве учебного пособия по Лиспу или как часть курсов, посвященных искусственному интеллекту или теории языков программирования. Профессиональные разработчики, желающие изучить Лисп, оценят простой, практический подход. Те, кто уже знаком с языком, найдут в книге множество полезных примеров и оценят ее как удобный справочник по стандарту ANSI Common Lisp.

Как пользоваться книгой

Лучший способ выучить Лисп – использовать его. Кроме того, намного интереснее изучать язык в процессе написания программ. Книга устроена так, чтобы читатель смог начать делать это как можно раньше. После небольшого введения в главе 2 объясняется все, что понадобится для создания первых Лисп-программ. В главах 3–9 рассматриваются ключевые элементы программирования на Лиспе. Особое внимание уделяется таким понятиям, как роль указателей в Лиспе, использование рекурсии и значимость функций как полноценных объектов языка.

Следующие материалы предназначены для читателей, которые хотят более основательно разобраться с техникой программирования на Лиспе. Главы 10–14 охватывают макросы, CLOS (объектная система Common

Lisp), операции со списками, оптимизацию, а также более сложные темы, такие как пакеты и макросы чтения. Главы 15–17 подводят итог предыдущих глав на трех примерах реальных приложений: программа для создания логических интерфейсов, HTML-генератор и встроенный объектно-ориентированный язык программирования.

Последняя часть книги состоит из четырех приложений, которые будут полезны всем читателям. Приложения A–D включают руководство по отладке, исходные коды для 58 операторов языка, описание основных отличий ANSI Common Lisp от предыдущих версий языка˚ и справочник по каждому оператору в ANSI Common Lisp.

Книга завершается комментариями, содержащими пояснения, ссылки, дополнительный код и прочие отступления. Наличие комментария помечается в основном тексте маленьким кружочком: ˚.

Код

Несмотря на то что книга посвящена ANSI Common Lisp, по ней можно изучать любую разновидность Common Lisp. Примеры, демонстрирующие новые возможности, обычно сопровождаются комментариями, поясняющими, как они могут быть адаптированы к более ранним реализациям.

Весь код из книги, ссылки на свободный софт, исторические документы, часто задаваемые вопросы и множество других ресурсов доступны по адресу:

http://www.eecs.harvard.edu/onlisp/

Анонимный доступ к коду можно получить по ftp:

ftp://ftp.eecs.harvard.edu:/pub/onlisp/

Вопросы и комментарии присылайте на *pg@eecs.harvard.edu*.

«On Lisp»

В этой книге я постарался показать уникальные особенности, которые выделяют Лисп из множества языков программирования, а также предоставляемые им новые возможности. Например, макросы – они позволяют разработчику писать программы, которые будут писать другие программы. Лисп – единственный язык, который позволяет с легкостью осуществлять это, потому что только он предлагает необходимые для этого абстракции. Читателям, которым интересно узнать больше о макросах и других интересных возможностях языка, я предлагаю познакомиться с книгой «On Lisp»[1], которая является продолжением данного издания.

[1] Paul Graham «On Lisp», Prentice Hall, 1993. – *Прим. перев.*

Благодарности

Из всех друзей, участвовавших в работе над книгой, наибольшую благодарность я выражаю Роберту Моррису, который на протяжении всего времени помогал совершенствовать это издание. Некоторые примеры, включая Henley (стр. 149) и сопоставление с образцом (стр. 254), взяты из написанного им кода.

Я был счастлив работать с первоклассной командой технических рецензентов: Сконом Бриттаном (Skona Brittain), Джоном Фодераро (John Foderaro), Ником Левином (Nick Levine), Питером Норвигом (Peter Norvig) и Дэйвом Турецки (Dave Touretzky). Вряд ли найдется хотя бы одна страница, к улучшению которой они не приложили руку. Джон Фодераро даже переписал часть кода к разделу 5.7.

Нашлись люди, которые согласились прочитать рукопись целиком или частично, включая Кена Андерсона (Ken Anderson), Тома Читама (Tom Cheatham), Ричарда Фейтмана (Richard Fateman), Стива Хайна (Steve Hain), Барри Марголина (Barry Margolin), Уолдо Пачеко (Waldo Pacheco), Вилера Румла (Wheeler Ruml) и Стюарта Рассела (Stuart Russel). Кен Андерсон и Вилер Румл сделали множество полезных замечаний.

Я благодарен профессору Читаму и Гарварду в целом за предоставление необходимых для написания книги возможностей. Также благодарю сотрудников лаборатории Айкен: Тони Хэртмана (Tony Hartman), Дейва Мазиерса (Dave Mazieres), Януша Джуду (Janusz Juda), Гарри Бохнера (Harry Bochner) и Джоанну Клис (Joanna Klys).

Я рад, что у меня снова появилась возможность поработать с сотрудниками Prentice Hall: Аланом Аптом (Alan Apt), Моной Помпили (Mona Pompili), Ширли МакГир (Shirley McGuire) и Ширли Майклс (Shirley Michaels). Обложка книги выполнена превосходным мастером Джино Ли (Gino Lee) из Bow & Arrow Press, Кэмбридж.

Эта книга была набрана с помощью LaTeX, языка, созданного Лесли Лэмпортом (Leslie Lamport) поверх TeX'а Дональда Кнута (Donald Knuth), с использованием дополнительных макросов авторства Л.А. Карра (L.A. Carr), Вана Якобсона (Van Jacobson) и Гая Стила (Guy Steele). Чертежи были выполнены с помощью программы Idraw, созданной Джоном Влиссидсом (John Vlissids) и Скоттом Стантоном (Skott Stanton). Предпросмотр всей книги был сделан с помощью программы Ghostview, написанной Тимом Тейзеном (Tim Theisen) на основе интерпретатора Ghostscript, созданного Л. Питером Дойчем (L. Peter Deutch).

Я должен поблагодарить и многих других людей: Генри Бейкера (Henry Baker), Кима Барретта (Kim Barrett), Ингрид Бассет (Ingrid Basset), Тревора Блеквелла (Trevor Blackwell), Пола Беккера (Paul Becker), Гарри Бисби (Gary Bisbee), Франка Душмана (Frank Dueschmann), Франсиса Дикли (Frances Dickey), Рича и Скотта Дрейвса (Rich and Scott Draves), Билла Дабак (Bill Dubuque), Дэна Фридмана (Dan Friedman),

Дженни Грэм (Jenny Graham), Эллис Хартли (Alice Hartley), Дэвида Хендлера (David Hendler), Майка Хьюлетта (Mike Hewlett), Гленна Холловота (Glenn Hollowat), Брэда Карпа (Brad Karp), Соню Кини (Sonya Keene), Росса Найтс (Ross Knights), Митсуми Комуро (Mutsumi Komuro), Стефи Кутзиа (Steffi Kutzia), Дэвида Кузника (David Kuznick), Мэди Лорд (Madi Lord), Джулию Маллози (Julie Mallozzi), Пола МакНами (Paul McNamee), Дэйва Муна (Dave Moon), Говарда Миллингса (Howard Mullings), Марка Ницберга (Mark Nitzberg), Ненси Пармет (Nancy Parmet) и ее семью, Роберта Пенни (Robert Penny), Майка Плуча (Mike Plusch), Шерил Сэкс (Cheryl Sacks), Хейзэма Сейеда (Hazem Sayed), Шеннона Спайреса (Shannon Spires), Лоу Штейнберга (Lou Steinberg), Пола Стоддарда (Paul Stoddard), Джона Стоуна (John Stone), Гая Стила (Guy Steele), Стива Страссмана (Steve Strassmann), Джима Вейча (Jim Veitch), Дэйва Уоткинса (Dave Watkins), Айдела и Джулианну Вебер (Idelle and Julian Weber), Вейкерсов (the Weickers), Дэйва Йоста (Dave Yost) и Алана Юлли (Alan Yuille).

Но больше всего я благодарен моим родителям и Джекки.

Дональд Кнут назвал свою известную серию книг «Искусство программирования». В прочитанной им Тьюринговской лекции он объяснил, что это название было выбрано не случайно, так как в программирование его привела как раз «возможность писать красивые программы».

Так же как и архитектура, программирование сочетает в себе искусство и науку. Программа создается на основе математических принципов, так же как здание держится согласно законам физики. Но задача архитектора не просто построить здание, которое не разрушится. Почти всегда он стремится создать нечто прекрасное.

Как и Дональд Кнут, многие программисты чувствуют, что такова истинная цель программирования. Так считают почти все Лисп-хакеры. Саму суть лисп-хакерства можно выразить двумя фразами. Программирование должно доставлять радость. Программы должны быть красивыми. Таковы идеи, которые я постарался пронести через всю книгу.

Пол Грэм

Предисловие к русскому изданию

Книга, перевод которой вы держите в руках, была издана в 1996 году, а написана и того раньше. К моменту подготовки перевода прошло 15 лет. Для такой стремительно развивающейся отрасли, как программирование, это огромный срок, за который изменили облик не только парадигмы и языки программирования, но и сама вычислительная техника.

Несмотря на это, данная книга и на настоящий момент представляет большую практическую ценность. Она соответствует стандарту языка, который не менялся с момента ее написания и, похоже, не будет меняться в течение ощутимого времени. Кроме того, в книге описаны модели и методы, пришедшие в программирование из Лиспа и в той или иной мере актуальные в современном программировании.

Автор не раз упоминает о том, что Лисп, несмотря на его долгую историю, не теряет актуальности. Теперь, когда с момента издания оригинала книги прошло 15 лет, а с момента создания языка Лисп более полувека, мы отчетливо видим подтверждение слов автора, наблюдая постоянный рост интереса к языку.

Тем не менее некоторые моменты в книге являются слегка устаревшими и требуют дополнительных комментариев.

В числе уникальных особенностей Лиспа Грэм выделяет интерактивность, автоматическое управление памятью, динамическую типизацию и замыкания. На момент написания книги Лисп конкурировал с такими языками, как C, C++, Паскаль, Фортран (на протяжении книги автор сравнивает Лисп именно с ними). Эти языки «старой закалки» действительно представляют полную противоположность Лиспу. На настоящий момент разработано множество языков, в которых в той или иной степени заимствованы преимущества Лиспа. Таким, например, является Perl, который вытесняется более продвинутым языком Python, а последний, несмотря на популярность, сам испытывает конкуренцию со стороны языка Ruby, известного как «Лисп с человеческим синтаксисом». Такие языки благодаря гибкости быстро находят свою нишу, оставаясь при этом средствами общего назначения. Так, Perl прочно занял нишу скриптового языка в Unix-подобных системах. Однако механизм макросов, лежащий в основе Лиспа, пока не был заимствован ни одним из языков, так как прочно связан с его синтаксисом. Кроме того, Лисп выгодно отличается от своих «последователей». Согласитесь, искусственное

добавление возможностей в язык с уже существующей структурой и идеологией существенно отличается от случая, когда язык изначально разрабатывался с учетом данных возможностей.

Время коснулось также и ряда идей и моделей, упомянутых в данной книге. Несколько странными могут показаться восторженные упоминания об объектно-ориентированном программировании. Прошло немало времени, и сегодня ООП уже больше не вызывает подобный восторг.

Многое изменилось и в мире реализаций Common Lisp. Автор сознательно не упоминает названия реализаций, так как их жизненный срок не определен. Многих реализаций языка уже нет в живых, но на их место пришли новые. Необходимо отметить, что сейчас имеется ряд блестящих реализаций Common Lisp, как коммерческих, так и свободных. Стандарт языка дает разработчикам довольно много свободы действий, и выпускаемые ими реализации как внешне, так и внутренне могут сильно отличаться друг от друга. Детали реализаций вас могут не волновать, а вот различия в их поведении могут смущать новичков. Когда речь заходит о взаимодействии с пользователем (например, о работе в отладчике), автор использует некий упрощенный унифицированный интерфейс, который он называет «гипотетической» реализацией. На деле, вам придется поэкспериментировать с выбранной реализацией, чтобы научиться эффективно ее использовать. Кроме того, сейчас имеется отличная среда разработки Slime[1], помимо прочего скрывающая разницу в поведении между реализациями.

Всеволод Демкин

Переводчик, Иван Хохлов, выражает благодарность Ивану Струкову, Сергею Катревичу и Ивану Чернецкому за предоставление ценных замечаний по переводу отдельных глав книги.

[1] Домашняя страница проекта – *http://common-lisp.net/project/slime/*.

1

Введение

Джон Маккарти со своими студентами начал работу над первой реализацией Лиспа в 1958 году. Не считая Фортрана, Лисп – это старейший из ныне используемых языков.° Но намного важнее то, что до сих пор он остается флагманом среди языков программирования. Специалисты, хорошо знающие Лисп, скажут вам, что в нем есть нечто, делающее его особенным по сравнению с остальными языками.

Отчасти его отличает изначально заложенная возможность развиваться. Лисп позволяет программисту определять новые операторы, и если появятся новые абстракции, которые приобретут популярность (например, объектно-ориентированное программирование), их всегда можно будет реализовать в Лиспе. Изменяясь как ДНК, такой язык никогда не выйдет из моды.

1.1. Новые инструменты

Зачем изучать Лисп? Потому что он позволяет делать то, чего не могут другие языки. Если вы захотите написать функцию, складывающую все числа, меньшие *n*, она будет очень похожа на аналогичную функцию в C:

```
; Lisp                    /* C */
(defun sum (n)            int sum(int n){
  (let ((s 0))              int i , s = 0;
    (dotimes (i n s)        for(i = 0; i < n; i++)
      (incf s i))))           s += i;
                            return(s);
                          }
```

Если вы хотите делать несложные вещи типа этой, в сущности, не имеет значения, какой язык использовать. Предположим, что теперь мы

хотим написать функцию, которая принимает число *n* и возвращает функцию, которая добавляет *n* к своему аргументу:

```
; Lisp
(defun addn (n)
  #'(lambda (x)
      (+ x n)))
```

Как функция `addn` будет выглядеть на C? Ее просто невозможно написать.

Вы, вероятно, спросите, зачем это может понадобиться? Языки программирования учат вас не хотеть того, что они не могут осуществить. Раз программисту приходится думать на том языке, который он использует, ему сложно представить то, чего он не может описать. Когда я впервые занялся программированием на Бейсике, я не огорчался отсутствием рекурсии, так как попросту не знал, что такая вещь имеет место. Я думал на Бейсике и мог представить себе только итеративные алгоритмы, так с чего бы мне было вообще задумываться о рекурсии?

Если вам не нужны лексические замыкания (а именно они были продемонстрированы в предыдущем примере), просто примите пока что на веру, что Лисп-программисты используют их постоянно. Сложно найти программу на Лиспе, написанную без использования замыканий. В разделе 6.7 вы научитесь пользоваться ими.

Замыкания – не единственные абстракции, которых нет в других языках. Другой, возможно даже более важной, особенностью Лиспа является то, что программы, написанные на нем, представляются в виде его же структур данных. Это означает, что вы можете писать программы, которые пишут программы. Действительно ли люди пользуются этим? Да, это и есть макросы, и опытные программисты используют их на каждом шагу. Вы узнаете, как создавать свои макросы, в главе 10.

С макросами, замыканиями и динамической типизацией Лисп превосходит объектно-ориентированное программирование. Если вы в полной мере поняли предыдущее предложение, то, вероятно, можете не читать эту книгу. Это важный момент, и вы найдете подтверждение тому в коде к главе 17.

Главы 2–13 поэтапно вводят все понятия, необходимые для понимания кода главы 17. Благодаря вашим стараниям вы будете ощущать программирование на C++ таким же удушающим, каким опытный программист C++ в свою очередь ощущает Бейсик. Сомнительная, на первый взгляд, награда. Но, быть может, вас вдохновит осознание природы этого дискомфорта. Бейсик неудобен по сравнению с C++, потому что опытный программист C++ знает приемы, которые невозможно осуществить в Бейсике. Точно так же изучение Лиспа даст вам больше, нежели добавление еще одного языка в копилку ваших знаний. Вы научитесь размышлять о программах по-новому, более эффективно.

1.2. Новые приемы

Итак, Лисп предоставляет такие инструменты, которых нет в других языках. Но это еще не все. Отдельные технологии, впервые появившиеся в Лиспе, такие как автоматическое управление памятью, динамическая типизация, замыкания и другие, значительно упрощают программирование. Взятые вместе, они создают критическую массу, которая рождает новый подход к программированию.

Лисп изначально гибок – он позволяет самостоятельно задавать новые операторы. Это возможно сделать, потому что сам Лисп состоит из таких же функций и макросов, как и ваши собственные программы. Поэтому расширить возможности Лиспа ничуть не сложнее, чем написать свою программу. На деле это так просто (и полезно), что расширение языка стало обычной практикой. Получается, что вы не только пишете программу в соответствии с языком, но и дополняете язык в соответствии с нуждами программы. Этот подход называется *снизу-вверх* (*bottom-up*).

Практически любая программа будет выигрывать, если используемый язык заточен под нее, и чем сложнее программа, тем большую значимость имеет подход «снизу-вверх». В такой программе может быть несколько слоев, каждый из которых служит чем-то вроде языка для описания вышележащего слоя. Одной из первых программ, написанных таким образом, был TEX. Вы имеете возможность писать программы снизу-вверх на любом языке, но на Лиспе это делать проще всего.

Написанные снизу-вверх программы легко расширяемы. Поскольку идея расширяемости лежит в основе Лиспа, это идеальный язык для написания расширяемых программ. В качестве примера приведу три программы, написанные в 80-х годах и использовавшие возможность расширения Лиспа: GNU Emacs, Autocad и Interleaf.

Кроме того, код, написанный данным методом, легко использовать многократно. Суть написания повторно используемого кода заключается в отделении общего от частного, а эта идея лежит в самой основе метода. Вместо того чтобы прилагать существенные усилия к созданию монолитного приложения, полезно потратить часть времени на построение своего языка, поверх которого затем реализовывать само приложение. Приложение будет находиться на вершине пирамиды языковых слоев и будет иметь наиболее специфичное применение, а сами слои можно будет приспособить к повторному использованию. Действительно, что может быть более пригодным для многократного применения, чем язык программирования?

Лисп позволяет не просто создавать более утонченные приложения, но и делать это быстрее. Практика показывает, что программы на Лиспе выглядят короче, чем аналоги на других языках. Как показал Фредерик Брукс, временные затраты на написание программы зависят в первую очередь от ее длины.° В случае Лиспа этот эффект усиливается его

динамическим характером, за счет которого сокращается время между редактированием, компиляцией и тестированием.

Мощные абстракции и интерактивность вносят коррективы и в принцип разработки приложений. Суть Лиспа можно выразить одной фразой – *быстрое прототипирование*. На Лиспе гораздо удобнее и быстрее написать прототип, чем составлять спецификацию. Более того, прототип служит проверкой предположения, является ли эффективным выбранный метод, а также гораздо ближе к готовой программе, чем спецификация.

Если вы еще не знаете Лисп достаточно хорошо, то это введение может показаться набором громких и, возможно, бессмысленных утверждений. Превзойти объектно-ориентированное программирование? Подстраивать язык под свои программы? Программировать на Лиспе в реальном времени? Что означают все эти утверждения? До тех пор пока вы не познакомитесь с Лиспом поближе, все эти слова будут звучать для вас несколько противоестественно, однако с опытом придет и понимание.

1.3. Новый подход

Одна из целей этой книги – не просто объяснить Лисп, но и продемонстрировать новый подход к программированию, который стал возможен благодаря этому языку. Подход, который вы чаще будете видеть в будущем. С ростом мощности сред разработки и увеличением абстрактности языков стиль программирования на Лиспе постепенно заменяет старую модель, в которой реализации предшествовало проектирование.

Согласно этой модели баги вообще не должны появляться. Программа, созданная по старательно разработанным заранее спецификациям, работает отлично. В теории звучит неплохо. К сожалению, эти спецификации разрабатываются и реализуются людьми, а люди не застрахованы от ошибок и каких-либо упущений. Кроме того, тяжело учесть все нюансы на стадии проектирования. В результате такой метод часто не срабатывает.

Руководитель проекта OS/360 Фредерик Брукс был хорошо знаком с традиционным подходом, а также с его результатами:

> Любой пользователь OS/360 вскоре начинал понимать, насколько лучше могла бы быть система. Более того, продукт не успевал за прогрессом, использовал памяти больше запланированного, стоимость его в несколько раз превосходила ожидаемую, и он не работал стабильно до тех пор, пока не было выпущено несколько релизов.°

Так он описывал систему, ставшую тогда одной из наиболее успешных.

Проблема в том, что такой подход не учитывает человеческий фактор. При использовании старой модели вы ожидаете, что спецификации не содержат серьезных огрехов и что остается лишь каким-то образом оттранслировать их в код. Опыт показывает, что это ожидание редко бы-

вает оправдано. Гораздо полезнее предполагать, что спецификация будет реализована с ошибками, а в коде будет полно багов.

Это как раз та идея, на которой построен новый подход. Вместо того чтобы надеяться на безукоризненную работу программистов, она пытается минимизировать стоимость допущенных ошибок. Стоимость ошибки – это время, необходимое на ее исправление. Благодаря мощным языкам и эффективным инструментам это время может быть существенно уменьшено. Кроме того, разработчик больше не удерживается в рамках спецификации, меньше зависит от планирования и может экспериментировать.

Планирование – это необходимое зло. Это ответ на риск: чем он больше, тем важнее планировать наперед. Мощные инструменты уменьшают риск, в результате уменьшается и необходимость планирования. Дизайн вашей программы может быть улучшен на основании информации из, возможно, самого ценного источника – опыта ее реализации.

Лисп развивался в этом направлении с 1960 года. На Лиспе вы можете писать прототипы настолько быстро, что успеете пройти несколько итераций проектирования и реализации раньше, чем закончили бы составлять спецификацию в старой модели. Все тонкости реализации будут осознаны в процессе создания программы, поэтому переживания о багах пока можно отложить в сторону. В силу функционального подхода к программированию многие баги имеют локальный характер. Некоторые баги (переполнения буфера, висящие указатели) просто невозможны, а остальные проще найти, потому что программа становится короче. И когда у вас есть интерактивная разработка, можно исправить их мгновенно, вместо того чтобы проходить через длинный цикл редактирования, компиляции и тестирования.

Такой подход появился не просто так, он действительно приносит результат. Как бы странно это ни звучало, чем меньше планировать разработку, тем стройнее получится программа. Забавно, но такая тенденция наблюдается не только в программировании. В средние века, до изобретения масляных красок, художники пользовались особым материалом – темперой, которая не могла быть перекрашена или осветлена. Стоимость ошибки была настолько велика, что художники боялись экспериментировать. Изобретение масляных красок породило множество течений и стилей в живописи. Масло «позволяет думать дважды».° Это дало решающее преимущество в работе со сложными сюжетами, такими как человеческая натура.

Введение в обиход масляных красок не просто облегчило жизнь художникам. Стали доступными новые, прогрессивные идеи. Янсон писал:

> Без масла покорение фламандскими мастерами визуальной реальности было бы крайне затруднительным. С технической точки зрения они являются прародителями современной живописи, потому что масло стало базовым инструментом художника с тех пор.°

Как материал темпера не менее красива, чем масло, но широта полета фантазии, которую обеспечивают масляные краски, является решающим фактором.

В программировании наблюдается похожая идея. Новая среда – это «объектно-ориентированный динамический язык программирования»; говоря одним словом, Лисп. Но это вовсе не означает, что через несколько лет все программисты перейдут на Лисп, ведь для перехода к масляным краскам тоже потребовалось существенное время. Кроме того, по разным соображениям темпера используется и по сей день, а порой ее комбинируют с маслом. Лисп в настоящее время используется в университетах, исследовательских лабораториях, некоторых компаниях, лидирующих в области софт-индустрии. А идеи, которые легли в основу Лиспа (например, интерактивность, сборка мусора, динамическая типизация), все больше и больше заимствуются в популярных языках.

Более мощные инструменты убирают риск из исследования. Это хорошая новость для программистов, поскольку она означает, что можно будет браться за более амбициозные проекты. Изобретение масляных красок, без сомнения, имело такой же эффект, поэтому период сразу после их внедрения стал золотым веком живописи. Уже есть признаки того, что подобное происходит и в программировании.

2

Добро пожаловать в Лисп

Цель этой главы – помочь вам начать программировать как можно скорее. Прочитав ее, вы узнаете достаточно о языке Common Lisp, чтобы начать писать программы.

2.1. Форма

Лисп – интерактивный язык, поэтому наилучший способ его изучения – в процессе использования. Любая Лисп-система имеет интерактивный интерфейс, так называемый *верхний уровень* (*toplevel*). Программист набирает выражения в *toplevel*, а система показывает их значения.

Чтобы сообщить, что система ожидает новые выражения, она выводит приглашение. Часто в качестве такого приглашения используется символ >. Мы тоже будем пользоваться им.

Одними из наиболее простых выражений в Лиспе являются целые числа. Если ввести 1 после приглашения,

```
> 1
1
>
```

система напечатает его значение, после чего выведет очередное приглашение, ожидая новых выражений.

В данном случае введенное выражение выглядит так же, как и полученное значение. Такие выражения называют самовычисляемыми. Числа (например, 1) – самовычисляемые объекты. Давайте посмотрим на более интересные выражения, вычисление которых требует совершения некоторых действий. Например, если мы хотим сложить два числа, то напишем

```
> (+ 2 3)
5
```

В выражении (+ 2 3) знак + – это *оператор*, а числа 2 и 3 – его *аргументы*.

В повседневной жизни вы бы написали это выражение как 2+3, но в Лиспе мы сначала ставим оператор +, следом за ним располагаем аргументы, а все выражение заключаем в скобки: (+ 2 3). Такую структуру принято называть *префиксной* нотацией, так как первым располагается оператор. Поначалу этот способ записи может показаться довольно странным, но на самом деле именно такому способу записи выражений Лисп обязан своими возможностями.

Например, если мы хотим сложить три числа, в обычной форме записи нам придется воспользоваться плюсом дважды:

```
2 + 3 + 4
```

в то время как в Лиспе вы всего лишь добавляете еще один аргумент:

```
(+ 2 3 4)
```

В обычной нотации[1] оператор + имеет два аргумента, один перед ним и один после. Префиксная запись дает большую гибкость, позволяя оператору иметь любое количество аргументов или вообще ни одного:

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 4)
9
> (+ 2 3 4 5)
14
```

Поскольку у оператора может быть произвольное число аргументов, нужен способ показать, где начинается и заканчивается выражение. Для этого используются скобки.

Выражения могут быть вложенными:

```
> (/ (- 7 1) (- 4 2))
3
```

Здесь мы делим разность 7 и 1 на разность 4 и 2.

Другая особенность префиксной нотации: все выражения в Лиспе – либо *атомы* (например, 1), либо *списки*, состоящие из произвольного количества выражений. Приведем допустимые Лисп-выражения:

```
2       (+ 2 3)       (+ 2 3 4)       (/ (- 7 1) (- 4 2))
```

[1] Ее также называют *инфиксной*. – *Прим. перев.*

В дальнейшем вы увидите, что весь код в Лиспе имеет такой вид. Языки типа C имеют более сложный синтаксис: арифметические выражения имеют инфиксную запись, вызовы функций записываются с помощью разновидности префиксной нотации, их аргументы разделяются запятыми, выражения отделяются друг от друга с помощью точки с запятой, а блоки кода выделяются фигурными скобками. В Лиспе же для выражения всех этих идей используется единая нотация.

2.2. Вычисление

В предыдущем разделе мы набирали выражения в *toplevel*, а Лисп выводил их значения. В этом разделе мы поближе познакомимся с процессом вычисления выражений.

В Лиспе + – это функция, а выражение вида `(+ 2 3)` – это вызов функции. Результат вызова функции вычисляется в 2 шага:

1. Сначала вычисляются аргументы, слева направо. В нашем примере все аргументы самовычисляемые, поэтому их значениями являются `2` и `3`.
2. Затем аргументы применяются к функции, задаваемой оператором. В нашем случае это оператор сложения, который возвращает `5`.

Аргумент может быть не только самовычисляемым объектом, но и другим вызовом функции. В этом случае он вычисляется по тем же правилам. Посмотрим, что происходит при вычислении выражения `(/ (- 7 1) (- 4 2))`:

1. Вычисляется `(- 7 1)`: `7` вычисляется в `7`, `1` – в `1`. Эти аргументы передаются функции `-`, которая возвращает `6`.
2. Вычисляется `(- 4 2)`: `4` вычисляется в `4`, `2` – в `2`, функция `-` применяется к этим аргументам и возвращает `2`.
3. Значения `6` и `2` передаются функции `/`, которая возвращает `3`.

Большинство операторов в Common Lisp – это функции, но не все. Вызовы функций всегда обрабатываются подобным образом. Аргументы вычисляются слева направо и затем передаются функции, которая возвращает значение всего выражения. Этот порядок называется *правилом вычисления* для Common Lisp.

Тем не менее существуют операторы, которые не следуют принятому в Common Lisp порядку вычислений. Один из них – `quote`, или оператор цитирования. `quote` – это *специальный оператор*; это означает, что у него есть собственное правило вычисления, а именно: ничего не делать. Фактически `quote` берет один аргумент и просто возвращает его текстовую запись:

```
> (quote (+ 3 5))
(+ 3 5)
```

Решение проблемы

Если вы ввели что-то, что Лисп не понимает, то он выведет сообщение об ошибке, и среда переведет вас в вариант toplevel, называемый *циклом прерывания* (*break loop*). Он дает опытному программисту возможность выяснить причину ошибки, но вам пока что потребуется знать только то, как выйти из этого цикла. В разных реализациях Common Lisp выход может осуществляться по-разному. В гипотетической реализации вам поможет команда `:abort`.

```
> (/ 1 0)
Error: Division by zero.
       Options: :abort, :backtrace
>> :abort
>
```

В приложении А показано, как отлаживать программы на Лиспе, а также приводятся примеры наиболее распространенных ошибок.

Для удобства в Common Lisp можно заменять оператор `quote` на кавычку. Тот же результат можно получить, просто поставив ' перед цитируемым выражением:

```
> '(+ 3 5)
(+ 3 5)
```

В явном виде оператор `quote` почти не используется, более распространена его сокращенная запись с кавычкой.

Цитирование в Лиспе является способом *защиты* выражения от вычисления. В следующем разделе будет показано, чем такая защита может быть полезна.

2.3. Данные

Лисп предоставляет все типы данных, которые есть в большинстве других языков, а также некоторые другие, отсутствующие где-либо еще. С одним типом данных мы уже познакомились. Это *integer* – целое число, записываемое в виде последовательности цифр: `256`. Другой тип данных, который есть в большинстве других языков, – *строка*, представляемая как последовательность символов, окруженная двойными кавычками: `"ora et labora"`. Так же как и целые числа, строки самовычисляемы.

В Лиспе есть два типа, которые редко используются в других языках, – символы и списки. *Символы* – это слова. Обычно они преобразуются к верхнему регистру независимо от того, как вы их ввели:

```
> 'Artichoke
ARTICHOKE
```

Символы, как правило, не являются самовычисляемым типом, поэтому, чтобы сослаться на символ, его необходимо цитировать, как показано выше.

Список – это последовательность из нуля или более элементов, заключенных в скобки. Эти элементы могут принадлежать к любому типу, в том числе могут являться другими списками. Чтобы Лисп не счел список вызовом функции, его нужно процитировать:

```
> '(my 3 "Sons")
(MY 3 "Sons")
> '(the list (a b c) has 3 elements)
(THE LIST (A B C) HAS 3 ELEMENTS)
```

Обратите внимание, что цитирование предотвращает вычисление всего выражения, включая все его элементы.

Список можно построить с помощью функции `list`. Как и у любой функции, ее аргументы вычисляются. В следующем примере внутри вызова функции `list` вычисляется значение функции `+`:

```
> (list 'my (+ 2 1) "Sons")
(MY 3 "Sons")
```

Пришло время оценить одну из наиболее важных особенностей Лиспа. *Программы, написанные на Лиспе, представляются в виде списков.* Если приведенные ранее доводы о гибкости и элегантности не убедили вас в ценности принятой в Лиспе нотации, то, возможно, этот момент заставит вас изменить свое мнение. Именно эта особенность позволяет программам, написанным на Лиспе, генерировать Лисп-код, что дает возможность разработчику создавать программы, которые пишут программы.

Хотя такие программы не рассматриваются в книге вплоть до главы 10, сейчас важно понять связь между выражениями и списками. Как раз для этого нам нужно цитирование. Если список цитируется, то результатом его вычисления будет сам список. В противном случае список будет расценен как код и будет вычислено его значение:

```
> (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)
```

Первый аргумент цитируется и остается списком, в то время как второй аргумент расценивается как вызов функции и превращается в число.

Список может быть пустым. В Common Lisp возможны два типа представления пустого списка: пара пустых скобок и специальный символ `nil`. Независимо от того, как вы введете пустой список, он будет отображен как `nil`.

```
> ()
NIL
> nil
NIL
```

Перед () необязательно ставить кавычку, так как символ nil самовычисляем.

2.4. Операции со списками

Построение списков осуществляется с помощью функции cons. Если второй ее аргумент – список, она возвращает новый список с первым аргументом, добавленным в его начало:

```
> (cons 'a '(b c d))
(A B C D)
```

Список из одного элемента также может быть создан с помощью cons и пустого списка. Функция list, с которой мы уже познакомились, – всего лишь более удобный способ последовательного использования cons.

```
> (cons 'a (cons 'b nil))
(A B)
> (list 'a 'b)
(A B)
```

Простейшие функции для получения отдельных элементов списка – car и cdr.° Функция car служит для вывода первого элемента списка, а cdr – для вывода всех элементов, кроме первого:

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

Используя комбинацию функций car и cdr, можно получить любой элемент списка. Например, чтобы получить третий элемент, нужно написать:

```
> (car (cdr (cdr '(a b c d))))
C
```

Однако намного проще достичь того же результата с помощью функции third:

```
> (third '(a b c d))
C
```

2.5. Истинность

В Common Lisp истинность по умолчанию представляется символом t. Как и nil, символ t является самовычисляемым. Например, функция listp возвращает истину, если ее аргумент – список:

```
> (listp '(a b c))
T
```

Функции, возвращающие логические значения «истина» либо «ложь», называются *предикатами*. В Common Lisp имена предикатов часто оканчиваются на «p».

Ложь в Common Lisp представляется с помощью `nil`, пустого списка. Применяя `listp` к аргументу, который не является списком, получим `nil`:

```
> (listp 27)
NIL
```

Поскольку `nil` имеет два значения в Common Lisp, функция `null`, имеющая истинное значение для пустого списка:

```
> (null nil)
T
```

и функция `not`, возвращающая истинное значение, если ее аргумент ложен:

```
> (not nil)
T
```

делают одно и то же.

Простейший условный оператор в Common Lisp – `if`. Обычно он принимает три аргумента: *test*-, *then*- и *else*-выражения. Сначала вычисляется *тестовое test*-выражение. Если оно истинно, вычисляется *then*-выражение (*«то»*) и возвращается его значение. В противном случае вычисляется *else*-выражение (*«иначе»*).

```
> (if (listp '(a b c))
      (+ 1 2)
      (+ 5 6))
3
> (if (listp 27)
      (+ 1 2)
      (+ 5 6))
11
```

Как и `quote`, `if` – это специальный оператор, а не функция, так как для функции вычисляются все аргументы, а у оператора `if` вычисляется лишь одно из двух последних выражений.

Указывать последний аргумент `if` необязательно. Если он пропущен, то автоматически принимается за `nil`.

```
> (if (listp 27)
      (+ 2 3))
NIL
```

Несмотря на то, что по умолчанию истина представляется в виде `t`, любое выражение, кроме `nil`, также считается истинным:

```
> (if 27 1 2)
1
```

Логические операторы `and` (и) и `or` (или) действуют похожим образом. Оба могут принимать любое количество аргументов, но вычисляют их до тех пор, пока не будет ясно, какое значение необходимо вернуть. Если все аргументы истинны (то есть не `nil`), то оператор `and` вернет значение последнего:

```
> (and t (+ 1 2))
3
```

Но если один из аргументов окажется ложным, то следующие за ним аргументы не будут вычислены. Так же действует и `or`, вычисляя значения аргументов до тех пор, пока среди них не найдется хотя бы одно истинное значение.

Эти два оператора – *макросы*. Как и специальные операторы, макросы могут обходить обычный порядок вычисления. В главе 10 объясняется, как писать собственные макросы.

2.6. Функции

Новые функции можно определить с помощью оператора `defun`. Он обычно принимает три или более аргументов: имя, список параметров и одно или более выражений, которые составляют тело функции. Вот как мы можем с помощью `defun` определить функцию `third`:

```
> (defun our-third (x)
    (car (cdr (cdr x))))
OUR-THIRD
```

Первый аргумент задает имя функции, в нашем примере это `our-third`. Второй аргумент, список `(x)`, сообщает, что функция может принимать строго один аргумент: `x`. Используемый здесь символ `x` называется *переменной*. Когда переменная представляет собой аргумент функции, как `x` в этом примере, она еще называется *параметром*.

Оставшаяся часть, `(car (cdr (cdr x)))`, называется *телом* функции. Она сообщает Лиспу, что нужно сделать, чтобы вернуть значение из функции. Вызов `our-third` возвращает `(car (cdr (cdr x)))`, какое бы значение аргумента `x` ни было задано:

```
> (our-third '(a b c d))
C
```

Теперь, когда мы познакомились с переменными, будет легче понять, чем являются символы. Это попросту имена переменных, существующие сами по себе. Именно поэтому символы, как и списки, нуждаются в цитировании. Так же как «закавыченные» списки не будут восприняты как код, так и «закавыченные» символы не будут перепутаны с переменными.

Функцию можно рассматривать как обобщение Лисп-выражения. Следующее выражение проверяет, превосходит ли сумма чисел `1` и `4` число `3`:

```
> (> (+ 1 4) 3)
T
```

Заменяя конкретные числа переменными, мы можем получить функцию, выполняющую ту же проверку, но уже для трех произвольных чисел:

```
> (defun sum-greater (x y z)
    (> (+ x y) z))
SUM-GREATER
> (sum-greater 1 4 3)
T
```

В Лиспе нет различий между программой, процедурой и функцией. Все это функции (да и сам Лисп по большей части состоит из функций). Не имеет смысла определять одну *главную* функцию[1], ведь любая функция может быть вызвана в toplevel. В числе прочего, это означает, что программу можно тестировать по маленьким кусочкам в процессе ее написания.

2.7. Рекурсия

Функции, рассмотренные в предыдущем разделе, вызывали другие функции, чтобы те выполнили часть работы за них. Например, `sum-greater` вызывала + и >. Функция может вызывать любую другую функцию, включая саму себя.

Функции, вызывающие сами себя, называются *рекурсивными*. В Common Lisp есть функция `member`, которая проверяет, есть ли в списке какой-либо объект. Ниже приведена ее упрощенная реализация:

```
(defun our-member (obj lst)
  (if (null lst)
      nil
      (if (eql (car lst) obj)
          lst
          (our-member obj (cdr lst)))))
```

Предикат `eql` проверяет два аргумента на идентичность. Все остальное в этом выражении вам должно быть уже знакомо.

```
> (our-member 'b '(a b c))
(B C)
> (our-member 'z '(a b c))
NIL
```

Опишем словами, что делает эта функция. Чтобы проверить, есть ли `obj` в списке `lst`, мы:

---

[1] Как, например, в C, где требуется введение основной функции `main`. – *Прим. перев.*

1. Проверяем, пуст ли список `lst`. Если он пуст, значит, `obj` не присутствует в списке.
2. Если `obj` является первым элементом `lst`, значит, он есть в этом списке.
3. В другом случае проверяем, есть ли `obj` среди оставшихся элементов списка `lst`.

Подобное описание порядка действий будет полезным, если вы захотите понять, как работает рекурсивная функция.

Многим поначалу бывает сложно понять рекурсию. Причина этому – использование ошибочной метафоры для функций. Часто считается, что функция – это особый агрегат, который получает параметры в качестве сырья, перепоручает часть работы другим функциям-агрегатам и в итоге получает готовый продукт – возвращаемое значение. При таком рассмотрении возникает парадокс: как агрегат может перепоручать работу сам себе, если он уже занят?

Более подходящее сравнение для функции – процесс. Для процесса рекурсия вполне естественна. В повседневной жизни мы часто наблюдаем рекурсивные процессы и не задумываемся об этом. К примеру, представим себе историка, которому интересна динамика численности населения в Европе. Процесс изучения им соответствующих документов будет выглядеть следующим образом:

1. Получить копию документа.
2. Найти в нем информацию об изменении численности населения.
3. Если в нем также упоминаются иные источники, которые могут быть полезны, также изучить и их.

Этот процесс довольно легко понять, несмотря на то, что он рекурсивен, поскольку его третий шаг может повлечь за собой точно такой же процесс.

Поэтому не стоит расценивать функцию `our-member` как некий агрегат, который проверяет присутствие какого-либо элемента в списке. Это всего лишь набор правил, позволяющих выяснить этот факт. Если мы будем воспринимать функции подобным образом, то парадокс, связанный с рекурсией, исчезает.°

2.8. Чтение Лиспа

Определенная в предыдущем разделе функция заканчивается пятью закрывающими скобками. Более сложные функции могут заканчиваться семью-восемью скобками. Это часто обескураживает людей, которые только начинают изучать Лисп. Как можно читать подобный код, не говоря уже о том, чтобы писать его самому? Как искать в нем парные скобки?

Ответ прост: вам не нужно делать это. Лисп-программисты пишут и читают код, ориентируясь по отступам, а не по скобкам. К тому же любой

хороший текстовый редактор, особенно если он поставляется с Лисп-системой, умеет выделять парные скобки. Если ваш редактор не делает этого, остановитесь и постарайтесь узнать, как включить подобную функцию, поскольку без нее писать Лисп-код практически невозможно.[1]

С хорошим редактором поиск парных скобок в процессе написания кода перестанет быть проблемой. Кроме того, благодаря общепринятым соглашениям касательно выравнивания кода вы можете легко читать код, не обращая внимания на скобки.

Любой Лисп-хакер, независимо от своего стажа, с трудом разберет определение `our-member`, если оно будет записано в таком виде:

```
(defun our-member (obj lst) (if (null lst) nil (if (eql (car lst) obj) lst
(our-member obj (cdr lst)))))
```

С другой стороны, правильно выравненный код будет легко читаться даже без скобок:

```
defun our-member (obj lst)
  if (null lst)
     nil
     if eql (car lst) obj
        lst
        our-member obj (cdr lst)
```

Это действительно удобный подход при написании кода на бумаге, в то время как в редакторе вы сможете воспользоваться удобной возможностью поиска парных скобок.

2.9. Ввод и вывод

До сих пор мы осуществляли ввод-вывод с помощью toplevel. Чтобы ваша программа была по-настоящему интерактивной, этого явно недостаточно. В этом разделе мы рассмотрим несколько функций ввода-вывода.

Основная функция вывода в Common Lisp – `format`. Она принимает два или более аргументов: первый определяет, куда будет напечатан результат, второй – это строковый шаблон, а остальные аргументы – это объекты, которые будут вставляться в нужные позиции заданного шаблона. Вот типичный пример:

```
> (format t "~A plus ~A equals ~A.~%" 2 3 (+ 2 3))
2 plus 3 equals 5.
NIL
```

Обратите внимание, что здесь отображены два значения. Первая строка – результат выполнения `format`, напечатанный в toplevel. Обычно функции типа `format` не вызываются напрямую в toplevel, а используются внутри программ, поэтому возвращаемые ими значения не отображаются.

---

[1] В редакторе vi эта опция включается командой `:set sm`. В Emacs `M-x lisp-mode` будет отличным выбором.

Первый аргумент для функции format, t, показывает, что вывод будет отправлен в стандартное место, принятое по умолчанию. Обычно это toplevel. Второй аргумент – это строка, служащая шаблоном для вывода. В ней каждое ~A определяет позицию для заполнения, а символ ~% соответствует переносу строки. Позиции по порядку заполняются значениями оставшихся аргументов.

Стандартная функция чтения – read (ее также называют считывателем). Вызванная без аргументов, она выполняет чтение из стандартного места, которым обычно бывает toplevel. Ниже приведена функция, которая предлагает ввести любое значение и затем возвращает его:

```
(defun askem (string)
  (format t "~A" string)
  (read))
```

Это работает так:

```
> (askem "How old are you? ")
How old are you? 29
29
```

Учтите, что для завершения считывания read будет ожидать нажатия вами Enter. Поэтому не стоит использовать функцию read, не печатая перед этим приглашение ко вводу, иначе может показаться, что программа зависла, хотя на самом деле она просто ожидает ввода.

Другая вещь, которую следует знать о функции read: это поистине мощный инструмент. Фактически это полноценный обработчик Лисп-выражений. Она не просто читает символы и возвращает их в виде строки – она обрабатывает введенное выражение и возвращает полученный лисповый объект. В приведенном выше примере функция read возвращает число.

Несмотря на свою краткость, определение askem содержит нечто, чего мы до сих пор не встречали – тело функции состоит из нескольких выражений. Вообще говоря, оно может содержать любое количество выражений. При вызове функции они вычисляются последовательно, и возвращается значение последнего выражения.

Во всех предыдущих разделах мы придерживались «чистого» Лиспа, то есть Лиспа без *побочных эффектов*. Побочный эффект – это событие, которое каким-либо образом изменяет состояние системы. При вычислении выражения (+ 1 2) возвращается значение 3, и никаких побочных эффектов не происходит. В последнем же примере побочный эффект производится функцией format, которая не только возвращает значение, но и печатает что-то еще. Это одна из разновидностей побочных эффектов.

Если мы зададимся целью писать код без побочных эффектов, то будет бессмысленно определять функции, состоящие из нескольких выражений, так как в качестве значения функции будет использоваться значение последнего аргумента, а значения предыдущих выражений будут

утеряны. Если эти выражения не будут вызывать побочных эффектов, то вы даже не узнаете, вычислял ли их Лисп вообще.

2.10. Переменные

Один из наиболее часто используемых операторов в Common Lisp – это `let`, который позволяет вам ввести новые локальные переменные:

```
> (let ((x 1) (y 2))
    (+ x y))
3
```

Выражение с использованием `let` состоит из двух частей. Первая содержит инструкции, определяющие новые переменные. Каждая такая инструкция содержит имя переменной и соответствующее ей выражение. Чуть выше мы создали две новые переменные, `x` и `y`, которым были присвоены значения `1` и `2`. Эти переменные действительны внутри тела `let`.

За списком переменных и значений следуют выражения, которые вычисляются по порядку. В нашем случае имеется только одно выражение, `(+ x y)`. Значением всего вызова `let` будет значение последнего выражения. Давайте напишем более избирательный вариант функции `askem` с использованием `let`:

```
(defun ask-number ()
  (format t "Please enter a number. ")
  (let ((val (read)))
    (if (numberp val)
        val
        (ask-number))))
```

Мы создаем переменную `val`, содержащую результат вызова `read`. Сохраняя это значение, мы можем проверить, что было прочитано. Как вы уже догадались, `numberp` – это предикат, проверяющий, является ли его аргумент числом.

Если введенное значение – нечисло, то `ask-number` вызовет саму себя, чтобы пользователь повторил попытку. Так будет повторяться до тех пор, пока функция `read` не получит число:

```
> (ask-number)
Please enter a number. a
Please enter a number. (no hum)
Please enter a number. 52
52
```

Переменные типа `val`, создаваемые оператором `let`, называются *локальными*, то есть действительными в определенной области. Есть также *глобальные* переменные, которые действительны везде.[1]

[1] Реальная разница между локальными и глобальными переменными будет пояснена в главе 6.

Глобальная переменная может быть создана с помощью оператора `defparameter`:

```
> (defparameter *glob* 99)
*GLOB*
```

Такая переменная будет доступна везде, кроме выражений, в которых создается локальная переменная с таким же именем. Чтобы избегать таких случайных совпадений, принято давать глобальным переменным имена, окруженные звездочками.

Также в глобальном окружении можно задавать константы, используя оператор `defconstant`[1]:

```
(defconstant limit (+ *glob* 1))
```

Нет необходимости давать константам имена, окруженные звездочками, потому что любая попытка определить переменную с таким же именем закончится ошибкой. Чтобы узнать, соответствует ли какое-то имя глобальной переменной или константе, воспользуйтесь `boundp`:

```
> (boundp '*glob*)
T
```

2.11. Присваивание

В Common Lisp наиболее общим средством присваивания значений является оператор `setf`. Он может присваивать значения переменным любых типов:

```
> (setf *glob* 98)
98
> (let ((n 10))
    (setf n 2)
    n)
2
```

Если `setf` пытается присвоить значение переменной, которая в данном контексте не является локальной, то переменная будет определена как глобальная:

```
> (setf x (list 'a 'b 'c))
(A B C)
```

Таким образом, вы можете создавать глобальные переменные неявно, просто присваивая им значения. Однако в файлах исходного кода считается хорошим тоном задавать их явно с помощью оператора `defparameter`.

---

[1] Как и для специальных переменных, для определяемых пользователем констант существует негласное правило, согласно которому их имена следует окружать знаками +. Следуя этому правилу, в примере выше мы бы определили константу `+limit+`. – *Прим. перев.*

Но можно делать больше, чем просто присваивать значения переменным. Первым аргументом setf может быть не только переменная, но и выражение. В таком случае значение второго аргумента вставляется в то *место*, на которое ссылается это выражение:

```
> (setf (car x) 'n)
N
> x
(N B C)
```

Первым аргументом setf может быть почти любое выражение, которое ссылается на какое-то место. Все такие операторы отмечены в приложении D как работающие с setf.

Также setf можно передать любое (четное) число аргументов. Следующий вызов

```
(setf a b
      c d
      e f)
```

эквивалентен трем последовательным вызовам setf:

```
(setf a b)
(setf c d)
(setf e f)
```

2.12. Функциональное программирование

Функциональное программирование – это понятие, которое означает написание программ, работающих через возвращаемые значения, а не изменения среды. Это основная парадигма в Лиспе. Большинство встроенных в Лисп функций не вызывают побочных эффектов.

Например, функция remove принимает список и объект и возвращает новый список, состоящий из тех же элементов, что и предыдущий, за исключением этого объекта.

```
> (setf lst '(c a r a t))
(C A R A T)
> (remove 'a lst)
(C R T)
```

Почему бы не сказать, что remove просто удаляет элемент из списка? Потому что она этого не делает. Исходный список остался нетронутым:

```
> lst
(C A R A T)
```

А что если вы хотите действительно удалить элементы из списка? В Лиспе принято предоставлять список в качестве аргумента какой-либо функции и присваивать возвращаемое ей значение с помощью setf. Чтобы удалить все a из списка x, мы скажем:

```
(setf x (remove 'a x))
```

В функциональном программировании избегают использовать `setf` и другие подобные вещи. Поначалу сложно вообразить, что это попросту возможно, не говоря уже о том, что желательно. Как можно создавать программы только с помощью возвращения значений?

Конечно, совсем без побочных эффектов работать неудобно. Однако, читая дальше эту книгу, вы будете все сильнее удивляться тому, как, в сущности, немного побочных эффектов необходимо. И чем меньше вы будете ими пользоваться, тем лучше.

Одно из главных преимуществ использования функционального программирования заключается в *интерактивном тестировании*. Это означает, что вы можете тестировать функции, написанные в функциональном стиле, сразу же после их написания, и при этом вы будете уверены, что результат выполнения всегда будет одинаковым. Вы можете изменить одну часть программы, и это не повлияет на корректность работы другой ее части. Такая возможность безбоязненно изменять код делает доступным новый стиль программирования, подобно тому как телефон в сравнении с письмами предоставляет новый способ общения.

2.13. Итерация

Когда мы хотим повторить что-то многократно, порой удобнее использовать итерацию, чем рекурсию. Типичный пример использования итерации – генерация таблиц. Следующая функция

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

печатает квадраты целых чисел от `start` до `end`:

```
> (show-squares 2 5)
2 4
3 9
4 16
5 25
DONE
```

Макрос `do` – основной итерационный оператор в Common Lisp. Как и в случае `let`, первый аргумент `do` задает переменные. Каждый элемент этого списка может выглядеть как:

(*variable initial update*)

где *variable* – символ, а *initial* и *update* – выражения. Исходное значение каждой переменной определяется значением *initial*, и на каждой итерации это значение будет меняться в соответствии со значением *update*. В примере `show-squares` `do` использует только одну переменную, `i`. На первой итерации значение `i` равно `start`, и после каждой успешной итерации оно будет увеличиваться на единицу.