Функциональное программирование

Питер Сайбель

Практическое использование Common Lisp

УДК 004.438Common Lisp
ББК 32.973.22
С14

Сайбель П.
С14 Практическое использование Common Lisp / пер. с англ. А.Я. Отта. – М.: ДМК Пресс, 2015. – 488 с.: ил.

ISBN 978-5-94074-627-0

В отличие от основной массы литературы про Lisp, эта книга не просто рассказывает о ряде возможностей языка, предоставляя читателю самостоятельно осваивать их на практике. Здесь будут описаны все функции языка, которые понадобятся вам для написания реальных программ. Более трети книги посвящено разработке нетривиальных программ – статистического фильтра для спама, библиотеки для разбора двоичных файлов и сервера для трансляции музыки в формате MP3 через сеть, включающего в себя базу данных (MP3-файлов) и веб-интерфейс.

Издание прнедназначено для программистов различной квалификации, как уже использующих Lisp в своей работе, так и только знакомящихся с этим языком.

УДК 004.438Common Lisp
ББК 32.973.22

ISBN 1-59059-239-5 (анг.)
ISBN 978-5-94074-627-0 (рус.)

Оглавление

Глава 2

Намылить, смыть, повторить: знакомство с REPL

В этой главе вы настроите среду программирования и напишете свои первые программы на Common Lisp. Мы воспользуемся лёгким в установке дистрибутивом Lisp in a Box, разработанным Matthew Danish и Mikel Evins, включающим в себя реализацию Common Lisp, Emacs – мощный текстовый редактор, прекрасно поддерживающий Lisp, а также SLIME[1] – среду разработки для Common Lisp, основанную на Emacs.

Этот набор предоставляет программисту современную среду разработки для Common Lisp, поддерживающую инкрементальный интерактивный стиль разработки, характерный для программирования на этом языке. Среда SLIME даёт дополнительное преимущество в виде унифицированного пользовательского интерфейса, не зависящего от выбранных вами операционной системы и реализации Common Lisp. В книге я буду ориентироваться на среду Lisp in a Box, но те, кто хочет изучить другие среды разработки, например графические интегрированные среды разработки (IDE – Integrated Development Environment), предоставляемые некоторыми коммерческими поставщиками, или среды, основанные на других текстовых редакторах, не должны испытывать больших трудностей в понимании[2].

2.1. Выбор реализации Lisp

Первое, что вам предстоит сделать, – выбрать реализацию Lisp. Это может показаться несколько странным для тех, кто раньше занимался программированием на таких

[1] Superior Lisp Interaction Mode for Emacs.

[2] Если у вас уже был неудачный опыт работы с Emacs, то вам следует рассматривать Lisp in a Box как IDE, которая использует Emacs в качестве текстового редактора. Однако для программирования на Lisp от вас не требуется быть гуру Emacs. С другой стороны, программировать на Lisp гораздо удобнее в редакторе, который имеет хотя бы минимальную поддержку этого языка. Наверняка вам захочется, чтобы редактор автоматически помечал парные скобки и сам мог расставить отступы в коде на Lisp. Так как Emacs почти целиком написан на одном из диалектов Lisp, Elisp, он имеет очень хорошую поддержку редактирования такого кода. История Emacs неразрывно связана с историей Lisp и культурой Lisp-хакеров: первые версии Emacs, как и его непосредственные предшественники TECMACS и TMACS, были написаны заинтересованными в Lisp разработчиками в Массачусетском технологическом институте (MIT). Редакторами, использовавшимися на Lisp-машинах, были версии Emacs, целиком написанные на Lisp. Под влиянием любви хакеров к рекурсивным акронимам две первые реализации Emacs для Lisp-машин были названы EINE и ZWEI, что означало «EINE Is Not Emacs» и «ZWEI Was EINE Initially» соответственно. Некоторое время был распространён производный от ZWEI редактор, названный более прозаично, ZMACS.

языках, как Perl, Python, Visual Basic (VB), C# или Java. Разница между Common Lisp и этими языками заключается в том, что Common Lisp определяется своим стандартом: не существует единственной его реализации, контролируемой «великодушным диктатором» (как в случае с Perl и Python), или канонической реализации, контролируемой одной компанией (как в случае с VB, C# или Java). Любой желающий может создать свою реализацию на основе стандарта. Кроме того, изменения в стандарт должны вноситься в соответствии с процессом, контролируемым Американским национальным институтом стандартов (ANSI). Этот процесс организован таким образом, что «случайные лица», такие как частные поставщики программных решений, не могут вносить изменения в стандарт по своему усмотрению[3]. Таким образом, стандарт Common Lisp – это договор между поставщиком Common Lisp и использующими Common Lisp разработчиками; этот договор подразумевает, что если вы пишете программу, использующую возможности языка так, как это описано в стандарте, вы можете рассчитывать, что эта программа запустится на любой совместимой реализации Common Lisp.

С другой стороны, стандарт может не описывать всё то, что вам может понадобиться в ваших программах. Более того, некоторые аспекты языка спецификация намеренно опускает, чтобы дать возможность экспериментальным путём решить спорные вопросы языка. Таким образом, каждая реализация предоставляет пользователям как возможности, входящие в стандарт, так и выходящие за его пределы. В зависимости от того, что вы хотите программировать, вы можете выбрать реализацию Common Lisp, поддерживающую именно те дополнительные возможности, которые вам больше всего понадобятся. С другой стороны, если вы пишете код, которым будут пользоваться другие разработчики, то вы, вероятно, захотите – конечно, в пределах возможного – писать переносимый код на Common Lisp. Для нужд написания переносимого кода, который в то же время использует возможности, не описанные в стандарте, Common Lisp предоставляет гибкий способ писать код, «зависящий» от возможностей текущей реализации. Вы увидите пример такого кода в главе 15, когда мы будем разрабатывать простую библиотеку, «сглаживающую» некоторые различия в обработке имён файлов разными реализациями Lisp.

Сейчас, однако, наиболее важная характеристика реализации – её способность работать в вашей любимой операционной системе. Сотрудники компании Franz, занимающейся разработкой Allegro Common Lisp, выпустили пробную версию своего продукта, работающего на GNU/Linux, Windows и OS X, предназначенную для использования с этой книгой. У читателей, предпочитающих реализации с открытым исходным кодом, есть несколько вариантов. SBCL[4] – высококачественная открытая реализация, способная компилировать в машинный код и работать на множестве различных UNIX-

[3] На самом деле существует очень малая вероятность пересмотра стандарта языка. Хотя есть некоторое количество недостатков, которые пользователи языка могут желать исправить, согласно процессу стандартизации ANSI, существующий стандарт не подлежит открытию для внесения небольших изменений, и эти недостатки на самом деле, не вызывают ни у кого серьёзных трудностей. Возможно, будущее стандартизации Common Lisp – за стандартами «де-факто», больше похожими на «стандартизацию» Perl и Python, когда различные разработчики экспериментируют с интерфейсами прикладного программирования (API) и библиотеками для реализации вещей, не описанных в стандарте языка, а другие разработчики могут принимать их; или заинтересованные программисты будут разрабатывать переносимые библиотеки для сглаживания различий между реализациями возможностей, не описанных в стандарте языка.

[4] Steel Bank Common Lisp.

систем, включая Linux и OS X. SBCL – «наследник» CMUCL[5] – реализации Common Lisp, разработанной в университете Carnegie Mellon, и, как и CMUCL, является всеобщим достоянием (public domain), за исключением нескольких частей, покрываемых BSD-подобными (Berkley Software Distributions) лицензиями. CMUCL – тоже хороший выбор, однако SBCL обычно легче в установке и поддерживает 21-разрядный Unicode[6]. OpenMCL будет отличным выбором для пользователей OS X: эта реализация способна компилировать в машинный код, поддерживать работу с потоками, а также прекрасно интегрируется с инструментальными комплектами Carbon и Cocoa. Кроме перечисленных, существуют и другие свободные и коммерческие реализации. В главе 32 перечислены источники для получения дополнительной информации.

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

2.2. Введение в Lisp in a Box

Поскольку Lisp in a Box спроектирован с целью быть «дружелюбным» к новичкам, а также предоставлять первоклассную среду разработки на Lisp, всё, что вам нужно для работы, – выбрать пакет, соответствующий вашей операционной системе, с веб-сайта Lisp in a Box (см. главу 32), а затем просто следовать инструкциям по установке.

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

Учебник также описывает много других полезных команд Emacs и вызывающих их комбинаций клавиш. Более того, у Emacs есть расширенная онлайн-документация, для просмотра которой используется специальный браузер – Info. Чтобы её вызвать, нажмите `C-h i`. У Info, в свою очередь, есть своя справка, которую можно вызвать, нажав клавишу h, находясь в браузере Info. Emacs предоставляет ещё несколько способов получить справку – это все сочетания клавиш, начинающиеся с `C-h`, – полный

[5] CMU Common Lisp.

[6] SBCL стал «ответвлением» CMUCL, так как его разработчики хотели сосредоточиться на упорядочивании его внутренней организации и сделать его легче в сопровождении. «Ответвление» вышло очень удачным. Исправления ошибок привели к появлению серьёзных различий между двумя проектами, и, как теперь поговаривают, их снова планируют объединить.

список по `C-h ?`. В этом списке есть две полезные вещи: `C-h k` «объяснит» комбинацию клавиш, а `C-h w` – команду.

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

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

2.3. Освободите свой разум: интерактивное программирование

При запуске Lisp in a Box вы должны увидеть примерно следующее приглашение:

```
CL-USER>
```

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

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

Все эти возможности встроены в язык и доступны через функции, определённые в стандарте языка. Если вы захотите, то можете построить достаточно приемлемую среду разработки только из REPL и текстового редактора, который знает, как правильно форматировать код Lisp. Но для истинного опыта Lisp-программирования вам необ-

ходима среда разработки типа SLIME, которая бы позволяла вам взаимодействовать с Lisp как посредством REPL, так и при редактировании исходных файлов. Например, вы ведь не хотите каждый раз копировать и вставлять куски кода из редактора в REPL или перезагружать весь файл только потому, что изменилось одно определение, ваше окружение должно позволять вам вычислять или компилировать как отдельные выражения, так и целые файлы из вашего редактора*.

2.4. Эксперименты в REPL

Для знакомства с REPL вам необходимо выражение Lisp, которое может быть прочитано, вычислено и выведено на экран. Простейшее выражение Lisp – это число. Если вы наберёте `10` в приглашении Lisp и нажмёте Enter, то увидите что-то наподобие:

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

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

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

Все, что в скобках, – это список, в данном случае список из трёх элементов: символ `+` и числа 2 и 3. Lisp, в общем случае, вычисляет списки, считая первый элемент именем функции, а остальные – выражениями для вычисления и передачи в качестве аргументов этой функции. В нашем случае символ `+` – название функции, которая вычисляет сумму. 2 и 3 вычисляются сами в себя и передаются в функцию суммирования, которая возвращает 5. Значение 5 отправляется на устройство вывода, которое отображает его. Lisp может вычислять выражения и другими способами, но нам пока не нужно вдаваться в такие детали. В первую очередь нам нужно написать...

2.5. «Hello, world» в стиле Lisp

Нет законченной книги по программированию без программы «hello, world»[7]. После того как интерпретатор запущен, нет ничего проще, чем набрать строку «hello, world».

```
CL-USER> "hello, world"
 "hello, world"
```

Это работает, поскольку строки, так же как и числа, имеют символьный синтаксис, понимаемый процедурой чтения Lisp, и являются самовычисляемыми объектами: Lisp

* Для использования русского языка необходима соотв. настройка Emacs, Slime(cvs-версия), и вашего интерпретатора Lisp. — *Прим. перев.*

[7] Досточтимая фраза «hello, world» предшествует даже классической книге по языку C Кернигана и Ритчи, которая сыграла огромную роль в её популяризации. Первоначальный «hello, world», похоже, пришёл из книги Брайана Кернигана «A Tutorial Introduction to the Language B», которая была частью *Bell Laboratories Computing Science Technical Report #8: The Programming Language B*, опубликованного в январе 1973 г. (Отчёт выложен в Интернете `http://cm.bell-labs.com/cm/cs/who/dmr/bintro.html`.)

считывает строку в двойных кавычках и создаёт в памяти строковой объект, который при вычислении вычисляется сам в себя и потом печатается в том же символьном представлении. Кавычки не являются частью строкового объекта в памяти – это просто синтаксис, который позволяет процедуре чтения определить, что этот объект – строка. Устройство вывода REPL напечатает кавычки тоже, потому что оно пытается выводить объекты в таком же виде, в каком понимает их процедура чтения.

Однако наш пример не может квалифицироваться как *программа* «hello, world». Это, скорее, *значение* «hello, world».

Вы можете сделать шаг к настоящей программе, напечатав код, который в качестве побочного эффекта отправит на стандартный вывод строку «hello, world». Common Lisp предоставляет несколько путей для вывода данных, но самый гибкий – это функция `FORMAT`. `FORMAT` получает переменное количество параметров, но только два из них обязательны: указание, куда осуществлять вывод, и строка для вывода. В следующей главе вы увидите, как строка может содержать встроенные директивы, которые позволяют вставлять в строку последующие параметры функции (а-ля `printf` или строка `%` из Python). До тех пор, пока строка не содержит символа `~`, она будет выводиться как есть. Если вы передадите `t` в качестве первого параметра, функция `FORMAT` направит отформатированную строку на стандартный вывод. Итак, выражение `FORMAT` для печати «hello, world» выглядит примерно так[8]:

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

Стоит отметить, что результатом выражения `FORMAT` является `NIL` в строке после вывода «hello, world». Этот `NIL` является результатом вычисления выражения `FORMAT`, напечатанного REPL. (`NIL` – это Lisp-версия false и/или null. Подробнее об этом рассказывается в главе 4.) В отличие от других выражений, рассмотренных ранее, нас больше интересует побочный эффект выражения `FORMAT` (в данном случае печать на стандартный вывод), чем возвращаемое им значение. Но каждое выражение в Lisp вычисляется в некоторый результат[9].

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

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

---

[8] Есть несколько других выражений, которые тоже выводят строку «hello, world»:

```
(write-line "hello, world")
```

или

```
(print "hello, world")
```

[9] На самом деле, как вы увидите, когда будет рассказано о возврате множественных значений, технически возможно написание выражений, которые не вычисляются ни в какие значения, но даже такие выражения рассматриваются как возвращающие `NIL` при вычислении в контексте, ожидающем возврата значения.

Выражение `hello-world`, следующее за `DEFUN`, является именем функции. В главе 4 мы рассмотрим, какие именно символы могут использоваться в именах, но сейчас будет достаточно сказать, что многие символы, такие как `-`, недопустимые в именах в других языках, можно использовать в Common Lisp. Это стандартный стиль Lisp – не говоря уже о том, что это больше соответствует нормальному английскому написанию, – формирование составных имён с помощью дефисов, как в `hello-world`, вместо использования знаков подчёркивания, как в `hello_world`, или использования заглавных букв внутри имени, как `helloWorld`. Скобки `()` после имени отделяют список параметров, который в данном случае пуст, так как функция не принимает аргументов. Остальное – это тело функции.

В какой-то мере это выражение подобно всем другим, которые вы видели, всего лишь ещё одно выражение для чтения, вычисления и печати, осуществляемых REPL. Возвращаемое значение в этом случае – это имя только что определённой функции[10]. Но, подобно выражению `FORMAT`, это выражение более интересно своими побочными эффектами, нежели возвращаемым значением. Однако, в отличие от выражения `FORMAT`, побочные эффекты невидимы: после вычисления этого выражения создаётся новая функция, не принимающая аргументов, с телом `(format t "hello, world")`, и ей даётся имя `HELLO-WORLD`.

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

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

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

2.6. Сохранение вашей работы

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

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

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

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

Сообщение будет исчезать, когда вы будете начинать печатать каждый новый элемент, и снова появляться после ввода пробела. При вводе определения в файл вы може-

[10] В главе 4 будет рассказано, почему имя преобразуется в верхний регистр.

те захотеть разбить определение после списка параметров так, чтобы оно занимало две строки. Если вы нажмёте Enter, а затем Tab, SLIME автоматически выровняет вторую строку соответствующим образом[11]:

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

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

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

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

Затем перекомпилируем её с помощью `C-c C-c` и перейдём в REPL, набрав `C-c C-z`, чтобы попробовать новую версию.

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

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

Теперь, для того чтобы попробовать перезагрузить эту функцию из файла с исходным кодом, вы должны выйти из Lisp и перезапустить его. Для выхода вы можете использовать клавишную комбинацию SLIME: находясь в REPL, наберите запятую. Внизу окна Emacs вам будет предложено ввести команду. Наберите `quit` (или `sayoonara`), а затем нажмите Enter. Произойдёт выход из Lisp, а все окна, созданные SLIME (такие как буфер REPL), закроются[12]. Теперь перезапустите SLIME, набрав `M-x slime`.

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

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

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

```
attempt to call ‘HELLO-WORLD’ which is an undefined function.
   [Condition of type UNDEFINED-FUNCTION]

Restarts:
   0: [TRY-AGAIN] Try calling HELLO-WORLD again.
```

[11] Вы также можете ввести определение в двух строках в REPL, так как REPL читает выражение целиком, а не по строкам.

[12] клавишные комбинации SLIME – это не часть Common Lisp, это команды SLIME.

```
    1: [RETURN-VALUE] Return a value instead of calling HELLO-WORLD.
    2: [USE-VALUE] Try calling a function other than HELLO-WORLD.
    3: [STORE-VALUE] Setf the symbol-function of HELLO-WORLD and call it again.
    4: [ABORT] Abort handling SLIME request.
    5: [ABORT] Abort entirely from this process.

Backtrace:
  0: (SWANK::DEBUG-IN-EMACS #<UNDEFINED-FUNCTION @ #x716b082a>)
  1: (FLET SWANK:SWANK-DEBUGGER-HOOK SWANK::DEBUG-IT)
  2: (SWANK:SWANK-DEBUGGER-HOOK #<UNDEFINED-FUNCTION @ #x716b082a>
                                #<Function SWANK-DEBUGGER-HOOK>)
  3: (ERROR #<UNDEFINED-FUNCTION @ #x716b082a>)
  4: (EVAL (HELLO-WORLD))
  5: (SWANK::EVAL-REGION "(hello-world)
" T)
```

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

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

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

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

Вернувшись в REPL, вы можете попробовать снова. Ошибка произошла, потому что Lisp не знает определения `hello-world`. Поэтому вам нужно предоставить Lisp определение, сохранённое нами в файле `hello.lisp`. Вы можете сделать это несколькими способами. Вы можете переключиться назад в буфер, содержащий файл (наберите `C-x b`, а затем введите `hello.lisp`), и перекомпилировать определение, как вы это делали ранее с помощью `C-c C-c`. Или вы можете загрузить файл целиком (что будет более удобным способом, если файл содержит множество определений) путём использования функции `LOAD` в REPL следующим образом:

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

`T` означает, что загрузка всех определений произошла успешно[13]. Загрузка файла с помощью `LOAD` в сущности эквивалентна набору каждого выражения этого файла в REPL в том порядке, в каком они находятся в файле. Таким образом, после вызова `LOAD hello-world` должен быть определён.

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

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

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

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

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

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

И даже более впечатляющий пример удалённой отладки произошёл в миссии NASA «Deep Space 1» в 1998 году. Через полгода после запуска космического корабля небольшой код на Lisp должен был управлять космическим кораблём в течение двух дней для

[13] Если по каким-то причинам `LOAD` не выполнилась успешно, вы получите другую ошибку и будете перенаправлены в отладчик. Если это произошло, наиболее вероятной причиной может быть то, что Lisp не может найти файл, возможно из-за того, что его текущая директория не совпадает с той, в которой находится файл. В этом случае вы можете выйти из отладчика, набрав `q`, а затем использовать клавиатурную комбинацию SLIME `cd` для изменения текущей директории – наберите запятую, а затем на приглашение к вводу команды – `cd` и имя директории, где был сохранён `hello.lisp`.

проведения серии экспериментов. Однако неуловимое состояние гонки (race condition) в коде не было выявлено при тестировании на земле и было обнаружено уже в космосе – в 100 миллионах миль от Земли. Команда смогла произвести диагностику и исправление работающего кода, что позволило завершить эксперимент[14]. Один из программистов сказал об этом следующее:

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

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

[14] `http://www.flownet.com/gat/jpl-lisp.html`.

Глава 3

Практикум: простая база данных

Очевидно, перед тем как создавать настоящие программы на Lisp, вам необходимо изучить язык. Но давайте смотреть правде в глаза – вы можете подумать «Practical Common Lisp? Не оксюморон ли это? Зачем тратить силы на изучение деталей языка, если на нем невозможно сделать что-то дельное?» Итак, для начала я приведу маленький пример того, что можно сделать с помощью Common Lisp. В этой главе вы напишете простую базу данных для организации коллекции CD. В главе 27 вы будете использовать схожую технику при создании базы данных записей в формате MP3 для вашего потокового MP3-сервера. Фактически можете считать это частью вашего программного проекта – в конце концов, для того чтобы иметь сколько-нибудь MP3-записей для прослушивания, совсем не помешает знать, какие записи у вас есть, а какие нужно извлечь с диска.

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

Одно замечание по терминологии: в этой главе я расскажу о некоторых операторах Lisp. В главе 4 вы узнаете, что Common Lisp предоставляет три разных типа операторов: функции, макросы и операторы специального назначения. Для целей этой главы вам необязательно понимать разницу. Однако я буду ссылаться на различные операторы как на функции, макросы или специальные операторы, в зависимости от того, чем они на самом деле являются, вместо того чтобы попытаться скрыть эти детали за одним словом – оператор. Сейчас вы можете рассматривать функции, макросы и специальные операторы как более или менее эквивалентные сущности[1].

Также имейте ввиду, что я не буду использовать все наиболее сложные техники Common Lisp для вашей первой после «Hello, world» программы. Цель этой главы не в том, чтобы показать, как вам следует писать базу данных на Lisp; скорее, цель в том, чтобы вы получили представление, на что похоже программирование на Lisp, и видение того, что даже относительно простая программа на Lisp может иметь много возможностей.

[1] Прежде чем я продолжу, очень важно, чтобы вы забыли все, что можете знать о «макросах» в стиле `#define`, реализованных в препроцессоре C. Макросы Lisp не имеют с ними ничего общего.

3.1. CD и записи

Чтобы отслеживать диски, которые нужно перекодировать в MP3, и знать, какие из них должны быть перекодированы в первую очередь, каждая запись в базе данных будет содержать название и имя исполнителя компакт-диска, оценку того, насколько он нравится пользователю, и флаг, указывающий, был ли диск уже перекодирован. Итак, для начала вам необходим способ представления одной записи в базе данных (другими словами, одного CD). Common Lisp предоставляет для этого много различных структур данных, от простого четырёхэлементного списка до определяемого пользователем с помощью CLOS класса данных.

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

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

Вы могли бы использовать четырёхэлементный список, отображающий позицию в списке на соответствующее поле записи. Однако другая существующая разновидность списков, называемая *property list* (список свойств) или, сокращённо, *plist*, в нашем случае гораздо удобнее. *Plist* – это такой список, в котором каждый нечётный элемент является *символом*, описывающим следующий (чётный) элемент списка. На этом этапе я не буду углубляться в подробности понятия *символ*; по своей природе это имя. Для символов, именующих поля в базе данных, мы можем использовать частный случай символов, называемый *символами-ключами* (*keyword symbol*). Ключ – это имя, начинающееся с двоеточия (`:`), например `:foo`*. Вот пример *plist*, использующего символы-ключи `:a`, `:b` и `:c` как имена свойств:

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

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

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

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

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

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

* `LIST` – по английски **СПИСОК**. Кстати, последние реализации Common Lisp позволяют писать и на родном для вас языке. Например, на русском можно создать макрос `список`, который будет вызывать `LIST`, например так: `(defmacro list (&body body) ‘(list ,@body))`. — *Прим. перев.*

* `foo`, `bar` – любимые имена переменных у англоговорящих программистов, пишущих книги и документацию. — *Прим. перев.*

```
  (list :title title :artist artist :rating rating :ripped ripped))
```

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

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

3.2. Заполнение CD

Впрочем, создание одной записи – ещё не создание базы данных. Вам необходима более комплексная структура данных для хранения записей. Опять же, простоты ради, список представляется здесь вполне подходящим выбором. Также для простоты вы можете использовать глобальную переменную *db*, которую можно будет определить с помощью макроса DEFVAR. Звёздочки (*) в имени переменной – это договорённость, принятая в языке Lisp при объявлении глобальных переменных[2].

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

Для добавления элементов в *db* можно использовать макрос PUSH. Но разумнее немного абстрагировать вещи и определить функцию add-record, которая будет добавлять записи в базу данных.

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

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

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

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

---

* Тем, для кого английский родной. — *Прим. перев.*

[2] Использование глобальной переменной имеет ряд недостатков – например, в каждый момент времени вы можете работать только с одной базой данных. В главе 27, имея за плечами уже солидный багаж знаний о Lisp, вы будете готовы к созданию более гибкой базы данных. В главе 6 вы также увидите, что даже использование глобальных переменных в Common Lisp более гибко, чем это возможно в других языках.

3.3. Просмотр содержимого базы данных

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

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

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

```
TITLE:    Home
ARTIST:   Dixie Chicks
RATING:   9
RIPPED:   T

TITLE:    Fly
ARTIST:   Dixie Chicks
RATING:   8
RIPPED:   T

TITLE:    Roses
ARTIST:   Kathy Mattea
RATING:   7
RIPPED:   T
```

Эта функция может выглядеть так:

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

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

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

Второй аргумент `FORMAT` – формат строки; он может содержать как символьный текст, так и управляющие команды, контролирующие работу этой функции, например то, как она должна интерпретировать остальные аргументы. Команды, управляющие форматом вывода, начинаются со знака тильды (`~`) (так же, как управляющие команды *printf* начинаются с `%`). `FORMAT` может принимать довольно много таких команд, каждую со сво-