Предисловие переводчиков

Я познакомился с редактором Gnu Emacs в 2000 году, приблизительно через полгода после того, как начал изучение операционной системы Linux. И с тех пор очень и очень редко покидаю эту замечательную, расширяемую, самодокументированную среду реального времени:).

Понемногу, по мере сил, времени и возможностей начал изучать Emacs Lisp, ведь работать в Emacs и не пытаться его настроить под свои конкретные нужды очень нелегко. Правда тут следует предупредить начинающих программировать на языке Лисп. Этот язык очень красив, слишком красив для нашей сегодняшней российской действительности, и если им очень сильно увлекаться, то расплата следует незамедлительно, сразу теряешь интерес ко всем другим менее гибким языкам программирования (C, C++, shell, perl). Поэтому сильно увлекающимся программистам я советую ознакомиться только с первыми пятью главами (Обработка Списков, Практика Вычислений, Написание функций, Что такое буфер и Первые трудности), а также с главой Инициализация Emacs всего этого будет достаточно для начальной настройки и перестройки Emacs.

Перевод этой книги я начал, решив совместить приятное с полезным, прочесть несложную книжку и попрактиковаться в переводе. О качестве перевода судить вам. С нетерпением (так принято обычно писать) жду ваших отзывов и предложений по адресу dubkov@rbcmail.ru.

Предисловие

Большая часть интегрированной среды GNU Emacs написана на языке программирования Emacs Lisp. Код написанный на этом языке программирования — это программное обеспечение, набор инструкций, которые говорят компьютеру что делать, когда вы даете эти команды. Emacs сконструирован так, что вы можете писать новый код на Emacs Lisp и легко устанавливать его как расширение к редактору. Вот почему Emacs называют "расширяемым редактором".

(На самом деле, поскольку Emacs может делать много больше, чем просто редактировать текст, его по праву можно назвать "расширяемой средой редактирования", но эта фраза слишком труднопроизносима. Также, все, что вы делаете в Emacs: находите даты по календарю Майа и фазы луны, упрощаете полином, отлаживаете программы, управляете файлами, читаете электронную почту, пишете книги — все эти действия являются редактированием в самом общем смысле слова).

Хотя о языке Emacs Lisp обычно вспоминают только в связи с текстовым редактором, это полноправный язык программирования. Вы можете использовать его так же как и любой другой язык программирования.

Возможно, вы хотите научиться программированию, или добавить какую-нибудь полезную функцию к Emacs; может быть, вы хотите стать программистом. Введение в Emacs Lisp написано, чтобы помочь вам в самом начале: оно станет вашим помощником в изучении основ программирования и, что более важно, покажет вам, как в дальнейшем вы сможете обучаться самостоятельно.

Как читать этот текст

В этой книге вы увидите небольшие простенькие программы, которые можно опробовать не выходя из Emacs. Если вы читаете этот документ с помощью встроенной в Emacs справочной системы Info, то вы можете запускать программы по мере их появления. (Это очень легко сделать, и я научу вас этому, когда появятся примеры). Вместо этого вы можете читать эту книгу в печатном виде сидя рядом с компьютером, на котором запущен Emacs. (Именно так я обычно делаю; я очень люблю книги). Если у вас нет под боком Emacs, то вы все равно сможете с пользой читать эту книгу, но в таком случае, лучше относиться к ней как к художественному произведению или как к путеводителю в незнакомую страну: интересно, но лучше все-таки побывать там.

Большая часть этой книги посвящена изучению и объяснению программ, используемых в GNU Emacs. Такой способ выбран не случайно: во-первых, я хочу, чтобы вы познакомились с настоящим, работающим кодом (функциями которыми вы пользуетесь каждый день); и во-вторых, чтобы вы познакомились с тем, как устроен Emacs. Очень полезно понимать как работает ваш редактор. Также я надеюсь, что у вас появится желание исследовать исходный код, написанный на Emacs Lisp, входящий в состав GNU Emacs. Благодаря этому вы сможете многому научиться. GNU Emacs — это сокровищница знаний, охраняемая драконом.

Кроме изучения Emacs как редактора и Emacs Lisp как языка программирования, примеры и исследование исходных кодов Emacs дадут вам возможность познакомиться с Emacs, как с интегрированной средой программирования. GNU

Emacs очень помогает при разработке программного обеспечения, поскольку предоставляет инструменты, с которыми вам будет удобно работать, например, сочетание клавиш M-. (Это сочетание запускает команду find-tag). Вы также научитесь использовать буфера и другие объекты, которые являются частью среды редактирования. Обучение этим возможностям Emacs похоже на изучение новых маршрутов в вашем родном городе.

Кроме этого я написал несколько программ в качестве более сложных примеров. Хотя это только примеры, они являются самыми настоящими программами. Я сам использую их. Другие люди используют их. Вы тоже можете использовать их. Если не считать фрагментов программ, которые приведены здесь только в "учебных целях, то, что вы увидите в этой книге, используется в реальной жизни. В этом огромное преимущество Emacs Lisp — его легко приспособить для повседневной работы.

Наконец, я надеюсь поделиться богатым опытом использования Emacs, чтобы вы могли понять аспекты программирования, которые вы еще не знаете. Вы часто сможете использовать Emacs для решения какой-нибудь задачи или для выяснения, как сделать что-то иным способом. Emacs придаст вам уверенность[1] в своих силах, а это не только приятно само по себе, но и является огромным преимуществом.

Для кого это написано

Этот текст написан как элементарное введение в программирование для людей, которые сами не являются профессиональными программистами. Если вы профессиональный программист, то вас может не устроить эта книга. Может быть, вы уже стали специалистом в чтении справочных руководств, и вас будет раздражать организация этого текста.

Опытный программист, который просмотрел этот текст сказал мне:

> *Я предпочитаю учиться из справочников. Я "ныряю" в каждый параграф и "вдыхаю" в промежутке между ними.*
> *Когда я заканчиваю параграф, то полагаю, что тема закрыта, и я знаю все, что нужно (есть исключения, когда следующий параграф начинает говорить об этом более детально). Я ожидаю, что хорошо написанное руководство не будет избыточным и будет иметь отлично организованный индекс и содержать ссылки на места, где я смогу найти всю необходимую мне информацию.*

Это введение написано не для такого человека!

Во-первых, я пробую сказать все как минимум три раза: первый раз — чтобы представить что-нибудь; второй раз — чтобы показать это в контексте; и третий раз — чтобы напомнить или показать информацию в другом ракурсе.

Во-вторых, я никогда не помещаю всю информацию о новом понятии в одном месте, тем более в одном параграфе. По моему мнению это возлагает слишком тяжелую ношу на читателя. Вместо этого я пробую объяснить только то, что вам надо знать к этому времени. (Иногда я включаю немного дополнительной информации, откладывая формальное объяснение до более позднего времени).

Когда вы будете читать этот текст, не ждите, что вы научитесь всему с первого раза.

1 Даже сверхуверенность. (Прим. переводчика)

Часто вы только, так сказать, "поверхностно" познакомитесь с некоторыми представленными темами. Я надеюсь, что неплохо структурировал текст и дал вам достаточно подсказок, для того, чтобы вы смогли понять, что более важно, и сконцентрировались на этом.

Иногда вам все таки придется "углубляться" в некоторые параграфы — нет другого способа усвоить их. Но я пытался уменьшить число таких мест в тексте. Эта книга более походит на пологий холм, чем на отвесную гору.

Это введение в Программирование на Emacs Lisp имеет вспомогательный документ, section 'The GNU Emacs Lisp Reference Manual' in The GNU Emacs Lisp Reference Manual. Справочное руководство содержит намного больше деталей, чем данная книга. В справочнике вся информация об одной теме сконцентрирована в одном месте. Вы можете использовать это руководство, если вы похожи на программиста о котором я упоминал выше. И конечно, после того как вы прочтете это Введение, то вы поймете, что GNU Emacs Lisp Reference Manual весьма полезен, когда вы пишете свои собственные программы.

Происхождение Emacs Lisp

Лисп был разработан в конце 1950-х годов в Массачусетском Технологическом Институте в результате проекта по исследованию искусственного интеллекта. Огромная мощь языка Лисп сделала его подходящим также и для других задач, например, для написания команд редактирования.

GNU Emacs Lisp в основном произошел от Maclisp, который был создан в МТИ в 1960-е годы. И кое-где основывается на Common Lisp, который стал стандартом в 1980-е годы. Однако Emacs Lisp намного проще чем Common Lisp. (В стандартном дистрибутиве Emacs есть файл расширения 'cl.el', который добавляет к Emacs Lisp много возможностей Common Lisp).

Замечание для новичков

Если вы не используете GNU Emacs, то прочтение этого документа все равно принесет вам пользу. Однако я настоятельно рекомендую вам освоить Emacs. Вы сможете научиться использовать Emacs с помощью встроенной обучающей программы (Tutorial). Чтобы запустить ее нажмите C-h t. (Это значит что вы нажимаете и отпускаете клавишу CTRL и h одновременно, а потом нажимаете и отпускаете клавишу t).

Также я часто буду упоминать стандартные команды Emacs, перечисляя клавиши, которые вы должны нажать, чтобы запустить эту команду и потом даю имя команды в скобках, например: M-C-\ (indent-region). Это означает, что обычно команда indent-region запускается нажатием сочетания клавиш M-C-\. (Вы можете, если захотите, изменить сочетания клавиш, которые надо набрать, чтобы запустить команду; это называется привязка. See section Таблицы ключей.) Сокращение M-C-\ означает, что вы одновременно нажимаете клавиши META, CTRL и \. Иногда подобные комбинации называют клавиатурными аккордами, поскольку это похоже на аккорды при игре на пианино. Если на вашей клавиатуре нет ни клавиши META, ни ALT, то в качестве префикса можно использовать клавишу ESC. В этом случае M-C-\ означает, что вы нажимаете и отпускаете клавишу ESC, а потом одновременно нажимаете и отпускаете клавиши CTRL и \.

Если вы читаете этот документ в Info с помощью GNU Emacs, то вы можете пролистать весь этот документ, нажимая клавишу пробела SPC. (Чтобы изучить Info, нажмите C-h i и затем выберите пункт Info).

Замечание по терминологии: когда я использую слово Лисп, я обычно имею в виду все различные диалекты Лиспа, но, когда я говорю о Emacs Lisp, я имею в виду именно Emacs Lisp.

Благодарности

Я искренне благодарен всем кто помог мне с этой книгой. Выражаю особую признательность Jim Blandy, Noah Friedman, Jim Kingdon, Roland McGrath, Frank Ritter, Randy Smith, Richard M. Stallman, и Melissa Weisshaus. Благодарю также Philip Johnson и David Stampe за их терпеливую поддержку. Все ошибки исключительно мои собственные.

1. Обработка списков

С первого взгляда Лисп кажется очень необычным языком программирования. В программах на Лисп очень много круглых скобок. Некоторые люди даже заявляют, что сокращение Lisp происходит от "Lots of Isolated Silly Parentheses"[2]. Но это утверждение несправедливо. LISP означает LISt Processing[3] и этот язык программирования обрабатывает списки (и списки списков), располагая их между круглых скобок. Круглые скобки ограничивают список. Иногда перед списком ставится один апостроф `'". Списки — основа Лиспа.

1.1 Списки в Лисп

В Лиспе список выглядит следующим образом: `'(роза фиалка маргаритка лютик)`. Перед этим списком поставлен один апостроф. Этот список можно переписать по другому, именно так в дальнейшем и будут выглядеть списки с которыми вам придется иметь дело:

```
'(роза
фиалка
маргаритка
лютик)
```

Элементами этого списка являются названия четырех разных цветов, разделенные друг от друга пробелами и окруженные круглыми скобками, подобно цветам в саду с каменной оградой.

Списки могут также содержать числа, например: `(+ 2 2)`. В этом списке первый элемент это знак плюс +, за которым следуют два числа, разделенные пробелами.

В Лиспе и данные, и программы представляются одинаковым образом, то есть в виде списков, состоящих из слов, чисел или других списков, разделенные пробельными символами и окруженные круглыми скобками. (Так как программы выглядят как данные, то одна программа легко может служить данными для другой программы — это очень мощное свойство Лиспа). (Кстати, эти два замечания в скобках не являются допустимыми списками Лиспа, поскольку они содержат знаки препинания ';' и '.').

Вот другой список, в этот раз со списком внутри себя:

```
'(этот список содержит список (внутри себя))
```

Элементами этого списка являются слова 'этот', 'список', 'содержит' и список '(внутри себя)'. Внутренний список составлен из слов 'внутри', 'себя'.

1.1.1 Атомы Лиспа

То, что мы называем словами, в Лиспе называют атомами. Этот термин происходит от исторического значения слова атом, что значит "неделимый". Что касается Лиспа, то слова, которые мы использовали в списках, нельзя разделить на меньшие части; то же самое можно сказать о числах и одиночных символах, таких как '+'. С другой

2 Множество изолированных глупых скобок. (Прим. переводчика)

3 Обработка списков. (Прим. переводчика)

стороны, в отличие от атома список можно разложить на составные части. (car cdr & cons: основные функции.)

В списке атомы разделяются друг от друга пробелами и могут находиться вплотную к скобкам.

Если рассматривать это с технической точки зрения, список в Лиспе состоит из круглых скобок, окружающих атомы, другие списки, или и списки, и атомы, разделенные пробелами. Список может иметь только один атом внутри себя или не иметь совсем ничего. Список, который ничего не содержит, выглядит следующим образом: () и называется пустым списком. В отличие от всего остального, пустой список считается и атомом, и списком одновременно.

Печатное представление и атомов, и списков называют символическим выражением, или говоря точнее s-выражениями. Слово выражение может относиться печатному представлению, к атому или списку, как они содержатся внутри компьютера. Люди часто используют термин выражение не по назначению. (Также во многих книгах по Лиспу, слово форма используется, как синоним слова выражение).

Кстати, атомы, из которых состоит наша вселенная, были названы так, когда их считали неделимыми; но позже выяснилось, что это не совсем правильно. Атом можно расщепить на две части примерно одинакового размера, или он может распасться сам. Следовательно, физические атомы были названы преждевременно, до того, как прояснилась их истинная природа. В Лиспе некоторые виды атомов, такие как массивы, можно разделить на части, но механизм разделения отличается от механизма разделения списка на составные части. Когда дело касается операций на списках, атомы списка неразделимы.

Как и в английском языке, значения букв, составляющих название атома Лиспа, отличается от значения букв, составляющих слово. Например, южно-американский ленивец по английски называется 'ai', и это полностью отличается от двух букв 'a' и 'i'.

В природе существует много различных видов атомов, но в Лиспе существует только несколько: например, числа, такие как 37, 511, или 1729; или символы,такие как '+', 'foo', 'forward-line'. Слова, которые мы перечислили выше, являются символами. В обычных разговорах программистов слово "атом" используется не так часто, потому что программисты обычно пытаются быть более конкретны при указании типов атомов, которые они используют. Программирование на Лиспе в основном имеет дело с символами (и иногда с числами) внутри списков. (Кстати, предыдущие четыре слова в скобках являются правильным списком для Лиспа, так как он состоит из атомов, которые в этом случае являются символами, разделенными пробелами и закрытые в скобки, без каких либо знаков препинания).

Добавим, что текст между двойными кавычками (даже предложения или параграфы) считается одним атомом. Например:

```
'(этот список включает "текст между двойными кавычками.")
```

В Лиспе весь текст между двойными кавычками, включая знаки препинания и пробельные символы, считается атомом. Этот вид атома называется строка (от "строка символов") и обычно используется для сообщений, которые компьютер печатает для того, чтобы пользователь мог их прочесть. Строки отличаются от чисел

и символов и используются по-другому.

1.1.2 Пробелы в списках

Количество пробелов в списке не имеет значения. С точки зрения синтаксиса Лиспа,

```
'(это одинаковые
   списки)
```

это точно такой же список как и:

```
'(это одинаковые списки)
```

Для Лиспа оба этих списка абсолютно одинаковы и состоят из символов 'это', 'одинаковые', 'списки' именно в таком порядке.

Дополнительные пробелы и символы новой строки используются для того, чтобы сделать список более удобным для чтения людьми. Когда Лисп считывает выражение, он избавляется от лишних пробелов (но для того, чтобы атомы можно было воспринимать раздельно, между ними должен быть по меньшей мере один пробел.)

Это может показаться удивительным, но мы рассмотрели почти все виды списков Лиспа! Любой другой список в Лиспе выглядит более или менее похожим на один из выше рассмотренных примеров, разве что список может быть больше и выглядеть более сложным. Короче говоря, список находится между круглых скобок, строка находится между двойными кавычками, символ выглядит как слово, а число выглядит как число. (В определенных случаях могут использоваться квадратные скобки, точки и некоторые другие специальные символы; однако мы можем зайти довольно далеко и без них).

1.1.3 GNU Emacs помогает вам программировать

Если вы набираете Лисп-выражение в GNU Emacs, используя основной режим Emacs Lisp или Lisp Interaction, то вы можете воспользоваться некоторыми командами, чтобы отформатировать Лисп-выражение так, чтобы его было легче воспринимать при чтении. Например, нажатие клавиши TAB автоматически выравнивает строку, в которой находится курсор, вправо. Команда для выравнивания области текста обычно привязана к сочетанию клавиш M-C-\. Выравнивание производится так, чтобы вы могли сразу увидеть, к какому списку принадлежат данные выражения списка (элементы вложенных списков расположены правее, чем элементы списков более верхнего уровня).

Кроме этого, когда вы печатаете закрывающую скобку, Emacs моментально переводит курсор к соответствующей ей открывающей скобке. Это очень полезная функция редактора, так как в каждом списке, который вы будете набирать, все скобки должны быть сбалансированы. (See section 'Major Modes' in The GNU Emacs Manual, для получения дополнительной информации о режимах Emacs).

1.2 Запуск Программы

Список в Лиспе (любой список) — это программа, готовая к запуску. Если вы запустите ее (на жаргоне Лиспа вычислите), то компьютер сделает следующее: 1) ничего не сделает, а только вернет вам сам список; 2) отобразит сообщение об

ошибке; 3) обработает первый символ в списке как команду сделать что-либо. (Обычно, именно последнее — это то, что вы хотите на самом деле!)

Одиночный апостроф ', который я ставил перед некоторыми из списков в предыдущих абзацах, называют цитатой (quote) и, когда перед списком стоит апостроф, это сигнал для Лиспа ничего не делать со списком, а взять его как есть. Но если перед списком нет апостофа, то первый символ в списке имеет особое значение: — это команда, которую компьютер должен выполнить. (В Лиспе эти команды называют функциями). Перед списком (+ 2 2) нет апострофа, поэтому Лисп будет рассматривать + как инструкцию, сделать что-то с остальными символами списка; в этом примере он просто сложит последующие числа.

Если вы читаете это в Info внутри Gnu Emacs, то вот как вы можете вычислить такой список: расположите курсор сразу после правой скобки следующего ниже списка и нажмите сочетание клавиш C-x C-e:

```
(+ 2 2)
```

Вы увидите, что в эхо-области появится число 4. (Говоря на жаргоне Лиспа, вы только что "вычислили список". Эхо-область — это строка внизу экрана, в которой отображается текст). Сейчас попробуйте сделать то же самое со списком, перед которым стоит апостроф: поместите курсор сразу за следующим списком и нажмите C-x C-e:

```
'(это список перед которым стоит апостроф)
```

В этом случае в эхо-области появится (это список перед которым стоит апостроф).

В обоих случаях вы давали команду "вычислить выражение" встроенной в GNU Emacs программе (которая называется интерпретатор Лиспа). Имя интерпретатора Лисп произошло от названия задачи, которую выполняет человек, объясняющий выражение, т.е. "интерпретирующий" его.

Вы можете также вычислить атом, который не является частью списка (не окружен круглыми скобками), и снова интерпретатор Лиспа переведет это из формы, понятной человеку, на язык компьютера. Но до того, как обсуждать это (see section 1.7 Переменные), мы рассмотрим, что сделает интерпретатор Лиспа, когда вы совершите ошибку.

1.3 Получаем сообщение об ошибке

Чтобы вы не беспокоились из-за случайно совершенной ошибки, мы сейчас дадим команду интерпретатору Лиспа, которая выдаст сообщение об ошибке. Это совсем безвредно; на самом деле мы часто будем выдавать сообщения об ошибках, когда это нам необходимо. Если вы понимаете терминологию, сообщения об ошибках могут быть очень полезными. Вместо того, чтобы называть их сообщения об "ошибках", назовем их "подсказками". Они как указатели для путешественника в незнакомую страну — разобраться в них сначала тяжело, но когда все ясно, они указывают путь.

Вот, что мы сейчас сделаем: мы вычислим список, перед которым не стоит апостроф, а первый элемент списка не является командой. Этот список мы уже использовали, сейчас мы уберем одиночный апостроф. Расположите курсор после

выражения и нажмите C-x C-e;

```
(это список перед которым стоит апостроф)
```

В этот раз вы увидите, что в эхо-области появится следующее сообщение:

```
Symbol's function definition is void: это
```

(Также терминал может подать вам звуковой сигнал или мигнуть. Это чтобы привлечь ваше внимание). Сообщение исчезнет как только вы нажмете какую-нибудь клавишу, даже чтобы просто переместить курсор.

Основываясь на том, что мы уже знаем, мы можем полностью понять это сообщение об ошибке. Вы знаете смысл слова 'Symbol (символ)'. В этом случае оно относится к первому атому списка, слову 'это'. Слово 'функция' мы уже упоминали ранее. Это очень важное слово. Для себя мы можем определить его следующим образом: функция — это набор инструкций для компьютера, которые сообщают ему, что необходимо сделать. (Строго говоря, символ говорит компьютеру, где найти инструкции, но это усложнение мы пока будем игнорировать).

Сейчас мы можем понять сообщении об ошибке: 'Symbol's function definition is void: это'. Символу (то есть слову 'это'), не соответствует никакой набор инструкций, которые мог бы выполнить компьютер.

Немного странное значение сообщения, 'function definition is void' (определение функции — пусто), объясняется тем, как реализован Emacs Lisp — то есть, когда с символом не связано никакого определения функции, то место, где должно содержаться это определение — пусто.

С другой стороны, так как мы смогли успешно сложить 2 и 2, то вычислив выражение (+ 2 2), можно полагать, что с символом + связан некоторый набор инструкций, выполняя которые, компьютер складывает числа, следующие за +.

1.4 Имена символов и определения функций

Мы сейчас обсудим одну особенность Лиспа, основываясь на приобретенных нами знаниях: символ, такой как +, сам не является набором инструкций, которые должен выполнить компьютер. Вместо этого символ используется (возможно временно) как указатель на определение или набор инструкций. То есть через имя можно найти эти инструкции. Имена людей используются похожим образом. Ко мне можно обратиться, как к 'Боб'; однако я — это не буквы 'б', 'о', 'б', а особая жизненная форма, обладающая сознанием. Я — это не имя, но его можно использовать, чтобы обратиться ко мне.

В Лиспе один и тот же набор инструкций может быть связан с несколькими именами. Например, инструкции для сложения чисел можно связать как с символом plus или плюс, так и с символом + (так и есть в некоторых диалектах Лиспа). Среди людей ко мне можно обращаться как к 'Роберту', так и к 'Бобу'.

С другой стороны, к символу можно прикрепить только одно определение функции. В противном случае компьютер не смог бы выбрать, какое определение использовать. Если бы так было среди людей, то только один человек в мире мог носить имя 'Боб'. Однако определение функции, к которому относится имя, можно легко изменить.

Поскольку Emacs Lisp огромен, то символы принято называть таким образом, чтобы сразу было ясно, к какой части Emacs принадлежит эта функция. Так все имена функций, которые имеют дело с Texinfo начинаются с 'texinfo-', а функции, которые имеют дело с электронной почтой, начинаются с 'rmail-'.

1.5 Интерпретатор Лиспа

Основываясь на том, что изучили, мы можем понять, что делает интерпретатор Лиспа, когда мы даем ему команду вычислить выражение. Первым делом интерпретатор смотрит, есть ли апостроф перед списком; если да, то он только возвращает выражение как есть. Если же апострофа нет, то интерпретатор анализирует первый элемент списка и пытается определить по этому символу определение функции. Если функция существует, то интерпретатор выполняет соответствующие ей инструкции. В противном случае он выдает сообщение об ошибке.

Вот как работает интерпретатор Лиспа. Это очень просто. Мы скоро рассмотрим некоторые нюансы, но, то что вы изучили, это основы. Конечно, чтобы написать программу на Лиспе, вам необходимо знать, как определять функцию и связывать ее с именем, и как сделать это, не запутав компьютер и себя.

Вот и первый нюанс. Кроме списков, интерпретатор Лиспа может вычислить символ, перед которым нет апострофа, и который не заключен в круглые скобки. В этом случае интерпретатор Лиспа попытается определить значение символа, считая его переменной. Такая ситуация описана в разделе о переменных в данной главе. (See section 1.7 Переменные.)

Второй нюанс связан с тем, что некоторые функции необычны и работают несколько странно. Их называют особыми формами. Их не очень много и они используются в специальных случаях, например, когда надо определить новую функцию. В следующих нескольких главах мы рассмотрим некоторые из наиболее важных особых форм.

Третий и последний нюанс: если функция, которую анализирует интерпретатор Лиспа, не является особой формой, и если она часть списка, то интерпретатор Лиспа смотрит, есть ли внутри этого списка вложенные списки. Если есть, тогда интерпретатор вначале вычисляет внутренние списки, а затем внешние. Он всегда в первую очередь обрабатывает внутренние списки. Это делается для того, чтобы выяснить, какой они возвращают результат. Этот результат может использоваться в списке верхнего уровня.

Если внутренних списков нет, то интерпретатор вычисляет выражения слева направо, от одного выражения к другому.

1.5.1 Байт-компиляция

Интерпретатор Лиспа может обрабатывать два вида данных: код, который может прочесть человек (мы и будем рассматривать его в этой книге) и особо обработанный код, не предназначенный для чтения людьми, так называемый байт-компилированный код. Байт-компилированный код выполняется гораздо быстрее чем обычный.

Вы можете перевести обычный код в байт-компилированный, запустив команду для компиляции, например, byte-compile-file. Байт-компилированный код обычно хранится в файлах, имеющих расширение '.elc', а файлы, которые можно читать, имеют расширение '.el'. Вы можете увидеть оба типа файлов в директории 'emacs/lisp'.

На практике многое из того, чем мы будем заниматьсяw{ ---} настраивать и расширять Emacsw{ ---} не потребует байт-компиляции; и я не буду обсуждать здесь эту тему. See section 'Byte Compilation' in The GNU Emacs Lisp Reference Manual, для более полного описания байт-компиляции.

1.6 Вычисление

Когда интерпретатор Лиспа анализирует выражение, то говорят, что он занимается вычислением. То есть интерпретатор "вычисляет выражение". Я уже использовал этот термин несколько раз раньше. Слово пришло из повседневного языка, "выяснить значение или количество; оценить", согласно "Webster's New Collegiate Dictionary".

После вычисления выражения, интерпретатор Лиспа обычно возвращает значение, которое вычислил компьютер, выполняя инструкции, найденные в определении функции; иногда он прерывает выполнение функции и выдает сообщение об ошибке. (Интерпретатор может также выполнить не ту функцию, или он может начать повторять что-нибудь снова и снова — зайти в так называемый "бесконечный цикл". Эти действия менее обычны, и мы можем проигнорировать их). В основном, интерпретатор возвращает какое-нибудь значение.

Одновременно с возвратом значения интерпретатор может сделать что-нибудь еще, например, переместить курсор или скопировать файл — этот дополнительный род деятельности называют побочным эффектом. Действия, которые людьми считаются важными, такие как печать результатов, часто являются только "побочными эффектами" для интерпретатора Лиспа. Это может звучать несколько необычно, но позже мы довольно легко научимся использовать побочные эффекты.

В общем вычисление символического выражения обычно заставляет интерпретатор Лиспа возвратить какое-нибудь значение и, возможно, выполнить побочное действие; или прервать выполнение, отобразив соответствующее сообщение об ошибке.

1.6.1 Вычисление внутренних списков

Если мы вычисляем список, внутри которого есть другой список, то при вычислении внешнего списка может использоваться значение, которое возвращается при вычислении внутреннего списка. Это объясняет, почему внутренние выражения вычисляются в первую очередь — значения, которые они возвращают используются внешними выражениями.

Мы можем исследовать этот процесс, вычислив еще один пример на сложение чисел. Расположите курсор после следующего выражения и нажмите C-x C-e:

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

В эхо-области появится число 8.

Это произошло, потому что интерпретатор Лиспа вначале вычислил внутреннее выражение, (+ 3 3), вычисление которого вернуло значение 6; а затем он вычислил внешнее выражение, которое теперь можно представить как (+ 2 6), вычисление которого вернуло значение 8. Так как в этом выражении больше нечего вычислять, то интерпретатор отобразил это значение в эхо-области.

Сейчас уже довольно легко понять название команды, которая запускается клавишами C-x C-e — она называется eval-last-sexp. Буквы sexp — это сокращение от "symbolic expression (символьное выражение)", а eval сокращение от "evaluate". Эта команда означает 'вычислить последнее символическое выражение' ("evaluate last symbolic expression").

В порядке эксперимента вы можете сейчас попробовать вычислить это же выражение, расположив курсор на пустой строке сразу за выражением, или внутри него.

Вот копия предыдущего выражения:

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

Если вы расположите курсор в начало пустой строки сразу за этим выражением и нажмете C-x C-e, то в эхо-области также отобразится 8. Сейчас попробуйте расположить курсор внутри выражения. Если вы поставите его за предпоследней скобкой (будет казаться, что курсор поверх последней скобки), в эхо-области появится 6! Это потому что сейчас мы вычислили выражение (+ 3 3).

Теперь расположите курсор сразу за каким-нибудь числом. Нажмите C-x C-e и вы получите само это число. В Лиспе, если вы вычисляете число, то возвращается само это число (этим числа отличаются от символов). Если вы вычислите список, начинающийся с символа +, то интерпретатор вернет вам значение, которое он получит, выполняя инструкции связанные, с этим символом. Если вы попытаетесь вычислить сам этот символ, произойдет что-то другое, и это мы рассмотрим в следующем разделе.

1.7 Переменные

В Лиспе к любому символу можно прикрепить какое-нибудь значение, точно также как к нему можно прикрепить функцию. Это две большие разницы. Функция — это набор инструкций, которые компьютер должен выполнить. С другой стороны, значение — это нечто, например, число или имя, то что может изменяться (vary) (поэтому такой символ назвали переменной (variable)). Значением символа может быть любое выражение, допустимое в Лиспе — такое как число, символ, список, или строка. Символ, который имеет значение, часто называют переменной.

С символом одновременно может быть связано и определение функции и какое-нибудь значение. Это два различных понятия. Похоже на то, как имя Кембридж носит город в штате Массачусетс, и с этим именем связана еще кое-какая информация — многие считают его "крупнейшим центром программирования".

(Кстати, в Emacs Lisp с символом могут быть связаны также список свойств и строка документации; они обсуждаются позже).

Есть другой способ разграничить эти понятия — представить себе символ как стол с выдвижными ящичками. Определение функции можно положить в один ящик, значение переменной в другой, и так далее. То, что мы положим в ящик для переменной, не повлияет на то, что находится в ящике для функции, и наоборот.

Переменная fill-column — пример символа, которому соответствует некоторое значение (в каждом буфере GNU Emacs этому символу присвоено какое-нибудь значение, обычно 70 или 72.) Чтобы узнать значение этого символа, вычислите его сами. Если вы читаете это в Info внутри GNU Emacs, вы можете сделать это, расположив курсор за символом и нажав C-x C-e:

```
fill-column
```

После того, как я нажал C-x C-e, Emacs напечатал в эхо-области число 72. Это значение я присвоил fill-column, когда писал эту книгу. В вашем случае оно может быть другим. Заметьте, что значение, которое вернулось после вычисления переменной, отпечаталось точно так же, как и значение, которое бы вернула функция. С точки зрения интерпретатора Лиспа возвращенное значение есть только возвращенное значение. Какого рода выражение мы вычисляли уже неважно, когда значение известно.

Символу может соответствовать любое значение или, используя жаргон Лиспа, мы можем связать с переменной любое значение — число, такое как 72; строку, такую как эта; список, такой как (ель сосна дуб); мы можем даже связать с переменной определение функции.

Значение можно связать с переменной несколькими способами. Дополнительную информацию ищите в See section Присваиваем Переменной Значение.

Заметьте, что вокруг слова fill-column не было скобок, когда мы вычислили его, чтобы найти связанное с ним значение. Это потому, что мы не собирались использовать его как функцию. Если бы fill-column было бы первым или единственным символом в списке, то интерпретатор Лиспа попытался бы найти определение функции, связанное с этим символом. Но с fill-column не связана никакая функция. Попробуйте вычислить следующее выражение:

```
(fill-column)
```

Вы получите следующее сообщение об ошибке:

```
Symbol's function definition is void: fill-column
```

1.7.1 Сообщение об ошибке для пустого символа

Если вы попытаетесь вычислить символ, с которым не связано никакое значение, то вы получите сообщение об ошибке. Вы можете проверить это, продолжив эксперимент с нашим примером на сложение 2 плюс 2. В следующем выражении расположите курсор после +, перед первым числом 2, и нажмите C-x C-e:

```
(+ 2 2)
```

Вы получите следующее сообщение об ошибке:

```
Symbol's value as variable is void: +
```

Оно отличается от первого сообщения об ошибке, которое мы видели: 'Symbol's function definition is void: это'. Во втором случае с символом, как с переменной, не было связано никакого значения; в первом случае с символом (то есть со словом 'это') не было связано определение функции.

В этом эксперименте с + мы заставили интерпретатор Лиспа вычислять символ + как переменную, вместо того, чтобы искать определение функции, связанное с этим символом. Мы сделали это, расположив курсор сразу за символом, а не после скобки, закрывающей список, как было раньше. Поэтому интерпретатор Лиспа вычислил предыдущее s-выражение, которое в нашем случае было символом +.

Так как с + не связано никакого значения, как с переменной, а только определение функции, то мы и получили сообщение об ошибке — значение символа как переменной пусто.

1.8 Аргументы

Чтобы понять, как функция получает необходимую ей информацию, давайте снова рассмотрим наш старый добрый пример: 2 + 2. В Лиспе это записывается следующим образом:

```
(+ 2 2)
```

После вычисления этого выражения, в эхо-области появится число 4. Интерпретатор Лиспа сложил числа, которые следовали за символом +.

Числа, которые складывает функция +, называют аргументами этой функции. Аргументы — это информация, которую мы сообщили или передали этой функции.

Слово "аргумент" здесь используется в математическом, а не повседневном смысле. Здесь оно обозначает информацию, переданную функции, в нашем случае к функции +. В Лиспе аргументы функции — это атомы или списки, которые следуют за именем функции. Значения, возвращаемые вычислением этих атомов или списков, передаются в функцию. Разные функции требуют разного числа аргументов, некоторые функции совсем не требуют аргументов.

1.8.1 Типы Данных аргументов

Типы данных, которые будут передаваться в функцию, зависят от того, какого рода информацию она использует. Аргументы функции, такой как +, должны иметь численные значения, поскольку + складывает числа. Другие функции используют аргументы другого типа.

Например, функция concat связывает вместе, или объединяет две или более строк, чтобы получить одну строку. Ее аргументы строки. Конкатенация двух символьных строк abc, def выдаст в результате строку abcdef. Это можно проверить вычислив следующее:

```
(concat "abc" "def")
```

Значение, произведенное вычислением, этого выражения равно abcdef.

Аргументами такой функции как substring являются и строки, и числа. Эта функция возвращает часть строки, подстроку своего первого аргумента. Функция требует три аргумента. Первый аргумент — строка символов, второй и третий аргументы — это числа, которые отмечают начало и конец строки. Числа — это количество символов (включая пробелы и знаки пунктуации) от начала строки.

Например если вы вычислите следующее выражение:

```
(substring "Четыре черненьких чумазеньких чертенка." 7 29)
```

то в эхо-области появится фраза "черненьких чумазеньких" Аргументами функции substring были строка и два числа.

Отметим, что строка, которую мы передали в функцию substring, это один атом, даже если он был составлен из нескольких слов, разделенных пробелами. Лисп считает все, то заключено между двойными кавычками, частью строки, включая пробелы. Вы можете считать функцию substring как своего рода "разделитель атома" поскольку она извлекает часть из по-другому неделимого атома. Однако substring способна выделить подстроку только из строки, а с другого рода атомами, такими как символы или числа, она не работает.

1.8.2 Аргумент как значение переменной или списка

Аргументом может быть и символ, который возвращает значение, когда его вычислят. Например, когда мы вычисляем символ fill-column, то он возвращает число. Это число может использоваться в сложении. Поставьте курсор после следующего выражения и нажмите C-x C-e:

```
(+ 2 fill-column)
```

Значение, которое вы получите будет числом, которое на два больше чем значение fill-column. В моем случае это было 74, поскольку fill-column установлено в 72.

Как мы только что видели, аргументом может быть символ, который возвращает некоторое значение, когда его вычисляют. Кроме того, аргумент может быть списком, который возвращает значение, когда его вычисляют. Например в следующем выражении, аргументами функции concat являются строка "рыжие лисички.", а также список (+2 fill-column).

```
(concat  (+ 2 fill-column) " рыжие лисички.")
```

Когда я вычислил это выражение, в эхо-области появилось "74 рыжие лисички.". (Обратите внимание на пробелы перед словом 'рыжие', чтобы отделить их от числа).

1.8.3 Аргументов может быть много

Некоторые функции, такие как concat, + или *, принимают произвольное число аргументов. (* — это символ умножения). Это можно проверить, вычислив каждое из следующих выражений обычным образом. То, что вы увидите в эхо-области, мы будем печатать в тексте после знака '=>', вы можете считать этот знак как "вычисляется в".

В первый раз функции не имеют аргументов:

```
(+)        => 0

(*)        => 1
```

Теперь у каждой функции один аргумент:

```
(+ 3)      => 3

(* 3)      => 3
```

Сейчас уже по три аргумента:

```
(+ 3 4 5) => 12

(* 3 4 5) => 60
```

1.8.4 А если аргумент неправильного типа

Если вы попытаетесь передать функции аргумент неправильного типа, то интерпретатор Лиспа отобразит сообщение об ошибке. Например, функция + ожидает, что ее аргументами будут числа. В порядке эксперимента, мы передадим ей вместо числа символ привет, перед которым поставим апостроф. Расположите курсор после следующего выражения и нажмите C-x C-e:

```
(+ 2 'привет)
```

Когда вы сделаете это, появится сообщение об ошибке. Это произошло когда интерпретатор Лиспа попытался сложить 2 со значением возвращаемым 'привет, но это значение — сам символ привет, а не число. Складывать можно только числа. Поэтому + и не смог выполнить сложение своих аргументов.

Как обычно сообщение об ошибке содержит полезную информацию, помогающую исправить ее. Оно гласит:

```
Wrong type argument: integer-or-marker-p, привет
```

Первая часть сообщения ясна: — она говорит 'Wrong type argument'.[4] Затем идет загадочное жаргонное словечко 'integer-or-marker-p'. Этим словом интерпретатор пытается сообщить вам, какого типа аргументы ожидались функцией +.

Символ integer-or-marker-p означает, что интерпретатор Лиспа считал, что информация, представленная аргументом (значение аргумента) это целое число или маркер (особый объект представляющий позицию в буфере). Так как функция + может складывать только числа или маркеры, то интерпретатор проверяет или по другому тестирует типы аргументов на соответствие этим двум типам. (В Emacs положение курсора в буфере записывается в виде маркера. Когда вы ставите метку, нажимая C-@ или C-SPC Emacs запоминает положение в буфере и запоминает его в маркере). Маркер можно упрощенно считать числом символов до точки от начала буфера. В Emacs Лисп + можно использовать для сложения числовых значений

4 Неправильный тип аргумента. Прим. переводчика.

маркеров.

Буква 'p' в integer-or-marker-p — наследие практики ранних дней программирования на Лиспе. 'p' означает 'predicate' (предикат). На жаргоне, используемым первыми исследователями Лиспа, предикатом называли функцию, которая определяла правдиво ли какое-нибудь суждение и возвращала либо истину, либо ложь. Так 'p' означает, что integer-or-marker-p — это имя функции, которая проверяет свой аргумент на соответствие числу или маркеру и возвращает true(истина) или false(ложь). В Лиспе много функций, чьи имена которые оканчиваются на 'p', например, zerop, которая тестирует свой аргумент на равенство нулю, или listp — функция, проверяющая, является ли ее аргумент списком.

И наконец, последняя часть сообщения об ошибке символ привет. Это значение аргумента, который мы передали к +. Если бы мы передали в функцию сложения правильный тип объекта, то значением аргумента было бы число, например, 37, а не символ привет. Но тогда бы вы не получили такого интересного сообщения об ошибке.

1.8.5 Функция message

Как и +, функция message принимает переменное число аргументов. Ее обычно используют для отображения информационных сообщений пользователю. Мы опишем ее здесь, поскольку это одна из наиболее часто применяемых функций.

Сообщение отображается в эхо-области. Например, вы можете напечатать сообщение в эхо-области, вычислив следующий список:

```
(message "Это сообщение появится в эхо-области!")
```

Вся строка между двойными кавычками — это один аргумент, и он появится в эхо-области. (Обратите внимание, что само сообщение появится в эхо-области в двойных кавычках — это потому что вы видите значение, которое вернула функция message. В большинстве случаев использования функции message в программах, которые мы будем писать, текст, печатаемый в эхо-области — это побочный эффект, без кавычек).

Однако если в строке найдется символ '%s', то функция message не будет печатать его, а вместо этого просмотрит аргументы, которые следуют за строкой. Она вычислит второй аргумент и напечатает его значение на месте '%s'.

Вы можете проверить это, расположив курсор после следующего выражения и нажав C-x C-e:

```
(message "Имя этого буфера: %s." (buffer-name))
```

В моем случае в эхо-области появилось, "Имя этого буфера: *info*.". Функция buffer-name вернула имя буфера в виде строки, которую функция message вставила на место %s.

Чтобы напечатать значение в виде десятичного числа, вы должны вместо '%s' использовать '%d'. Например, чтобы напечатать в эхо-области значение fill-column, вычислите следующее:

```
(message "Значение переменной fill-column: %d." fill-column)
```

Когда я сделал это в моей эхо-области отобразилось "Значение переменной fill-column: 72."

Если в строке больше чем одна '%s', то значение первого аргумента после строки печатается на месте первого появления '%s', а значение второго аргумента — на месте второго появления '%s', и так далее. Например, если вы вычислите следующее:

```
(message "Ничего себе, в офисе %d %s!"
         (- fill-column 14) "розовых слонов")
```

тогда в вашей эхо-области появиться довольно забавное сообщение. У меня напечаталось: "Ничего себе, в офисе 58 розовых слонов!".

Выражение (- fill-column 14) при вычислении вернуло число, которое подставилось на место '%d'; а строка в двойных кавычках "розовых слонов", как единый аргумент, вставилась на место '%s'. (То есть строки между двойными кавычками при вычислении возвращает сами себя, как и числа).

Наконец, мы рассмотрим довольно сложный пример, который не только проиллюстрирует вычисление числа, но также покажет, как вы можете использовать вложенные выражения, чтобы получить текст, который заменит '%s':

```
(message "Он увидел %d %s"
         (- fill-column 34)
         (concat "рыжих " "прыгающих "
                 (substring
                  "Шустрые, лисицы весело прыгали." 9 14)))
```

В этом примере у функции message три аргумента — строка, "Он увидел %d %s", выражение (- fill-column 32), и выражение, которое начинается с функции concat. Значение, которое возвращается после вычисления (- fill-column 32), вставляется на место '%d', а значение от вычисления списка, где первый элемент concat, вставляется на место '%s'.

Когда я вычислил это выражение, в моей эхо-области появилось сообщение: "Он увидел 38 рыжих прыгающих лисиц".

1.9 Присваиваем переменной значение

Есть несколько способов, для того чтобы переменной можно было присвоить какое-нибудь значение. Например, с помощью функций set и setq. Или с помощью функции let (see section 3.6 let). (На Лисп жаргоне это называют связать переменную со значением.)

В следующем разделе мы не только опишем функции set и setq, но также покажем, как передавать аргументы.

1.9.1 Используя set

Чтобы связать с символом цветы список '(роза фиалка маргаритка лютик), вычислите

следующее выражение, расположив курсор после него и нажав C-x C-e.

```
(set 'цветы '(роза фиалка маргаритка лютик))
```

В эхо-области появится список (роза фиалка маргаритка лютик) . Это то, что вернула функция set. Как побочный эффект символ цветы теперь связан со списком — то есть, символ цветы, можно теперь считать переменной, значением которой является список. (Этот процесс между прочим показывает как побочный эффект для интерпретатора Лиспа для нас является основным. Это потому что каждая функция Лиспа должна возвращать какое-нибудь значение, но побочный эффект имеют только некоторые из них).

После вычисления выражения set, вы можете теперь вычислить символ цветы и посмотреть, какое он вернет значение. Расположите курсор после этого символа и нажмите C-x C-e.

```
цветы
```

Когда вы вычислите символ цветы, то в эхо-области появиться список (роза фиалка маргаритка лютик).

Вспомним, что если вы вычислите 'цветы — тот же символ, но на этот раз с апострофом, то перед ним в эхо-области появиться сам этот символ цветы. Ниже символ, перед которым мы поставили апостроф, так что проверьте это:

```
'цветы
```

Заметьте, что когда вы используете set, вам нужно ставить апостроф перед обеими аргументами set, если вы не хотите, чтобы интерпретатор вычислял их. В нашем случае мы не хотим, чтобы аргументы вычислялись: ни переменная цветы ни список (роза фиалка маргаритка лютик), поэтому перед каждым мы поставили апостроф. (Если бы мы не поставили апостроф перед первым аргументом set, интерпретатор вычислил бы его перед выполнением функции. Если с символом цветы не было бы связано никакое значение, вы бы получили сообщение об ошибке 'Symbol's value as variable is void'; если бы вычисление символа цветы все же вернуло бы какое-нибудь значение, тогда функция set попыталась бы связать список именно с этим возвращенным значением. Иногда так и надо делать; но такие ситуации весьма редки.)

1.9.2 Используя setq

На практике вы почти всегда ставите апостроф перед первым аргументом set. Сочетание set с первым аргументом, перед которым стоит апостроф, встречается так часто, что была введена особая форма setq. Эта особая форма действует точно также, как и set, но перед первым аргументом не надо ставить апостроф, так как он не вычисляется. Также добавлена еще одна полезная возможность — setq разрешает вам назначить одновременно значения нескольким переменным в одном выражении.

Чтобы установить значением переменной плотоядные список '(лев тигр леопард) с помощью setq, можно использовать следующее выражение:

```
(setq плотоядные '(лев тигр леопард))
```

Это тоже самое, что и использование set, но первый аргумент защищен от вычисления setq. ('q' в setq означает quote). С set выражение выглядело бы по-другому:

```
(set 'плотоядные '(лев тигр леопард))
```

Также setq можно применять для назначения разных величин разным переменным. Первый аргумент связывается со значением второго аргумента, третий аргумент связывается со значением четвертого аргумента и так далее. Например, вы можете использовать следующее выражение, чтобы связать символ деревья со списком деревьев, а список травоядных с символом травоядные:

```
(setq деревья '(сосна ель дуб клен)
      травоядные '(газель антилопа зебра))
```

(Это выражение можно было бы поместить в одну строку, но более длинные выражения не поместятся на страницу, кроме того людям легче читать красиво отформатированные списки.)

Хотя я использовал термин 'назначить', существует другой способ представления работы функций set и setq — представьте себе, что функции set и setq заставляют символ указывать на список. Этот последний способ представления очень обычен и в последующих главах я буду использовать его более часто, мы встретим по меньшей мере один символ в имени которого используется 'указатель'. Это название выбрано потому, что символ имеет значение — например, список, связанный с ним; или если по другому выразить, этот символ указывает на список.

1.9.3 Счетчик

Здесь пример как использовать setq в счетчике. Вы можете использовать его, чтобы сосчитать сколько раз ваша программа повторилась. Вначале назначьте переменной значение равное нулю, а затем добавляйте по единице каждый раз, когда программа повторяется. Чтобы выполнить это, вам понадобится переменная, которая будет служить счетчиком, и два выражения: первое setq, которое устанавливает счетчик в ноль; и второе — setq, которое увеличивает счетчик каждый раз, когда его вычисляют.

```
(setq counter 0)                ; Назовем это инициализацией.

(setq counter (+ counter 1))    ; Здесь увеличиваем счетчик.

counter                         ; А это счетчик.

(Текст после ';' --- это комментарий.)
```

Если вы сначала вычислите первое из этих выражений — инициализатор (setq counter 0), а затем третье выражение, counter, то в эхо-области появится 0. Если вы вычислите второе выражение для приращения счетчика — (setq counter (+ counter 1)), то счетчик получит значение 1. Так что если вы снова вычислите counter, то теперь в эхо-области появиться число 1. Каждый раз, когда вы будете вычислять второе выражение, значение счетчика будет увеличиваться.

Когда вы вычисляете выражение приращения, (setq counter (+ counter 1)), интерпретатор Лиспа вначале вычисляет внутренний список — то есть сложение. Для этого он должен вычислить переменную counter и число 1. После вычисления переменной counter, возвращается ее текущее значение. Затем оно вместе с единицей передается функции +, которая складывает их вместе. Итоговая сумма возвращается как значение этой функции и передается setq, которая связывает переменную counter с этим новым значением. Таким образом и изменяется значение этой переменной.

1.10 Резюме

Изучение Лиспа подобно восхождению на холм, где первая часть подъема самая трудная. Мы только что преодолели самую трудную часть — то что осталось, намного легче и вы почувствуете это, когда мы будем продвигаться вперед.

Если просуммировать:

- Программы на Лиспе состоят из выражений, которые являются списками или одиночными атомами.
- Списки состоят из нескольких атомов или списков, разделенных пробельными символами и окруженных круглыми скобками. Список может быть пустым.
- Атомы — это много-буквенные символы, такие как forward-paragraph; одно-буквенные символы, как +; символьные строки, ограниченные двойными кавычками; и числа.
- После вычисления числа возвращается само число.
- После вычисления строки, заключенной между двойными кавычками, возвращается сама строка.
- Если вы вычислите символ, то интерпретатор вернет связанное с ним, как с переменной, значение.
- Когда вы вычисляете список, интерпретатор Лиспа анализирует первый символ списка и затем ищет определение функции, связанное с этим символом. После этого выполняются инструкции этой функции.
- Одиночная кавычка ', говорит интерпретатору Лиспа, что следующее выражение не надо вычислять, а надо просто вернуть его, как оно есть.
- Аргументы — это информация, которую мы передаем функции. Аргументы функции получаются после вычисления оставшихся элементов списка, где первый элемент это имя функции.
- Любая функция всегда возвращает какое-нибудь значение после выполнения (если не было никакой ошибки); вдобавок она может также выполнить некоторые действия называемые "побочным эффектом". При выполнении многих функций главная цель — это побочный эффект.

1.11 Упражнения

Вот несколько простых упражнений:

- Вызовите сообщение об ошибке, вычисляя какой-нибудь символ, не окруженный скобками.
- Вызовите сообщение об ошибке, вычисляя какой-нибудь символ, окруженный скобками.
- Создайте счетчик, который увеличивается на два за одно приращение.
- Напишите какое-нибудь выражение, которое печатает сообщение в эхо-

области, когда его вычисляешь.

2. Практика вычислений

До того, как я научу вас создавать собственные функции на Emacs Lisp, полезно немного попрактиковаться в вычислении различных выражений, которые уже написаны. Эти выражения будут списками, где первый и часто единственный элемент списка — имя какой-нибудь функции. Так как одни из самых простых функций в Emacs — это функции связанные с буферами, то мы начнем с них; к тому же они очень интересны и познавательные. В этой главе мы изучим наиболее простые из этих функций. В следующей главе разберемся с более сложными функциями, связанными с буферами Emacs, и мы увидим, как они реализованы на Emacs Lisp.

Как вычислить

> *Когда вы выполняете какую-нибудь команду редактирования, такую как перемещение курсора или прокрутка экрана, вы вычисляете выражение, первым элементом которого является функция. Именно так и работает Emacs.*

Когда вы нажимаете клавиши, вы заставляете интерпретатор Лиспа вычислить выражение, и так вы получаете результаты. Даже просто набирая какой-нибудь текст, вы вычисляете функцию, в данном случае self-insert-command, которая просто вставляет в текст набранные вами символы. Функции, которые вы вычисляете нажатием клавиш, называются 'интерактивными' функциями, или командами; как сделать функцию интерактивной показано в следующей главе, где мы научимся определять собственные функции в Emacs Лисп. See section Making a Function Interactive.

Кроме выполнения интерактивных команд, мы уже знаем другой способ вычислить выражение — расположить курсор после списка и нажать C-x C-e. Именно так мы и будем делать до конца этой главы. Существуют и другие способы вычислить выражение — они будут описаны в последующих главах.

Можно добавить, что функции, которые мы будем вычислять до конца этой главы, полезны сами по себе. Изучение этих функций поможет понять разницу между буферами и файлами; каким образом можно переключиться в нужный буфер, и как определить положение курсора в текущем буфере.

2.1 Имя буфера

Две функции, buffer-name и buffer-file-name, показывают разницу между файлом и буфером. Когда вы вычислите следующее выражение, (buffer-name), в эхо-области появиться имя этого буфера. Когда вы вычислите (buffer-file-name), в эхо-области появится имя файла, соответствующего данному буферу. Обычно имя, которое возвращает функция (buffer-name) — это имя редактируемого файла, а имя, возвращаемое (buffer-file-name) полное имя файла (вместе с именами каталогов).

Файл и буфер — это два совершенно различных понятия. Файл — это информация хранимая в компьютере постоянно (если вы конечно не удалите его сами). Буфер, наоборот — это информация, которая хранится в Emacs и она пропадет в конце сессии редактирования (или когда вы уничтожите буфер). Обычно буфер содержит информацию, которую вы скопировали из файла; мы говорим, что буфер посетил

этот файл. Именно эту копию вы изменяете. Изменения в буфере не меняют содержимое файла до тех пор, пока вы не сохраните буфер. Когда вы сохраняете буфер, он копируется обратно в файл, и таким образом все изменения фиксируются постоянно.

Если вы читаете это в Info внутри GNU Emacs, то вы можете вычислить каждое из следующих выражений, расположив курсор после него и нажав C-x C-e.

```
(buffer-name)

(buffer-file-name)
```

Когда я сделал это, в эхо-области появилась строка "'introduction.texinfo"' — это значение, которое вернула функция (buffer-name), функция (buffer-file-name) вернула другое значение "'/gnu/work/intro/introduction.texinfo"'. Первое — это имя буфера, а второе — это имя файла. (При вычислении этих выражений круглые скобки говорят интерпретатору Лисп обрабатывать buffer-name и buffer-file-name как функции; без скобок интерпретатор попытается вычислить эти символы как переменные).

Несмотря на существующую разницу между файлами и буферами, вы часто обнаруживаете, что люди говорят о файле, когда они имеют ввиду буфер и наоборот. В самом деле, большинство говорит, "Я редактирую файл" а, не "Я редактирую буфер, который вскоре сохраню в файле." Однако из контекста почти всегда ясно, что они имеет ввиду. Однако когда дело касается компьютерных программ очень важно понимать эту разницу, так как компьютеры не так сообразительны как люди.

Слово 'буфер' между прочим первоначально означало подушку, которая смягчает силу коллизии. В старых компьютерах буфер смягчал коллизии между файлами и центральным процессором. Барабаны и ленты, которые содержали файл и процессор были устройствами, которые очень отличались друг от друга и работали с разной скоростью, рывками. Буфер позволял таким различным устройствам сосуществовать вместе, и эффективно работать. Постепенно буфер превратился из промежуточного временного устройства в то место, где происходить основная работа. Это превращение похоже на то, как маленький портовый городишко превратился в огромный мегаполис — когда-то это было место, где команда отдыхала на временной стоянке, а теперь это крупный культурный и деловой центр.

Не все буфера связаны с существующими файлами. Например, когда вы запускаете сессию Emacs, набрав команду emacs в командной строке, не перечисляя никаких файлов, то Emacs вначале отобразит на экране буфер '*scratch*'. Этот буфер не связан ни с каким файлом. То же самое можно сказать про буфер '*Help*'.

Если вы переключитесь в буфер '*scratch*' наберете (buffer-name), поставите курсор после этого выражения и нажмете C-x C-e, чтобы вычислить его, то в эхо-области появится имя "*scratch*", как результат вычисления данной функции. "*scratch*" — это имя буфера. Однако, если вы попробуете вычислить выражение (buffer-file-name), находясь в буфере '*scratch*', то в эхо-области появится nil. nil — это 'ничего' в переводе с латинского; в нашем случае это означает, что буфер '*scratch*' не связан ни с каким из файлов. (В Лиспе, nil также используют в значении 'ложь' и также как синоним для пустого списка, ().)

Кстати, если вы находитесь в буфере '*scratch*' и хотите, чтобы возвращаемое

значение появилось в самом буфере '*scratch*', а не в эхо-области, то нажмите C-u C-x C-e вместо C-x C-e. Тогда возвращенное значение появится после выражения. Буфер будет выглядеть следующим образом:

```
(buffer-name)"*scratch*"
```

Вы не сможете выполнить этого в Info, поскольку буфер Info открыт в режиме только для чтения, и поэтому в нем запрещены какие-либо изменения. Но вы можете сделать так в любом буфере, который вы можете изменять; а когда вы пишете программы или документацию (например такую книгу), то такая возможность очень полезна.

2.2 Как вернуть буфер

Функция buffer-name возражает имя буфера; чтобы получить сам буфер, нужна другая функция: current-buffer. Вы можете использовать эту функцию в своих программах, когда вам надо получить сам буфер.

Имя и объект, или сущность, к которой относится это имя отличаются друг от друга. Вы — не ваше имя. Вы личность, к которой другие люди могут обращаться по имени. Если вы захотите поговорить с Джоном, и кто-то протянет вам лист бумаги с буквами 'Д', 'ж', 'о', 'н', вас это может быть позабавит, но вы же хотели совсем другого. Вы не хотите разговаривать с именем, вам нужен человек, которого так зовут. С буфером дело обстоит таким же образом — имя буфера scratch это '*scratch*', но имя это не сам буфер. Чтобы получить сам буфер вам нужна какая-нибудь функция — например current-buffer.

Однако существует определенная тонкость — если вы вычислите выражение current-buffer в каком-нибудь выражении, как мы вскоре и сделаем, то, то что вы увидите — это печатное представление буфера без его содержимого. Emacs делает это по двум причинам — в буфере может быть тысячи строк (это неудобно отображать) и другой буфер может иметь такое же содержание, и очень важно отличать их друг от друга.

Ниже выражение с нужной нам функцией:

```
(current-buffer)
```

Если вы вычислите это выражение, как обычно, то в эхо-области появится '#<buffer *info*>'. Особый формат показывает, что функция возвращает буфер, а не его имя.

Кстати, хотя вы можете использовать в своих программах числа и символы, вы не можете делать это с печатным представлением буфера — единственный способ получить буфер в вашей программе, это использовать функцию такую как current-buffer.

Похожая функция other-buffer. Она возвращает буфер, в котором вы недавно побывали. Если вы переключитесь в буфер '*scratch*', а затем вернетесь обратно, то функция other-buffer вернет буфер '#<buffer *scratch*>'.

Вы можете проверить это, вычислив следующее выражение:

```
(other-buffer)
```

В эхо-области появится '#<buffer *scratch*>' или имя какого-нибудь другого буфера, из которого вы недавно переключились.

2.3 Смена буфера

Функции, возвращающие буфер, обычно используют в более сложных выражениях — например, когда какой-нибудь функции требуется буфер как один из ее аргументов. Мы увидим это на примере функции switch-to-buffer, которая используется для переключения в другой буфер.

Но вначале краткое введение в функцию switch-to-buffer. Когда вы переключаетесь из буфера Info в буфер '*scratch*' и обратно, вычисляя (buffer-name), то вы обычно нажимает C-x b и затем в мини-буфере вводите '*scratch*' в качестве имени буфера, в который вы хотите переключиться. Нажатие клавиш C-x b заставляет интерпретатор Лиспа вычислить интерактивную функцию Emacs Lisp switch-to-buffer. Как мы уже говорили, именно так и работает Emacs — разные клавиши вызывают или запускают разные функции. Например, C-f вызывает forward-char, M-e вызывает forward-sentence, и так далее.

Написав выражение с функцией switch-to-buffer и задав ей буфер, в который мы хотим переключиться, в качестве аргумента, мы можем переключится в этот буфер аналогично тому, как мы это делаем, нажимая сочетание клавиш C-x b.

Вот необходимое Лисп выражение:

```
(switch-to-buffer (other-buffer))
```

Символ switch-to-buffer — первый элемент списка, поэтому интерпретатор будет обрабатывать его как функцию и выполнит соответствующие ей инструкции. Но до того, как вызвать эту функцию, интерпретатор заметит внутренний вложенный список (other-buffer) и сначала вычислит внутреннее выражение. other-buffer первый (и в нашем случае единственный) элемент внутреннего списка, так что интерпретатор Лиспа вызовет или запустит эту функцию. Она возвратит другой буфер. Затем интерпретатор вызовет switch-to-buffer, передав ей в качестве аргумента другой буфер, в который Emacs и переключится. Если вы читаете это в Info, попробуйте это прямо сейчас. Вычислите выражение. (Чтобы вернуться обратно, нажмите C-x b RET).

В последующих главах вы будете встречать функцию set-buffer чаще чем switch-to-buffer. Это потому, что между людьми и компьютерными программами существует разница — у людей есть глаза, и они привыкли видеть то, над чем они сейчас работают. Это настолько ясно, что об этом даже не говорят. Однако у программ нет глаз. Когда компьютерная программа работает над буфером, его необязательно отображать на экране компьютера.

switch-to-buffer предназначена для людей и выполняет две разные вещи — она переключает внимание Emacs на какой-нибудь буфер; и отображает новый выбранный буфер на экране компьютера. set-buffer, действует немного по-другому, эта функция делает только одно — она переключает внимание компьютерной программы на другой буфер. Буфер, отображаемый на экране компьютера, остается без изменений (конечно, там обычно ничего не произойдет до тех пор, пока программа не завершится).

Также, мы недавно использовали другой термин из сленга программистов — слово вызывать. Когда вы вычисляете список, в котором первый символ это имя функции, то вы вызываете эту функцию. Этот термин появился, когда заметили, что функция — это сущность которая может выполнить что нибудь для вас, если вы вызовете ее (как водопроводчик сущность, которая может устранить утечку, если вы вызовете его).

2.4 Размер буфера и местоположение точки

В конце главы давайте обратимся к совсем простым функциям: buffer-size, point, point-min и point-max. Они дают информацию о размере буфера и местоположении точки в нем.

Функция buffer-size показывает размер текущего буфера; то есть эта функция возвращает число символов в данном буфере.

```
(buffer-size)
```

Вы можете вычислить это как обычно, расположив курсор за выражением и нажав C-x C-e.

В Emacs текущая позиция курсора называется точкой. Выражение (point) возвращает число, которое сообщает вам, где расположен курсор, считая символы от начала буфера до точки.

Вы можете увидеть число символов для текущего положения курсора в этом буфере, вычислив следующее выражение обычным способом:

```
(point)
```

Когда я сам вычислил это выражение, значение точки было 65724. Функция point часто используется в последующих примерах этой книги.

Значение точки зависит, конечно, от ее местоположения внутри буфера. Если вы вычислите следующее выражение, то число будет больше:

```
(point)
```

Для меня значение точки в этом месте было 66043 — это означает, что между этими двумя выражениями 319 символов (включая пробелы).

Функция point-min очень похожа на point, но она возвращает минимально возможное значение точки в текущем буфере. Обычно это число 1, если не включено сужение ("narrowing"). (Сужение — это механизм, каким вы можете ограничить себя или программу для действия только на части буфера. See section Narrowing and Widening.)) Аналогично, функция point-max возвращает максимально возможное значение точки в текущем буфере.

2.5 Упражнения

Откройте какой-нибудь текстовый файл и переместите курсор в середину этого файла. Найдите имя файла, имя буфера, его длину, и вашу текущую позицию в нем.

3. Написание функций

Когда интерпретатор Лиспа вычисляет список, он анализирует, связано ли с первым символом списка определение функции; или по-другому, указывает ли символ на какую-нибудь функцию. Если указывает, то компьютер выполняет инструкции этой функции. Символ, с которым связано определение функции, обычно называют просто функцией (хотя строго говоря, сама функция — это ее определение, а символ только указывает на нее).

Немного о примитивных функциях

Все функции определяются в терминах других функций, кроме нескольких примитивных функций, которые написаны на языке программирования C. Когда вы пишете определение функции, вы пишете его на Emacs Лисп и используете другие функции как строительные блоки. Некоторые из функций, которые вы используете, сами написаны на Emacs Лиспе (возможно даже вами), а некоторые будут примитивами, написанными на языке C. Примитивные функции используются точно так же, как те, которые написаны на Emacs Лиспе, и ведут себя точно таким же образом. Они написаны на языке C, поэтому мы можем легко переносить GNU Emacs на любую платформу, которая поддерживает язык C.

Еще раз подчеркнем следующее — когда вы пишете код на Emacs Лиспе, вы не различаете функции написанные на C и функции, написанные на Emacs Лиспе. Разница здесь не играет особой роли. Я упомянул о ней только потому, что это полезно знать. В самом деле, если вы попытаетесь исследовать этот вопрос, то вы не узнаете на каком языке написана уже существующая функция: на C или на Emacs Лиспе.

3.1 Особая форма defun

В Лиспе с символом, таким как mark-whole-buffer, связан некоторый код, который выполняется компьютером, когда вызывается эта функция. Этот код называют определением функции, и он создается вычислением Лисп-выражения, которое начинается с символа defun (это сокращение от define function). Поскольку defun не вычисляет свои аргументы обычным образом, то она называется особой формой.

В следующих разделах мы рассмотрим определения функций, входящие в дистрибутив Emacs, такие как, например, mark-whole-buffer. А в этом разделе мы опишем несколько простых функций так, чтобы вы представляли себе, как это выглядит. Это будут арифметические функции, поскольку они наиболее просты. Некоторые не любят арифметические примеры; однако, если вы один из таких людей, не отчаивайтесь. Едва ли какой нибудь код, который мы будем изучать в оставшихся главах, будет связан с арифметикой или математикой. В основном мы будем обрабатывать текст тем или иным способом.

Определение функции может содержать до пяти частей, которые следуют после слова defun:

1. Имя символа, с которым это определение функции будет связано.
2. Список аргументов, которые можно будет передать в эту функцию. Если функция не требует аргументов, тогда это пустой список ().
3. Строка документации, описывающая работу функции. (Формально не

обязательна, но настоятельно рекомендуемая).
4. Необязательное выражение, чтобы сделать функцию интерактивной, чтобы вы могли использовать ее, нажав M-x и набрав имя функции; или связать с ней какое-нибудь сочетание клавиш или клавишу.
5. Сами инструкции, которые будет выполнять компьютер — так называемое тело определения функции.

Полезно, для запоминания думать о пяти частях в определении функции, как о следующем шаблоне, где есть место для каждой части:

```
(defun имя-функции (аргументы...)
  "необязательная-документация..."
  (interactive как-получить-аргументы)     ; необязательно
  тело...)
```

Как первый пример мы рассмотрим определение функции, которая умножает свой аргумент на 7. (Это пример не интерактивной функции. See section Making a Function Interactive, для получения дополнительной информации).

```
(defun умножить-на-семь (number)
  "Умножить NUMBER на семь."
  (* 7 number))
```

Это определение начинается со скобок и символа defun, за которым следует имя функции.

За именем функции идет список, который содержит аргументы, передаваемые этой функции. Этот список называется списком аргументов. В нашем случае в списке только один элемент — символ number. Когда будет выполняться функция, этот символ будет связан со значением аргумента, который передали функции при вызове.

Вместо того, чтобы назвать наш аргумент словом number, я мог бы выбрать любое другое имя. Например, слово multiplicand. Я выбрал слово 'number', поскольку это подсказывает, какого рода значение ожидает функция; но слово 'multiplicand' тоже весьма неплохо, поскольку оно подскажет, какую роль этот аргумент будет играть в теле нашей функции. Если бы я назвал аргумент foogle, это было бы намного хуже, потому что такое имя не несет никакой полезной информации. Выбор имен это обязанность программиста, и лучше если они будут помогать в понимании функции.

На самом деле вы можете выбрать любое имя, какое пожелаете для символа в списке аргументов, даже имя, которое используется в какой-нибудь другой функции — имя, которое вы используете для аргумента, локально для этого конкретного определения функции. В этом определении имя относится к одному объекту, а такое же имя вне определения функции — к совсем другой. Положим, что в вашей семье вас зовут 'Коротышка'; когда кто-нибудь из вашей семьи говорит 'Коротышка' — они имеют в виду вас. Но вне вашей семьи, например, в каком нибудь фильме, 'Коротышкой' могут звать кого-нибудь еще. Так как имя для символа в списке аргументов локально для данного определения функции, то вы можете изменять значение такого символа внутри тела функции, не изменяя его значения вне функции. Похожего эффекта можно достигнуть использованием выражения let. (See section let.)

Заметьте также, что мы обсуждаем слово 'number' в двух различных значениях — как символ, который появится в программе, и как имя чего-то, что будет заменено чем-то другим во время вычисления функции. В первом случае number — это символ, а не число; так получилось, что в пределах функции, это переменная, чье значение — искомое число, но наш главный интерес в нем, как в символе. С другой стороны, когда мы говорим о функции, нас интересует, что за число будет подставлено за место слова number. Чтобы четко это различать, мы используем различные шрифты для отображения слов в этих двух обстоятельствах. Когда мы говорим об этой функции, или о том как это работает, мы соотносимся к числу как number. В теле функции мы пишем это, как number.

За списком аргументов следует строка документации, которая описывает данную функцию. Это ее вы видите, когда нажимаете сочетание клавишC-h f и набираете имя функции. Кстати, если вы пишете строку документации, такую как эту, то вы должны сделать первую строку завершенным предложением, так как некоторые команды, такие как apropos, печатают только первую строку многострочной документации. Так же вы не должны выравнивать вторую строку строки документации, если она у вас появляется, потому что это выглядит странно, когда вы используете сочетание клавиш C-h f (describe-function). Строка документации необязательна, но очень полезна, поэтому ее надо включать во все функции, которые вы создаете.

Третья строка нашего примера — это тело определения функции. (Большинство определений функций, конечно, намного больше чем это). В нашем случае тело — это список (* 7 number), которое умножает значение number на 7. (В Emacs Лиспе * функция умножения, как + функция сложения).

Когда вы будете использовать функцию умножить-на-семь, аргумент number будет заменен конкретным числом, которое вы захотите использовать. Ниже пример использования функции умножить-на-семь, но не пытайтесь вычислить ее!

```
(умножить-на-семь 3)
```

С символом number, который описан в определении функции в предыдущем разделе, при данном вызове функции "связывается" фактическое значение 3. Заметьте, что, хотя number в определении функции был в скобках, но при вызове функции умножить-на-семь, он передавался уже без скобок. Скобки используются в определении функции для того, чтобы компьютер мог отделить список аргументов от тела функции.

Если вы вычислите этот пример, то наверняка получите сообщение об ошибке. (Смелей, пробуйте!) Это потому что хотя мы и написали определение функции, но еще не проинформировали компьютер об этом определении (не внесли определение этой функции в Emacs). Инсталляция функции — это процесс передачи интерпретатору Лиспа определения функции. Она описана в следующем разделе.

3.2 Установка определения функции

Если вы читаете это в Info внутри Emacs, то вы можете попробовать выполнить функцию умножить-на-семь, вначале вычислив определение функции, а затем вычислив (умножить-на-семь 3). Копия определения функции расположена ниже. Поставьте курсор за последней скобкой определения функции и нажмите C-x C-e.

Когда вы сделаете это, в эхо-области появится строка умножить-на-семь. (Это означает, что когда вычисляется определение функции, возвращаемым значением является имя определяемой функции). Одновременно с этим мы устанавливаем функцию.

```
(defun умножить-на-семь (number)
  "Умножить NUMBER на семь."
  (* 7 number))
```

Вычислив defun, вы только что установили умножить-на-семь в Emacs. Эта функция сейчас точно такая же часть Emacs, как и forward-word, или любая другая функция редактирования, которую вы используете. (умножить-на-семь останется в системе, до тех пор пока вы не завершите сессию Emacs. Чтобы загружать код автоматически, когда вы запускаете Emacs, смотрите Installing Code Permanently.)

Вы можете увидеть эффект установки умножить-на-семь вычислив следующий пример. Расположите курсор после выражения и нажмите C-x C-e. В эхо-области появиться число 21.

```
(умножить-на-семь 3)
```

Если хотите, вы сейчас можете прочесть документацию для этой функции нажав C-h f (describe-function) и затем набрав в мини-буфере имя нашей функции, умножить-на-семь. Когда вы сделает это, на экране появится окно '*Help*', которое отобразит следующее:

```
умножить-на-семь:
Умножить NUMBER на семь.
```

(Чтобы закрыть окно помощи нажмите C-x 1.)

3.2.1 Изменяем определение функции

Если вы хотите изменить код, связанный с умножить-на-семь, только перепишите его. Чтобы установить новую версию на месте старой, снова вычислите определение функции. Именно так вы модифицируете код в Emacs. Все очень просто.

Например, вы можете изменить функцию умножить-на-семь так, чтобы она семь раз складывала это число, вместо того чтобы умножать его на семь. Мы получим тот же самый ответ, но другим способом. Одновременно мы добавим к определению комментарий; комментарий — это текст, который интерпретатор Лиспа игнорирует, но люди находят весьма и весьма полезным. В нашем случае комментарий будет говорить, что это "вторая версия".

```
(defun умножить-на-семь (number)        ; Вторая версия.
  "Умножить NUMBER на семь."
  (+ number number number number number number number))
```

Комментарии следуют после точки с запятой ';'. Согласно синтаксису Лиспа все, что следует после ; — комментарий. Конец линии — конец комментария. Чтобы растянуть комментарий на две или более строк, начинайте каждую строку с точки с запятой.

See section Начало файла '.emacs', и section 'Comments' in The GNU Emacs Lisp Reference Manual, для дополнительной информации о комментариях.

Вы можете установить эту версию умножить-на-семь точно так же, как и предыдущую, расположив курсор за последней скобкой и нажав C-x C-e.

Если просуммировать, именно так вы и пишете код на Emacs Лисп — сначала вы создаете функцию, устанавливаете ее, тестируете; потом исправляете ошибки или улучшаете и инсталлируете ее вновь.

3.3 Делаем функцию интерактивной

Вы можете сделать функцию интерактивной, поместив список, который начинается с особой формы interactive сразу после документации. Пользователь запускает интерактивную функцию нажав M-x и потом набрав имя функции в мини-буфере; или нажав клавишу, к которой эта функция привязана, например, C-n для next-line, или C-x h для mark-whole-buffer.

Когда вы вызываете интерактивную функцию интерактивно, то возвращаемое ей значение автоматически не отображается в эхо-области. Это от того, что вы часто вызываете интерактивную функцию из-за ее побочных эффектов, например, перемещение курсора на слово или строку, а не из-за возвращаемого значения. Если бы возвращаемое значение каждый раз, когда вы нажимаете клавишу, отображалось бы в эхо-области, то это бы очень сильно отвлекало внимание пользователя.

Использование особой формы interactive и способ отобразить значение в эхо-области можно проиллюстрировать интерактивной версией умножить-на-семь.

Вот и текст программы:

```
(defun умножить-на-семь (number)        ; Интерактивная версия.
  "Умножить NUMBER на семь."
  (interactive "p")
  (message "Итог %d" (* 7 number)))
```

Вы можете установить эту функцию, расположив курсор после определения и нажав C-x C-e. В эхо-области появится имя функции. Теперь вы можете использовать эту функцию, нажав C-u, затем какое-нибудь число и потом M-x умножить-на-семь, а в конце ВВОД. В эхо-области появится фраза `Итог ...', где на месте ... будет стоять итоговое произведение.

То есть, вы можете запустить функцию, подобную этой, двумя способами:

Задав как префикс-аргумент число, передаваемое в функцию, потом нажав M-x и имя функции, например C-u 3 M-x forward-sentence; или,
Нажав клавишу или сочетание клавиш, с которой связана эта функция, например, C-u 3 M-e.
Оба только что упомянутых примера работают одинаковым образом --- они перемещают точку на два предложения. (Так как умножить-на-семь не привязана ни к какой клавише, то ее нельзя вычислить вторым способом.)

(See section Привязки клавиш, чтобы научиться, как связывать команду с клавишей.)

Префикс-аргумент можно передать интерактивной функции, либо нажав клавишу META, затем число, например M-3 M-e, или нажав C-u, и потом число, например, C-u 3 M-e. (Если вы просто нажмете C-u по умолчанию префикс-аргумент будет равен 4).

3.3.1 Интерактивная умножить-на-семь.

Давайте рассмотрим использование особой формы interactive и функции message в интерактивной версии умножить-на-семь. Напомним, как выглядит определение функции:

```
(defun умножить-на-семь (number)        ; Интерактивная версия.
  "Умножить NUMBER на семь."
  (interactive "p")
  (message "Итог %d" (* 7 number)))
```

В этой функции, выражение (interactive "p") — список из двух элементов. "p" говорит Emacs передать префикс-аргумент в функцию и использовать его значение как значение аргумента.

Аргумент будет числом. Это значит, что символ number будет связан с этим числом в строке:

```
(message "Итог %d" (* 7 number))
```

Например, если префикс-аргумент 5, то интерпретатор Лиспа будет вычислять эту строку, как будто бы она выглядит следующим образом:

```
(message "Итог %d" (* 7 5))
```

(Если вы читаете это в GNU Emacs,то вы можете вычислить это выражение сами). Первым делом интерпретатор вычислит внутренний список — в нашем случае (* 7 5). Возвращенное значение 35. Затем он вычислит внешний список, передав значения второго и последующих элементов списка функции message.

Как мы уже знаем, message — это функция Emacs Лисп, специально предназначенная для отображения однострочных сообщений пользователю (See section The message function.). Напомним, что функция message печатает свой первый аргумент в эхо-области как есть, кроме следующих символов '%d', '%s', '%c'. Когда она встречает одну из таких управляющих последовательностей, эта функция вычисляет второй и последующие аргументы и вставляет их значения на место соответствующей управляющей последовательности.

В интерактивной версии умножить-на-семь управляющая строка '%d', которая требует число, а значение, возвращаемое вычислением (*7 5) это число 35. Поэтому число 35 печатается на месте '%d' и конечное сообщение 'Итог 35'.

(Заметьте, что, когда вы вызываете функцию умножить-на-семь, то сообщение печатается без кавычек, но если вызвать саму функцию message, то текст напечатается в двойных кавычках. Это потому что при вычислении функции message в эхо-области появляется значение, которое возвращает эта функция, но когда функция message используется в более сложном выражении, то текст отображается без кавычек как побочный эффект.)

3.4 Различные опции для interactive

В предыдущем примере умножить-на-семь как аргумент при вызове interactive использовалась строка "p". Это указание интерпретатору Emacs Лиспа передать в функцию численное значение префикс-аргумента, который вы можете задать, нажав либо C-u, либо META и затем набрав какое-нибудь число. В Emacs более 20 символов, предопределенных для использования с interactive. Почти во всех случаях, та или другая из этих опций позволит вам передать интерактивно в функцию всю необходимую информацию. (See section `Code Characters for interactive' in The GNU Emacs Lisp Reference Manual.)

Например, символ 'r' заставляет Emacs передать начало и конец области (текущие значения точки и метки) в функцию как два различных аргумента. Его используют следующим образом:

```
(interactive "r")
```

Другой символ 'B' — указание для Emacs запросить от вас имя буфера, которое будет передано в функцию. В этом случае Emacs отобразит в мини-буфере подсказку, используя строку, которая следует за 'B', например, "BДобавить к буферу: ". Emacs не только отобразит подсказку в мини-буфере, но также позволит воспользоваться функцией автодополнения, что здорово помогает (просто наберите первые несколько символов и нажмите TAB).

Функция, которая требует два и более аргумента, может получить информацию для каждого из них, добавляя к строке за interactive различные части. Когда вы сделаете это, информация передается в каждый аргумент в том же самом порядке как они описаны в списке аргументов interactive. В строке каждая часть отделяется друг от друга '\n' — символом новой строки. Например, если за interactive будет следовать строка "BДобавить к буферу: \nr", то это заставит интерпретатор Лиспа передать в функцию значения точки и метки, а также запросить у вас имя буфера (в целом три аргумента).

В этом случае определение функции будет выглядеть следующим образом: buffer, start и end — символы, которые interactive свяжет с буфером и текущими значениями начала и конца блока текста:

```
(defun имя-функции (buffer start end)
  "документация..."
  (interactive "BДобавить к буферу: \nr")
  тело-функции...)
```

(Пробел после двоеточия в подсказке для лучшего отображения в эхо-области. Функция append-to-buffer выглядит точно таким образом. See section The Definition of append-to-buffer.)

Если функции не требуются аргументы, тогда и у interactive их не будет. Такая функция будет содержать только выражение (interactive). Именно так выглядит функция mark-whole-buffer.

Если ни один из существующих буквенных кодов не подходит для вашего приложения, то вы можете передать ваши собственные аргументы к interactive в виде списка. See section 'Using Interactive' in The GNU Emacs Lisp Reference Manual, для

дополнительной информации об этом методе.

3.5 Устанавливаем код надолго

Когда вы устанавливаете функцию, вычисляя ее определение, то она остается внутри Emacs до тех пор, пока вы не завершите работу с ним. Если вы снова запустите Emacs, то функция не будет установлена, до тех пор пока вы не вычислите ее определение снова.

Иногда вам может понадобиться, чтобы функция устанавливалась автоматически, когда вы запускаете новую сессию Emacs. Это можно выполнить несколькими способами:

Если некоторая функция нужна только вам, то вы можете сохранить ее определение в своем файле инициализации '.emacs'. Когда вы запускаете Emacs, ваш файл '.emacs' автоматически вычисляется, и все определения функций, расположенные в нем, устанавливаются. See section Ваш файл '.emacs'.
Если ваша функция будет полезна всем пользователям компьютера, ее определение можно разместить в файле, который называется 'site-init.el' и который загружается при компиляции Emacs. (Прочитайте файл 'INSTALL', который входит в дистрибутив Emacs).
Альтернатива этому — расположить определения функций, которые вы хотите установить, в один или несколько файлов и использовать функцию load, которая заставит Emacs вычислить эти файлы и таким образом установит каждую функцию, расположенную в этих файлах. See section Загрузка файлов.
Наконец, если вы считаете что ваш код может быть полезен всем пользователям Emacs в мире, то вы можете опубликовать его в Internet или послать копию Free Software Foundation. (Когда вы будете это делать, пожалуйста включите в состав кода copyleft notice до того, как публиковать его). Если вы пошлете копию вашего кода Free Software Foundation, nj его могут включить в следующую версию Emacs. В основном именно так Emacs и разрастался в течении многих лет, с помощью пользователей.

3.6 let

Выражение let — это особая форма в Lisp, которую вы будете часто использовать при создании своих функций. Поскольку она используется очень часто, то давайте рассмотрим ее в этой главе.

let используют, чтобы связать значение с символом таким образом, чтобы интерпретатор Лиспа не спутал переменную с таким же именем, но определенную вне функции. Чтобы оценить важность этой особой формы, рассмотрим ситуацию.

Чтобы понять всю важность этой особой формы, рассмотрим житейский пример — например, у вас есть собственная дача и в предложении "Не мешало бы дачу покрасить.", вы имеете в виду именно свою фазенду. Если вы навестили загородный участок друга, то такая фраза сказанная вами будет уже относиться к даче друга, а если ее только недавно построили, то вы рискуете нарваться на неприятности. Та же самая ситуация может случиться и в Лиспе, если переменная, которая используется внутри одной функции названа также как и переменная используемая в другой функции, и у этих переменных не обязательно должно быть одинаковое значение.

Особая форма let защищает от такого рода путаницы. let создает имя для локальной

переменной, которая затеняет любое использование такого же имени вне выражения let. Это как понимание того, что, когда вы сказали 'дача', то вы имели в виду свою дачку, а не друга. (Символы используемые в списке аргументов действуют точно таким же образом. See section Особая Форма defun.).

Локальные переменные, создаваемые выражением let, сохраняют свои значения только внутри самого выражения let (внутри выражений, которые вы вызываете из выражения let); локальные переменные не действуют вне выражения let.

let может создать более одной переменной одновременно. Так же let назначает каждой переменной при создании первоначальное значение, или то, которое задаете вы, или nil. (На жаргоне Лисп программистов это означает 'связать переменную со значением'). После того как let создал переменные и назначил им какое-нибудь значение, выполняется код в теле let и возвращается результат последнего значение в теле, как значение всего выражения let.

('Выполнить' — это жаргонный термин, который означает вычислить список; он произошел от использования слова в значении 'получить практический результат' (Oxford English Dictionary). Поскольку вы вычисляете выражение, чтобы исполнить действие, то 'выполнить' используется, как синоним `вычислить'.)

3.6.1 Части выражения let

Выражение let — это список, состоящий из трех частей. Первая часть — сам символ let. Вторая часть — это список, который называют список переменных, каждый элемент которого или символ или двухэлементный список, первый элемент которого символ. Третья часть выражения let — это основное тело выражения. Тело обычно состоит из одного или более списков.

Шаблон для выражения let выглядит следующим образом:

```
(let список_переменных тело...)
```

Символы в списке переменных — это переменные, которым особая форма let присваивает начальное значение. Если символ только один, то ему присваивается значение nil; если символ входит в состав двухэлементного списка, то ему назначается значение, которое возвращает интерпретатор Лиспа после вычисления второго элемента списка.

Так, список переменных может выглядеть следующим образом — (нитки (иголки 3)). В этом случае в выражении let, Emacs свяжет символ нитки со значением nil, а символ иголки со значением 3.

Когда вы будете писать выражение let, то вам надо будет заполнить соответствующими выражениями места в шаблоне для выражения let.

Если список переменных состоит из двухэлементных списков, как это часто и случается, то шаблон для выражения let будет таким:

```
(let ((переменная значение)
      (переменная значение)
      ...)
      тело...)
```

3.6.2 Пример выражения let

В следующем выражении создаются и инициализируются две переменные зебра и тигр. Телом выражения let является список, в котором вызывается функция message.

```
(let ((зебра 'полосаты)
      (тигр 'свирепы))
  (message "Некоторые животные %s, а другие %s."
   зебра тигр))
```

Здесь список переменных `((зебра 'полосаты) (тигр 'свирепы))`.

Две переменные зебра и тигр. Каждая переменная является первым элементом двухэлементного списка, и ей назначается результат вычисления второго элемента двухэлементного списка. В списке переменных Emacs связывает переменную зебра со значение полосаты, а переменную тигр со значением свирепы. В нашем случае перед символами, которые будут представлять первоначальное значение переменных, стоит апостроф. На их месте мог бы стоять любой список или строка. Основное тело let начинается после списка переменных. В этом случае тело является списком, который использует функцию message, чтобы распечатать строку в эхо-области.

Вы можете вычислить этот пример обычным образом, поместив курсор за последней закрывающей скобкой и нажав C-x C-e. Когда вы сделаете это, в эхо-области появится следующая строка:

```
"Некоторые животные полосаты, а другие свирепы."
```

Как мы уже знаем, функция message печатает свой первый аргумент полностью, кроме спецсимволов, например, '%s'. В этом случае значение переменной зебра напечатается на месте первого '%s', а значение переменной тигр на месте второго '%s'.

3.6.3 Неинициализированные переменные в операторе let

Если вы не назначили переменным в выражении let какое-нибудь свое первоначальное значение, то им автоматически присваивается значение nil, как в следующем выражении:

```
(let ((береза 3)
      сосна
      ель
      (дуб 'что-то))
  (message
   "Вот %d переменные со значениями %s, %s, и %s."
   береза сосна ель дуб))
```

Вот список переменных этого выражения: ((береза 3) сосна ель (дуб 'что-то))

Если вы вычислите это выражение как обычно, в эхо-области появится следующая фраза:

```
"Вот 3 переменные со значениями nil, nil, и что-то."
```

В этом случае Emacs присвоил символу береза число 3, символам сосна и ель значения nil, а символу дуб значение что-то.

Обратите внимание, что в списке переменных let переменные сосна и ель стоят сами по себе, как атомы, и не окружены скобками — поэтому им при инициализации назначается значение nil. Но дуб в составе списка (дуб 'что-то) и поэтому ему присваивается значение что-то. Аналогично березе присваивается число 3, так как она в списке, где вторым элементом является число 3. (Напомним, что поскольку число вычисляется само собой, то его не надо предварять апострофом. Также число печатается в сообщении на месте '%d', а не '%s'). Вся группа из четырех переменных помещается в скобки, чтобы ограничить их от основного тела let.

3.7 Особая форма if

Третья особая форма, которую мы изучим в этой главе, в добавок к defun и let — это условная if. Эту форму используют когда надо проинструктировать компьютер принять решение. Можно конечно создавать функции и без использования этой формы, но поскольку ее все же используют довольно часто, и она очень важна, то мы рассмотрим ее здесь. Ее используют, например, в определении функции beginning-of-buffer.

Основная идея if — "если какое-то условие истинно, то тогда выполняем какие-то действия". Если условие ложно, то действия не выполняются. Например, вы можете принять следующее решение "Если будет хорошая погода, то тогда я пойду на пляж!"

В выражении if, написанном на Лиспе, слово 'then' не используют; тест и действие — это соответственно второй и третий элемент списка, где первый элемент сам if. Тем не менее тестовую часть выражения if часто называют if-часть, а следующую часть — then-часть.

Так же, когда записывается выражение if, проверка-истинна-ложь обычно записывается на той же самой строке, где и символ if, но действие, которое будет выполнятся, если проверка истинна, на второй и последующих строках. Таким образом оформленное выражение if легче воспринимается.

```
(if проверка-истинна-или-ложь
    выполняемые-действия-если-проверка-истинна)
```

Проверка-истинна-ложь — это выражение, которое будет вычисляться интерпретатором Лиспа.

Ниже пример, который вы можете вычислить обычным образом. Там проверяется, правда ли, что число 5, больше чем число 4. Так как это похоже на правду, будет отображено следующее сообщение '5 больше чем 4!'.

```
(if (> 5 4)                             ; if-part
    (message "5 больше чем 4!"))        ; then-part
```

(Функция > проверяет больше ли ее первый аргумент, чем второй, и возвращает истину в случае успеха).

Конечно, в настоящей программе тестовое условие для выражения if не будет так

жестко фиксировано, как в этом примере (> 5 4). Вместо этого по крайней мере одна из переменных будет связана со значением, которое не известно заранее. (Если бы мы знали его заранее, зачем тогда было бы его проверять!)

Например, значение может быть связано с аргументом в определении функции. В следующем определении функции характеристика животного это значение, которое передается в функцию. Если значение аргумента характеристика — свирепость, тогда печатается сообщение `Это тигр!'; в противном случае вернется nil.

```
(defun тип-животного (характеристика)
  "Напечатать сообщение в эхо-области. В зависимости от ХАРАКТЕРИСТИКИ.
     Если ХАРАКТЕРИСТИКА `свирепость', тогда напечатать предупреждение."
  (if (equal характеристика 'свирепость)
      (message "Это тигр!")))
```

Если вы читаете это внутри Emacs, то вы можете вычислить определение этой функции как обычно, чтобы установить ее в Emacs, а потом вычислить два следующих выражения:

```
(тип-животного 'свирепость)

(тип-животного 'зебра)
```

Когда вы вычислите (тип-животного 'свирепость), в эхо-области появится следующее сообщение: "Это тигр!"; а когда вы вычислите (тип-животного 'зебра), в эхо-области будет напечатано nil.

3.7.1 Подробное рассмотрение функции type-of-animal

Давайте рассмотрим функцию тип-животного подробнее.

Эта функция написана стандартным образом путем заполнения соответствующих мест в двух шаблонах — один для определения функции defun, а второй для выражения if.

Шаблон для каждой неинтерактивной функции:

```
(defun имя-функции (список-аргументов)
  "документация..."
  тело...)
```

Части функции, соответствующие шаблону, выглядят следующим образом:

```
(defun тип-животного (характеристика)
  "Напечатать сообщение в эхо-области. В зависимости от ХАРАКТЕРИСТИКИ.
Если ХАРАКТЕРИСТИКА 'свирепость', тогда напечатать предупреждение."
    тело:  if выражение)
```

В этом случае имя функции тип-животного; ей передается значение одного аргумента. За списком аргументов следует двухстрочный комментарий. Мы включили документацию в наш пример, так как надо выработать у себя привычку писать документацию к каждой функции. Тело функции состоит из выражения if.

Шаблон для выражения if такой:

```
(if проверка-истинна-ложь
    выполняемые-действия-если-проверка-истинна)
```

В определении функции тип-животного, настоящий код для if выглядит следующим образом:

```
(if (equal характеристика 'свирепость)
    (message "Это тигр!")))
```

Здесь проверка-истинна-ложь это выражение:

```
(equal характеристика 'свирепость)
```

В Лиспе, функция equal проверяет на равенство два своих аргумента. В нашем случае второй аргумент символ 'свирепость, а первый — значение символа характеристика (другими словами аргумент, который передали в нашу функцию.)

В начале мы передали в функцию тип-животного аргумент 'свирепость. Поскольку свирепость равна свирепость, то вычисление выражения (equal характеристика 'свирепость), вернуло истинное значение. Когда это произошло, if вычислило свой второй аргумент или then-часть: (message "Это тигр!")

Во втором примере мы передали функции тип-животного аргумент зебра. зебра не равна свирепости, поэтому then-часть не вычислялось и выражение if вернуло nil.

3.8 Выражения if--then--else

У выражения if может быть необязательный третий аргумент, который называют else-часть, это на тот случай, когда проверка истинна-ложь окончится неудачей, то есть вернет ложь. Когда это происходит, второй аргумент или then-часть не вычисляется, а вычисляется третья часть или else-часть выражения if. Вы можете считать это альтернативой для дождливого дня в следующем решении `если будет солнечный день, тогда пойду на пляж, иначе буду читать книгу!'.

Слово "else" не надо записывать согласно синтаксису Лиспа; else-часть в выражении if идет сразу за then-частью. Обычно, else-часть записывают на отдельной строке и выравнивают немного левее чем then-часть:

```
(if проверка-истинна-ложь
    выполняемые-действия-если-проверка-истинна)
  действия-выполняемые-если-проверка-ложна)
```

Например, следующее выражение if напечатает сообщение '4 не больше чем 5' если вы вычислите его обычным образом:

```
(if (> 4 5)                             ; if-часть
    (message "5 больше чем 4!")         ; then-часть
  (message "4 не больше чем 5!"))       ; else-часть
```

Обратите внимание, что разные уровни выравнивания сразу позволяют отделить then-часть от else-части. (В GNU Emacs существуют несколько команд, которые автоматически выравнивают выражения if. See section GNU Emacs Помогает Вам Программировать.)

Вы можете расширить функцию тип-животного, чтобы включить else-часть, просто добавив ее к выражению if.

Затем посмотрите, что изменится, если вы как обычно вычислите следующую версию тип-животного, для того чтобы установить ее, и потом вычислите два примера ее использования с различными аргументами.

```
(defun тип-животного (характеристика)  ; Вторая версия
  "Напечатать сообщение в эхо-области. В зависимости от
ХАРАКТЕРИСТИКИ.
     Если ХАРАКТЕРИСТИКА `свирепость, тогда напечатать
предупреждение."
  (if (equal характеристика 'свирепость)
      (message "Это тигр!")
    (message "Оно не свирепое!")))

(тип-животного 'свирепость)

(тип-животного 'зебра)
```

Когда вы вычислите (тип-животного 'свирепость), в эхо-области будет напечатано следующее сообщение "Это тигр!", а когда вы измените аргумент на зебра, вы увидите "Оно не свирепое!".

(Конечно, если характеристика будет жестокость, сообщение "Оно не свирепое!" может быть обманчивым! Когда вы пишете реальные программы, вы всегда должны принимать в рассмотрение возможность таких неожиданных аргументов и соответствующим образом реагировать на них).

3.9 Истина и ложь в Lisp

Существует важный аспект, касающийся проверки на истинность в выражении if. Пока мы говорили об 'истине' и 'лжи' как о значении предикатов, как будто это какие-то новые объекты в Лиспе. На самом деле 'ложь' — это наш старый друг nil. Все остальное — это 'истина'.

Выражение, которое осуществляет проверку условия, возвращает true, если результат вычисления проверочного условия не nil. Другими словами, результат проверки считается истинным, если возвращенное значение число такое как 47; строка, например "привет"; или символ (любой кроме nil), например цветы; или список, или даже буфер!

До того как проиллюстрировать выше сказанное, мы поподробнее объясним значение nil.

В Лиспе, символ nil имеет два значения. Во-первых — это означает пустой список. Во-вторых — это означает false (ложь), и именно это значение возвращает проверка истинна-ложь, если проверка прошла не успешно. nil можно записать как пустой список (), или как nil. Когда дело доходит до интерпретатора Лиспа, то () и nil считаются одинаковыми. Людям, однако, больше нравится использовать nil для значения ложь, и () как обозначение пустого списка.

В Лиспе, любое значение, которое не равно nil (или не пустой список) считается истиной. Это означает, что если вычисление проверки истинна-ложь в выражении if вернет что угодно кроме пустого списка, то это будет считаться истиной. Например, если на месте проверяемого условия будет число, то при вычислении оно вернет само себя, именно как это делают числа, когда их вычисляют. В этом случае проверка истина-ложь будет считаться выполненной успешно. Проверка будет считаться неудачной только тогда, когда при вычислении проверяемого выражения оно вернет nil или пустой список.

Вы можете проверить это, вычислив два следующих примера.

В первом из них, в проверке выражения if вычисляется число 4, и так как она возвращает сама себя, это считается истиной, и поэтому вычисляется then-часть выражения if — в результате в эхо-области появляется 'истина'. Во втором примере nil означает ложь — поэтому вычисляется else-часть выражения if, и в эхо-области печатается 'ложь'.

```
(if 4
    'истина
  'ложь)

(if nil
    'истина
  'ложь)
```

Кстати, если какое-нибудь полезное значение не доступно для возвращения при тестовом условии, которое завершается успешно, то тогда интерпретатор Лиспа вернет символ t, который в таких случаях означает истина. Например, выражение (> 5 4) вернет t в результате вычисления, как вы можете проверить сами:

```
(> 5 4)
```

С другой стороны, эта функция вернет nil, если проверка завершится неудачей.

```
(> 4 5)
```

3.10 save-excursion

Функция save-excursion — четвертая и последняя особая форма, которую мы обсудим в этой главе.

В основном программы на Emacs Лиспе используются для редактирования, и функция save-excursion встречается очень часто. Она сохраняет местоположение точки и метки, выполняет тело функции, и затем восстанавливает точку и метку в их первоначальные значения. Основная роль этой функции — оберегать пользователя от неожиданных перемещений точки или метки при выполнении команд редактирования.

До обсуждения save-excursion может быть полезным вначале вспомнить что такое точка (point) и метка (mark) в GNU Emacs. Точка (point) — это текущее положение курсора. Там где курсор, там и точка. Если быть более точным на терминалах, где курсор кажется поверх символа, точка расположена, как раз перед этим символом. В Emacs Лиспе, точка — это целое число. Первый символ в буфере — число один,

второй — число два, и так далее. Функция point возвращает текущую позицию курсора как число. В каждом буфере свое собственное значение для точки.

Метка (mark) — это другая позиция в буфере; ее можно установить с помощью команды, такой как C-SPC (set-mark-command). Если метка установлена, то вы можете использовать команду C-x C-x (exchange-point-and-mark). Это заставит курсор переместиться к метке и установить метку на месте прошлого положения курсора. В добавок, если вы установите другую метку, то позиция предыдущей будет сохранена в кольце меток. Вы можете переместить курсор на место сохраненной метки, нажимая C-u C-SPC один или несколько раз.

Часть буфера между точкой и меткой называется блок (region). Много команд работают над блоком текста, например, center-region, count-lines-region, kill-region и print-region.

Особая форма save-excursion сохраняет местоположение точки и метки и восстанавливает эти позиции после того, как тело этой особой формы будет вычислено интерпретатором Лиспа. Например, если точка была в начале какого-нибудь блока текста, и некоторый код переместил в процессе вычислений точку в конец буфера, то тогда save-excursion восстановит первоначальное положение точки, после того как закончится вычисление тела save-excursion.

В Emacs, функции часто перемещают точку в процессе внутренней работы, хотя пользователь и не ожидает этого. Например так делает функция count-lines-region. Чтобы оградить пользователя от раздражающих, и с его точки зрения ненужных прыжков курсора, часто используют функцию save-excursion, чтобы сохранять положение точки и метки в местах, ожидаемых пользователем. Использование save-excursion — хороший стиль программирования.

Чтобы убедиться, что все в порядке, save-excursion восстанавливает значения точки и метки, даже если что-то идет не так в теле функции (или если быть более точным и использовать надлежащий жаргон в "случае ненормального завершения" (abnormal exit)). Эта возможность может быть очень полезной.

Кроме восстановления значений точки и метки, save-excursion запоминает текущий буфер и также восстанавливает его. Это означает, что вы можете написать программу, которая меняет буфер в процессе своей работы, и потом save-excursion переключит вас обратно в первоначальный буфер. Именно так save-excursion используется в функции append-to-buffer. (See section The Definition of append-to-buffer.)

3.10.1 Шаблон для выражения save-excursion

Шаблон для использования в программах выражения save-excursion очень прост:

```
(save-excursion
  тело...)
```

Тело функции — это одно или несколько выражений, которые последовательно будут вычислены интерпретатором Лиспа. Если в теле более одного выражения, то значение последнего будет считаться значением, возвращаемым всей функцией save-excursion. Другие выражения в теле функции будут вычислены только ради их побочных эффектов; кстати, можно сказать, что и сама функция save-excursion

используется только из-за своего побочного эффекта (восстановления точки и метки).

Более детально шаблон для save-excursion выглядит следующим образом:

```
(save-excursion
  первое-выражение-в-теле
  второе-выражение-в-теле
  третье-выражение-в-теле
   ...
  последнее-выражение-в-теле)
```

Выражения, конечно, могут быть или просто символами или списками.

В программах на Emacs Лиспе выражение save-excursion часто помещают в тело выражения let. Это выглядит следующим образом:

```
(let список-переменных
  (save-excursion
    тело...))
```

3.11 Обзор

В последней главе мы ввели в рассмотрение довольно значительное число функции и особых форм. Здесь я попытаюсь кратко повторить их все, плюс представлю несколько простых, но очень полезных функций, которые вы еще не знаете.

`eval-last-sexp`
Вычислить последнее символическое выражение перед текущим положением курсора. Возвращенное значение будет напечатано в эхо-области, если эта функция запущена без префикс-аргумента; если функция запущена с префикс-аргументом, то результат печатается в текущем буфере. Эта команда обычно привязана к сочетанию клавиш C-x C-e.
`defun`
Define function (Определить функцию). Эта особая форма может включать до пяти частей — имя, шаблон для аргументов, передаваемых в функцию, необязательную документацию, необязательное интерактивное объявление, и само тело функции.
Например:

```
(defun back-to-indentation ()
  "Переместить точку к первому видимому символу на линии."
  (interactive)
  (beginning-of-line 1)
  (skip-chars-forward " \t"))
```

`interactive`
Объявить интерпретатору, что эта функция может использоваться интерактивно. За этой особой формой может следовать строка, состоящая из нескольких частей, о том, как передать информацию в эту функцию. В этой строке могут так же содержаться подсказки отображаемые в эхо-области. Части строки разделяются друг от друга символами новой строки '\n'.
Наиболее часто используемые символы:

`b`
Имя существующего буфере.
`f`
Имя существующего файла.
`p`
Числовой префикс-аргумент. (Обратите внимание, что 'p' набран в нижнем регистре).
`r`
Точка и метка, как два числовых аргумента, самое маленькое первым. Это единственный символ, который сразу описывает два последовательных аргумента, а не один.
See section 'Code Characters for 'interactive'' in The GNU Emacs Lisp Reference Manual, для более полного списка символов, предопределенных для interactive.

`let`
Объявляет список переменных, которые будут использоваться внутри тела let, и присваивает им первоначальные значения, или nil или заданное программистом; затем вычисляет оставшиеся выражения в теле let и возвращает результат вычисления последнего из них. Внутри тела let интерпретатор Лиспа игнорирует переменные с теми же именами, которые существуют вне выражения let.
Например:

```
(let ((foo (buffer-name))
      (bar (buffer-size)))
  (message
   "В этом буфере %s ровно %d characters."
   foo bar))
```

`save-excursion`
Запоминает значение точки, метки и текущего буфера перед вычислением тела этой особой формы. Затем восстанавливает их значения к первоначальным.
Например,

```
(message "Мы на расстоянии %d символов от начала буфера."
       (- (point)
          (save-excursion
           (goto-char (point-min)) (point))))
```

if
Вычисляет первый аргумент особой формы; если результат --- истинна, вычисляет второй аргумент; иначе вычисляет третий аргумент, если он существует.
Особая форма if называется условной формой. В Emacs Лиспе существуют и другие условные формы, но if наиболее часто используемая.

Например,

```
(if (string= (int-to-string 19)
             (substring (emacs-version) 10 12))
    (message "Это 19 версия Emacs")
  (message "Это не 19 версия Emacs"))
```

equal
eq
Проверяют два объекта на равенство. equal возвращает истину если два объекта имеют одинаковую структуру и содержание. Другая функция eq возвращает истину, если два аргумента на самом деле один и тот же объект.
<
>
<=
>=
Функция < проверяет, меньше ли ее первый аргумент чем второй. Соответственно функция > проверяет больше ли ее первый аргумент чем второй. <= проверяет меньше или равен первый аргумент второго и >= соответственно больше либо равен первый аргумент второму. Эти функции работают только с численными аргументами.
message
Печатает сообщение в эхо-области. Длина сообщения ограничена только одной строкой. Первый аргумент --- это строка, которая может содержать символы `%s', `%d', `%c', на месте которых будут подставлены последующие аргументы функции message. Аргумент, подставляемый на место `%s', должен быть строкой или символом; на место `%d' подставляется число. Аргумент, который используется с `%c' тоже должен быть числом, оно будет напечатано как код для ASCII символа.
setq
set
Функция setq устанавливает значением своего первого аргумента значение второго аргумента. Первый аргумент setq не вычисляется автоматически. Эту функцию можно использовать и сразу с несколькими аргументами. Другая функция set принимает только два аргумента, и после вычисления обоих назначает значению, возвращенному первым аргументом, значение, возвращенное вычислением второго аргумента.
buffer-name
Используется без аргумента, возвращает имя буфера в виде строки.
buffer-file-name
Используется без аргумента, возвращает имя файла, связанного с данным буфером.
current-buffer
Возвращает текущий активный буфер Emacs --- это необязательно должен быть буфер, который отображен на экране.
other-buffer
Возвращает недавно выбранный буфер.
switch-to-buffer
Устанавливает буфер, который задан как аргумент активным для Emacs и одновременно отображает его в текущем окне. Обычно эта команда связана с C-x b.
set-buffer
Переключает внимание Emacs на другой буфер. Не изменяет содержимое текущего окна Emacs.
buffer-size
Возвращает число символов в текущем буфере.
point
Возвращает значение текущей позиции курсора как целое, считая число символов с начала буфера.
point-min
Возвращает минимально возможное значение точки в текущем буфере. Обычно 1, если не включено сужение.
point-max

Возвращает максимально возможное значение точки в текущем буфере. Обычно конец буфера, если не включено сужение.

3.12 Упражнения

Напишите не интерактивную функцию, которая удваивает значение своего аргумента, числа. Сделайте эту функцию интерактивной.
Напишите функцию, которая проверяет: больше ли текущее значение fill-column, чем значение аргумента, переданного в функцию, и если да, то печатает соответствующее сообщение в эхо-области.