Emacs Lisp 元素

Protesilaos Stavrou*

2025 年 4 月 14 日

本书由 Protesilaos Stavrou（亦称“Prot”）撰写，提供了 Emacs Lisp 编程语言的宏观视角。此处提供的信息对应于 1.0.0 稳定版本，发布于 2025-04-12。

- 官方页面：`https://protesilaos.com/emacs/emacs-lisp-elements`
- Git 仓库：`https://github.com/protesilaos/emacs-lisp-elements`

目录

*info@protesilaos.com

1 开始使用 Emacs Lisp

本书旨在为你提供 Emacs Lisp（亦称“Elisp”）的宏观视角。这是你用来扩展 Emacs 的编程语言。Emacs 是一个可编程的文本编辑器：它解释 Emacs Lisp 并相应地执行操作。你可以在不编写任何一行代码的情况下使用 Emacs：它已经具备了许多功能。不过，你随时可以通过对你自己编写或从他人处（例如以包的形式）获取的 Elisp 代码进行求值，来对其编程以精确地执行你想要的操作。

对你自己的文本编辑器进行编程既有用又有趣。例如，你可以将一系列你反复执行的操作组合成一个单一命令，然后将其分配给一个按键绑定：按下该键——砰！——一次性执行所有中间任务。这让你更高效，同时也将编辑器变成了一个舒适的工作环境。

有趣的部分在于你如何着手编写代码。没有任何你必须遵守的义务。完全没有！你为了编程而编程。这是一种扩展你视野的娱乐活动。此外，你还能培养你的 Elisp 技能，如果你将来选择修改 Emacs 的某些行为，这些技能可能会很有用。

捣鼓 Emacs 是体验的一部分。它教会你毫不妥协地坚持自己对于编辑器如何工作的看法。关键在于掌握足够的 Elisp 知识，这样你就不会因为一些琐碎的事情不起作用而花费太多时间去“玩乐”或感到沮丧。我写这篇文章时，自己就是一个没有计算机科学或相关学科背景的捣鼓者：我是通过在编辑器中反复试

验、摸索来学习 Emacs Lisp 的。我名义上的目标是改进我一遍又一遍重复的某些微小动作：我追求效率，结果却发现了更深刻的东西。学习扩展我的编辑器是一次充实的体验，结果我变得更有效率了。Emacs 做了我想让它做的事情，我对此很满意。

本书的每一章通常都简短且切中要点。有些对初学者更友好，而另一些则深入探讨了高级主题。各章之间存在链接，这正是一个参考手册应有的做法。你可以来回查阅以找到你需要的内容。

你在这里会看到文本是散文和代码的结合。后者可能是实际的 Elisp 代码，也可能是捕捉底层模式的伪代码。我鼓励你在 Emacs 内部或随时可以打开 Emacs 的情况下阅读本书。这样，你可以尝试运行我给你的函数，以进一步体会它们的细微之处。

我采用的"宏观视角"方法旨在涵盖我在使用 Emacs Lisp 时经常遇到的概念。本书不能替代 Emacs Lisp 参考手册，并且绝不应被视为我对任何 Elisp 形式进行评论的真理之源。

祝你好运，享受过程！

2 求值 Emacs Lisp

你在 Emacs 中做的每一件事都会调用某个函数。它对 Emacs Lisp 代码进行求值，读取返回值并产生副作用 (副作用与返回值)。

你在键盘上键入一个键，一个字符就被写入当前缓冲区。那是一个绑定到按键的函数。它实际上是一个 *交互式*函数，因为你是通过按键绑定而不是通过某个程序来调用它。交互式函数被称为"命令"。不过，不要让交互性这个实现细节分散你的注意力，要认识到你在 Emacs 中执行的每一个动作都涉及 Emacs Lisp 的求值。

另一种常见的交互模式是使用 M-x (`execute-extended-command`) 键，它默认运行命令 ~execute-extended-command~：它会产生一个 minibuffer 提示，要求你按名称选择一个命令，然后继续执行它。

Emacs 可以从任何地方求值 Elisp 代码。如果你的缓冲区中有一些 Elisp 代码，你可以将光标放在其闭合括号的末尾，然后键入 C-x C-e (`eval-last-sexp`)。类似地，你可以使用命令 `eval-buffer` 和 `eval-region` 分别对当前缓冲区或高亮区域进行操作。

`eval-last-sexp` 也适用于符号 (符号、平衡表达式与引用)。例如，如果你将光标放在变量 `buffer-file-name` 的末尾并使用 C-x C-e (`eval-last-sexp`)，你将得到该变量的值，这个值要么是 ~nil~，要么是你正在编辑的文件在文件系统中的路径。

有时上述方法不适用于你想要做的事情。假设你打算编写一个复制当前缓冲区文件路径的命令。为此，你的代码需要测试变量 `buffer-file-name` 的值 (作为数据结构的缓冲区)。但你不想在你的实际文件中输入 `buffer-file-name~`，然后使用前面提到的某个 `Elisp` 求值命令，最后再撤销你的编辑。这既繁琐又容易出错！在当前缓冲区中运行 `Elisp` 的最佳方法是键入 `{{{kbd(M-:)}}}` (`~eval-expression`): 它会打开 minibuffer 并期望你写入想要求值的代码。从那里键入 RET 以继续。求值是在最后一个缓冲区作为当前缓冲区的情况下完成的（即调用 `eval-expression` 之前的当前缓冲区)。

这里有一些 Emacs Lisp 代码，你可能想在 (i) 对应于文件的缓冲区与 (ii) 与磁盘上任何文件都无关的缓冲区中尝试运行。

```
;; Use `eval-expression' to evaluate this code in a file-visiting
```

```
;; buffer versus a buffer that does not visit any file.
(if buffer-file-name
    (message "The path to this file is `%s'" buffer-file-name)
  (message "Sorry mate, this buffer is not visiting a file"))
```

当你在试验代码时，你想测试它的行为。使用命令 `ielm` 打开一个交互式 shell。它会把你带到一个提示符下，你可以在那里键入任何 Elisp 代码并按 RET 来求值它。返回值会打印在紧随其后下方。或者，切换到 `*scratch*` 缓冲区。如果它正在使用主模式 `lisp-interaction-mode~`（这是变量 `~initial-major-mode` 的默认值），那么你可以在该缓冲区中自由移动，并在某些代码的末尾键入 C-j (`eval-print-last-sexp`) 来对其求值。这与 `eval-last-sexp` 的工作方式几乎相同，但附加效果是把返回值放在你刚刚求值的表达式正下方。

除此之外，你可以依赖 Emacs 的自文档特性来弄清楚当前的状态。例如，要了解变量 `major-mode` 的缓冲区局部值，你可以执行 C-h v (`describe-variable`)，然后搜索该变量。生成的帮助缓冲区将告知你 `major-mode` 的当前值。这个帮助命令以及许多其他命令，如 `describe-function~`、`~describe-keymap~`、`~describe-key` 和 ~describe-symbol~，提供了关于 Emacs 对给定对象所知信息的深入了解。帮助缓冲区将显示相关信息，例如定义给定函数的文件路径，或者一个变量是否被声明为缓冲区局部变量。

Emacs 是“自文档化的”，因为它报告其自身状态。你不需要显式更新帮助缓冲区。这通过求值相关代码自动发生：Emacs 实际上向你展示你正在处理的任何东西的最新值。

3 副作用与返回值

Emacs Lisp 拥有函数。它们接受输入并产生输出。在其最纯粹的形式中，函数是一个仅返回值而不改变其环境任何东西的计算。一个函数的返回值被用作另一个函数的输入，这实际上形成了一个计算链。因此，你可以依赖一个函数的返回值来表达类似“如果这个成功了，那么也做另一件事，否则做别的事情甚至什么都不做”的意思。

Elisp 是扩展和控制 Emacs 的语言。这意味着它也会影响编辑器的状态。当你运行一个函数时，它可以进行永久性更改，例如在光标处插入一些文本、删除一个缓冲区、创建一个新窗口等等。这些更改将对未来的函数调用产生影响。例如，如果前一个函数删除了某个缓冲区，那么本应写入该缓冲区的下一个函数就无法再执行其任务了：缓冲区已经消失了！

当你编写 Elisp 时，你必须同时考虑返回值和副作用。如果你很草率，你会因为所有那些未经深思熟虑的环境更改而得到意想不到的结果。但如果你细致地使用副作用，你就能够将 Elisp 的潜力发挥到极致。例如，想象你定义了一个函数，它遵循这样的逻辑：“创建一个缓冲区，转到那里，写入一些文本，将缓冲区保存到我偏好的位置的文件中，然后回到我调用此函数之前的位置，同时保持创建的缓冲区打开。”所有这些都是副作用，而且它们都很有用。你的函数可能也有一些有意义的返回值，你可以将其用作另一个函数的输入。例如，你的函数可能会返回它生成的缓冲区对象，以便下一个函数可以在那里做些事情，比如在一个单独的 frame 中显示该缓冲区并使其文本变大。

其思想是操纵编辑器的状态，让 Emacs 做你设想的事情。有时这意味着你的代码有副作用。其他时候，副作用是无用的，甚至与你预期结果背道而驰。随着你获得更多经验和扩展技能范围，你将不断提炼你对于需要做什么的直觉 (符号、平衡表达式与引用)。没问题；别紧张！

4 作为数据结构的缓冲区

缓冲区以字符序列的形式持有数据。例如，这些数据就是你打开文件时看到的文本。每个字符都存在于一个给定的位置，这是一个数字。函数 `point` 给出你所在点位（point）的位置，这通常对应于光标所在的位置 (求值 Emacs Lisp)。在缓冲区的开头，~point~ 返回值 `1` (副作用与返回值)。有大量函数返回缓冲区位置，例如 `point-min~`、`~point-max~`、`~line-beginning-position` 和 `re-search-forward~`。其中一些会有副作用，比如 `~re-search-forward` 会将光标移动到给定的匹配项。

当你在 Emacs Lisp 中编程时，你经常依赖缓冲区来做以下一些事情:

将文件内容提取为字符串 将缓冲区视为一个大字符串。你可以使用函数 `buffer-string` 将其全部内容获取为一个可能非常巨大的字符串。你也可以使用 `buffer-substring` 函数或其对应的 `buffer-substring-no-pro` 函数获取两个缓冲区位置之间的子字符串 (文本自身具有属性)。想象一下，你将其作为更广泛操作的一部分来执行，该操作 (i) 打开文件，(ii) 转到特定位置，(iii) 复制找到的文本，(iv) 切换到另一个缓冲区，以及 (v) 将找到的内容写入这个新缓冲区。

呈现某些操作的结果 你可能有一个显示即将到来的假期的函数。你的代码在后台进行计算，并最终将一些文本写入缓冲区。最终产品被展示出来。取决于你如何处理它，你可能需要求值函数 `get-buffer-create` 或其更严格的替代方案 `get-buffer~`。如果你需要清除已存在缓冲区的内容，你可能会使用 `~with-current-buf` 宏临时切换到你目标指向的缓冲区，然后要么调用函数 `erase-buffer` 删除所有内容，要么使用 `delete-region` 将删除范围限制在两个缓冲区位置之间。最后，函数 `display-buffer` 或 `pop-to-buffer` 会将缓冲区置于 Emacs 窗口中。

将变量与给定缓冲区关联 在 Emacs Lisp 中，变量可以采用缓冲区局部值，该值与其全局对应值不同。有些变量甚至被声明为始终是缓冲区局部的，例如 `buffer-file-name~`、`~fill-column` 和 `default-directory~`。假设你正在做类似返回访问给定目录中文件的缓冲区列表的事情。你会遍历 `~buffer-list` 函数的返回值，通过测试 `buffer-file-name` 的特定值来相应地过滤结果 (使用 `if~`、`~cond` 等进行基本控制流)。这个特定变量始终可用，不过你总是可以使用 `setq-local` 宏将值赋给当前缓冲区中的变量。

后一点也许是最开放的一点。缓冲区就像一组变量的集合，包括它们的内容、它们正在运行的主模式以及它们拥有的所有缓冲区局部值。在下面的代码块中，我使用 `seq-filter` 函数遍历函数 `buffer-list` 的返回值 (符号、平衡表达式与引用)。

```
(seq-filter
 (lambda (buffer)
   "Return BUFFER if it is visible and its major mode derives from `text-mode'."
   (with-current-buffer buffer
     ;; The convention for buffers which are not meant to be seen by
     ;; the user is to start their name with an empty space.  We are
     ;; not interested in those right now.
     (and (not (string-prefix-p " " (buffer-name buffer)))
```

```
          (derived-mode-p 'text-mode))))
 (buffer-list))
```

这将返回一个缓冲区对象列表，这些对象通过了以下测试：(i) 对用户“可见”且 (ii) 它们的主模式要么是 `text-mode` 要么是从中派生的。上述代码也可以这样写 (何时使用命名函数或 lambda 函数)：

```
(defun my-buffer-visble-and-text-p (buffer)
  "Return BUFFER if it is visible and its major mode derives from `text-mode'."
  (with-current-buffer buffer
    ;; The convention for buffers which are not meant to be seen by
    ;; the user is to start their name with an empty space.  We are
    ;; not interested in those right now.
    (and (not (string-prefix-p " " (buffer-name buffer)))
         (derived-mode-p 'text-mode))))

(seq-filter #'my-buffer-visble-and-text-p (buffer-list))
```

与缓冲区一样，Emacs 窗口和 frame 也有它们自己的参数。我不会涵盖那些，因为它们的用途更为专门化，而且概念是相同的。只需知道它们是数据结构，你可以利用它们为你带来优势，包括遍历它们 (遍历列表元素)。

5 文本自身具有属性

正如像数据结构一样工作的缓冲区 (作为数据结构的缓冲区)，任何文本也可能有与其关联的属性。这是你可以使用 Emacs Lisp 来检查的元数据。例如，当你在某个编程缓冲区中看到语法高亮时，这就是文本属性的效果。某个函数负责“属性化（propertise）”或“字体化（fontify）”相关文本，并决定将一个称为“face”的对象应用于它。Face 是将排版和颜色属性捆绑在一起的构造，例如字体家族和字重，以及前景和背景色调。要获取一个包含光标处文本属性信息的帮助缓冲区，请键入 M-x (`execute-extended-command`) 然后调用命令 ~describe-char~。它会告诉你它看到的字符、它是用什么字体渲染的、它是哪个代码点，以及它的文本属性是什么。

假设你正在编写你自己的主模式。在实验的早期阶段，你想手动将文本属性添加到缓冲区中所有出现的短语 `I have properties` 上，该缓冲区的主模式是 ~fundamental-mode~，所以你可以这样做（上次搜索的匹配数据）：

```
(defun my-add-properties ()
  "Add properties to the text \"I have properties\" across the current buffer."
  (goto-char (point-min))
  (while (re-search-forward "I have properties" nil t)
    (add-text-properties (match-beginning 0) (match-end 0) '(face error))))
```

实际尝试一下。使用 C-x b (`switch-to-buffer`)，输入一些与现有缓冲区不匹配的随机字符，然后按

RET 访问该新缓冲区。它运行 fundamental-mode~，意味着没有“字体化”发生，因此 ~my-add-properties 将按预期工作。现在粘贴以下内容：

```
This is some sample text. Will the phrase "I have properties" use the `bold' face?

What does it even mean for I have properties to be bold?
```

继续使用 M-: (eval-expression) 并调用函数 ~my-add-properties~。成功了吗？它应用的 face 叫做 ~error~。忽略这个词的语义：我选择它仅仅是因为它通常以相当强烈和明显的方式进行样式化（尽管你当前的主题可能会有不同的做法）。

有一些函数可以查找给定缓冲区位置的属性，还有一些函数可以向前和向后搜索给定的属性。具体的细节现在并不重要。我只想让你记住，文本不仅仅是其组成的字符。要了解更多详情，请键入 M-x (execute-extended-command) 来调用命令 ~shortdoc~。它会询问你一个文档组。选择 text-properties 以了解更多信息。嗯，对那里列出的所有内容都使用 ~shortdoc~。我一直都这样做。

6 符号、平衡表达式与引用

对于不熟悉 Emacs Lisp 的人来说，这是一种括号非常多的语言！这是一个简单的函数定义：

```
(defun my-greet-person (name)
  "Say hello to the person with NAME."
  (message "Hello %s" name))
```

我刚刚定义了一个名为 my-greet-person 的函数。它有一个参数列表，具体来说，是一个包含一个参数的列表，参数名为 name=。然后是可选的文档字符串，供用户理解代码和/或函数的意图。~my-greet-person~ 接受 =name 并将其作为参数传递给 message 函数，最终打印出问候语。~message~ 函数将文本记录在 *Messages* 缓冲区中，你可以使用 C-h e (view-echo-area-messages) 直接访问该缓冲区。无论如何，以下是你如何用它期望的一个参数来调用 ~my-greet-person~：

```
(my-greet-person "Protesilaos")
```

现在用多个参数做同样的事情：

```
(defun my-greet-person-from-country (name country)
  "Say hello to the person with NAME who lives in COUNTRY."
  (message "Hello %s of %s" name country))
```

并这样调用它：

```
(my-greet-person-from-country "Protesilaos" "Cyprus")
```

即使对于最基本的任务，你也会用到很多括号。但不要害怕！这些实际上使结构化地理解你的代码变得更简单。如果现在感觉不是这样，那是因为你还不习惯。一旦你习惯了，就回不去了。

任何 Lisp 方言（Emacs Lisp 是其中之一）的基本思想是，你有用来界定列表的括号。列表由元素组成。列表要么被求值以产生某些计算的结果，要么按原样返回以用于其他求值 (副作用与返回值)：

列表作为函数调用 当一个列表被求值时，第一个元素是函数名，其余元素是传递给该函数的参数。你已经在上面看到了这一点，我是如何用 `"Protesilaos"` 作为参数调用 `my-greet-person` 的。~my-greet-person-from-country~ 也是同样的原理，参数是 `"Protesilaos"` 和 =”Cyprus”=。

列表作为数据 当一个列表不被求值时，它的任何元素在一开始都没有特殊含义。它们都作为一个列表原样返回，没有进一步的改变。当你不希望你的列表被求值时，你在它前面加上一个单引号字符。例如，=’(”Protesilaos” ”Prot” ”Cyprus”)= 是一个包含三个元素的列表，应该按原样返回。

考虑后一种情况，你还没见过。你有一个元素列表，你想从中获取一些数据。在最基本的层面上，函数 `car` 和 `cdr` 分别返回第一个元素和所有剩余元素的列表：

```
(car '("Protesilaos" "Prot" "Cyprus"))
;; => "Protesilaos"

(cdr '("Protesilaos" "Prot" "Cyprus"))
;; => ("Prot" "Cyprus")
```

这里的单引号至关重要，因为它指示 Emacs 不要对列表求值。否则，对这个列表的求值会将第一个元素，即 =”Protesilaos”=，视为函数名，并将列表的其余部分视为该函数的参数。由于你没有定义这样的函数，你会得到一个错误。

Emacs Lisp 中的某些数据类型是“自求值的”。这意味着如果你对它们求值，它们的返回值就是你已经看到的东西。例如，字符串 `"Protesilaos"` 的返回值是 =”Protesilaos”=。这对于字符串、数字、关键字、符号以及特殊的 `nil` 或 `t` 都成立。下面是一个包含这些类型示例的列表，你可以通过调用函数 `list` 来构建它：

```
(list "Protesilaos" 1 :hello 'my-greet-person-from-country nil t)
;; => ("Protesilaos" 1 :hello 'my-greet-person-from-country nil t)
```

`list` 函数对其传递的参数进行求值，除非它们被引用。你得到没有明显变化的返回值的原因是自求值。注意 `my-greet-person-from-country` 的引用方式与我们引用一个不希望求值的列表的方式相同。如果没有它，~my-greet-person-from-country~ 将被求值，除非它也被定义为一个变量，否则将返回一个错误。

将单引号视为一个明确的指令：“不要对下面的内容求值”。更具体地说，它是一个指令，在通常会发生求值的情况下不执行求值 (列表内部的部分求值)。换句话说，你不想在一个被引用的列表内部引用某些东西，因为那等同于对其引用两次：

```
;; This is the correct way:
'(1 :hello my-greet-person-from-country)

;; It is wrong to quote `my-greet-person-from-country' because the
;; entire list would not have been evaluated anyway.  The mistake here
;; is that you are quoting what is already quoted, like doing
;; ''my-greet-person-from-country.
'(1 :hello 'my-greet-person-from-country)
```

现在你可能想知道为什么我们引用了 `my-greet-person-from-country` 而没有引用其他东西?原因是你在那里看到的其他所有东西实际上都是“自引用的”,即自求值的另一面。而 `my-greet-person-from-country` 是一个符号。一个“符号”是对自身以外某物的引用：它要么表示某个计算——一个函数——要么表示一个变量的值。如果你写一个符号而不引用它，你实际上是在告诉 Emacs“给我这个符号所代表的值”。就 `my-greet-person-from-country` 而言，如果你尝试这样做会得到一个错误，因为这个符号不是一个变量，因此尝试从中获取值是行不通的。

请记住，Emacs Lisp 有一个“宏”的概念，它基本上是一个模板系统，用于编写实际扩展成其他代码然后被求值的代码。在一个宏内部，你控制引用的完成方式，这意味着前面提到的规则可能不适用于涉及宏的调用，即使它们仍然在宏的扩展形式内部使用 (宏或特殊形式内部的求值)。

随着你接触更多的 Emacs Lisp 代码，你会遇到前面带有井号的引号，例如 =#'some-symbol=。这个“锐引号”(sharp quote)，正如它被称呼的那样，与常规引号相同，但增加了特指一个函数的语义。程序员因此可以更好地表达给定表达式的意图，而字节编译器可以在内部执行必要的检查和优化。有鉴于此，请阅读关于函数 `quote` 和 `function` 的内容，它们分别对应于常规引号和锐引号。

7 列表内部的部分求值

你已经对 Emacs Lisp 代码的样子有了一些概念 (符号、平衡表达式与引用)。你有一个列表，它要么被求值，要么按原样获取。还有另一种情况，即列表应该被部分求值，或者更具体地说，它应该被当作数据而不是函数调用，同时其中某些元素仍然需要被求值。

在下面的代码块中，我定义了一个名为 `my-greeting-in-greek` 的变量，这是一个希腊语中常用的短语，字面意思是“祝你健康”，发音为“yah sou”。为什么要用希腊语? 好吧，你已经有了产生 Lisp 这一切的 ~lambda~，所以你不妨也了解其余的部分 (何时使用命名函数或 lambda 函数)!

```
(defvar my-greeting-in-greek "Γ    "
  "Basic greeting in Greek to wish health to somebody.")
```

现在我想用 `message` 函数做些实验，以更好地理解求值是如何工作的。让我从引用列表，从而按原样获取它的场景开始:

```
(message "%S" '(one two my-greeting-in-greek four))
;;=> "(one two my-greeting-in-greek four)"
```

你会注意到变量 `my-greeting-in-greek` 没有被求值。我得到了符号，即实际的 ~my-greeting-in-greek~，而不是它所代表的值。这是预期的结果，因为整个列表都被引用了，因此，其中的所有内容都不会被求值。

现在检查下一个代码块，以理解我如何告诉 Emacs 我希望整个列表仍然被引用，但特别地让 `my-greeting-in-gre` 被求值，以便它被其值替换：

```
(message "%S" `(one two ,my-greeting-in-greek four))
;; => "(one two \"Γ    \" four)"
```

请密切注意这里的语法。我使用的是反引号或称后引号，而不是单引号，在我们的例子中，这也被称为“准引用”。它的行为类似于单引号，但前面带有逗号的任何内容除外。逗号是一个“对后面的东西求值”的指令，并且只在准引用的列表内部有效。后面跟着的“东西”要么是一个符号，要么是一个列表。列表当然可以是一个函数调用。那么让我使用 `concat` 来问候某个人，同时将所有内容作为一个列表返回：

```
(message "%S" `(one two ,(concat my-greeting-in-greek " " "Π    ") four))
;; => "(one two \"Γ    Π    \" four)"
```

请记住，如果你根本不引用这个列表，你会得到一个错误，因为第一个元素 `one` 会被视为一个函数的符号，该函数将用所有其他元素作为其参数来调用。很可能 `one` 在你当前的 Emacs 会话中没有被定义为函数，或者那些参数对它来说无论如何都没有意义。此外，~two~ 和 `four` 随后会被视为变量，因为它们没有被引用，在这种情况下，那些变量也必须被定义，否则会导致更多错误。

除了逗号操作符，还有 `,@`（这到底怎么发音？也许是“comma at”？），这是“拼接（splicing）”的表示法。这是行话，用来替代说“返回值是一个列表，我希望你移除它最外层的括号”。实际上，原本会返回 `'(one two three)` 的代码现在返回 =one two three=。这种差异在孤立的情况下可能没有多大意义，但一旦你将这些元素视为应该独立工作的表达式，而不是仅仅作为被引用列表的元素时，它就有意义了。我不会在这里详细阐述一个例子，因为我认为这最好在定义宏的背景下进行介绍 (宏或特殊形式内部的求值)。

你很可能不需要使用部分求值的知识。它在宏中更常见，但可以在任何地方应用。无论如何都要意识到它，因为在某些情况下，你至少需要理解你所依赖的某些代码正在做什么。

最后，既然我向你介绍了一些希腊语单词，我现在把你当作我的朋友了。这里有一个我小时候的笑话。我试图向我的英语老师解释某个事件。由于我缺乏表达自己的词汇，我开始使用希腊语单词。我的老师有一个严格的只用英语回应的政策，所以她说：“It is all Greek to me.”（对我来说都是希腊语）。我不知道她的回答是一个习语，意思是“我不明白你在说什么”，我轻率地回答说：“Yes, Greek madame; me no speak England very best.”（是的，希腊语，夫人；我英语说得不是最好）。我当时其实不是初学者，但我不会放过取笑这种情况的机会。就像你应该记得享受捣鼓 Emacs 的时光一样。但说够了！回到阅读这本书。

8 宏或特殊形式内部的求值

在最基本的 Emacs Lisp 代码情况下，你有要么被求值要么不被求值的列表 (符号、平衡表达式与引用)。如果你玩得更花哨一点，你有只被部分求值的列表 (列表内部的部分求值)。但有时，你看着一段代码，却

无法理解为什么常规的引用和求值规则不适用。在你看到实际例子之前，先检查一个典型的函数调用，它也涉及一个变量的求值：

```
(concat my-greeting-in-greek " " "Π     ")
```

你在关于部分求值的部分遇到过这段代码。你这里有一个对函数 `concat` 的调用，后面跟着三个参数。其中一个参数是一个变量，即 ~my-greeting-in-greek~。当这个列表被求值时，Emacs 实际做的是首先对参数（包括 ~my-greeting-in-greek~）进行求值，以获取它们各自的值，然后才用这些值调用 ~concat~。你可以将整个操作过程想象如下：

- 这里有一个列表。
- 它没有被引用。
- 所以你应该对它求值。
- 第一个元素是函数名。
- 剩余的元素是传递给该函数的参数。
- 检查参数是什么。
- 对每个参数求值，以将其解析为其真实值。
- 字符串是自求值的，而 `my-greeting-in-greek` 是一个变量。
- 你现在拥有了每个参数的值，包括符号 `my-greeting-in-greek` 的值。
- 用你得到的所有值调用 ~concat~。

换句话说，下面两个产生相同的结果（假设 `my-greeting-in-greek` 是常量）：

```
(concat my-greeting-in-greek " " "Π     ")
```

```
(concat "Γ     " " " "Π     ")
```

这是可预测的。它遵循单引号的基本逻辑：如果它被引用了，就不要对它求值并按原样返回，否则就对它求值并返回其值。但是你会发现很多情况下，这种预期的模式似乎没有被遵循。考虑这个使用 `setq` 将符号绑定到给定值的常见情况：

```
(setq my-test-symbol "Protesilaos of Cyprus")
```

上面的表达式看起来像一个函数调用，这意味着 (i) 列表没有被引用，(ii) 第一个元素是函数名，以及 (iii) 剩余的元素是传递给该函数的参数。在某种程度上，这都是正确的。不过你可能会期望 `my-test-symbol` 被当作一个变量，它会被原地求值以返回其结果，而这个结果反过来将是传递给函数的实际参数。然而，这并不是 `setq` 的工作方式。原因是它是一个特殊情况，内部执行的是这个：

```
(set 'my-test-symbol "Protesilaos of Cyprus")
```

在这里，事情就如预期了。幕后没有发生什么魔法。那么，~setq~ 是为了方便用户不必每次都引用符号。是的，这使得推理它变得有点困难，但你会习惯它，最终一切都会变得有意义。希望你会习惯这种特殊形式，就像你在 `setq` 以及 `defun` 等许多其他形式中发现的那样。这是一个你已经见过的函数：

```
(defun my-greet-person-from-country (name country)
  "Say hello to the person with NAME who lives in COUNTRY."
  (message "Hello %s of %s" name country))
```

如果常规的求值规则适用，那么参数列表应该被引用。否则，你会期望 `(name country)` 被解释为一个函数调用，其中 `name` 是函数符号，~country~ 是它的参数，而 `country` 本身也会是一个变量。但这并不是实际发生的情况，因为 `defun` 会在内部将该参数列表视为已被引用。

另一个常见的场景是使用 `let` (使用 `if-let*` 及其相关形式进行控制流)。它的一般形式如下：

```
;; This is pseudo-code
(let LIST-OF-LISTS-AS-VARIABLE-BINDINGS
  BODY-OF-THE-FUNCTION)
```

`LIST-OF-LISTS-AS-VARIABLE-BINDINGS` 是一个列表，其中每个元素都是 `(SYMBOL VALUE)` 形式的列表。这里有一些实际的代码：

```
(let ((name "Protesilaos")
      (country "Cyprus"))
  (message "Hello %s of %s" name country))
```

继续特殊形式的主题，如果 `let` 是一个典型的函数调用，那么 `LIST-OF-LISTS-AS-VARIABLE-BINDINGS` 就必须被引用。否则，它会被求值，在这种情况下，第一个元素将是函数名。但这会返回一个错误，因为函数名将对应于另一个列表，即 `(name "Protesilaos")=`，而不是一个符号。使用 `~let~` 时一切正常，因为它内部对其 `=LIST-OF-LISTS-AS-VARIABLE-BINDINGS` 进行了引用。

对于许多特殊形式以及宏，比如流行的 ~use-package~（用于在你的 Emacs 初始化文件中配置包），预期会有类似的行为。这些宏各自如何工作取决于它们的设计方式。我不会在这里深入探讨技术细节，因为我希望这本书能够长期有用，侧重于原则而不是可能随时间变化的实现细节。

要了解给定宏实际扩展成什么，请将光标放在其闭合括号的末尾，并调用命令 ~pp-macroexpand-last-sexp~。它将生成一个新的缓冲区，显示扩展后的 Emacs Lisp 代码。这才是实际替代宏被求值的内容。

有了这些基础，是时候编写一个宏了。这就像一个模板，使你能够避免重复自己。在语法上，宏很可能依赖于准引用、逗号操作符以及拼接机制的使用 (列表内部的部分求值)。这里有一个简单的场景，我们希望在一个临时缓冲区中运行一些代码，同时将 `default-directory` 设置为用户的主目录。

```
(defmacro my-work-in-temp-buffer-from-home (&rest expressions)
  "Evaluate EXPRESSIONS in a temporary buffer with `default-directory' set to the user's home."
  `(let ((default-directory ,(expand-file-name "~/")))
     (with-temp-buffer
       (message "Running all expression from the `%s' directory" default-directory)
       ,@expressions)))
```

在这个定义中，=&rest= 使得后面的参数成为一个列表。所以你可以向它传递任意数量的参数，所有这些参数都将被收集到一个名为 EXPRESSIONS 的列表中。审慎使用部分求值确保了宏不会立即被求值，而只在被调用时才求值。传递给它的参数将被放置在你指定的位置。这是一个使用此宏的调用：

```
(progn
  (message "Now we are doing something unrelated to the macro")
  (my-work-in-temp-buffer-from-home
   (message "We do stuff inside the macro")
   (+ 1 1)
   (list "Protesilaos" "Cyprus")))
```

如果你将光标放在 my-work-in-temp-buffer-from-home 的闭合括号处，你可以通过键入 M-x (execute-extende 然后调用命令 pp-macroexpand-last-sexp 来确认它扩展成了什么。这是我得到的结果：

```
(let ((default-directory "/home/prot/"))
  (with-temp-buffer
    (message "Running all expression from the `%s' directory" default-directory)
    (message "We do stuff inside the macro")
    (+ 1 1)
    (list "Protesilaos" "Cyprus")))
```

将其与其上下文中的其余代码拼凑在一起，我得到这个：

```
(progn
  (message "Now we are doing something unrelated to the macro")
  (let ((default-directory "/home/prot/"))
    (with-temp-buffer
      (message "Running all expression from the `%s' directory" default-directory)
      (message "We do stuff inside the macro")
      (+ 1 1)
      (list "Protesilaos" "Cyprus"))))
```

记住这个例子，将 Elisp 宏视为一种表达方式：“这个小东西帮助我更简洁地表达这个更大的过程，而实际运行的代码仍然是后者的代码。”

我上面写的宏，其主体以准引用开始，所以你无法体会到其中求值的细微差别。让我向你展示另一种方法，我编写一个宏，让它可以定义几个几乎相同的交互式函数 (让你的交互式函数也能从 Lisp 调用中工作)。

```
(defmacro my-define-command (name &rest expressions)
  "Define command with specifier NAME that evaluates EXPRESSIONS."
  (declare (indent 1))
  (unless (symbolp name)
    (error "I want NAME to be a symbol"))
  (let ((modifined-name (format "modified-version-of-%s" name)))
    `(defun ,(intern modifined-name) ()
       (interactive)
       ,(message "The difference between `%s' and `%s'" modifined-name name)
       ,@expressions)))
```

`my-define-command` 大致可以分为两部分：(i) 直接求值的部分和 (ii) 展开以供进一步求值的部分。后一部分以准引用开始。当调用宏时，这种区别很重要，因为前一部分会立即执行，所以如果我们遇到错误，它将永远不会展开然后运行 =EXPRESSIONS=。对下面的例子使用 `pp-macroexpand-last-sexp` 来看看我的意思。为方便起见，我在每种情况下面都包含了宏展开。

```
(my-define-command first-demo
  (message "This is what my function does")
  (+ 1 10)
  (message "And this"))
;; =>
;;
;; (defun modified-version-of-first-demo nil
;;   (interactive)
;;   "The difference between ‘modified-version-of-first-demo’ and ‘first-demo’"
;;   (message "This is what my function does")
;;   (+ 1 10)
;;   (message "And this"))


(my-define-command second-demo
  (list "Protesilaos" "Cyprus")
  (+ 1 1)
  (message "Arbitrary expressions here"))
;; =>
;;
;; (defun modified-version-of-second-demo nil
;;   (interactive)
```

```
;;   "The difference between ‘modified-version-of-second-demo’ and ‘second-demo’"
;;   (list "Protesilaos" "Cyprus")
;;   (+ 1 1)
;;   (message "Arbitrary expressions here"))


(my-define-command "error scenario"
  (list "Will" "Not" "Reach" "This")
  (/ 2 0))
;; => ERROR...
```

你需要宏吗? 不总是需要，但有些情况下，一个定义良好的宏会使你的代码更优雅。重要的是你对求值如何工作有一个概念，这样你就不会被所有那些括号搞糊涂。否则，你可能会期望发生与实际得到的结果不同的事情。

9 遍历列表元素

编程中的一个常见例程是遍历一个项目列表并对每个项目执行一些计算。Emacs Lisp 有通用的 `while` 循环，以及一系列更专门用于遍历列表元素的函数，例如 `mapcar~`、`~mapc~`、`~dolist~`、`~seq-filter~`、`~seq-remove` 等等。根据你正在做的事情，你遍历元素的目的是产生一些副作用和/或测试返回值 (副作用与返回值)。我将向你展示一些例子，让你决定哪种工具最适合手头的任务。

从 `mapcar` 开始，它将一个函数应用于列表的每个元素。然后它获取每次迭代的返回值并将它们收集到一个新列表中。这就是 `mapcar` 整体的返回值。在下面的代码块中，我使用 `mapcar` 遍历一个数字列表，将它们增加 =10=，并返回一个包含增加后数字的新列表。

```
(mapcar
 (lambda (number)
   (+ 10 number))
 '(1 2 3 4 5))
;; => (11 12 13 14 15)
```

在上面的代码块中，我使用了一个 ~lambda~，即匿名函数 (何时使用命名函数或 lambda 函数)。下面是相同的代码，但使用了同名函数，即命名函数:

```
(defun my-increment-by-ten (number)
  "Add 10 to NUMBER."
  (+ 10 number))

(mapcar #'my-increment-by-ten '(1 2 3 4 5))
;; => (11 12 13 14 15)
```

注意这里我们引用了同名函数 (符号、平衡表达式与引用)。

`mapcar` 将返回值收集到一个新列表中。有时这是无用的。假设你想对一个保存所有访问文件的未保存缓冲区的函数求值。在这种情况下，你不在乎累积结果：你只想获得直接保存缓冲区的副作用。为此，你可以使用 ~mapc~，它总是返回它操作的列表：

```
(mapc
 (lambda (buffer)
   (when (and (buffer-file-name buffer)
              (buffer-modified-p buffer))
     (save-buffer)))
 (buffer-list))
```

上述方法的替代方案是 ~dolist~，它用于产生副作用但总是返回 ~nil~：

```
(dolist (buffer (buffer-list))
  (when (and (buffer-file-name buffer)
             (buffer-modified-p buffer))
    (save-buffer)))
```

你会注意到 `dolist` 是一个宏，所以它的某些部分似乎与基本列表及其适用的求值规则表现不同 (宏或特殊形式内部的求值)。这需要习惯代码的表达方式。

何时使用 `dolist` 而不是 `mapc` 是风格问题。如果你使用命名函数，在我看来 `mapc` 更简洁。否则 `dolist` 更容易阅读。这是我使用一些伪代码的方法：

```
;; I like this:
(mapc #'NAMED-FUNCTION LIST)

;; I also like a `dolist' instead of a `mapc' with a `lambda':
(dolist (element LIST)
  (OPERATE-ON element))

;; I do not like this:
(mapc
 (lambda (element)
   (OPERATE-ON element))
 LIST)
```

虽然 `dolist` 和 `mapc` 是为了副作用，但你仍然可以在 `let` 和相关形式的帮助下，利用它们来累积结果 (使用 `if-let*` 及其相关形式进行控制流)。根据具体情况，这种方法可能比依赖 `mapcar` 更有意义。这是一个带注释的草图：

```
;; Start with an empty list of `found-strings'.
(let ((found-strings nil))
  ;; Use `dolist' to test each element of the list '("Protesilaos" 1 2 3 "Cyprus").
  (dolist (element '("Protesilaos" 1 2 3 "Cyprus"))
    ;; If the element is a string, then `push' it to the `found-strings', else skip it.
    (when (stringp element)
      (push element found-strings)))
  ;; Now that we are done with the `dolist', return the new value of `found-strings'.
  found-strings)
;; => ("Cyprus" "Protesilaos")


;; As above but reverse the return value, which makes more sense:
(let ((found-strings nil))
  (dolist (element '("Protesilaos" 1 2 3 "Cyprus"))
    (when (stringp element)
      (push element found-strings)))
  (nreverse found-strings))
;; => ("Protesilaos" "Cyprus")
```

为了完整起见，前面的例子如果使用 `mapcar` 将必须这样做：

```
(mapcar
 (lambda (element)
   (when (stringp element)
     element))
 '("Protesilaos" 1 2 3 "Cyprus"))
;; => ("Protesilaos" nil nil nil "Cyprus")


(delq nil
      (mapcar
       (lambda (element)
         (when (stringp element)
           element))
       '("Protesilaos" 1 2 3 "Cyprus")))
;; => ("Protesilaos" "Cyprus")
```

因为 `mapcar` 会愉快地累积所有的返回值，它返回一个包含 `nil` 的列表。如果你想要那样，你可能甚至不会费心在那里使用 `when` 子句。因此，~delq~ 被应用于 `mapcar` 的返回值，以删除所有 `nil` 的实例。现在将这项繁琐的工作与 `seq-filter` 进行比较：

```
(seq-filter #'stringp '("Protesilaos" 1 2 3 "Cyprus"))
;; => ("Protesilaos" "Cyprus")
```

当你只需要测试元素是否满足谓词函数然后返回该元素时，~seq-filter~ 是最佳工具。但你不能返回其他东西。而 `mapcar` 会毫无怨言地接受任何返回值，例如以下：

```
(delq nil
      (mapcar
       (lambda (element)
         (when (stringp element)
           ;; `mapcar' accumulates any return value, so we can change
           ;; the element to generate the results we need.
           (upcase element)))
       '("Protesilaos" 1 2 3 "Cyprus")))
;; => ("PROTESILAOS" "CYPRUS")

(seq-filter
 (lambda (element)
   (when (stringp element)
     ;; `seq-filter' only returns elements that have a non-nil return
     ;; value here, but it returns the elements, not what we return
     ;; here.  In other words, this `lambda' does unnecessary work.
     (upcase element)))
 '("Protesilaos" 1 2 3 "Cyprus"))
;; => ("Protesilaos" "Cyprus")
```

你如何遍历列表元素将取决于你想要做什么。没有一个单一的函数能为你做所有事情。理解细微差别，你就可以开始了。哦，还有，一定要看看内置的 `seq` 库（使用 M-x (`execute-extended-command`)，调用 `find-library~`，然后搜索 `~seq~`)。你现在正在查看 `=seq.el=` 的源代码：它定义了许多函数，如 `~seq-take~`、`~seq-find~`、`~seq-union~`。另一种方法是调用命令 `~shortdoc` 并阅读关于文档组 `list` 以及 `sequence` 的内容。

10 上次搜索的匹配数据

当你使用 Emacs Lisp 时，你会遇到“匹配数据（match data）”的概念以及伴随的函数 `match-data~`、`~match-beginning~`、`~match-string` 等等。这些指的是上次搜索的结果，通常由函数 `re-search-forward~`、`~looking-at~`、`~string-match` 及相关函数执行。每次你执行搜索时，匹配数据都会更新。请注意这个常见的副作用 (副作用与返回值)。如果你忘记了它，你的代码很可能不会做正确的事情。

在下面的代码块中，我定义了一个函数，它在当前缓冲区中执行搜索，并返回一个不带文本属性（如果相关）的匹配数据列表 (文本自身具有属性)。

```
(defun my-get-match-data (regexp)
  "Search forward for REGEXP and return its match data, else nil."
  (when (re-search-forward regexp nil t)
    (list
     :beginning (match-beginning 0)
     :end (match-end 0)
     :string (match-string-no-properties 0))))
```

然后你可以用一个字符串参数来调用它，该参数表示一个 Emacs Lisp 正则表达式：

```
(my-get-match-data "Protesilaos.*Cyprus")
```

如果正则表达式匹配，那么你会得到匹配数据。这是一些示例文本：

```
Protesilaos lives in the mountains of Cyprus.
```

将光标放在该文本之前，并使用 M-: (`eval-expression`) 来求值带有我上面显示的 regexp 的 ~my-get-match-data~。你会得到一个返回值，正如预期的那样。

按照 `my-get-match-data` 的编写方式，它做了两件事：(i) 它具有将光标移动到找到的文本末尾的副作用，以及 (ii) 它返回一个包含我指定的匹配数据的列表。在许多情况下，你不希望有前面提到的副作用：光标应该停留在原来的位置。因此，你可以将你的代码包装在 `save-excursion` 中 (切换到另一个缓冲区、窗口或 narrowed 状态)：它会做它必须做的事情，并最终恢复 `point` (运行一些代码或回退到其他代码)：

```
(defun my-get-match-data (regexp)
  "Search forward for REGEXP and return its match data, else nil."
  (save-excursion ; we wrap our code in a `save-excursion' to inhibit the side effect
    (when (re-search-forward regexp nil t)
      (list
       :beginning (match-beginning 0)
       :end (match-end 0)
       :string (match-string-no-properties 0)))))
```

如果你对这个版本的 `my-get-match-data` 求值，然后重试我上面的函数调用，你会注意到你是如何得到预期的返回值，而没有光标移动到匹配文本末尾的副作用。在实践中，这是一个有用的工具，可以与 `save-match-data` 结合使用。想象一下，你想在你正在执行的另一次搜索内部进行一次向前搜索，例如仅仅测试上下文中是否存在某个正则表达式的匹配，但需要抑制对你计划操作的匹配数据的修改。因此：

```
(defun my-get-match-data-with-extra-check (regexp)
  "Search forward for REGEXP followed by no spaces and return its match data, else nil."
  (save-excursion
    (when (and (re-search-forward regexp nil t)
```

```
            (save-match-data (not (looking-at "[\s\t]+"))))
      ;; Return the match data of the first search.  The second one
      ;; which tests for spaces or tabs is just an extra check, but we
      ;; do not want to use its match data, hence the `save-match-data'
      ;; around it.
      (list
       :beginning (match-beginning 0)
       :end (match-end 0)
       :string (match-string-no-properties 0)))))
```

对函数 my-get-match-data-with-extra-check 求值，然后用 M-: (eval-expression) 调用它来测试，它在下面的第二个例子中返回一个非 nil 值，但在第一个例子中不返回。这是预期的结果。

```
(my-get-match-data-with-extra-check "Protesilaos.*Cyprus")
;; => nil



;; Protesilaos, also known as "Prot", lives in the mountains of Cyprus   .

(my-get-match-data-with-extra-check "Protesilaos.*Cyprus")
;; => (:beginning 41988 :end 42032 :string "Protesilaos lives in the mountains of Cyprus")



;; Protesilaos lives in the mountains of Cyprus.
```

11 切换到另一个缓冲区、窗口或 narrowed 状态

当你使用 Emacs Lisp 以编程方式做事时，你会遇到需要离开当前位置的情况。你可能需要切换到另一个缓冲区，切换到给定缓冲区的窗口，甚至修改你正在编辑的缓冲区中可见的内容。在任何时候，这都涉及一个或多个副作用，这些副作用很可能应该在你的函数完成其工作时被撤销 (副作用与返回值)。

也许最常见的情况是恢复 point~。你有一些代码在缓冲区中向后或向前移动以执行对给定文本片段的匹配。但是之后，你需要将光标留在它原来的位置，否则用户会失去方向感。将你的代码包装在 ~save-excursion 中，你就搞定了，正如我在别处展示的那样 (上次搜索的匹配数据):

```
(save-excursion ; restore the `point' after you are done
  MOVE-AROUND-IN-THIS-BUFFER)
```

save-window-excursion 的原理相同，它允许你选择另一个窗口（例如使用 ~select-window~），在其缓冲区中移动，然后恢复窗口的原状:

```
(save-window-excursion
```

```
  (select-window SOME-WINDOW)
  MOVE-AROUND-IN-THIS-BUFFER)
```

save-restriction 允许你恢复缓冲区的当前 narrowing 状态。然后你可以选择要么 widen 要么 narrow-to-region~（以及像 ~org-narrow-to-subtree 这样的相关命令），做你必须做的事情，然后将缓冲区恢复到其原始状态。

```
;; Here we assume that we start in a widened state.  Then we narrow to
;; the current Org heading to get all of its contents as one massive
;; string.  Then we widen again, courtesy of `save-restriction'.
(save-restriction
  (org-narrow-to-subtree)
  (buffer-string))
```

根据具体情况，你可能需要组合使用上述方法。请注意 save-restriction 的文档告诉你将 save-excursion 作为最外层的调用。除此之外，你还会发现一些情况需要不同的方法来执行某些条件行为 (运行一些代码或回退到其他代码)。

12 使用 if~、~cond 等进行基本控制流

你不需要任何条件逻辑来执行基本操作。例如，如果你编写一个向下移动 15 行的命令，当它无法移动超过你指定的数量时，它自然会在缓冲区末尾停止。使用 defun~，你编写一个交互式函数（即一个“命令”）来无条件地使用 ~forward-line 向下移动 15 行（用负数调用它以向相反方向移动）：

```
(defun my-15-lines-down ()
  "Move at most 15 lines down."
  (interactive)
  (forward-line 15))
```

my-15-lines-down 几乎是最简单的：它包装了一个基本函数，并向其传递一个固定的参数，本例中是数字 =15=。使用 M-x (execute-extended-command) 然后按名称调用此命令。它有效！一旦你决定仅在满足给定条件时才执行某些操作，事情就会变得更加复杂。这种逻辑序列不同分支之间的“控制流”是用 if~、~when~、~unless 和 cond 等来表达的。根据具体情况，~and~ 以及 or 可能就足够了。

让你的 my-15-lines-down 变得更聪明一点怎么样？当它处于缓冲区的绝对末尾时，让它向上移动 15 行。为什么？因为这是一个演示，所以为什么不呢？测试点位是否在缓冲区末尾的谓词函数是 eobp~。一个“谓词”是一个函数，当其条件满足时返回 true（技术上是非 ~nil~），否则返回 ~nil (副作用与返回值)。至于这个奇怪的名字，Emacs Lisp 中的惯例是以 p 后缀结束谓词函数：如果函数名由多个单词组成（通常用破折号分隔），则谓词函数命名为 =NAME-p=，例如 ~string-match-p~；否则命名为 =NAMEp=，例如 ~stringp~。

```
(defun my-15-lines-down-or-up ()
```

```
  "Move at most 15 lines down or go back if `eobp' is non-nil."
  (interactive)
  (if (eobp)
      (forward-line -15)
    (forward-line 15)))
```

对这个函数求值，然后键入 M-x (execute-extended-command) 并调用 my-15-lines-down-or-up 来感受一下。下面是一个类似的想法，如果 eobp 返回非 ~nil~，它会抛出错误并退出正在做的事情：

```
(defun my-15-lines-down-or-error ()
  "Throw an error if `eobp' returns non-nil, else move 15 lines down."
  (interactive)
  (if (eobp)
      (error "Already at the end; will not move further")
    (forward-line 15)))
```

Emacs Lisp 的一个怪癖（或许一直以来都是一个特性）是它的缩进方式。只需标记你写好的代码并键入 TAB: Emacs 会负责按照应有的方式对其进行缩进。在 if 语句的情况下，“then”部分比逻辑的“else”部分缩进得更深。这种缩进没有特殊含义：你可以把所有东西写在一行上，比如 =(if COND THIS ELSE)=，顺便说一下，这看起来就像你典型的列表 (符号、平衡表达式与引用)。缩进的作用是帮助你识别括号的不平衡。如果不同的表达式都以看起来奇怪的方式对齐，那么你很可能缺少一个括号或括号太多了。通常，同一级别的表达式都会以相同的方式对齐。更深层次的表达式会有更多的缩进，依此类推。经验会让你能够发现括号不匹配的错误。但即使你没有识别出来，你最终也会得到一个错误。请放心!

if 的写法就像一个接受两个或更多参数的函数。“或更多”的部分都算作“else”逻辑的一部分。因此，=(if COND THIS)= 没有“else”后果，而 (if COND THIS ELSE1 ELSE2 ELSE3) 将按顺序运行 ELSE1=、=ELSE2 和 ELSE3 作为“else”分支的一部分。当你考虑到适当的缩进时，它看起来是这样的：

```
(if COND
    THIS
  ELSE1
  ELSE2
  ELSE3)
```

那么如果 THIS 部分需要多于一个函数调用呢? Elisp 有 progn 形式，你可以用它来包装函数调用并将它们作为一个单一参数传递。把它们放在一起，你的代码现在会像这样：

```
(if COND
    (progn
      THIS1
      THIS2
      THIS3)
```

```
  ELSE1
  ELSE2
  ELSE3)
```

如果你不需要“else”部分，使用 when 来表达你的意图。在内部，这是一个宏，实际上代表 (if COND (progn EXPRESSIONS))=，其中 =EXPRESSIONS 是一个或多个表达式。一个 when 看起来像这样：

```
(when COND
  THIS1
  THIS2
  THIS3)
```

类似地，~unless~ 的意思是 =(when (not COND) EXPRESSIONS)=。它也是一个宏，扩展为一个 if 语句：

```
(unless COND
  THIS1
  THIS2
  THIS3)
```

当你测试的条件有多个部分时，你可以依赖 and 以及 ~or~：

```
(when (or THIS THAT)
  EXPRESSIONS)

(when (and THIS THAT)
  EXPRESSIONS)

(when (or (and THIS THAT) OTHER)
  EXPRESSIONS)
```

根据具体情况，多个 if~、~when~、~or~、~and 的组合会看起来很别扭。你可以使用 cond 将逻辑分解为不同的条件，这些条件从上到下依次测试。~cond~ 的写法是一个列表的列表，这些列表不需要引用(宏或特殊形式内部的求值)。抽象地说，它看起来像这样：

```
(cond
 (CONDITION1
  CONSEQUENCES1)
 (CONDITION2
  CONSEQUENCES2)
 (CONDITION3
```

```
  CONSEQUENCES3)
 (t
  CONSEQUENCES-FALLBACK))
```

每个后果可以是任意数量的表达式，就像你上面看到的 `when` 一样。这是一个展示 `cond` 行为的玩具函数：

```
(defun my-toy-cond (argument)
  "Return a response depending on the type of ARGUMENT."
  (cond
   ((and (stringp argument)
         (string-blank-p argument))
    (message "You just gave me a blank string; try harder!"))
   ((stringp argument)
    (message "I see you can do non-blanks string; I call that progress."))
   ((null argument)
    (message "Yes, the nil is an empty list like (), but do not worry about it"))
   ((listp argument)
    (message "Oh, I see you are in the flow of using lists!"))
   ((symbolp argument)
    (message "What's up with the symbols, mate?"))
   ((natnump argument)
    (message "I fancy those natural numbers!"))
   ((numberp argument)
    (message "You might as well be a math prodigy!"))
   (t
    (message "I have no idea what type of thing your argument `%s' is" argument))))
```

我希望你对其求值并传递不同的参数来测试它做了什么 (求值 Emacs Lisp)。这里有两个例子：

```
(my-toy-cond "")
;; => "You just gave me a blank string; try harder!"

(my-toy-cond '(1 2 3))
;; => "Oh, I see you are in the flow of using lists!"
```

以上所有在 Emacs Lisp 中都很常见。另一个强大的宏是 `pcase`，由于其特殊性，我们将单独考虑它 ([[#h:pattern-match-with-pcase-and-related][使用 `pcase` 及相关形式进行模式匹配]])。

13 使用 `if-let*` 及其相关形式进行控制流

`let` 和 `let*` 声明仅在当前作用域内（即 `let` 的 `BODY` 部分）可用的变量。因此：

```
(let BINDINGS
  BODY)

(let ((variable1 value1)
      (variable2 value2))
  BODY)
```

BINDINGS 是一个列表的列表，不需要被引用 (宏或特殊形式内部的求值)。而 BODY 由一个或多个表达式组成，我在本书的其他地方也将其命名为 =EXPRESSIONS=。~let~ 和 ~let*~（读作“let star”）的区别在于，后者使较早的绑定可用于较晚的绑定。像这样：

```
;; This works because `greeting' can access `name' and `country',
;; courtesy of `let*':
(let* ((name "Protesilaos")
       (country "Cyprus")
       (greeting (format "Hello %s of %s" name country)))
  (DO-STUFF-WITH greeting))

;; But this fails...
(let ((name "Protesilaos")
      (country "Cyprus")
      (greeting (format "Hello %s of %s" name country)))
  (DO-STUFF-WITH greeting))
```

有时你想要做的是，当且仅当这些绑定都非 nil 时才创建它们。如果它们的值是 nil~，那么它们对你来说是无用的，在这种情况下你会做别的事情 ([[#h:basic-control-flow-with-if-cond-and-others][使用 ~if~、~cond 等进行基本控制流]])。当你使用函数调用或某个其他变量的返回值创建绑定时，值可能是 nil 也可能不是。你总是可以编写这样的代码：

```
(let ((variable1 (SOME-FUNCTION SOME-ARGUMENT))
      (variable2 (OTHER-FUNCTION OTHER-ARGUMENT)))
  (if (and variable1 variable2) ; simply test both for non-nil
      THIS
    ELSE))
```

但是你可以用 if-let* 做同样的事情，其中 THIS 部分仅在所有绑定都非 nil 时运行：

```
(if-let* ((variable1 (SOME-FUNCTION SOME-ARGUMENT))
          (variable2 (OTHER-FUNCTION OTHER-ARGUMENT)))
    THIS
  ELSE)
```

在 ELSE 部分，绑定 variable1 和 variable2 不存在：它们只存在于代码的 THIS 部分。

when-let* 与 when 相同，意味着它没有“else”逻辑。如果它的某个绑定是 nil~，那么整个 ~when-let* 返回 ~nil~。无需赘述这一点。

随着你深入研究 Emacs Lisp 生态系统，你会遇到 if-let* 的用法，它 (i) 像 let 或 let* 一样创建多个绑定，但 (ii) 也调用一个谓词函数来测试它们是否应该继续执行逻辑的 THIS 部分。记住，如果 if-let* 的某个绑定返回 ~nil~，它会直接转到 =ELSE=。考虑这个例子：

```
(if-let* ((variable1 (SOME-FUNCTION SOME-ARGUMENT))
          ;; The _ signifies intent: "do not bind this; I only care
          ;; about the return value being non-nil".  What we are doing
          ;; here is test if `variable1' is a string: if it is, we
          ;; continue with the bindings, otherwise we move to the ELSE
          ;; part of the code.
          (_ (string-match-p variable1))
          (variable2 (OTHER-FUNCTION OTHER-ARGUMENT)))
    THIS
  ELSE)
```

没有天生优越的做事方式。关键在于为手头的任务使用正确的工具。有时你希望创建绑定，即使它们的值是 ~nil~。选择有意义的方式。

14 使用 pcase 及相关形式进行模式匹配

一旦你掌握了用 Emacs Lisp 表达思想的流程，你将能熟练使用 if~、~cond 及类似形式 (使用 if~、~cond 等进行基本控制流)。如果使用 if-let*~，你甚至可能玩得更花哨 ([[#h:control-flow-with-if-let-and-friends][使用 ~if-let* 及其相关形式进行控制流]])。然而，无论你怎么做，有些情况下，更简洁的表达方式无疑更有益。这就是 pcase 发挥作用的地方。在其最基本的表述中，它类似于 cond~，因为它测试给定表达式的返回值与一系列条件的匹配情况。这里有一个例子，将变量 ~major-mode 的缓冲区局部值与几个已知的符号进行相等性比较：

```
(pcase major-mode
  ('org-mode (message "You are in Org"))
  ('emacs-lisp-mode (message "You are in Emacs Lisp"))
  (_ (message "You are somewhere else")))
```

以上与这个 cond 的想法相同：

```
(cond
 ((eq major-mode 'org-mode)
  (message "You are in Org"))
```

```
 ((eq major-mode 'emacs-lisp-mode)
  (message "You are in Emacs Lisp"))
 (t
  (message "You are somewhere else")))
```

一些程序员可能会争辩说 pcase 更优雅。我认为在这个具体的例子中确实如此，但我保持灵活和务实：我会使用任何对我正在编写的代码更有意义的方式。谈到优雅，我应该告诉你，几乎所有的条件逻辑都可以用一种看似意想不到的方式来完成。考虑一下我在本书中的例子是如何重复使用 message 的，而实际上唯一改变的部分是传递给该函数的实际字符串/参数。这样做同样有效：

```
(message
 (pcase major-mode
   ('org-mode "You are in Org")
   ('emacs-lisp-mode "You are in Emacs Lisp")
   (_ "You are somewhere else")))
```

对于 if~、~when 和其余的也是同样的想法。

回到 pcase 有何不同的主题。如果你阅读它的文档，你会意识到它有自己的迷你语言，或称“领域特定语言”(DSL)。这对于宏来说很常见 (宏或特殊形式内部的求值)。它们定义了求值如何完成以及哪种表达式被特殊处理。那么让我送你这个玩具函数，它说明了现在正在讨论的 DSL 的一些主要特性：

```
(defun my-toy-pcase (argument)
  "Use `pcase' to return an appropriate response for ARGUMENT."
  (pcase argument
    (`(,one ,_ ,three)
     (message "List where first element is `%s', second is ignored, third is `%s'" one three))
    (`(,one . ,two)
     (message "Cons cell where first element is `%s' and second is `%s'" one two))
    ((pred stringp)
     (message "The argument is a string of some sort"))
    ('hello
     (message "The argument is equal to the symbol `hello'"))
    (_ (message "This is the fallback"))))
```

去对那个函数求值，然后试用一下 (求值 Emacs Lisp)。下面是几个例子：

```
(my-toy-pcase '("Protesilaos" "of" "Cyprus"))
;; => "List where first element is ‘Protesilaos’, second is ignored, third is ‘Cyprus’"

(my-toy-pcase '("Protesilaos" . "Cyprus"))
;; => "Cons cell where first element is ‘Protesilaos’ and second is ‘Cyprus’"
```

其中一些子句是表达 cond 的不同方式。可以说更好，但在我看来并非明显的赢家。令人印象深刻且真正带来范式转变的是“解构（destructuring）”的概念，即对表达式进行的模式匹配，它有效地将列表或 cons cell 的元素通过 let 绑定到它们对应的索引。用于这种解构的语法是晦涩难懂的，直到你将其与用于部分求值的准引用和逗号联系起来 (列表内部的部分求值)。考虑到这一点，将 pcase-let~、~pcase-let*~、~pcase-lambda 和 pcase-dolist 视为普通的 let~、~let*~、~lambda 和 dolist 的变体，增加了支持解构的特性。不过，它们并没有做 pcase 的任何额外功能——只是在它们熟悉的行为之上增加了析构！这在你处理函数返回值为列表时特别有用。我不会详细阐述，因为这是一个高级用例。如果你已经达到那个水平，你不需要我告诉你该写什么。对于我们其余的人，像我一样，通常处理更简单的代码，~pcase-let~ 足以说明这个原则：

```
(defun my-split-string-at-space (string)
  "Split STRING at the space, returning a list of strings."
  (split-string string "\s"))

(pcase-let ((`(,one ,_ ,three) (my-split-string-at-space "Protesilaos of Cyprus")))
  (message "This is like `let', but we got `%s' and `%s' via destructuring" one three))
;; => "This is like ‘let’, but we got ‘Protesilaos’ and ‘Cyprus’ via destructuring"
```

你是否使用 pcase 和一般的解构取决于你。你不需要它们来编写高质量的代码。不过，你可能会同意那些认为它们本质上更优雅的人的观点，并因此选择使用它们来使代码简洁而富有表现力。

15 运行一些代码或回退到其他代码

你典型的代码将依赖 if~、~cond 等进行控制流 (使用 if~、~cond 等进行基本控制流)。根据你的具体需求或风格考虑，它甚至可能包括 pcase (使用 pcase 及相关形式进行模式匹配) 以及 if-let* (使用 if-let* 及其相关形式进行控制流)。然而，有些情况下，你必须在主要操作结束或退出后运行额外的代码。其思想是清理你创建的任何中间状态。逻辑是“用所有必要的副作用做这件事，然后无论发生什么，现在也做那件事，以便（除其他外）撤销副作用。”这是“回溯保护（unwinding）”的概念，通过 unwind-protect 实现。

在下面的代码块中，我定义了一个函数，它会产生一个 minibuffer 提示，要求你提供 y 或 n 的答案，这是“是”或“否”的简写表示法。它测试 y-or-n-p 的返回值来决定需要做什么。当提示符打开时，该函数会高亮显示当前缓冲区中正则表达式 (defun 的所有实例。在你完成 minibuffer 及其后果后，这些高亮必须消失。

```
(defun my-prompt-with-temporary-highlight ()
  "Ask for confirmation and highlight all instances of a regexp while waiting."
  (let ((regexp "(defun"))
    (unwind-protect
        (progn
          (highlight-regexp regexp)
          (if (y-or-n-p "Should we proceed or not? ")
              (message "You have decided to proceed")
```

```
          (message "You prefer not to continue")))
    (unhighlight-regexp regexp))))
```

在你的 Emacs 中尝试上面的代码来感受一下。当“是或否”提示处于活动状态时，也执行 C-g (keyboard-quit) 或 C-] (abort-recursive-edit) 来确认即使代码从未通过提示阶段，高亮也会被移除。你甚至可以修改函数以产生错误：它将创建一个回溯 (backtrace)，在你从 *Backtrace* 窗口执行 q (debugger-quit) 后，仍然会产生回溯保护的效果。

```
(defun my-prompt-with-temporary-highlight-try-with-error ()
  "Ask for confirmation and highlight all instances of a regexp while waiting."
  (let ((regexp "(defun"))
    (unwind-protect
        (progn
          (highlight-regexp regexp)
          (error "This error makes no sense here; close the backtrace to test the unwinding")
          (if (y-or-n-p "Should we proceed or not? ")
              (message "You have decided to proceed")
            (message "You prefer not to continue")))
      (unhighlight-regexp regexp))))
```

退一步看，你会发现 unwind-protect 是像 save-excursion 和 save-restriction (切换到另一个缓冲区、窗口或 narrowed 状态) 这样的专门形式的更通用形式，同时它支撑着 save-match-data (上次搜索的匹配数据) 以及许多其他函数/宏，例如 with-temp-buffer 和 save-window-excursion~。~unwind-protect 不做的是对信号（例如来自 error 函数的信号）做出特殊响应：它会允许错误发生，这意味着将显示回溯并且你的代码将在那里退出（但是回溯保护仍然会起作用，正如我已经解释过的，一旦你关闭回溯）。要让你的代码以更可控的方式处理信号，你必须依赖 ~condition-case~。

使用 condition-case~，你可以完全控制代码的行为，包括它应该如何处理错误。换句话说，你的 Elisp 将表达这样的意图：“我想做这个，但如果我得到一个错误，我想做那个来代替。”有许多信号需要考虑，所有这些都来自 ~signal 函数。这些包括符号 error~、~user-error~、~args-out-of-range~、~wrong-type-argument~、~wrong-length-argument 和 quit~，此外还有程序员可能认为必要的任何其他信号。在下面的代码块中，我向你展示 ~condition-case 的样子。请记住，有时由于底层形式的实现方式，你不会像通常那样进行引用 (宏或特殊形式内部的求值)。我使用的例子与我用于 unwind-protect 的例子相同。

```
(defun my-prompt-with-temporary-highlight-and-signal-checks ()
  "Ask for confirmation and highlight all instances of a regexp while waiting."
  (let ((regexp "(defun"))
    (condition-case nil
        (progn
          (highlight-regexp regexp)
          (if (y-or-n-p "Should we proceed or not? ")
```

```
            (user-error "You have decided to proceed; but we need to return a `user-error'")
          (error "You prefer not to continue; but we need to return an `error'")))
     (:success
      (unhighlight-regexp regexp)
      (message "No errors, but still need to unwind what we did, plus whatever else we want here
     (quit
      (unhighlight-regexp regexp)
      (message "This is our response to the user aborting the prompt"))
     (user-error
      (unhighlight-regexp regexp)
      (message "This is our response to the `user-error' signal"))
     (error
      (unhighlight-regexp regexp)
      (message "This is our response to the `error' signal")))))
```

上面的函数说明了前面提到的回溯保护概念和处理信号的机制。~condition-case~ 的抽象结构在我看来像是 `let~`、`~unwind-protect` 和 `cond` 的混合体。这些条件可能包括特殊的处理程序 =:success=，正如我在那里展示的那样。诚然，我写的代码永远不会导致那个特定的成功情况，但你可以修改提示符之后发生的事情，比如说，调用 `message` 而不是 `user-error` 函数，这将被视为一个成功的结论。否则，我认为我写的表达式准确地告诉你这个程序如何响应它接收到的信号。

我还没有涵盖的是 `condition-case` 类似 `let` 的方面，即它如何将错误数据绑定到此作用域内的变量。在我上面的实现中，它是你看到的 ~nil~，意味着我选择不执行这样的绑定，因为我不需要它的数据。下面我决定使用它，仅仅是为了演示。

```
(defun my-prompt-with-temporary-highlight-and-signal-checks-with-error-report ()
  "Ask for confirmation and highlight all instances of a regexp while waiting."
  (let ((regexp "(defun"))
    (condition-case error-data-i-got
        (progn
          (highlight-regexp regexp)
          (if (y-or-n-p "Should we proceed or not? ")
              (user-error "You have decided to proceed; but we need to return a `user-error'")
            (error "You prefer not to continue; but we need to return an `error'")))
      (:success
       (unhighlight-regexp regexp)
       (message "No errors, but still need to unwind what we did, plus whatever else we want here
       (message "The error is `%s' and its data is `%S'" (car error-data-i-got) (cdr error-data-i
      (quit
       (unhighlight-regexp regexp)
       (message "This is our response to the user aborting the prompt")
       (message "The error is `%s' and its data is `%S'" (car error-data-i-got) (cdr error-data-i
      (user-error
```

```
      (unhighlight-regexp regexp)
      (message "This is our response to the `user-error' signal")
      (message "The error is `%s' and its data is `%S'" (car error-data-i-got) (cdr error-data-i
     (error
      (unhighlight-regexp regexp)
      (message "This is our response to the `error' signal")
      (message "The error is `%s' and its data is `%S'" (car error-data-i-got) (cdr error-data-i
```

有时 unwind-protect 和 condition-case 是适合这项工作的工具。我希望这些例子给了你宏观的视角，你现在已经准备好用 Emacs Lisp 编写你自己的程序了。

16 何时使用命名函数或 lambda 函数

lambda 是一个匿名函数。它与 defun 相对，后者定义一个具有给定名称的函数。何时使用哪一个很大程度上是风格问题。不过在某些情况下，某种方法更合适。经验法则是：如果你需要多次使用该函数，那么给它一个名字，然后通过名字调用它。否则，你实际上每次都会重新定义它，这使得重写你的程序变得困难。相比之下，如果函数只是临时相关的，那么使用 lambda 就可以了。

在某些情况下，你会有一个内部使用 lambda 的命名函数。修改你在本书中找到的一个例子 (遍历列表元素)：

```
(defun my-increment-numbers-by-ten (numbers)
  "Add 10 to each number in NUMBERS and return the new list."
  (mapcar
   (lambda (number)
     (+ 10 number))
   numbers))

(my-increment-numbers-by-ten '(1 2 3))
;; => (11 12 13)
```

命名函数内部的 lambda 也可以在 let 的帮助下用于重复做某事。例如，你可能有一个函数需要使用 mapc 作为副作用来问候一个人员列表，并且你不想多次定义同一个函数：

```
(defun my-greet-teams (&rest teams)
  "Say hello to each person in TEAMS and return list with all persons per team.
Each member of TEAMS is a list of strings."
  (let* ((greet-name (lambda (name)
                       (message "Hello %s" name)))
         (greet-team-and-names (lambda (team)
                                 (message "Greeting the team of `%s'..." team)
                                 (mapc greet-name team))))
```

```
    (mapcar greet-team-and-names teams)))

(my-greet-teams
 '("Pelé" "Ronaldo")
 '("Maradona" "Messi")
 '("Beckenbauer" "Neuer")
 '("Platini" "Zidane")
 '("Baresi" "Maldini")
 '("Eusebio" "Cristiano Ronaldo")
 '("Xavi" "Iniesta")
 '("Charlton" "Shearer")
 '("Puskas" "Kubala")
 '("All of the Greece Euro 2004 squad ;)"))
;; => (("Pelé" "Ronaldo") ("Maradona" "Messi") ...)
```

问候语在这种情况下是副作用,并且可以在 *Messages* 缓冲区中找到。你可以使用 C-h e (view-echo-area-messa
快速访问该缓冲区。~my-greet-teams~ 具体在做什么并不重要。关注命名函数和其内部匿名函数的组合。

17 让你的交互式函数也能从 Lisp 调用中工作

当函数使用 interactive 规范声明时，它们可以交互式地使用。这会将它们变成“命令”。它们可以通过名称调用，首先执行 M-x (execute-extended-command) 然后找到该命令。它们也可以分配给一个按键并直接通过按该键调用。在其最简单的形式中，~interactive~ 规范是一个未引用的列表，如 (interactive)~。这里有一个简单的例子，它调用 ~read-string 来产生一个 minibuffer 提示，该提示接受用户输入并将其作为字符串返回：

```
(defun my-greet-person ()
  (interactive)
  (message "Hello %s" (read-string "Whom to greet? ")))
```

上述实现的问题在于它仅在交互式使用中有用。如果你想通过程序非交互式地发出这样的问候，你需要编写另一个函数，该函数做几乎相同的事情，只是它接受一个 NAME 参数。像这样：

```
(defun my-greet-person-with-name (name)
  "Greet person with NAME."
  (message "Hello %s" name))
```

你不需要编写两个实际上做同样事情的独立函数。相反，你可以有一个带有参数的函数，它根据是交互式调用还是编程方式调用来决定如何获取传递给它的参数的值。考虑这种情况：

```
(defun my-greet-interactive-and-non-interactive (name)
```

```
  "Greet person with NAME.
When called interactively, produce a minibuffer prompt asking for NAME.

When called from Lisp, NAME is a string."
  (interactive (list (read-string "Whom to greet? ")))
  (message "Hello %s" name))
```

我在那里写的文档准确地告诉你发生了什么。不过让我进一步解释 `interactive~`：它接受一个参数，该参数是一个列表，对应于当前 `~defun` 的参数列表。在这种情况下，~defun~ 有一个包含单个元素 `NAME` 的参数列表。因此，~interactive~ 也有一个包含一个元素的列表，其值对应于 =NAME=。如果参数不止一个，那么 `interactive` 必须相应地编写：它的每个元素将对应于列表中相同索引处的参数。

你传递给 `interactive` 的这个表达式列表本质上是将值绑定到参数的准备工作。当你交互式地调用上面的函数时，你实际上告诉 Emacs 在这种情况下 `NAME` 是调用 `read-string` 的返回值。对于更多参数，原理相同，但我还是写下来以明确说明：

```
(defun my-greet-with-two-parameters (name country)
  "Greet person with NAME from COUNTRY.
When called interactively, produce a minibuffer prompt asking for NAME
and then another prompt for COUNTRY.

When called from Lisp, NAME and COUNTRY are strings."
  (interactive
   (list
    (read-string "Whom to greet? ")
    (read-string "Where from? ")))
  (message "Hello %s of %s" name country))

(my-greet-with-two-parameters "Protesilaos" "Cyprus")
;; => "Hello Protesilaos of Cyprus"
```

仔细编写 `interactive` 规范，你最终会得到一个既经济又灵活的丰富代码库。

18 版权信息

19 GNU 自由文档许可证

20 索引

20.1 函数索引

20.2 变量索引

20.3 概念索引

“‘