Emacs Lisp Elements

Protesilaos Stavrou*

2025 年 4 月 14 日

This book, written by Protesilaos Stavrou, also known as "Prot", provides a big picture view of the Emacs Lisp programming language.

The information furnished herein corresponds to stable version 1.0.0, released on 2025-04-12.

- Official page: `https://protesilaos.com/emacs/emacs-lisp-elements`
- Git repository: `https://github.com/protesilaos/emacs-lisp-elements`

目录

*info@protesilaos.com

1 Getting started with Emacs Lisp

The purpose of this book is to provide you with a big picture view of Emacs Lisp, also known as "Elisp". This is the programming language you use to extend Emacs. Emacs is a programmable text editor: it interprets Emacs Lisp and behaves accordingly. You can use Emacs without ever writing a single line of code: it already has lots of features. Though you can, at any time, program it to do exactly what you want by evaluating some Elisp that either you wrote yourself or got from another person, such as in the form of a package.

Programming your own text editor is both useful and fun. You can, for example, streamline a sequence of actions you keep doing by combining them in a single command that you then assign to a key binding: type the key and—bam!—perform all the intermediate tasks in one go. This makes you more efficient while it turns the editor into a comfortable working environment.

The fun part is how you go about writing the code. There are no duties you have to conform with. None! You program for the sake of programming. It is a recreational activity that expands your horizons.

Plus, you cultivate your Elisp skills, which can prove helpful in the future, should you choose to modify some behaviour of Emacs.

Tinkering with Emacs is part of the experience. It teaches you to be unapologetically opinionated about how your editor works. The key is to know enough Elisp so that you do not spend too much time having fun or getting frustrated because something trivial does not work. I am writing this as a tinkerer myself with no background in computer science or neighbouring studies: I learnt Emacs Lisp through trial and error by playing around with the editor. My nominal goal was to improve certain micro-motions I was repeating over and over: I sought efficiency only to discover something much more profound. Learning to extend my editor has been a fulfilling experience and I am more productive as a result. Emacs does what I want it to do and I am happy with it.

Each chapter herein is generally short and to-the-point. Some are more friendly to beginners while others dive deeper into advanced topics. There are links between the chapters, exactly how a reference manual is supposed to be done. You may then go back and forth to find what you need.

The text you will find here is a combination of prose and code. The latter may be actual Elisp or pseudo-code which captures the underlying pattern. I encourage you to read this book either inside of Emacs or with Emacs readily available. This way, you can play around with the functions I give you, to further appreciate their nuances.

The "big picture view" approach I am adopting is about covering the concepts that I encounter frequently while working with Emacs Lisp. This book is no substitute for the Emacs Lisp Reference Manual and should by no means be treated as the source of truth for any of the Elisp forms I comment on.

Good luck and enjoy!

2 Evaluate Emacs Lisp

Everything you do in Emacs calls some function. It evaluates Emacs Lisp code, reading the return values and producing side effects (Side effect and return value).

You type a key on your keyboard and a character is written to the current buffer. That is a function bound to a key. It actually is an *interactive* function, because you are calling it via a key binding rather than through some program. Interactive functions are known as "commands". Though do not let the implementation detail of interactivity distract you from the fact that every single action you perform in Emacs involves the evaluation of Emacs Lisp.

Another common pattern of interaction is with the M-x (`execute-extended-command`) key, which by default runs the command `execute-extended-command`: it produces a minibuffer prompt that asks you to select a command by its name and proceeds to execute it.

Emacs can evaluate Elisp code from anywhere. If you have some Elisp in your buffer, you can place the cursor at the end of its closing parenthesis and type C-x C-e (`eval-last-sexp`). Similarly, you can use the commands `eval-buffer` and `eval-region` to operate on the current buffer or highlighted region,

respectively.

The `eval-last-sexp` also works on symbols (Symbols, balanced expressions, and quoting). For example, if you place the cursor at the end of the variable `buffer-file-name` and use C-x C-e (`eval-last-sexp`), you will get the value of that variable, which is either `nil` or the file system path to the file you are editing.

Sometimes the above are not appropriate for what you are trying to do. Suppose you intend to write a command that copies the file path of the current buffer. To do that, you need your code to test the value of the variable `buffer-file-name` (Buffers as data structures). But you do not want to type out `buffer-file-name` in your actual file, then use one of the aforementioned commands for Elisp evaluation, and then undo your edits. That is cumbersome and prone to mistakes! The best way to run Elisp in the current buffer is to type M-: (`eval-expression`): it opens the minibuffer and expects you to write the code you want to evaluate. Type RET from there to proceed. The evaluation is done with the last buffer as current (the buffer that was current prior to calling `eval-expression`).

Here is some Emacs Lisp you may want to try in (i) a buffer that corresponds to a file versus (ii) a buffer that is not associated with any file on disk.

```
;; 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"))
```

When you are experimenting with code, you want to test how it behaves. Use the command `ielm` to open an interactive shell. It puts you at a prompt where you can type any Elisp and hit RET to evaluate it. The return value is printed right below. Alternatively, switch to the `*scratch*` buffer. If it is using the major mode `lisp-interaction-mode`, which is the default value of the variable `initial-major-mode`, then you can move around freely in that buffer and type C-j (`eval-print-last-sexp`) at the end of some code to evaluate it. This works almost the same way as `eval-last-sexp`, with the added effect of putting the return value right below the expression you just evaluated.

In addition to these, you can rely on the self-documenting nature of Emacs to figure out what the current state is. For example, to learn about the buffer-local value of the variable `major-mode`, you can do C-h v (`describe-variable`), and then search for that variable. The resulting Help buffer will inform you about the current value of `major-mode`. This help command and many others like `describe-function`, `describe-keymap`, `describe-key`, and `describe-symbol`, provide insight into what Emacs knows about a given object. The Help buffer will show relevant information, such as the path to the file that defines the given function or whether a variable is declared as buffer-local.

Emacs is “self-documenting” because it reports on its state. You do not need to explicitly update the Help buffers. This happens automatically by virtue of evaluating the relevant code: Emacs effectively shows you the latest value of whatever it is you are working with.

3 Side effect and return value

Emacs Lisp has functions. They take inputs and produce outputs. In its purest form, a function is a computation that only returns a value: it does not change anything in its environment. The return value of a function is used as input for another function, in what effectively is a chain of computations. You can thus rely on a function's return value to express something like "if this works, then also do this other thing, otherwise do something else or even nothing."

Elisp is the language that extends and controls Emacs. This means that it also affects the state of the editor. When you run a function, it can make permanent changes, such as to insert some text at the point of the cursor, delete a buffer, create a new window, and so on. These changes will have an impact on future function calls. For example, if the previous function deleted a certain buffer, the next function which was supposed to write to that same buffer can no longer do its job: the buffer is gone!

When you write Elisp, you have to account for both the return value and the side effects. If you are sloppy, you will get unintended results caused by all those ill-considered changes to the environment. But if you use side effects meticulously, you are empowered to take Elisp to its full potential. For instance, imagine you define a function that follows the logic of "create a buffer, go there, write some text, save the buffer to a file at my preferred location, and then come back where I was before I called this function, while leaving the created buffer open." All these are side effects and they are all useful. Your function may have some meaningful return value as well that you can employ as the input of another function. For example, your function would return the buffer object it generated, so that the next function can do something there like display that buffer in a separate frame and make its text larger.

The idea is to manipulate the state of the editor, to make Emacs do what you envision. Sometimes this means your code has side effects. At other times, side effects are useless or even run counter to your intended results. You will keep refining your intuition about what needs to be done as you gain more experience and expand the array of your skills (Symbols, balanced expressions, and quoting). No problem; no stress!

4 Buffers as data structures

A buffer holds data as a sequence of characters. For example, this data is the text you are looking at when you open a file. Each character exists at a given position, which is a number. The function `point` gives you the position at the point you are on, which typically corresponds to where the cursor is (Evaluate Emacs Lisp). At the beginning of a buffer, `point` returns the value of `1` (Side effect and return value). There are plenty of functions that return a buffer position, such as `point-min`, `point-max`, `line-beginning-position`, and `re-search-forward`. Some of those will have side effects, like `re-search-forward` which moves the cursor to the given match.

When you program in Emacs Lisp, you frequently rely on buffers to do some of the following:

Extract file contents as a string Think of the buffer as a large string. You can get the entirety of

its contents as one potentially massive string by using the function `buffer-string`. You may also get a substring between two buffer positions, such as with the `buffer-substring` function or its `buffer-substring-no-properties` counterpart (Text has its own properties). Imagine you do this as part of a wider operation that (i) opens a file, (ii) goes to a certain position, (iii) copies the text it found, (iv) switches to another buffer, and (v) writes what it found to this new buffer.

Present the results of some operation You may have a function that shows upcoming holidays. Your code does the computations behind the scenes and ultimately writes some text to a buffer. The end product is on display. Depending on how you go about it, you will want to evaluate the function `get-buffer-create` or its more strict `get-buffer` alternative. If you need to clear the contents of an existing buffer, you might use the `with-current-buffer` macro to temporarily switch to the buffer you are targetting and then either call the function `erase-buffer` to delete everything or limit the deletion to the range betweeen two buffer positions with `delete-region`. Finally, the functions `display-buffer` or `pop-to-buffer` will place the buffer in an Emacs window.

Associate variables with a given buffer In Emacs Lisp, variables can take a buffer-local value which differs from its global counterpart. Some variables are even declared to always be buffer-local, such as the `buffer-file-name`, `fill-column`, and `default-directory`. Suppose you are doing something like returning a list of buffers that visit files in a given directory. You would iterate through the return value of the `buffer-list` function to filter the results accordingly by testing for a certain value of `buffer-file-name` (Basic control flow with `if`, `cond`, and others). This specific variable is always available, though you can always use the `setq-local` macro to assign a value to a variable in the current buffer.

The latter point is perhaps the most open-ended one. Buffers are like a bundle of variables, which includes their contents, the major mode they are running, and all the buffer-local values they have. In the following code block, I am using the `seq-filter` function to iterate through the return value of the function `buffer-list` (Symbols, balanced expressions, and quoting).

```
(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))
```

This will return a list of buffer objects that pass the test of (i) being “visible” to the user and (ii) their major mode is either `text-mode` or derived therefrom. The above may also be written thus (When to use a named function or a lambda function):

```
(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))
```

As with buffers, Emacs windows and frames have their own parameters. I will not cover those as their utility is more specialised and the concepts are the same. Just know that they are data structures that you may use to your advantage, including by iterating through them (Mapping through a list of elements).

5 Text has its own properties

Just as with buffers that work like data structures (Buffers as data structures), any text may also have properties associated with it. This is metadata that you inspect using Emacs Lisp. For example, when you see syntax highlighting in some programming buffer, this is the effect of text properties. Some function takes care to "propertise" or to "fontify" the relevant text and decides to apply to it an object known as "face". Faces are constructs that bundle together typographic and colour attributes, such as the font family and weight, as well as foreground and background hues. To get a Help buffer with information about the text properties at the point of the cursor, type M-x (`execute-extended-command`) and then invoke the command `describe-char`. It will tell you about the character it sees, what font it is rendered with, which code point it is, and what its text properties are.

Suppose you are writings your own major mode. At the early stage of experimentation, you want to manually add text properties to all instances of the phrase `I have properties` in a buffer whose major mode is `fundamental-mode`, so you do something like this (The match data of the last search):

```
(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))))
```

Actually try this. Use C-x b (`switch-to-buffer`), type in some random characters that do not match an existing buffer, and then hit RET to visit that new buffer. It runs `fundamental-mode`, meaning that there is no "fontification" happening and, thus, `my-add-properties` will work as intented. Now paste the following:

```
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?
```

Continue with M-: (`eval-expression`) and call the function `my-add-properties`. Did it work? The face it is applying is called `error`. Ignore the semantics of that word: I picked it simply because it typically is styled in a fairly intense and obvious way (though your current theme may do things differently).

There are functions which find the properties at a given buffer position and others which can search forward and backward for a given property. The specifics do not matter right now. All I want you to remember is that the text can be more than just its constituent characters. For more details, type M-x (`execute-extended-command`) to call the command `shortdoc`. It will ask you for a documentation group. Pick `text-properties` to learn more. Well, use `shortdoc` for everything listed there. I do it all the time.

6 Symbols, balanced expressions, and quoting

To someone not familiar with Emacs Lisp, it is a language that has so many parentheses! Here is a simple function definition:

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

I just defined the function with the name `my-greet-person`. It has a list of parameters, specifically, a list of one parameter, called `name`. Then is the optional documentation string, which is for users to make sense of the code and/or understand the intent of the function. `my-greet-person` takes `name` and passes it to the function `message` as an argument to ultimately print a greeting. The `message` function logs the text in the `*Messages*` buffer, which you can visit directly with C-h e (`view-echo-area-messages`). At any rate, this is how you call `my-greet-person` with the one argument it expects:

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

Now do the same with more than one parameters:

```
(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))
```

And call it thus:

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

Even for the most basic tasks, you have lots of parentheses. But fear not! These actually make it simpler to have a structural understanding of your code. If it does not feel this way right now, it is because you are not used to it yet. Once you do, there is no going back.

The basic idea of any dialect of Lisp, Emacs Lisp being one of them, is that you have parentheses which delimit lists. A list consists of elements. Lists are either evaluated to produce the results of some computation or returned as they are for use in some other evaluation (Side effect and return value):

The list as a function call When a list is evaluated, the first element is the name of the function and the remaining elements are the arguments passed to it. You already saw this play out above with how I called `my-greet-person` with `"Protesilaos"` as its argument. Same principle for `my-greet-person-from-country`, with `"Protesilaos"` and `"Cyprus"` as its arguments.

The list as data When a list is not evaluated, then none of its elements has any special meaning at the outset. They are all returned as a list without further changes. When you do not want your list to be evaluated, you prefix it with a single quote character. For example, `'("Protesilaos" "Prot" "Cyprus")` is a list of three elements that should be returned as-is.

Consider the latter case, which you have not seen yet. You have a list of elements and you want to get some data out of it. At the most basic level, the functions `car` and `cdr` return the first element and the list of all remaining elements, respectively:

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

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

The single quote here is critical, because it instructs Emacs to not evaluate the list. Otherwise, the evaluation of this list would treat the first element, namely `"Protesilaos"`, as the name of a function and the remainder of the list as the arguments to that function. As you do not have the definition of such a function, you get an error.

Certain data types in Emacs Lisp are "self-evaluating". This means that if you evaluate them, their return value is what you are already seeing. For example, the return value of the string of characters `"Protesilaos"` is `"Protesilaos"`. This is true for strings, numbers, keywords, symbols, and the special `nil` or `t`. Here is a list with a sample of each of these, which you construct by calling the function `list`:

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

The `list` function evaluates the arguments passed to it, unless they are quoted. The reason you get the return value without any apparent changes is because of self-evaluation. Notice that `my-greet-person-from-coun`

is quoted the same way we quote a list we do not want to evaluate. Without it, `my-greet-person-from-country` would be evaluated, which would return an error unless that was also defined as a variable.

Think of the single quote as an unambiguous instruction: "do not evaluate the following." More specifically, it is an instruction to not perform evaluation if it would have normally happened in that context (Partial evaluation inside of a list). In other words, you do not want to quote something inside of a quoted list, because that is the same as quoting it twice:

```
;; 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)
```

Now you may be wondering why did we quote `my-greet-person-from-country` but nothing else? The reason is that everything else you saw there is effectively "self-quoting", i.e. the flip-side of self-evaluation. Whereas `my-greet-person-from-country` is a symbol. A "symbol" is a reference to something other than itself: it either represents some computation—a function—or the value of a variable. If you write a symbol without quoting it, you are effectively telling Emacs "give me the value this symbol represents." In the case of `my-greet-person-from-country`, you will get an error if you try that because this symbol is not a variable and thus trying to get a value out of it is not going to work.

Keep in mind that Emacs Lisp has a concept of "macro", which basically is a templating system to write code that actually expands into some other code which is then evaluated. Inside of a macro, you control how quoting is done, meaning that the aforementioned may not apply to calls that involve the macro, even if they are still used inside of the macro's expanded form (Evaluation inside of a macro or special form).

As you expose yourself to more Emacs Lisp code, you will encounter quotes that are preceded by the hash sign, like `#'some-symbol`. This "sharp quote", as it is called, is the same as the regular quote with the added semantics of referring to a function in particular. The programmer can thus better articulate the intent of a given expression, while the byte compiler may internally perform the requisite checks and optimisations. In this light, read about the functions `quote` and `function` which correspond to the quote and sharp quote, respectively.

7 Partial evaluation inside of a list

You already have an idea of how Emacs Lisp code looks like (Symbols, balanced expressions, and quoting). You have a list that is either evaluated or taken as-is. There is another case where a list should be partially evaluated or, more specifically, where it should be treated as data instead of a function call with some

elements inside of it still subject to evaluation.

In the following code block, I am defining a variable called `my-greeting-in-greek`, which is a common phrase in Greek that literally means "health to you" and is pronounced as "yah sou". Why Greek? Well, you got the `lambda` that engendered this whole business with Lisp, so you might as well get all the rest (When to use a named function or a lambda function)!

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

Now I want to experiment with the `message` function to better understand how evaluation works. Let me start with the scenario of quoting the list, thus taking it as-is:

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

You will notice that the variable `my-greeting-in-greek` is not evaluated. I get the symbol, the actual `my-greeting-in-greek`, but not the value it represents. This is the expected result, because the entire list is quoted and, ipso facto, everything inside of it is not evaluated.

Now check the next code block to understand how I can tell Emacs that I want the entire list to still be quoted but for `my-greeting-in-greek` in particular to be evaluated, so it is replaced by its value:

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

Pay close attention to the syntax here. Instead of a single quote, I am using the backtick or back quote, which is also known as a "quasi quote" in our case. This behaves like the single quote except for anything that is preceded by a comma. The comma is an instruction to "evaluate the thing that follows" and only works inside of a quasi-quoted list. The "thing" that follows is either a symbol or a list. The list can, of course, be a function call. Let me then use `concat` to greet a certain person all while returning everything as a list:

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

Bear in mind that you would get an error if you were not quoting this list at all, because the first element `one` would be treated as the symbol a function, which would be called with all other elements as its arguments. Chances are that `one` is not defined as a function in your current Emacs session or those arguments are not meaningful to it, anyway. Plus, `two` and `four` would then be treated as variables, since they are not quoted, in which case those would have to be defined as well, else more errors would ensue.

Other than the comma operator, there is the `,@` (how is this even pronounced? "comma at", perhaps?), which is notation for "splicing". This is jargon in lieu of saying "the return value is a list and

I want you to remove the outermost parentheses of it." In effect, the code that would normally return `'(one two three)` now returns `one two three`. This difference may not make much sense in a vacuum, though it does once you consider those elements as expressions that should work in their own right, rather than simply be elements of a quoted list. I will not elaborate on an example here, as I think this is best covered in the context of defining macros (Evaluation inside of a macro or special form).

Chances are you will not need to use the knowledge of partial evaluation. It is more common in macros, though can be applied anywhere. Be aware of it regardless, as there are scenaria where you will, at the very least, want to understand what some code you depend on is doing.

Lastly, since I introduced you to some Greek words, I am now considering you my friend. Here is a joke from when I was a kid. I was trying to explain some event to my English instructor. As I lacked the vocabulary to express myself, I started using Greek words. My instructor had a strict policy of only responding to English, so she said "It is all Greek to me." Not knowing that her answer is an idiom for "I do not understand you", I blithely replied, "Yes, Greek madame; me no speak England very best." I was not actually a beginner at the time, though I would not pass on the opportunity to make fun of the situation. Just how you should remember to enjoy the time spent tinkering with Emacs. But enough of that! Back to reading this book.

8 Evaluation inside of a macro or special form

In the most basic case of Emacs Lisp code, you have lists that are either evaluated or not (Symbols, balanced expressions, and quoting). If you get a little more fancy, you have lists that are only partially evaluated (Partial evaluation inside of a list). Sometimes though, you look at a piece of code and cannot understand why the normal rules of quoting and evaluation do not apply. Before you see this in action, inspect a typical function call that also involves the evaluation of a variable:

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

You encountered this code in the section about partial evaluation. What you have here is a call to the function `concat`, followed by three arguments. One of these arguments is a variable, the `my-greeting-in-greek`. When this list is evaluated, what Emacs actually does is to first evaluate the arguments, including `my-greeting-in-greek`, in order to get their respective values and only then to call `concat` with those values. You can think of the entire operation as follows:

- Here is a list.
- It is not quoted.
- So you should evaluate it.
- The first element is the name of the function.
- The remaining elements are arguments passed to that function.

- Check what the arguments are.
- Evaluate each of the arguments to resolve it to its actual value.
- Strings are self-evaluating, while the `my-greeting-in-greek` is a variable.
- You now have the value of each of the arguments, including the value of the symbol `my-greeting-in-greek`.
- Call `concat` with all the values you got.

In other words, the following two yield the same results (assuming a constant `my-greeting-in-greek`):

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

(concat "Γ     " " " "Π     ")
```

This is predictable. It follows the basic logic of the single quote: if it is quoted, do not evaluate it and return it as-is, otherwise evaluate it and return its value. But you will find plenty of cases where this expected pattern is seemingly not followed. Consider this common case of using `setq` to bind a symbol to the given value:

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

The above expression looks like a function call, meaning that (i) the list is not quoted, (ii) the first element is the name of a function, and (iii) the remaining elements are arguments passed to that function. In a way, this is all true. Though you would then expect the `my-test-symbol` to be treated as a variable, which would be evaluated in place to return its result which would, in turn, be the actual argument passed to the function. However, this is not how `setq` works. The reason is that it is a special case that internally does this:

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

This is where things are as expected. There is no magic happening behind the scenes. The `setq`, then, is a convenience for the user to not quote the symbol each time. Yes, this makes it a bit more difficult to reason about it, though you get used to it and eventually it all makes sense. Hopefully, you will get used to such special forms, as you find them with `setq` but also with `defun`, among many others. Here is a function you have already seen:

```
(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))
```

If the normal rules of evaluation applied, then the list of parametes should be quoted. Otherwise, you would expect `(name country)` to be interpreted as a function call with `name` as the symbol of the function and `country` as its argument which would also be a variable. But this is not what is happening because `defun` will internally treat that list of parameters as if it was quoted.

Another common scenario is with `let` (Control flow with `if-let*` and friends). Its general form is as follows:

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

The `LIST-OF-LISTS-AS-VARIABLE-BINDINGS` is a list in which each element is a list of the form `(SYMBOL VALUE)`. Here is some actual code:

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

Continuing with the theme of special forms, if `let` was a typical function call, the `LIST-OF-LISTS-AS-VARIABLE-B` would have to be quoted. Otherwise, it would be evaluated, in which case the first element would be the name of the function. But that would return an error, as the name of the function would correspond to another list, the `(name "Protesilaos")`, rather than a symbol. Things work fine with `let` because it internally does the quoting of its `LIST-OF-LISTS-AS-VARIABLE-BINDINGS`.

Expect similar behaviour with many special forms as well as with macros such as the popular `use-package`, which is used to configure packages inside of your Emacs initialisation file. How each of those macros works depends on the way it is designed. I will not delve into the technicalities here, as I want the book to be useful long-term, focusing on the principles rather than the implementation details that might change over time.

To learn what a given macro actually expands to, place the cursor at the end of its closing parenthesis and call the command `pp-macroexpand-last-sexp`. It will produce a new buffer showing the expanded Emacs Lisp code. This is what is actually evaluated in the macro's stead.

With those granted, it is time to write a macro. This is like a template, which empowers you to not repeat yourself. Syntactically, a macro will most probably depend on the use of the quasi-quote, the comma operator, and the mechanics of splicing (Partial evaluation inside of a list). Here is a simple scenario where we want to run some code in a temporary buffer while setting the `default-directory` to the user's home directory.

```
(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)))
```

In this definition, the `&rest` makes the following parameter a list. So you can pass an arbitrary number of arguments to it, all of which will be collected into a single list called `EXPRESSIONS`. The judicious use of partial evaluation ensures that the macro will not be evaluated right now but only when it is called. The arguments passed to it will be placed where you have specified. Here is a call that uses this macro:

```
(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")))
```

If you place the cursor at the closing parenthesis of `my-work-in-temp-buffer-from-home`, you will be able to confirm what it expands to by typing M-x (`execute-extended-command`) and then invoking the command `pp-macroexpand-last-sexp`. This is what I get:

```
(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")))
```

Piecing it together with the rest of the code in its context, I arrive at this:

```
(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"))))
```

With this example in mind, consider Elisp macros to be a way of saying “this little thing here helps me express this larger procedure more succinctly, while the actual code that runs is still that of the latter.”

The above macro I wrote has its body start with a quasi-quote, so you do not get to appreciate the nuances of evaluation within it. Let me show you this other approach, instead, where I write a macro that lets me define several almost identical interactive functions (Make your interactive function also work from Lisp calls).

```
(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)))
```

The `my-define-command` can be broadly divided into two parts: (i) what gets evaluated outright and (ii) what gets expanded for further evaluation. The latter part starts with the quasi-quote. This distinction is important when we call the macro, because the former part will be executed right away so if we hit the error, it will never expand and then run the `EXPRESSIONS`. Try `pp-macroexpand-last-sexp` with the following to see what I mean. For your convenience, I include the macro expansions right below each case.

```
(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...
```

Do you need macros? Not always, though there will be cases where a well-defined macro makes your code more elegant. What matters is that you have a sense of how evaluation works so that you do not get confused by all those parentheses. Otherwise you might expect something different to happen than what you actually get.

9 Mapping through a list of elements

A common routine in programming is to work through a list of items and perform some computation on each of them. Emacs Lisp has the generic `while` loop, as well as a whole range of more specialised functions to map over a list of elements, such as `mapcar`, `mapc`, `dolist`, `seq-filter`, `seq-remove`, and many more. Depending on what you are doing, you map through elements with the intent to produce some side effect and/or to test for a return value (Side effect and return value). I will show you some examples and let you decide which is the most appropriate tool for the task at hand.

Starting with `mapcar`, it applies a function to each element of a list. It then takes the return value at each iteration and collects it into a new list. This is the return value of `mapcar` as a whole. In the following code block, I use `mapcar` over a list of numbers to increment them by `10` and return a new list of the incremented numbers.

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

In the code block above, I am using a `lambda`, else an anonymous function (When to use a named function or a lambda function). Here is the same code, but with an eponymous function, i.e. a named function:

```
(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)
```

Notice that here we quote the eponymous function (Symbols, balanced expressions, and quoting).

The `mapcar` collects the return values into a new list. Sometimes this is useless. Suppose you want to evaluate a function that saves all unsaved buffers which visit a file. In this scenario, you do not care about accumulating the results: you just want the side effect of saving the buffer outright. To this end, you may use `mapc`, which always returns the list it operated on:

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

An alternative to the above is `dolist`, which is used for side effects but always returns `nil`:

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

You will notice that the `dolist` is a macro, so some parts of it seem to behave differently than with basic lists and the evaluation rules that apply to them (Evaluation inside of a macro or special form). This is a matter of getting used to how the code is expressed.

When to use a `dolist` as opposed to a `mapc` is a matter of style. If you are using a named function, a `mapc` looks cleaner to my eyes. Otherwise a `dolist` is easier to read. Here is my approach with some pseudo-code:

```
;; 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)
```

While `dolist` and `mapc` are for side effects, you can still employ them in the service of accumulating results, with the help of `let` and related forms (Control flow with `if-let*` and friends). Depending on the specifics, this approach may make more sense than relying on a `mapcar`. Here is an annotated sketch:

```
;; 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")
```

For completeness, the previous example would have to be done as follows with the use of `mapcar`:

```
(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")
```

Because `mapcar` happily accumulates all the return values, it returns a list that includes `nil`. If you wanted that, you would probably not even bother with the `when` clause there. The `delq` is thus applied to the return value of the `mapcar` to delete all the instances of `nil`. Now compare this busy work to `seq-filter`:

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

The `seq-filter` is the best tool when all you need is to test if the element satisfies a predicate function and then return that element. But you cannot return something else. Whereas `mapcar` will take any return value without complaints, such as the following:

```
(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")
```

How you go about mapping over a list of elements will depend on what you are trying to do. There is no one single function that does everything for you. Understand the nuances and you are good to go. Oh, and do look into the built-in `seq` library (use M-x (`execute-extended-command`), invoke `find-library`, and then search for `seq`). You are now looking at the source code of `seq.el`: it defines plenty of functions

like `seq-take`, `seq-find`, `seq-union`. Another way is to invoke the command `shortdoc` and read about the documentation groups `list` as well as `sequence`.

10 The match data of the last search

As you work with Emacs Lisp, you will encounter the concept of "match data" and the concomitant functions `match-data`, `match-beginning`, `match-string`, and so on. These refer to the results of the last search, which is typically performed by the functions `re-search-forward`, `looking-at`, `string-match`, and related. Each time you perform a search, the match data gets updated. Be mindful of this common side effect (Side effect and return value). If you forget about it, chances are your code will not do the right thing.

In the following code block, I define a function that performs a search in the current buffer and returns a list of match data without text properties, where relevant (Text has its own properties).

```
(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))))
```

You may then call it with a string argument, representing an Emacs Lisp regular expression:

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

If the regular expression matches, then you get the match data. Here is some sample text:

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

Place the cursor before that text and use M-: (`eval-expression`) to evaluate `my-get-match-data` with the regexp I show above. You will get a return value, as intended.

The way `my-get-match-data` is written, it does two things: (i) it has the side effect of moving the cursor to the end of the text it found and (ii) it returns a list with the match data I specified. There are many scenaria where you do not want the aforementioned side effect: the cursor should stay where it is. As such, you can wrap your code in a `save-excursion` (Switching to another buffer, window, or narrowed state): it will do what it must and finally restore the `point` (Run some code or fall back to some other code):

```
(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)))))
```

If you evaluate this version of `my-get-match-data` and then retry the function call I had above, you will notice how you get the expected return value without the side effect of the cursor moving to the end of the matching text. In practice, this is a useful tool that may be combined with `save-match-data`. Imagine you want to do a search forward inside of another search you are performing, such as to merely test if there is a match for a regular expression in the context, but need to inhibit the modification of the match data you planned to operate on. As such:

```
(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)))))
```

Evaluate the function `my-get-match-data-with-extra-check` and then call with M-: (`eval-expression`) to test that it returns a non-`nil` value with the second example below, but not the first one. This is the expected outcome.

```
(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 Switching to another buffer, window, or narrowed state

As you use Emacs Lisp to do things programmatically, you encounter cases where you need to move away from where you are. You may have to switch to another buffer, change to the window of a given buffer, or even modify what is visible in the buffer you are editing. At all times, this involves one or more side effects which, most probably, should be undone when your function finishes its job (Side effect and return value).

Perhaps the most common case is to restore the `point`. You have some code that moves back or forth in the buffer to perform a match for a given piece of text. But then, you need to leave the cursor where it originally was, otherwise the user will lose their orientation. Wrap your code in a `save-excursion` and you are good to go, as I show elsewhere (The match data of the last search):

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

Same principle for `save-window-excursion`, which allows you to select another window, such as with `select-window`, move around in its buffer, and then restore the windows as they were:

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

The `save-restriction` allows you to restore the current narrowing state of the buffer. You may then choose to either `widen` or `narrow-to-region` (and related commands like `org-narrow-to-subtree`), do what you must, and then restore the buffer to its original state.

```
;; 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))
```

Depending on the specifics, you will want to combine the aforementioned. Beware that the documentation of `save-restriction` tells you to use `save-excursion` as the outermost call. Other than that, you will also find cases that require a different approach to perform some conditional behaviour (Run some code or fall back to some other code).

12 Basic control flow with `if`, `cond`, and others

You do not need any conditional logic to perform basic operations. For example, if you write a command that moves 15 lines down, it will naturally stop at the end of the buffer when it cannot move past the number you specified. Using `defun`, you write an interactive function (i.e. a "command") to unconditionally move down 15 lines using `forward-line` internally (call it with a negative number to move in the opposite direction):

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

The `my-15-lines-down` is about as simple as it gets: it wraps around a basic function and passes to it a fixed argument, in this case the number `15`. Use M-x (`execute-extended-command`) and then call this command by its name. It works! Things get more involved as soon as you decide to perform certain actions only once a given condition is met. This "control flow" between different branches of a logical sequence is expressed with `if`, `when`, `unless`, and `cond`, among others. Depending on the specifics of the case, `and` as well as `or` may suffice.

How about you make your `my-15-lines-down` a bit smarter? When it is at the absolute end of the buffer, have it move 15 lines up. Why? Because this is a demonstration, so why not? The predicate function that tests if the point is at the end of the buffer is `eobp`. A "predicate" is a function that returns true, technically non-`nil`, when its condition is met, else it returns `nil` (Side effect and return value). As for the weird name, the convention in Emacs Lisp is to end predicate functions with the `p` suffix: if the name of the function consists of multiple words, typically separated by dashes, then the predicate function is named `NAME-p`, like `string-match-p`, otherwise it is `NAMEp`, like `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)))
```

Evaluate this function, then type M-x (`execute-extended-command`) and invoke `my-15-lines-down-or-up` to get a feel for it. Below is a similar idea, which throws and error and exits what it was doing if `eobp` returns non-`nil`:

```
(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)))
```

A quirk of Emacs Lisp, which may be a feature all along, is how indentation is done. Just mark the code you have written and type TAB: Emacs will take care to indent it the way it should be done. In the case of the `if` statement, the "then" part is further in than the "else" part of the logic. There is no special meaning to this indentation: you could write everything on a single line like `(if COND THIS ELSE)`, which looks like your typical list, by the way (Symbols, balanced expressions, and quoting). What the indentation does is help you identify imbalances in your parentheses. If the different expressions all line up in a way that looks odd, then you are most probably missing a parentheses or have too many of them. Generally, expressions at the same level will all line up the same way. Those deeper in will have more indentation, and so on. Experience will allow you to spot mistakes with mismatching parentheses. But even if you do not identify them, you will get an error eventually. Rest assured!

The way `if` is written is like a function that takes two or more arguments. The "or more" all counts as part of the "else" logic. As such, `(if COND THIS)` has no "else" consequence, while `(if COND THIS ELSE1 ELSE2 ELSE3)` will run `ELSE1`, `ELSE2`, and `ELSE3` in order as part of the "else" branch. Here is how this looks once you factor in proper indentation:

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

Now what if the `THIS` part needs to be more than one function call? Elisp has the `progn` form, which you can use to wrap function calls and pass them as a single argument. Putting it all together, your code will now look this like:

```
(if COND
    (progn
      THIS1
      THIS2
      THIS3)
  ELSE1
  ELSE2
  ELSE3)
```

If you do not need the "else" part, use `when` to express your intention. Internally, this is a macro which actually stands for `(if COND (progn EXPRESSIONS))`, where `EXPRESSIONS` is one or more expressions. A `when` looks like this:

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

Similarly, the `unless` has the meaning of `(when (not COND) EXPRESSIONS)`. It, too, is a macro that expands to an `if` statement:

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

When the condition you are testing for has multiple parts, you can rely on `and` as well as `or`:

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

(when (and THIS THAT)
  EXPRESSIONS)

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

Depending on the specifics of the case, the combination of multiple `if`, `when`, `or`, `and` will look awkward. You can break down the logic to distinct conditions, which are tested in order from top to bottom, using `cond`. The way `cond` is written is as a list of lists, which do not need quoting (Evaluation inside of a macro or special form). In abstract, it looks like this:

```
(cond
 (CONDITION1
  CONSEQUENCES1)
 (CONDITION2
  CONSEQUENCES2)
 (CONDITION3
  CONSEQUENCES3)
 (t
  CONSEQUENCES-FALLBACK))
```

Each of the consequences can be any number of expressions, like you saw above with `when`. This is a toy function to show how `cond` behaves:

```
(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))))
```

I want you to evaluate it and pass it different arguments to test what it does (Evaluate Emacs Lisp). Here are two examples:

```
(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!"
```

All of the above are common in Emacs Lisp. Another powerful macro is `pcase`, which we will consider separately due to its particularities (Pattern match with `pcase` and related).

13 Control flow with `if-let*` and friends

The `let` and `let*` declare variables that are available only within the current scope, else the `BODY` of the `let`. As such:

```
(let BINDINGS
  BODY)
```

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

The BINDINGS is a list of lists, which does not need to be quoted (Evaluation inside of a macro or special form). While BODY consists of one or more expressions, which I have also named EXPRESSIONS elsewhere in this book. The difference between let and let* (pronounced "let star") is that the latter makes earlier bindings available to later bindings. Like this:

```
;; 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))
```

Sometimes what you want to do is create those bindings if—and only if—they are all non-nil. If their value is nil, then they are useless to you, in which case you do something else (Basic control flow with if, cond, and others). Values may or may not be nil when you are creating a binding with the return value of a function call or some other variable. You could always write code like this:

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

But you can do the same with if-let*, where the THIS part runs only if all the bindings are non-nil:

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

In the ELSE part, the bindings variable1 and variable2 do not exist: they only exist for the THIS part of the code.

The `when-let*` is the same as `when`, meaning that it has no "else" logic. If one of its bindings is `nil`, then the whole `when-let*` returns `nil`. No need to belabour that point.

As you dig dipper into the Emacs Lisp ecosystem, you will come across uses of `if-let*` that (i) create multiple bindings like `let` or `let*` but (ii) also call a predicate function to test if they should continue with the `THIS` part of their logic. Remember that `if-let*` goes straight to `ELSE` if one of its bindings returns `nil`. Consider this example:

```
(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)
```

There is no inherently superior way of doing things. It is a matter of using the right tool for the task at hand. Sometimes you want the bindings to be created, even if their value is `nil`. Choose what makes sense.

14 Pattern match with pcase and related

Once you get in the flow of expressing your thoughts with Emacs Lisp, you will be fluent in the use of `if`, `cond`, and the like (Basic control flow with `if`, `cond`, and others). You might even get more fancy if `if-let*` (Control flow with `if-let*` and friends). However you go about it, there are some cases that arguably benefit from more succinct expressions. This is where `pcase` comes in. At its more basic formulation, it is like `cond`, in that it tests the return value of a given expression against a list of conditions. Here is an example that compared the buffer-local value of the variable `major-mode` for equality against a couple of known symbols:

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

The above is the same idea as this `cond`:

```
(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")))
```

Some programmers may argue that `pcase` is more elegant. I think it is true in this specific example, though I remain flexible and practical: I will use whatever makes more sense for the code I am writing. While on the topic of elegance, I should inform you that practically all of the conditional logic can be done in a way that may seem unexpected. Consider how my examples in this book make repetitive use of `message`, when in reality the only part that changes is the actual string/argument passed to that function. This will work just as well:

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

Same idea for `if`, `when`, and the rest.

Back to the topic of what `pcase` does differently. If you read its documentation, you will realise that it has its own mini language, or "domain-specific language" (DSL). This is common for macros (Evaluation inside of a macro or special form). They define how evaluation is done and what sort of expressions are treated specially. Let me then gift you this toy function that illustrates some of the main features of the DSL now under consideration:

```
(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"))))
```

Go ahead and evaluate that function and then try it out (Evaluate Emacs Lisp). Below are a couple of examples:

```
(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’ "
```

Some of those clauses are a different way to express `cond`. Arguably better, but not a clear winner in my opinion. What is impressive and a true paradigm shift is the concept of "destructuring", else the pattern matching done to the expression that effectively `let` binds elements of a list or cons cell to their corresponding index. The syntax used for this destructuring is arcane, until you relate it to the quasi-quote and the comma which are used for partial evaluation (Partial evaluation inside of a list). With this in mind, consider `pcase-let`, `pcase-let*`, `pcase-lambda`, and `pcase-dolist`, as variations of the plain `let`, `let*`, `lambda`, and `dolist` with the added feature of supporting destructuring. They are not doing any of the extras of `pcase` though—just destructuring on top of their familiar behaviour! This is especially useful when you are working with the return value of a function which comes as a list. I will not elaborate at length, as this is an advanced use-case. If you are already at that level, you do not need me to tell you what to write. For the rest of us who, like me, typically work with simpler code, the `pcase-let` serves as a sufficient illustration of the principle:

```
(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"
```

Whether you use `pcase` and destructuring in general is up to you. You do not require them to write high quality code. Though you might agree with those who consider them inherently more elegant and opt to use them for this very reason to have code that is succinct yet highly expressive.

15 Run some code or fall back to some other code

Your typical code will rely on `if`, `cond`, and the like for control flow (Basic control flow with `if`, `cond`, and others). Depending on your specific needs or stylistic considerations, it may even include `pcase` (Pattern match with `pcase` and related) as well as `if-let*` (Control flow with `if-let*` and friends). There are some cases, nonetheless, that make it imperative you run additional code after your primary operation concludes or exits. The idea is to clean up whatever intermediate state you created. The logic is "do this with all the necessary side effects, then whatever happens to it do that now to, inter alia, undo the side effects." This is the concept of "unwinding", which is implemented via `unwind-protect`.

In the following code block, I define a function which produces a minibuffer prompt asking you

to provide a `y` or `n` answer, which is shorthand notation for "yes" or "no". It tests the return value of `y-or-n-p` to determine what it needs to do. While the prompt is open, the function highlights all instances of the regular expression `(defun` in the current buffer. Those highlights must go away after you are done with the minibuffer and its consequences.

```
(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))))
```

Try the above in your Emacs to get a feel for it. While the "yes or no" prompt is active, also do C-g (`keyboard-quit`) or C-] (`abort-recursive-edit`) to confirm that the highlights are removed even though the code never gets past the prompting phase. You may even modify the function to produce an error: it will create a backtrace, which will still have the effect of unwinding after you do q (`debugger-quit`) from the `*Backtrace*` window.

```
(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))))
```

Taking a step back, you will figure out how `unwind-protect` is a more general form of specialists like `save-excursion` and `save-restriction` (Switching to another buffer, window, or narrowed state), while it underpins the `save-match-data` (The match data of the last search) among many other functions/macros, such as `with-temp-buffer` and `save-window-excursion`. What `unwind-protect` does not do is respond specially to signals, such as those coming from the `error` function: it will allow the error to happen, meaning that a backtrace will be displayed and your code will exit right there (but the unwinding will still work, as I already explained, once you dismiss the backtrace). To make your code treat signals in a more controlled fashion, you must rely on `condition-case`.

With `condition-case` you assume full control over the behaviour of your code, including how it should deal with errors. Put differently, your Elisp will express the intent of "I want to do this, but if I get an error I want to do that instead." There are many signals to consider, all of which come from the `signal` function. These include the symbols `error`, `user-error`, `args-out-of-range`, `wrong-type-argument`, `wrong-length-argument`, and `quit`, in addition to anything else the programmer may consider necessary. In the following code blocks, I show you how `condition-case` looks like. Remember that sometimes you do not do quoting the usual way because of how the underlying form is implemented (Evaluation inside of a macro or special form). The example I am using is the same I had for `unwind-protect`.

```
(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")))))
```

The above function illustrates both the aforementioned concept of unwinding and the mechanics of handling signals. The abstract structure of `condition-case` looks to me like an amalgamation of `let`, `unwind-protect`, and `cond`. These conditions may include the special handler of `:success`, as I show there. Granted, the code I wrote will never lead to that specific success case, though you can modify what happens after the prompt to, say, call `message` instead of the `user-error` function, which will then count as a successful conclusion. Otherwise, I think the expressions I wrote tell you exactly how this program responds to the signals it receives.

What I have not covered yet, is the aspect of `condition-case` that is like the `let`, namely, how it binds the error data to a variable within this scope. In my implementation above, it is the `nil` you see there, meaning that I choose not to perform such a binding, as I have no use for its data. Below I decide to use it, just for the sake of demonstration.

```
(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
```

There will be times when `unwind-protect` and `condition-case` are the right tools for the job. My hope is that these examples have given you the big picture view and you are now ready to write your own programs in Emacs Lisp.

16 When to use a named function or a lambda function

The `lambda` is an anonymous function. It stands in juxtaposition to `defun`, which defines a function with a given name. When to use one or the other is largely a matter of style. Though there are some cases where a certain approach is more appropriate. The rule of thumb is this: if you need to use the function more than once, then give it a name and then call it by its name. Otherwise, you will effectively be redefining it each time, which makes it hard for you to rewrite your program. By contrast, if the function is only relevant ad-hoc, then a `lambda` is fine.

In some cases, you will have a named function that employs a `lambda` internally. To modify one of the examples you will find in this book (Mapping through a list of elements):

```
(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)
```

A `lambda` inside of a named function may also be used to do something over and over again, with the help of `let`. You may, for instance, have a function that needs to greet a list of people as a side effect with `mapc` and you do not want to define the same function more than once:

```
(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") ...)
```

The greetings are a side effect in this case and are available in the `*Messages*` buffer. You can quickly access that buffer with C-h e (`view-echo-area-messages`). It does not really matter what `my-greet-teams` is doing. Focus on the combination of a named function and anonymous functions inside of it.

17 Make your interactive function also work from Lisp calls

Functions can be used interactively when they are declared with the `interactive` specification. This turns them into "commands". They can be called via their name by first doing M-x (`execute-extended-command`) and then finding the command. They may also be assigned to a key and invoked directly by pressing that key. In its simplest form, the `interactive` specification is an unquoted list like `(interactive)`. Here is a trivial example that calls `read-string` to produce a minibuffer prompt which accepts user input and returns it as a string:

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

The problem with the above implementation is that it is only useful in interactive use. If you want to issue such a greeting non-interactively through a program, you need to write another function that does practically the same thing except that it takes a `NAME` argument. Like this:

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

You do not need to write two separate functions which practically do the same thing. Instead, you can have one function, with its parameters, which decides how to get the values of the arguments passed to it depending on if it is called interactively or programmatically. Consider this scenario:

```
(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))
```

The documentation I wrote there tells you exactly what is happening. Though let me explain `interactive` in further detail: it takes an argument, which is a list that corresponds to the argument list of the current `defun`. In this case, the `defun` has a list of arguments that includes a single element, the `NAME`. Thus, `interactive` also has a list with one element, whose value corresponds to `NAME`. If the parameters were more than one, then the `interactive` would have to be written accordingly: each of its elements would correspond to the parameter at the same index on the list.

This list of expressions you pass to `interactive` essentially is the preparatory work that binds values to the parameters. When you call the above function interactively, you practically tell Emacs that in this

case `NAME` is the return value of the call to `read-string`. For more parameters, you get the same principle but I write it down just to be clear:

```
(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"
```

Write `interactive` specifications with care and you will end up with a rich corpus of code that is economical and flexible.

18 COPYING

19 GNU Free Documentation License

20 Indices

20.1 Function index

20.2 Variable index

20.3 Concept index