Emacs Lisp Elements

Protesilaos Stavrou (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" (Chapter 2 [Basics of how Lisp works], page 2). 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 into 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 in a more refined way.

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 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 and rewarding. 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. The progression from one chapter to the other should be linear, such that you have enough information to proceed. 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 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 code I write here is simple, trivial even. I want you to make sense of the given principle and to not distract you with irrelevant complexity. The goal is to show you how the pieces are put together. It is then up to you to write elaborate programs.

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 Basics of how Lisp works

Emacs Lisp is a programming language whose basic form has a simple syntax: there are fundamental objects and lists of them. For example, we have numbers and strings of characters inside double quotes (also known as “strings”). These are some of the fundamental objects. Symbols are another fundamental object: they are made up of alphanumeric characters and/or symbols. They will often look like ordinary words, though they are not inside a pair of double quotes, otherwise they would be strings. A symbol is commonly used as the proper name of some variable or function: they “symbolise” something other than themselves. Though symbols can be self-referential, such that they symbolise themselves. For example, the value of `nil` is `nil`.

Every computation is expressed as a list. This is a list of fundamental objects or, in other words, one or more fundamental objects inside of a pair of parentheses. The first element of the list is the symbol of the function. The remaining elements, if any, are the arguments passed to the function. “Arguments” here means that they are the exact values that correspond to the specified parameters of the function. A function has “parameters”, meaning that it expects certain inputs with which to carry out its work. Functions can define zero or more parameters and internally decide how to handle them. Some parameters can be mandatory and others optional. The number of arguments must match that of the mandatory parameters, otherwise we get an error.

Let us take a look at a basic function call. For example, this is how we print a message, which we can then find in the ‘`*Messages*`’ buffer:

```
(message "Hello world, this is my message to you!")
;; => "Hello world, this is my message to you!"
```

Look at the syntax we have used. Here `message` is a symbol. It is the first element of the list, therefore `message` is interpreted as the proper name of a function. The second element of the list is a string which, in this case, is understood as the first argument passed to the `message` function. The entire expression is a list, hence the parentheses around it. This expression is a function call: it performs a computation.

When we use the term “evaluate” and derivatives, we are effectively describing how we ask the computer to perform some computation for us, such as to get the value of a variable or call a function for the work it does and get back whatever it returns. In the above example, I wrote the list and then evaluated it with *`C-x C-e`* (`eval-last-sexp`). The text on the second line is my way of representing the return value of this computation (Chapter 5 [Side effect and return value], page 11).

What you just read explains most of Emacs Lisp. Though it can still be difficult to understand a program because of how all the parentheses line up. It can be hard to figure out where something starts and where it ends when you have something like this:

```
(upcase (message "Hello world, the number is %s" (+ 1 1 1)))
;; => "HELLO WORLD, THE NUMBER IS 3"
```

What we have above are three function calls inside of each other. If you look at the closing three parentheses, you may panic as they are already more than what you are used to. Relax! Elisp is cool because you learn it by keeping calm. Let me read it to you from left to right. Put the previous code block on one side and the following explanation on the other, so that you can follow along more easily.

- I am asked to evaluate this piece of Emacs Lisp.
- I have found a parenthesis, which means that this is a list that stands for a function call.
- Let me check the first element of this list.
- I found the symbol `upcase`. Fine, I will use this.
- I know that `upcase` will work on a string to make all its letters upper case (Chapter 4 [Introspecting Emacs], page 9).
- Now I will check the next element of this function call. It has to be the argument passed to `upcase`.
- The argument I have found starts with a parenthesis. This means that it is not a fundamental object, but a function call in its own right.
- Before I do anything with `upcase`, I have to evaluate this function call.
- I found `message`, so I will use that. Time to check the next element of this list.
- I found the string. Good! It has a '`%s`' placeholder, which means that there is another argument whose value will go where '`%s`' now is.
- Here is another parenthesis. We have a third function call.
- The symbol I found here is '`+`'. This is a valid name for a function. I will check what the arguments are.
- I checked the arguments. They are all numbers. So I will apply '`+`' to them.
- I got the result of this function call, which is the number '`3`'. Now I can go back to `message` with '`3`' as its second argument (the first argument is that string I found which had the '`%s`' placeholder). I have all the values to make it work.
- Fine, I called `message`. It returns a string.
- With this string I got back, I am finally ready to run `upcase`.

What I just described is how Emacs Lisp code is expressed in its simplest form. This is enough to write a program.

Though you will eventually want to control when something is evaluated. To this end, Emacs Lisp has the concept of quoting, which is done with the single quote '`'`'. When the single quote is added as a prefix to an object, then it means "do not evaluate this right now and take it as-is" (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). Using the example from earlier with `message`:

```
(message "I got this: %s" '(one two three))
;; => "I got this: (one two three)"
```

When Emacs reads this, it now finds only one function call, which starts with the opening parenthesis for `message`. Then it reads one list of data, which starts with the quoted list of symbols we have there. By using the quote, we have successfully controlled where evaluation should happen and where it should just take what we give it without trying to interpret it.

Remember that when we use "quote" and related words in Emacs Lisp, we are referring to this single quote, which is always a prefix. We do not have pairs of single quotes around words or lists. No. It is only a prefix for the object that follows.

If we did not have the single quote there, then we would effectively be telling Emacs to evaluate '`(one two three)`' as a function call where '`one`' would be the name of the function and the rest would be its arguments. Unless you have a function whose proper name is ‘`one`’, this will result in an error.

Earlier in this chapter I wrote “every computation is expressed as a list”. When you are writing Emacs Lisp, you may find that this does not seem to be the case. For example, you can have a variable that you evaluate on its own. Let us first declare a variable and give it a string value:

```
(defvar my-name "Protesilaos")
;; => my-name
```

We have now defined the symbol `my-name`, whose value is the string of characters ‘`Protesilaos`’. What you read here looks exactly like the function calls we discussed in the previous examples. This is what Lisp is all about. Now you may wonder what happens if we evaluate `my-name` without doing a function call. Like this:

```
my-name
;; => "Protesilaos"
```

We get back the value. So is what I wrote earlier about everything being a list incorrect? While you do not have any parentheses there, it is helpful to think that Emacs makes things easier for you by internally using the `symbol-value` function to interpret the symbol `my-name`:

```
(symbol-value 'my-name)
;; => "Protesilaos"
```

Note that the `symbol-value` here is important because it is the function that takes the `my-name` symbol as an argument. But you cannot evaluate a variable by writing it as a function call. This is erroneous:

```
(my-name)
;; => ERROR...
```

The error pertains to the fact that `my-name` is not a function: it is a variable. Emacs has different namespaces for functions and variables, meaning that you can have `my-name` as a symbol of a function and as the symbol of a variable. Those are two separate objects that we distinguish by how they are called. Let me show you:

```
(defun my-name nil
  "I am the function that returns this string which includes Protesilaos")
;; => my-name
```

Again, you notice in the code above the idea of everything being a list: `defun` is what we call to interpret all the other elements. We now have defined `my-name` as a function. All this function does is return a string, which in practice is not really useful: we could have done this with a variable. Though what we are interested in at present is to understand how we can have `my-name` be both a variable and a function which we tell apart based on how they are used in context. It is time to test our work:

```
;; Here I am evaluating the variable.  When you evaluate a variable
;; directly, it is helpful to think that internally Emacs translates
;; your request to something like this: (symbol-value 'my-name)
my-name
;; => "Protesilaos"
;; Here I am evaluating the function.
(my-name)
;; => "I am the function that returns this string which includes Protesilaos"
```

What we have discussed thus far should help you make sense of all these nested parentheses you will be dealing with when working with Emacs Lisp. As you gain more experience, you will discover cases where the syntax varies slightly (Chapter 17 [Basic control flow with `if`, `cond`, and others], page 45). This is because programmers want to economise on typing the same things over and over, so they design "special forms" or "Lisp macros" that handle some things internally (Chapter 13 [Evaluation inside of a macro or special form], page 33). Specifically, you will encounter cases where quoting seems to be different than what you have learnt here. Worry not. The principles are the same. Those exceptions will ultimately be easy to reason about.

To avoid complex language such as "a list of list of lists with quoted lists and fundamental objects" we use the term "expression". In general, an expression is anything that Emacs can evaluate, be it a list, list of lists, or symbol (Chapter 6 [Symbols, balanced expressions, and quoting], page 12).

3 Evaluate Emacs Lisp

Everything you do in Emacs calls some function (Chapter 2 [Basics of how Lisp works], page 2). Click a button? That is a function. Write a few words? That is a series of functions. Use the arrow keys to move around? Functions; functions all the way down! Emacs evaluates Emacs Lisp code, reading the return values while dealing with side effects (Chapter 5 [Side effect and return value], page 11).

The fact that a computer program does computation is not peculiar to Emacs. What is special is the combination of two qualities: (i) Emacs is free software and (ii) it is extended in the same programming language it is written in. Its inherent freedom means that every single line of code that is evaluated is available to you for potential inspection, modification, and redistribution (Chapter 4 [Introspecting Emacs], page 9).

Nothing is hidden from you. While the uniform extensibility through Emacs Lisp means that the code you are reading and the code you are writing are treated the same way. You do not need to learn different, typically incompatible programming languages or configuration formats, to make Emacs perform the computations you want. Emacs Lisp is all there is.

Freedom and uniformity of computation together give you maximum control over your text editor (Chapter 24 [Emacs as a computing environment], page 68). The promise, then, is that learning to program in Elisp grants you access to the full power of Emacs, such that you do exactly what you want with it, give some basic concepts such as buffers and text manipulation (Chapter 7 [Buffers as data], page 17).

You type a key on your keyboard and a character is written to the current buffer at the point where the cursor is. This 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 other Emacs Lisp program (the latter is also described as "calling it from Lisp" or "Lisp call"). Interactive functions are known as "commands". They are computations which we can invoke interactively on demand.

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. A command will typically rely on the computations of other functions to do its work. For example, if I have a function that increments a list of numbers, I am internally calling another function for adding one number to another, and then yet another function to apply this operation over each element of the list (Chapter 14 [Mapping over a list of elements], page 38).

Apart from direct key bindings, the other pattern of interaction is with the *`M-x`* (`execute-extended-command`) key: it produces a minibuffer prompt that asks you to select a command by its name. You press the *`RET`* key and Emacs proceeds to execute the command you selected. What you get in this prompt is a list of all the commands that have currently been evaluated or automatically loaded (Chapter 19 [Autoloading symbols], page 52).

Emacs can evaluate Elisp code from anywhere. Normally, you would want to do this in a buffer whose major mode is designed for editing Emacs Lisp (Chapter 10 [What are major and minor modes], page 23). But Emacs will blithely evaluate any Elisp expression from any buffer, including those whose major mode is not designed to treat Elisp code in some special way. 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-buffer` only makes sense in a buffer that contains only Elisp, otherwise you will probably get errors from asking Emacs to evaluate text that does not correspond to known functions or variables and whose syntax does not consist of lists.

To evaluate some code is to make its results available to the running Emacs session. You are, in effect, teaching Emacs something new. It will know about it for as long as the session is active and you have not overwritten this knowledge, as it were, with some new definition of it. This is how you add new functionality to Emacs, redefine existing one, or simply get the value out of some code you are using.

The `eval-last-sexp` also works on symbols (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). 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 in the current buffer. Internally, it is like evaluating `(symbol-value 'buffer-file-name)`, as we covered in the chapter about the basics of Lisp (Chapter 2 [Basics of how Lisp works], page 2).

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` (Chapter 7 [Buffers as data structures], page 17). 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. Such is a cumbersome way of doing what is needed, plus it is 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`). This is especially useful if you want to do a quick test of some part of a bigger procedure you are working on.

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, such as the '`*scratch*`' buffer.

```
;; 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 and confirm you are getting the expected results out of it. Since Lisp consists of lists that can feed into each other, you can evaluate smaller snippets of code while working towards the final version of the program. Use the command `ielm` to open an interactive shell for Emacs Lisp evaluation. 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 (Chapter 4 [Introspecting Emacs], page 9).

4 Introspecting Emacs

Emacs is "self-documenting" because it is aware of its state and can report on it. When you define a variable, such as `my-name`, Emacs retains in its running memory the symbol `my-name` and its corresponding value of, say, '`Protesilaos`'. If you invoke the command `describe-variable`, it will produce a minibuffer prompt from where you can select `my-name` among all other known variables. Once you select that, a '`*Help*`' buffer will be displayed. There you will find information about `my-name`, such as its current value, any documentation it may have and whether it has a buffer-local value (Chapter 7 [Buffers as data structures], page 17).

At all times, you update the state of Emacs by evaluating Elisp code (Chapter 3 [Evaluate Emacs Lisp], page 6). Let us consider the example of a variable that we define for the first time:

```
(defvar my-name "Protesilaos"
  "This is some documentation about my name.
A good Elisp program will document all the functions and variables it
provides.  Emacs developers have a strong documentation culture.  It
complements the software freedom we stand for, because it empowers users
to learn from the tools available to them and thus exercise their
freedom.")
```

We use the special form `defvar` to declare `my-name` with an initial value of '`Protesilaos`' as a string (Chapter 13 [Evaluation inside of a macro or special form], page 33). The second line specifies the documentation of this variable. Documentation strings, also known as "docstrings" are optional though strongly recommended as a matter of convention. As you can tell from the sample above, docstrings are how we make software freedom more accessible to users. Like me, they can learn by reading this information and then by trying to understand the code.

A combination of consistent reading habits and the ease of trial and error that Emacs provides makes it possible to learn Elisp by doing more with it, one small step at a time. This is how every Emacs user can learn to program in Emacs Lisp or, at least, better understand what their tool is doing. Such is the realisation of the promise of free software: everyone, not just professional programmers, benefits from it. And is why, in my opinion, Emacs epitomises the stated objectives of the GNU project for software freedom: it incorporates them and puts them to practice, all in one powerful program.

Once we evaluate the `defvar` for `my-name`, `describe-variable` will produce a '`*Help*`' buffer with the following contents about it:

```
my-name's value is "Protesilaos"

This is some documentation about my name.
A good Elisp program will document all the functions and variables it
provides.  Emacs developers have a strong documentation culture.  It
complements the software freedom we stand for, because it empowers users
to learn from the tools available to them and thus exercise their
freedom.
```

Now go to the source code of `my-name` and modify its value to '`Prot`' between double quotes. Then invoke the command `eval-defun` to evaluate the definition anew. If you now call `describe-variable`, you will get a '`*Help*`' buffer with the updated value. This is how Emacs works for all symbols it has evaluated. When the source code is in a file, the '`*Help*`' buffer will include a link to it. If you find such a link, click on it to navigate to the file and at the locus where the source code is.

Depending on how the code is written, Emacs will have access to metadata that can feed into those '`*Help*`' buffers, among other uses. For example, when a variable is defined, it can specify its original value (Chapter 9 [Add metadata to symbols], page 21). What matters at this point is that Emacs will always know what the current state is. Make a habit out of using the commands `describe-variable`, `describe-function`, `describe-keymap`, `describe-key`, and `describe-symbol`, among others, whenever you have a question about what Emacs knows.

Do this before you search for further information online: it will teach you to rely more on the primary source of Emacs and less on secondary sources on the Internet. Plus, the resulting '`*Help*`' buffers will link you to the source code, from where you may learn something more than just what you are searching for, such as how a certain function is autoloaded (Chapter 19 [Autoloading symbols], page 52). This is how you build up your capacity for working with Elisp, which is how you get to extend Emacs the way you want.

5 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 can be 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 the way I expect, 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, permanently change the value of some variable or function, and so on. These changes may have an impact on future computations. 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! We call all these changes to the environment "side effects". They are the byproducts of the computation, which itself results in some value, i.e. the "return value".

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 use 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 instrumental to your goal. Your function may have some meaningful return value as well that you can employ as the input of another function. For example, your function could return the buffer object it generated, so that the next function can do something useful with it 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 patterns of behaviour. You will keep refining your intuition about what needs to be done as you gain more experience and expand the array of your skills (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). No problem; no stress!

6 Symbols, balanced expressions, and quoting

To someone not familiar with Emacs Lisp, it is a language that has so many nested parentheses (Chapter 2 [Basics of how Lisp works], page 2)! 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 (Chapter 4 [Introspecting Emacs], page 9).

By convention, we write in the imperative voice ("Say hello and do THIS" instead of "Says hello and does THIS") and use upper case letters to refer to the parameters or something that works as a placeholder. Even in that parenthetic comment I made about the imperative voice I followed the convention of using all capital letters to denote a variable/placeholder with '`THIS`'. Looks neat!

`my-greet-person` takes '`name`' as input and passes it to the function `message` as an argument to ultimately print a greeting. The `message` function returns the string it formats and also produces the side effect of appending it to the '`*Messages*`' buffer. You can visit the '`*Messages*`' buffer 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 parentheses everywhere: the uniformity of Lisp means that we have lists all the way down! These actually establish a clear structure, which make it easier to reason about your program. If Lisp does not feel easy 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 (Chapter 5 [Side effect and return value], page 11):

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 of elements without further changes. When you do not want your code 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. The quote prefix can be applied to symbols as well, such as `(symbol-function 'my-greet-person)` to get the actual value out of the function object whose symbol is `my-greet-person` (Chapter 2 [Basics of how Lisp works], page 2).

Consider the latter case of lists as data, which you have not seen yet. You have a list of elements and you want to get something 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 working with. 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-country` is quoted the same way we quote a list: we do not want to evaluate at this moment, but only to include its symbol in the resulting list. Without it, `my-greet-person-from-country` would be evaluated, which would return an error of a void/undefined variable, unless that symbol was also defined as a variable.

Pay close attention to the above code block and the resulting value I included in the comment below. You will notice that `my-greet-person-from-country` was quoted but is not quoted anymore. You can do this with any symbol or list (Chapter 3 [Evaluate Emacs Lisp], page 6). Every time you evaluate a quote expression, you get back the expression without the quote. This means that if you evaluate it again, you will get something different, such as a function call, the value of a variable, or some error.

As such, it is helpful to think of the single quote prefix as an instruction to “do not evaluate this right now, but pass it over to the next evaluation”. In its simplest form though, you can just say “do not evaluate the following”, which is exactly what happens. More specifically, it is an instruction to not perform evaluation if it would have normally happened in that context (Chapter 12 [Evaluate some elements inside of a list], page 31).

In other words, you do not want to quote something inside of a quoted list, because that is the same as quoting it twice. The way the quote prefix works is that it passes the "do not evaluate" instruction to its entire form, which in the case of a list is from one parenthesis all the way to the closing matching parenthesis, covering everything in between.

```
;; 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 usually a reference to something other than itself: it either represents some computation—a function—or the value of a variable. The symbols `nil` and `t` are special as they always refer to themselves, meaning that when you evaluate them you get back exactly what you evaluated (Chapter 9 [Add metadata to symbols], page 21).

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 (the `symbol-value` to be precise) is not going to work.

To make it easier to describe our actions, we say that we are evaluating "expressions". These are the objects that Emacs can evaluate, such as a list, a list of lists, or fundamental objects in their own right, like strings and symbols. These expressions need to be balanced, meaning that either they are self-contained (e.g. a string) or have parentheses around them and start and end where they are supposed to be.

A common error in Elisp programs is to have unbalanced parentheses, such as by forgetting one parenthesis at the end or mistyping an extra one. Some programmers use packages that automatically balance Elisp code. The built-in `electric-pair-mode` minor mode can help in this regard (Chapter 10 [What are major and minor modes], page 23). It automatically inserts delimiters as pairs, so you will probably have the correct number of parentheses each time. You may like to write code this way. Though it is a matter of style. I am among those who prefer to do things manually. I find such "electric" behaviour to be more trouble than it is worth, given how I like to move around in a buffer. Perhaps, then, the name "electric" draws from the ability of this mode to shock you at times.

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. Macros are not some other language with its own syntax. They are Emacs Lisp as well, only they make more advanced use of quoting to control how evaluation is done and where. Inside of a macro, you specify what is evaluated now as opposed to later, meaning that the aforementioned may not apply directly to calls that involve the macro, even if they are still used inside of the macro's expanded form (Chapter 13 [Evaluation inside of a macro or special form], page 33).

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.

This is not merely a matter of semantics though. When you define a symbol, there is a difference between storing in it a value or a closure. A "closure" is an object consisting of a `lambda` with its parameters and the body of form. In other words, defining a variable and defining a value leads to a different result. Consider how the functions `symbol-value` and `symbol-function` work: they both examine a symbol. `symbol-value` treats the symbol as a variable and will return its value, if any. `symbol-function` treats the symbol as a function and returns its closure, if any.

Here is how it works:

```
(defvar my-name "Protesilaos")
;; => my-name

(symbol-value 'my-name)
;; => "Protesilaos"

(symbol-function 'my-name)
;; => nil

(defun my-greet (person)
  (format "Hello %s" person))
;; => my-greet

(symbol-function 'my-greet)
;; => #[(person) ((format "Hello %s" person)) nil]

(symbol-value 'my-greet)
;; => ERROR...
```

Notice that '`[(person) ((format "Hello %s" person)) nil]`' is the printed representation of a closure. This is the object corresponding to the following:

```
(lambda (person)
  (format "Hello %s" person))
```

In fact, when you use `defun` it actually creates an alias for such a lambda. Place the cursor at the closing parenthesis of the `defun` expression and invoke the command `pp-macroexpand-last-sexp` to confirm as much. I am printing it below for your convenience:

```
(defun my-greet (person)
  (format "Hello %s" person))
;; => (defalias 'my-greet (lambda (person) (format "Hello %s" person)))
```

In light of these, you can better appreciate the nuances between the functions `quote` and `function`, as pertains to symbols. When you write `(quote some-symbol)` this is the same as `'some-symbol`, which is then expected to be evaluated by `symbol-value`. Whereas `(function some-symbol)` is the way to state `#'some-symbol` and to suggest that `symbol-function` is how to get the data out of the symbol in question.

While these technicalities matter, you do not need to be an expert in them to write decent code. What you need to remember when working with Elisp is that you can tell from the context alone whether something is treated as a function call or as a variable.

7 Buffers as data

In its simplest form, a buffer holds data as a sequence of characters. When you open a file, for example, Emacs reads the file contents from the disk, loads them into its running memory, and displays the contents in a buffer. This buffer now has as its data the contents of the file and anything else that can be extracted therefrom. Each character exists at a given position, which is a number. The function `point` function gives you the numeric position at the point you are on, which typically corresponds to where the cursor is (Chapter 3 [Evaluate Emacs Lisp], page 6).

At the beginning of a buffer, `point` returns the value of '`1`' (Chapter 5 [Side effect and return value], page 11). There are plenty of functions that return a buffer position, such as `point-min`, `point-max`, `re-search-forward`, and `line-beginning-position`. 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 (Chapter 8 [Text has its own properties], page 19). 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 between some boundaries, (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. When you present data, you may also choose to have some specialised major mode that sets the appropriate settings (Chapter 10 [What are major and minor modes], page 23).

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 (Chapter 17 [Basic control flow with if, cond, and others], page 45). The special form setq-local sets the buffer-local value of a variable (Chapter 6 [Symbols, balanced expressions, and quoting], page 12).

The latter point is perhaps the most open-ended one. Buffers are like a bundle of variables, which includes their contents, any overlays or text properties associated with the contents, the active major mode, any active minor modes, and all the buffer-local values of other variables (Chapter 10 [What are major and minor modes], page 23). In the following code block, I am using the seq-filter function to iterate through the return value of the function buffer-list to get a list of buffers that (i) are not hidden from the user's view and (ii) have text-mode as their major mode or a derivative thereof (Chapter 14 [Mapping over a list of elements], page 38).

```
(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 or nil. The above relies on a lambda to do its work. This lambda exists only inside the given function call. If we ever need to perform the same check, it is better to give a name to the lambda, i.e. to define a function. As such, we may write the following (Chapter 22 [When to use a named function or a lambda function], page 63):

```
(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 objects that you may use in your programs.

8 Text has its own properties

Just as with buffers that work like data objects (Chapter 7 [Buffers as data], page 17), any text may also have properties associated with it—and symbols too, by the way (Chapter 9 [Add metadata to symbols], page 21). This is metadata that you inspect using Emacs Lisp functions. For example, when you have syntax highlighting in some programming buffer, it is the effect of underlying text properties. Some function takes care to "propertise" or to "fontify" the relevant text. It applies to it an object known as a "face".

Faces are constructs of the Emacs display engine that bundle together typography and colour attributes, such as the font family and weight, as well as foreground and background hues. Every piece of text uses the `default` face at minimum. Whenever an colour is present or some other typographic style, then there are more faces present.

To get a '`*Help*`' buffer with information about the text properties where the cursor is, type *`M-x`* (`execute-extended-command`) and then invoke the command `describe-char`. It will tell you about the character it identifies, what font it is rendered in, which code point it is, and what its text properties are, if any. If there is a face, that '`*Help*`' buffer will link to it. Following the link shows you what the face's effective attributes are.

Suppose you are writing 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 produce a function that adds text properties where relevant, like the following (Chapter 15 [The match data of the last search], page 42):

```
(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 test this. Use *`C-x b`* (`switch-to-buffer`), type in some random characters that do not match an existing buffer, and then press *`RET`* to visit that new buffer. It runs `fundamental-mode`, meaning that no special "fontification" is in effect and, thus, `my-add-properties` will work as intented (Chapter 10 [What are major and minor modes], page 23). Now paste this snippet of text:

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

What does it even mean for I have properties to be rendered as `error'?
```

Continue with *`M-:`* (`eval-expression`) and call the function `my-add-properties`, i.e. `(my-add-properties)`. Did it work? The face this function is applying is called `error`. Ignore the semantics of that word: I picked `error` 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 certain property. The specifics do not matter right now. All I want you to remember is that the text may hold data beyond 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, actually use `shortdoc` for everything listed there—I do it frequently!

The example above with `add-text-properties` is not how major modes typically do things (Chapter 10 [What are major and minor modes], page 23). Emacs has a more sophisticated mechanism, called “font-lock”. It is responsible for fontifying the buffer. A major mode will thus set up the `font-lock-defaults` variable (minor modes will typically modify the `font-lock-keywords`). How exactly this works is not important for our purposes. The general idea is that for each style, say level 1 headings in Org, you either write a regular expression or, better, a function that does pattern matching. You then specify which face corresponds to the matched pattern or subgroups thereof and how should the faces be applied in case of competing fontification rules.

What ultimately matters is that we have buffers as data objects and text that has metadata. We use this to our advantage to parse the text available to us.

9 Add metadata to symbols

A symbol can be a reference to itself or to some value (Chapter 2 [Basics of how Lisp works], page 2). For variables, `nil` returns `nil`, `t` returns `t`, and `my-name` will return '`Protesilaos`' if it is defined thus. For functions, the symbol actually is an alias for a given computation, i.e. a `lambda` expression whose evaluation may produce side effects and return a value (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). We can get the value of a variable either by evaluating it without a quote or by passing its symbol as an argument to the function `symbol-value` (Chapter 5 [Side effect and return value], page 11). Consider these:

```
;; Define the variable `my-name' and give it a value.
(defvar my-name "Protesilaos")
;; => my-name

;; What is the value of the symbol `my-name', where the symbol is a variable?
(symbol-value 'my-name)
;; => "Prot"

;; What is the value of the variable `my-name' if I do not quote it?
my-name
;; => "Prot"
```

Similarly, we can get the function value of a symbol by calling `symbol-function`:

```
(defun my-hello (person)
  "Say hello to PERSON."
  (meesage "Hello %s" person))
;; => my-hello

;; What is the symbol `my-hello' as a function?
(symbol-function 'my-hello)
;; => #[(person) ((meesage "Hello %s" person)) nil nil "Say hello to PERSON."]

;; What happens if I call `my-hello' without parentheses and/or a quote?
my-hello
;; => ERROR
```

What we know up until now about symbols is that they can be associated with a data point, be it as variables or functions. That is their value. They may also have metadata, which in Emacs Lisp terminology is known as a "property" or "symbol properties". In its simplest form, a symbol `my-name` is associated with arbitrary metadata through the use of the `put` function:

```
(put 'my-name 'my-nicknames '("Prot" "The dog" "The Cypriot" "The Greek-mainlander"))
;; => ("Prot" "The dog" "The Cypriot" "The Greek-mainlander")
```

Similarly, the `get` function returns the value of the given symbol's property.

```
(get 'my-name 'my-nicknames)
;; => ("Prot" "The dog" "The Cypriot" "The Greek-mainlander")
```

In the above two examples, the symbol we are operating on is `my-name` and its property is `my-nicknames`. The value given to the property can be any Elisp object, such as a number, a string, or a list. In the above examples we have a list of strings. To retrieve all the properties of a symbol, we use the function `symbol-plist`, where "plist" stands for "property list".

```
(symbol-plist 'my-name)
;; => (variable-documentation "This is some documentation about my name." my-nicknames
```

What `symbol-plist` returns is an even-numbered list where each odd number is the key and each even number is its corresponding value. This is the plist. Looking at what we have above, we can use `plist-get` to extract a value from the property list, thus:

```
(plist-get (symbol-plist 'my-name) 'my-nicknames)
;; => ("Prot" "The dog" "The Cypriot" "The Greek-mainlander")
```

Though in this specific case we already have `get` do the same work, so `plist-get` is superfluous—it is applicable in other contexts where you need to work with a plist.

The `my-name` is the symbol of a variable, though the symbol of a function will work the same way. Let me show you how the `my-hello` we defined further above will get its '`cool-factor`':

```
(put 'my-hello 'cool-factor t)
```

You `get` how this works. In working code, you will probably not need to use properties in this way. Some macro will handle them internally. Consider this real-world example from the Emacs source code where the function `string-chop-newline` is declared with concomitant metadata (the relevant symbols for `declare`, such as `pure` and `side-effect-free`, are documented in the Emacs Lisp Reference Manual):

```
(defun string-chop-newline (string)
  "Remove the final newline (if any) from STRING."
  (declare (pure t) (side-effect-free t))
  (string-remove-suffix "\n" string))
```

In practice, this boils down to a `put` operation, as illustrated above. You just do not need to specify it yourself. Same principle for variables. For example, the Lisp macro `defcustom` defines a variable with some special metadata which renders it as a so-called "user option": it is a variable that the programmer has designed for users to modify and, therefore, is meant to have a predictable, well-supported behaviour. Internally, `defcustom` will add properties to associate the variable with its original value and to do everything necessary to render concrete its particularities, such as whether the variable is safe as a buffer-local value (Chapter 7 [Buffers as data], page 17).

Chances are that you will not need to deal with symbol properties, especially not while you are getting started. Though know that there are Emacs Lisp programs that make extensive use of them. If your program needs to work with symbol properties, you have already gotten enough exposure to them to proceed with confidence. May you make the most out of this occasion!

10 What are major and minor modes

Buffers are data objects that can be associated with more data (Chapter 7 [Buffers as data], page 17). A common use-case is to have buffer-local variables that control how certain functions behave when called with the given buffer as current. If the text is read-only, for example, this is due to a buffer-local variable that inhibits the ordinary behaviour of text editing commands. When a certain key binding does something different in one buffer than in another, this too is because of buffer-local variables. Or, to be more precise, the buffer applies its own local keymap which overrides the global keymap.

In essence, major and minor modes are commands which activate settings once they are invoked, such as with *M-x* (`execute-extended-command`) (minor mode commands have a toggle behaviour, but more on that below). If you look at the definition of a major mode, for instance, you will notice that all it seems to do is specify buffer-local values for certain variables. Those typically pertain to standardised definitions, such as what is a paragraph, what constitutes a comment, how should indentation be handled, whether there is an outline or some kind or index with points of interest in the buffer, and the fontification settings or other text properties (Chapter 8 [Text has its own properties], page 19). Here is a simple example from the source code of Emacs, specifically `text.el`:

```
(define-derived-mode text-mode nil "Text"
  "Major mode for editing text written for humans to read.
In this mode, paragraphs are delimited only by blank or white lines.
You can thus get the full benefit of adaptive filling
 (see the variable `adaptive-fill-mode').
\\{text-mode-map}
Turning on Text mode runs the normal hook `text-mode-hook'."
  (setq-local text-mode-variant t)
  (setq-local require-final-newline mode-require-final-newline)

  ;; Enable text conversion in this buffer.
  (setq-local text-conversion-style t)
  (add-hook 'context-menu-functions 'text-mode-context-menu 10 t)
  (when (eq text-mode-ispell-word-completion 'completion-at-point)
    (add-hook 'completion-at-point-functions #'ispell-completion-at-point 10 t)))
```

The docstring of `text-mode` has some markup that controls what the '`*Help*`' buffer will show (Chapter 4 [Introspecting Emacs], page 9). The combination of backtick and single quote makes '`*Help*`' treat the given text as a symbol that it links to. Thus, by reading about `text-mode`, we also get a link to `adaptive-fill-mode`. Following that link yields the relevant documentation. The braces or curly brackets, the opening of which is escaped with backslashes, is a placeholder to get the contents of a keymap listed there. What we will then get in the '`*Help*`' buffer is a listing with all the keys and the commands they are bound to. Beside these technicalities, we have some `setq-local` and `add-hook` calls that also have a local scope (Chapter 11 [Hooks and the advice mechanism], page 26).

What you learn from studying the definition of `text-mode` is that is effectively is a bundle of settings. By setting a buffer to use the major mode `text-mode`, we are practically opting in to those settings. If you create a new buffer, whose major mode is `fundamental-mode` by default, you can manually set all of the above without explicitly calling `text-mode`. Doing so will give you most—though not all!—of what `text-mode` does. Actually, do not try this at home: nobody has time for such cumbersome labour and it anyway is not doing what you actually need.

If you do proceed though, know that you are not getting the full effect of `text-mode`. Internally, Emacs takes care of a few more things when you invoke the command of the major mode. Specifically, it updates the variable `major-mode` to point to the symbol of the currently active mode. It also specifies the parent major mode, if any, from where the current major mode inherits its settings. Furthermore, Emacs updates the local keymap and also calls the hook of the given major mode. And if you are really curious, it even changes the applicable syntax table and table of abbreviations. The syntax table determines such things as what is a symbol and what is punctuation, while the abbreviations' table, known as "abbrev table", holds data that is used by the built-in minor mode called `abbrev-mode`.

Where do all these come from? Well, `define-derived-mode` is a macro that instantiates everything using the name of the mode as a prefix (Chapter 13 [Evaluation inside of a macro or special form], page 33). If, for example, there is a major mode called `football-mode`, in reference to the game where players kick a ball around the pitch instead of hugging it in confusion, then there will be a `football-mode-hook`, `football-mode-map`, `football-mode-syntax-table`, and `football-mode-abbrev-table`.

Major modes are mutually exclusive. There can only be one per buffer, though there are clever ways to have the effects of many major modes in one buffer, like what Org mode does with source code blocks that retain some of the qualities of the programming modes they reference. The reason a buffer has one major mode is due to the conflicts that would otherwise arise. For example, Emacs would have an unpredictable behaviour if one major mode defined, say, dashes as punctuation while another treated them as constituent characters of symbols. A programmer could come up with increasingly complex heuristics, though there is a certain elegance to the "one buffer, one major mode" approach. In some contexts, like in modern web development, the major mode effectively is a superset of other major modes, such that it can work with a mixture of JavaScript, HTML, and CSS.

By contrast, there can be more than one minor mode per buffer. Sometimes the functionality of one will directly contradict another, though they generally combine harmoniously. For example, Emacs comes with lots of useful minor modes out-of-the-box. One of them highlights the current line. It is called `hl-line-mode`. Another displays line numbers on the side of the window: its symbol is `display-line-numbers-mode`. Those can work together in a buffer together with other minor modes, such as `show-paren-mode`, that highlights matching parentheses, and `electric-pair-mode` that inserts certain characters as pairs, like double quotes and parentheses. Because minor modes usually have complementary functionality, they are designed to work both locally and globally. Whatever conflicts are up to the user to resolve.

Otherwise, minor modes follow the rationale of major modes: they are commands that typically set some variables (not to imply they are simply though, as they can do anything they want). Since minor modes can, in principle, coexist, calling their command with *`M-x`* (`execute-extended-command`) toggles them on and off. Such a switch would not be of much use for major modes: it would be the same as activating `fundamental-mode`.

Once you delve deeper, minor modes are also supposed to have a different scope of application. A minor mode should not be messing around with the syntax table, for example, as then there could be a minor mode that breaks the current major mode. This could be done though, but the programmer must be extra careful how they go about dealing with the resulting complexity, given that other minor modes may also be in effect while major modes may be doing something of their own that has to be accounted for. The macro which defines a minor mode, namely, `define-minor-mode` will also not do exactly what `define-derived-mode` does for major modes. It will not specify a parent mode as there is no such concept of inheritance for minor modes. There will also be no abbrev table, no syntax table, and no keymap. A keymap can be associated with a minor mode, though it must be done explicitly with a special keyword. The documentation of `define-minor-mode` covers the details (Chapter 4 [Introspecting Emacs], page 9).

In general, you will benefit from thinking of Emacs as one big mutable state. Within it there are compartments with their own peculiarities. Emacs is designed to give priority to the particular over the general. The specific/local overrides the generic/global. This is true for variables, keymaps, abbrev tables, faces (via face remapping that we do not need to cover), and, in an analogous way, is also true for `let` bindings where the innermost overrides the outer ones (Chapter 13 [Evaluation inside of a macro or special form], page 33). For Emacs' modes, a major mode will override the settings of the major mode it is derived from, while a minor mode will take precedence over a major mode. Be mindful of this so that you can use it to your advantage.

11 Hooks and the advice mechanism

You know that each major and minor mode runs its own hook (Chapter 10 [What are major and minor modes], page 23). A hook is a variable that holds a list of functions. These are evaluated in sequence when the hook runs. The functions are normally called without an argument, so they must have their parameters set up correctly to avoid the error with the mismatching number of mandatory arguments. Some hooks are considered "abnormal" in that they call their functions with one or more arguments. How many arguments is up to each hook. I will show you in practice.

The function `add-hook` adds a function to a hook. It looks like the function call in Emacs Lisp (Chapter 2 [Basics of how Lisp works], page 2):

```
(add-hook 'text-mode-hook #'visual-line-mode)
```

The `text-mode-hook` is exactly what its name suggests: the hook that `text-mode` runs. It is evaluated after the `text-mode` is done setting all its variables. Generally, it is common for hooks to run at the end of the function that calls them, though it is also possible to run a hook before or during some operation. Notable examples are `before-save-hook` and `pre-command-hook`. The rule of thumb is this: if the hook's symbol does not imply otherwise, the hook is run at the end.

In the above example, you noticed how I added a quote to the variable `text-mode-hook` and a sharp quote to the function `visual-line-mode` (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). The expression would have still worked fine with a regular quote instead of the sharp quote. Though these added semantics can help us catch problems during byte compilation when we are developing packages.

The `add-hook` has a global effect by default: it updates the global value of the variable, not its buffer-local value (Chapter 7 [Buffers as data], page 17). For example, if you want to do something with, say, `hl-line-mode-hook` only in a specific file, then you need to arrange for the `add-hook` side effect to be applied in the buffer that visits the given file (Chapter 5 [Side effect and return value], page 11). This is done by passing a non-`nil` argument to the '`LOCAL`' parameter of `add-hook`, as we learn from its documentation (Chapter 4 [Introspecting Emacs], page 9):

```
(defun my-display-line-numbers-toggle ()
  "Enable `display-line-numbers-mode' is `hl-line-mode' is active.
Else disable `display-line-numbers-mode'."
  (if hl-line-mode
      (display-line-numbers-mode 1)
    (display-line-numbers-mode -1)))

(add-hook 'hl-line-mode-hook #'my-display-line-numbers-toggle nil t)
```

The function `my-display-line-numbers-toggle` is now set up to run in the current buffer whenever `hl-line-mode` is enabled or disabled. Try it! Here you will also notice a technicality. Each minor mode has a function and a variable of the same symbol. When the minor mode is enabled, the variable is set to a non-`nil` value.

Internally, a function that runs a hook does it with a call to `run-hooks`:

```
(run-hooks 'text-mode-hook)
```

The `run-hooks` is plural because it has a '`&rest`' keyword in its parameters, meaning that we can give it as many arguments as we want and it will collect them all into a single list that will then be handled appropriately. Thus:

```
;; No matter how many arguments we pass, they will be collected into a
;; single HOOKS list, owning to the &rest keyword.
(run-hooks 'text-mode-hook 'hl-line-mode-hook)
```

To remove a function from a hook variable, we use `remove-hook`. It is similar to `add-hook`, though check its documentation because its parameters are not the same:

```
;; Remove globally
(remove-hook 'hl-line-mode-hook #'my-display-line-numbers-toggle)

;; Remove locally
(remove-hook 'hl-line-mode-hook #'my-display-line-numbers-toggle t)
```

Thus far we have examined "normal hooks", i.e. hooks whose functions are called without any arguments. By convention, a normal hook has the '`-hook`' prefix. "Abnormal hooks" are called with one or more arguments. Each hook decides how many those are and what they are expected to be. Abnormal hooks have the suffix '`-functions`'. The `add-hook` and `remove-hook` work the same way for abnormal hooks. What is different is the functions we have to add to them, in order to get the expected results: they must be able to handle the arguments passed to them. Here is an example of defining and then calling a hook. Remember that hooks are ordinary variables:

```
;; Example with a normal hook:
(defvar my-normal-hook nil
  "Normal hook for `my-hello-world'.")

(defun my-hello-world ()
  "Print a hello message and run `my-normal-hook'."
  (message "Hello world")
  (run-hooks 'my-normal-hook))

;; Example with an abnormal hook:
(defvar my-normal-functions nil
  "List of functions to call after `my-say-hello'.
Each function takes two arguments, the name of the person and their
country of residence.")

(defun my-say-hello (person country)
  "Say hello to PERSON from COUNTRY."
  (message "Hello %s from %s" person country)
  (run-hook-with-args 'my-normal-functions person country))
```

The `run-hook-with-args` takes one hook and the arguments that are passed to its functions. Otherwise the functionality is the same between running a normal hook and an abnormal hook.

The logic of a hook is simple "let the user do something before or after some event". The only constraint is that the programmer has provided such a point of entry. There are times though where you want to do something, but there is no hook for it. This is where the advice mechanism comes in. We can "advise" a function to run some code before or after or even around some other function. In practice, we can redefine the original function by wrapping it in our function. The documentation string of `add-function` provides a concise overview, but let me show you a couple of common use-cases. Before you proceed, promise me to use the advice mechanism with care, because you can produce unexpected results and create bugs in other packages.

Probably the simplest use of the advice mechanism is to do something after a given function is called. Conceptually, this is like a hook in the ordinary scenario. You do it with the function `advice-add`:

```
(defun my-after-next-line (&rest _)
  "Message the current line number while ignoring any arguments."
  (message "Current line: %s" (line-number-at-pos)))

(advice-add #'next-line :after #'my-after-next-line)
```

Once you evaluate this, move down a line and notice how you get the message with the current line number. Now do `advice-remove` to undo the effect:

```
(advice-remove #'next-line #'my-after-next-line)
```

Go back to the definition of `my-after-next-line`. It says that it ignores any arguments. This is done by writing the '`&rest`' keyword and then the name of some parameter prefixed by an underscore. If we do not care about the name either, we just write the underscore. The underscore is how we convey the meaning of "I shall ignore this, mate; cheers!". The reason I wrote it this way is because the advice is called with the arguments of the original function. If I wanted to do something with one or more of them, I could have written the list of parameters in one of the following ways:

```
;; Two mandatory parameters.
(defun my-demo (parameter-one parameter-two)
  ...)

;; A mandatory parameter and a list of zero or more other acceptable arguments.
(defun my-demo (parameter-one &rest all-others)
  ...)

;; As above but we declare the intention to ignore ALL-OTHERS.
(defun my-demo (parameter-one &rest _all-others)
  ...)

;; Two optional parameters, meaning that the function can be called
;; without them.  But if the function is called with arguments, those
;; must be maximum two.
(defun my-demo (&optional parameter-one parameter-two)
  ...)

;; One mandatory and one optional parameter.
(defun my-demo (parameter-one &optional parameter-two)
  ...)

;; One optional parameter and a list of zero or more acceptable arguments.
(defun my-demo (&optional parameter-one &rest all-others)
  ...)

;; As above, but ALL-OTHERS is meant to be ignored.
(defun my-demo (&optional parameter-one &rest _all-others)
  ...)

;; One mandatory, one optional parameter, and then a list of zero or
;; more acceptable arguments.
(defun my-demo (parameter-one &optional parameter-two &rest all-others)
  ...)
```

Keep the above in mind while adding an advice. If you get the parameters wrong, then errors will ensue. The easiest is '`(&rest _)`' though it may not be what you need to do.

Other than the '`:after`' advice, the '`:around`' is common. You want to use it to do something before and/or after the original function is called. For example, I want the `fundamental-mode` to actually have a hook, so I do this, which also happens to tie together what we have covered in this chapter:

```
(defvar my-fundamental-mode-hook nil
  "Normal hook for `fundamental-mode' (which is missing by default).")

(defun my-fundamental-mode-run-hook (&rest args)
  "Apply ARGS and then run `my-fundamental-mode-hook'."
  (apply args)
  (run-hooks 'my-fundamental-mode-hook))

(advice-add #'fundamental-mode :around #'my-fundamental-mode-run-hook)
```

After installing this advice, calling `fundamental-mode` will also run the `my-fundamental-mode-run-hook`. Neat!

Check how the function `my-fundamental-mode-run-hook` makes use of the '`args`' it gets. The `(apply args)` effectively means "call the original function with its original arguments". It works because the function and its arguments are all part of one list, namely '`args`', where the symbol of the function is the first element and the arguments to it are the remaining elements—exactly how a function is called. Now, it so happens that `fundamental-mode` is called without any arguments, but the code will still work even if `fundamental-mode` is rewritten to have parameters. What matters is that the pattern of '`&rest args`' and then `(apply args)` is robust.

I like to think of `apply` as (i) "here is a quoted list, remove the quote and evaluate it" or (ii) "here is the symbol of a function and then a quoted list of arguments, so append the symbol to the following list, remove the quote, and evaluate it". In other words, these two examples will yield the same results (Chapter 6 [Symbols, balanced expressions, and quoting], page 12):

```
(apply '(+ 1 1 1))
```

```
;; => 3

(apply #'+ '(1 1 1))
;; => 3
```

Because the function `my-fundamental-mode-run-hook` I was showing you earlier has full control over when to make use of the ‘`args`’, it gives me the power to do something before I call `apply`. I do not need it here, but you get the idea.

The advice mechanism is powerful, though you have to use it judiciously. Emacs gives you the power. It is then up to you to act responsibly. Otherwise, you will encounter nasty bugs and create problems for others.

12 Evaluate some elements inside of a list

You already have an idea of how Emacs Lisp code looks like (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). You have a list that is either evaluated or taken as-is (Chapter 2 [Basics of how Lisp works], page 2). There is another case where a list should be selectively 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 (Chapter 22 [When to use a named function or a lambda function], page 63)!

```
(defvar my-greeting-in-greek "Γεια σου"
  "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)"
```

The `message` function has the side effect of appending the message it produces to the '`*Messages*`' buffer (Chapter 5 [Side effect and return value], page 11). You can review the log this way. Now check the code we have here. 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 pay attention to the next code block to understand how I can tell Emacs that I want the entire list to still be quoted but for the singular element `my-greeting-in-greek` to be evaluated. This means that `my-greeting-in-greek` is replaced by its value while everything else is taken as-is:

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

The syntax here is special. Instead of a single quote, I am using the backtick character or back quote or grave accent, to write an expression that is "quasi quoted". The backtick behaves like the single quote except for anything that is preceded by a comma. The comma only has meaning inside of a quasi quoted list, otherwise you get an error. The comma is an instruction to "evaluate the thing that follows". It written as a prefix without a space after it and applies to the thing it is attached to. The "thing" that follows is either a symbol or a list. The list can, of course, be a function call of arbitrary complexity. 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 " " "Πρωτεσίλαε") four))
;; => "(one two \"Γεια σου Πρωτεσίλαε\" four)"
```

Do not forget that you will get an error if you are not quoting this list at all, because the first element `one` will be treated as the symbol a function, which must then 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` will then have to exist as variables, since they are not quoted either. Else more errors ensue. By quasi quoting the list, we get the best of both worlds: evaluate what we need and skip the evaluation of all the rest.

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 from 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 provide an example here, as I think this is best covered in the context of defining macros (Chapter 13 [Evaluation inside of a macro or special form], page 33).

Chances are you will not need to use the knowledge of selective 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. Besides, when you write something like '``(one ,two ,three)`' you can alternatively call the `list` function like '`(list 'one two three)`' to get the same results. Sometimes the backtick with all its comma operators looks right while at others the `list` is more appropriate. The way I approach this is in terms of style: what would look more clean in the given context is the more appropriate option. Mine is an aesthetic principle, if you will. In this example, I prefer '`(list 'one two three)`' over '``(one ,two ,three)`': it has no special syntax beside the basics of Lisp and is thus easier to reason about (Chapter 2 [Basics of how Lisp works], page 2).

Lastly, since I introduced you to some Greek words, I am now considering you my potential 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 resorted to the use of Greek words. My instructor had a strict policy of only responding in 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, madame, Greek; me no speak England very best." I was not actually a beginner at the time. I simply could 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 the elements of Emacs Lisp.

13 Evaluation inside of a macro or special form

In the most basic case of Emacs Lisp code, you have lists (Chapter 2 [Basics of how Lisp works], page 2). Those are either evaluated or not (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). If you get a little more fancy, you have lists whose elements are selectively evaluated (Chapter 12 [Evaluate some elements inside of a list], page 31). Sometimes, though, you come across 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 " " "Πρωτεσίλαε")
```

You encountered this code in the section about the evaluation of some elements inside of a list. 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 resolve their respective values. Once it collects those it calls `concat` with them. 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 get its value.
- Strings are self-evaluating (a string evaluates to itself), while the `my-greeting-in-greek` is a variable that likely has some value other than itself.
- 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 " " "Πρωτεσίλαε")

(concat "Γεια σου" " " "Πρωτεσίλαε")
```

This is predictable. It follows the basic logic of the single quote: if a symbol is quoted, do not evaluate it and return it as-is, otherwise evaluate it and return its value. Here `my-greeting-in-greek` is not quoted, so it is evaluated. All clear! But you will find plenty of cases where this expected pattern is seemingly not followed. Consider this common scenario of using `setq` to bind a symbol to the given value (i.e. set the value of a variable):

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

The above expression looks like a function call, meaning that (i) the entire 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 so-called "special form" that internally calls set while quoting the symbol it got, thus:

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

With set 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 have a sense of when you are dealing with special forms. Those include setq, defun, let, and a few others. Here is a function you have already seen in this book:

```
(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 parameters 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. Imagine this:

```
;; PSEUDO-CODE to illustrate how the `defun' special form could be
;; expressed if normal quoting was applied.
(PSEUDO-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))
```

Now you can appreciate defun a little bit more.

Another common scenario is with let (Chapter 18 [Control flow with if-let* and friends], page 50). 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-BINDINGS' 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'. Behold the pseudo-code that illustrates my point:

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

From a user perspective, there is no distinction between special forms and Lisp macros. The former are implemented in C, as part of the engine at the heart of Emacs. While Lisp macros are written in Emacs Lisp, the same way as everything else you see in this book. In practice, macros is how we define our own special form counterparts, in the sense that macros can implement their own rules of evaluation. A macro can even define its own mini-language, such that it looks profoundly different from your average lisp expression. Depending on how you feel about it, something like `cl-loop` will be a thing of beauty or the bane of your existence (Chapter 20 [Pattern match with `pcase` and related], page 55).

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. A cursory review of some `use-package` declaration will suffice: you are not looking at your average Lisp program and what you know may not even be valid syntax for this macro (Chapter 2 [Basics of how Lisp works], page 2).

To learn what a given macro 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 backtick, the comma operator, and the mechanics of splicing (Chapter 12 [Evaluate some elements inside of a list], page 31). 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
  `(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`' keyword makes the following parameter a list of arbitrary length. So you can pass any number of arguments to it, all of which are collected into a single '`expressions`' variable. The judicious use of selective evaluation inside of a quasi quoted list ensures that the macro will not be evaluated right now but only when its expansion is called. We thus appreciate how the quote is, in effect a "do not evaluate now but pass it over to the next evaluation" (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). The arguments given 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 out of the macroexpansion:

```
(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 the following:

```
(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 backtick, 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 (Chapter 23 [Make your interactive function also work from Lisp calls], page 65).

```
(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 at a later point. The latter part starts with the backtick. This distinction is important when we call the macro, because the former part will be executed outright: if we hit the error, the macro will never expand the rest of the form which includes 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 and, indeed, more powerful. Experience will teach you when a macro is the right tool for the job. What matters here is that you have a sense of how evaluation works so that you do not get confused by all those parentheses, quotes, quasi quoting, and splicing. Otherwise, you might expect something different to happen than what you actually get. Use `pp-macroexpand-last-sexp` meticulously and try to reason about what it shows you.

14 Mapping over 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 (Chapter 5 [Side effect and return value], page 11). 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 (Chapter 22 [When to use a named function or a lambda function], page 63). 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 (Chapter 6 [Symbols, balanced expressions, and quoting], page 12).

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 (Chapter 2 [Basics of how Lisp works], page 2). This is a matter of getting used to how the code is expressed (Chapter 13 [Evaluation inside of a macro or special form], page 33).

When to use a `dolist` as opposed to a `mapc` is largely 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 (Chapter 18 [Control flow with `if-let*` and friends], page 50). 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`’.

15 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 `looking-at`, `re-search-forward`, `string-match`, and related. Each time you perform a search, the match data gets updated. Be mindful of this common side effect (Chapter 5 [Side effect and return value], page 11). 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 (Chapter 8 [Text has its own properties], page 19).

```
(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` (Chapter 16 [Switching to another buffer, window, or narrowed state], page 44): it will do what it must and finally restore the `point` (Chapter 21 [Run some code or fall back to some other code], page 58):

```
(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 it 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 Cypr


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

If all you want is to check for a regular expression in the buffer, you wrap your search in a `save-match-data`. If, on the other hand, you need to test for the presence of a regular expression in a string, without modifying—or indeed wanting to access—the match, then you can rely on the function `string-match-p`. Given that much of the work you do in Emacs involves buffers, it is important to familiarise yourself with how match data work (Chapter 7 [Buffers as data], page 17). Another use-case for match data is if you are writing fontification rules for a major or minor mode, though that is probably more of a niche requirement (Chapter 10 [What are major and minor modes], page 23).

16 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 in order to perform some computation. 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 once your program concludes its operations (Chapter 5 [Side effect and return value], page 11).

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 (Chapter 15 [The match data of the last search], page 42):

```
(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 visibility state of the buffer. This pertains to whether the buffer is narrowed or not and the exact boundaries of the narrowed view. When the buffer is narrowed, the functions `point-min` and `point-max` return the minimum and maximum positions within those boundaries. Your program may need to work outside of them, yet still respect them after it is done. Same principle if you need to narrow to some portion of the buffer to better target a piece of text. 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 (Chapter 21 [Run some code or fall back to some other code], page 58).

At any rate, because Emacs effectively is one large mutable state, you have to write code that does not produce permanent side effects that are not necessary. Guard against them and you should be good to go (Chapter 24 [Emacs as a computing environment], page 68).

17 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 (Chapter 2 [Basics of how Lisp works], page 2).

```
(defun my-15-lines-down ()
  "Move at most 15 lines down."
  (interactive)
  ;; Give it a negative number to move in the opposite direction.
  (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` (Chapter 5 [Side effect and return value], page 11). 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 peculiarity of Emacs Lisp 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. Do not leave parentheses on their own line. Put them at the end of a balanced expression, as you have seen in all my examples. 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 (Chapter 6 [Symbols, balanced expressions, and quoting], page 12).

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 parenthesis or have too many of them. Generally, expressions at the same level of depth 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 (Chapter 13 [Evaluation inside of a macro or special form], page 33). The "or more" 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 like this:

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

The else clause does not need a `progn` because internally it is functionally equivalent to using a '`&rest`' for the arguments. You may remember some of the examples I demonstrated with '`&rest`' (Chapter 11 [Hooks and the advice mechanism], page 26). In short, it will automatically collect all arguments into a single list. But because we can only have one '`&rest`' parameter, the first argument passed to `if` cannot be done the same as the else. Besides, how would we be able to disambiguate something like this?

```
(if COND ONE TWO THREE)
```

Without knowing where the arguments start and end, we cannot have a reliable way forward. Whereas this is clear:

```
(if COND (progn ONE TWO) THREE)
```

This would also work, but is unnecessary:

```
(if COND (progn ONE TWO) (progn THREE FOUR))
```

```
;; Same as above because of how there is a de facto &rest for the "else" consequences
(if COND (progn ONE TWO) THREE FOUR)
```

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)
```

I personally always opt for `when` instead of writing an `if` without an "else" branch, because this way I am clear about what I am trying to do. When I look back at my code, I do not need to double-check every `if` statement to confirm that it is balanced the way it ought to.

Similarly, the `unless` has the meaning of '`(when (not COND) EXPRESSIONS)`'. It, too, is a macro that expands to an `if` statement and, again, I rely on it whenever I want to make my intent clear:

```
(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 what you are doing, you may not even need to write anything more than `and` or `or`. For example, suppose you want to express the logic of "if something is true, return it, else return an error. You may write this using an `if`, but an `or` will also do, thus:

```
;; Return THIS if it is non-nil
(if THIS
    THIS
  (error "The THIS cannot be nil"))

;; This is practically the same as above
(or THIS
    (error "The THIS cannot be nil"))
```

Like `or`, `and` can also do the same trick:

```
;; These are practically the same.
```

```
(when (and THIS THAT)
  (FUNCTION-FOR THIS THAT))

(and THIS
     THAT
     (FUNCTION-FOR THIS THAT))
```

Depending on the specifics of the case, the combination of multiple `if`, `when`, `or`, `and` will look awkward. You can break down the logic into 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 because `cond` is a special form (Chapter 13 [Evaluation inside of a macro or special form], page 33). 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` or the `if COND (progn EXPRESSIONS)` pattern. 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 (Chapter 3 [Evaluate Emacs Lisp], page 6). 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 (Chapter 20 [Pattern match with `pcase` and related], page 55).

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

A function may need to process data that is local to it; data that is located in the context of the function, else in its "lexical scope". This means that whatever variables or values thereof should not be exposed to the environment outside of the function. The `let` and `let*` create and bind variables that are available only within their scope, else the '`BODY`' of the `let`. These bindings can be arbitrary new symbols or existing symbols with a new value. Those will take precedence over the value of the same variable outside of this lexical scope. Again, you will notice how Emacs is consistent about the principle of the specific/local taking precedence over the generic/global (Chapter 10 [What are major and minor modes], page 23). 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 (Chapter 13 [Evaluation inside of a macro or special form], page 33). 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 (Chapter 17 [Basic control flow with `if`, `cond`, and others], page 45). 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)
```

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` immeditately.

The bindings of both `if-let*` and `when-let*` have an implied `and` logic to them, exactly because they do not accept a `nil` value: as soon as a `nil` is returned, they are aborted. Concretely, if a `if-let*` has three bindings and the second is `nil`, the third will never be bound to what you expect. In the following example, the `if-let*` has the third binding of ‘`(country "Cyprus")`’ which, however, is not established because `if-let*` exits at the `nil` it gets from its second binding. Consequently, the ‘`ELSE`’ branch of the logic get `nil` out of `country` instead of what we thought we were assigning it to.

```
(if-let* ((name "Protesilaos")
          (nickname nil)
          (country "Cyprus"))
    (message "This does not run")
  (message "The country is %s" country))
;; => "The country is nil"
```

Remember this implicit `and` for the `if-let*` and `when-let*` bindings, otherwise your code will not do what you expect.

As you dig deeper into the Emacs Lisp ecosystem, you will come across uses of `if-let*` or `when-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`, while `when-let*` exits immediately and 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 and thus refrain from binding
          ;; `variable2' to whatever it would otherwise be bound to.
          (_ (stringp variable1))
          (variable2 (OTHER-FUNCTION OTHER-ARGUMENT)))
    THIS
  ELSE)
```

I personally enjoy using `if-let*` and `when-let*`. They have the same syntax as `let*`, which means that I do not need to remember the intricacies of yet another Lisp macro. Plus, they express intent which is always good as a way of making the code self-evident or, at least, easier to reason about. Ultimately though, 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`, in which case `let` or `let*` are what you need. Choose what makes sense.

19 Autoloading symbols

In general, Emacs knows the functions and variables it has evaluated (Chapter 3 [Evaluate Emacs Lisp], page 6). Though it is also aware of all symbols that are marked for automatic loading inside the `load-path`. The `load-path` is a variable whose value is a list of directories. Each of those directories is expected to contain Elisp files. Those are specified with the '`.el`' file type extension, such as `my-hello.el`. An Elisp file may choose to mark for automatic loading some of the symbols it defines. When a symbol is marked for automatic loading, also known as an "autoload", Emacs will do the following when that symbol is called for evaluation:

- If the file with the given symbol's source code have been loaded, then Emacs will simply evaluate the symbol in the given call.
- If the file has not been loaded, Emacs will first load and evaluate the file, and then evaluate the symbol from where it is being called.

In this way, we can write programs that load their source code only when needed. Such is a concept that is also described by members of the community as "lazy loading". The idea is that we do not need to evaluate potentially millions of lines of code as soon as we launch Emacs. Most of that is likely not needed each time. But when we wish to do something with one of those definitions, its code base becomes available to us.

Programs will typically have functions that are there purely for internal purposes and others which are meant to be called by the user. The latter may be autoloaded. Though the programmer must take care to autoload only what is the actual point of entry to the package and not every single symbol that may be called at some point. For example, the built-in `ielm.el` autoloads the command `ielm`. This makes perfect sense. If we want to start using that package, we will do it by invoking `ielm`. The program defines many other commands, such as `ielm-return`. Those only work inside of the IELM shell. It thus is wrong to autoload them and, indeed, `ielm-return` and all related commands are not autoloaded. Compare the differences by doing the following:

- Call the command `find-library`. It will ask you for a file anywhere in the `load-path` that contains Emacs Lisp.
- Select '`ielm`' to visit the file `ielm.el`.
- Search for `(defun ielm (`. Notice that the definition of this command has a special '`;;;###autoload`' directive right above it.
- Now search for `(defun ielm-return`. This one does not have the '`;;;###autoload`'.

In this case, `ielm` is declared for autoloading, while `ielm-return` is not. The latter will be available only after the former has been called (or the entire feature is explicitly loaded by the user or another program). This is the correct way to organise the code of a program.

From the programmer's persective, autoloading is set up with those special '`;;;###autoload`' comments. For example:

```
;;;###autoload
(defun my-hello-world ()
  "Greet the world."
  (message "Hello everyone! My name is Protesilaos, also known as Prot!"))
```

If the code you want to autoload is the result of a macro, then you would need to write it this way (Chapter 13 [Evaluation inside of a macro or special form], page 33):

```
;;;###autoload (autoload 'my-hello-world "my-hello")
(my-hello-define-command ARGUMENTS-FOR-THE-MACRO-THAT-WILL-YIELD-my-hello-world-DEFINI
```

Here the `my-hello-define-command` is an Elisp macro that expands into a function like the `my-hello-world` shown further above. Because it is the latter you wish to autoload, you need to write the exact function call that would yield the desired result, hence '`;;;###autoload (autoload 'my-hello-world "my-hello")`'.

When you write '`;;;###autoload`' on its own, you are effectively asking to operate on the following symbol being defined. Whereas an '`;;;###autoload WHAT-TO-LOAD-EXACTLY`' gives you complete control over what to autoload by specifying the exact function call to be used. Generally, '`;;;###autoload`' is enough.

The '`;;;###autoload`' statements are read by the function `loaddefs-generate`. This is typically done when you install a package, such as with the `package-install` command. The relevant machinery will read every '`.el`' file to generate a new file that contains all the autoloads. This file will be in the same directory as the rest of the source code, all of which is part of a directory among those specified in the `load-path`.

If you have ever installed a package, then you most likely have files with autoloads. You can check what they look like by doing the following:

- Visit the directory where you Emacs configuration file is. It might be at '`~/.emacs.d/`' or '`~/.config/emacs/`'. When in doubt, inspect the value of the variable `user-emacs-directory`.
- If you have installed packages, you will find an '`elpa`' directory.
- Therein are all the directories of the packages you have.
- Inside those you will find '`.el`' files, which are the original source code, '`.elc`' files which are the byte compiled counterparts of the former, and a '`PACKAGE-autoloads.el`' file with all the collected autoloads. Here '`PACKAGE`' is a placeholder for the name of the package.

When you start up Emacs, it reads all those `PACKAGE-autoloads.el` and thus knows where the specified symbols are defined. From then on, the lazy loading happens when it must, i.e. when the affected symbol is called for evaluation.

Every Emacs user experiences the benefits of autoloading from day one. Consider the scenario where you launch a pristine Emacs for the first time. This can be done at any moment by running the following on the command-line or terminal emulator:

```
emacs -q
```

This generic Emacs session should take a short amount of time to initialise its state and be ready for further input. Do *M-x* (`execute-extended-command`) and press the *TAB* key. A '`*Completions*`' buffer will pop up, informing you about the thousands of commands that are available as candidates of the present interactive function call. Emacs has not actually evaluated the code of all those: that would take a lot more time than what you experienced at startup. Instead, Emacs is aware of all of them via the autoload mechanism. Once you invoke a command, Emacs will take care to load the corresponding file and then perform whatever computation the command specifies (Chapter 23 [Make your interactive function also work from Lisp calls], page 65).

What is usually autoloaded are commnads, though variables can be handled the same way. It all depends on what the program is doing.

20 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 (Chapter 17 [Basic control flow with `if`, `cond`, and others], page 45). You might even get more fancy with `if-let*` and `when-let*` (Chapter 18 [Control flow with `if-let*` and friends], page 50). However you go about it, there are some cases that arguably benefit from more succinct expressions. "Arguably" is the operative term, because code that is more terse can also be harder to reason about. At any rate, this is where the `pcase` Lisp macro comes in (Chapter 13 [Evaluation inside of a macro or special form], page 33). 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. Unlike `cond`, it can perform pattern matching and create binding that it can then used based on the pattern it is matching.

Let me show you how `pcase` works like a `cond`. Here is an example that compares the buffer-local value of the variable `major-mode` for equality against a couple of known major mode symbols, namely `org-mode` and `emacs-lisp-mode` (Chapter 10 [What are major and minor modes], page 23):

```
(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 the following `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 already 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. If we care about "elegance", we might prefer this way of expressing ourselves:

```
(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. It works, but now you have to keep in your head the `message` that is waiting for the `pcase` that is testing for many conditions.

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 (Chapter 13 [Evaluation inside of a macro or special form], page 33).

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
    (`(,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 (Chapter 3 [Evaluate Emacs Lisp], page 6). What it showcases is the various ways we write the cases that `pcase` checks for. You wil notice the use of the backtick and the comma, which is similar to the quasi quoting I have covered before (Chapter 12 [Evaluate some elements inside of a list], page 31). Those perform the pattern matching against the expression that is given to `pcase`, in this case the '`argument`'. If the pattern matches, the elements of the list are bound to the corresponding elements returned by '`argument`'. Then they can be used inside of that lexical scope (Chapter 18 [Control flow with `if-let*` and friends], page 50).

Below are a couple of examples with `my-toy-pcase`:

```
(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 in `my-toy-pcase` 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 in the list. The syntax used for this destructuring is arcane, until you relate it to the backtick and the comma which are used for selective evaluation (Chapter 12 [Evaluate some elements inside of a list], page 31).

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! Perhaps calling something `pcase-let` when it does not handle cases is wrong, but you will eventually get used to the '`pcase-`' prefix being a reference to destructuring capabilities. Those are especially useful when you are working with the return value of a function which comes as a list.

Consider a `pcase-dolist`. It maps through a list of elements for side effects only, just like `dolist`, while doing destructuring (Chapter 14 [Mapping through a list of elements], page 38). For example, evaluate the following `defvar`, then the `pcase-dolist`, and then check the '`*Messages*`' buffer to confirm what it did. Once you do that, come back and evaluate the `dolist` and check the '`*Messages*`' once more.

```
(defvar my-sample-data-for-pcase-dolist
  '(("Craig Gordon" "Goalkeeper" "Hearts")
    ("Aaron Hickey" "Defender" "Brentford")
    ("Grant Hanley" "Defender" "Hibernian")
    ("Scott McKenna" "Defender" "GNK Dinamo")
    ("Andy Robertson" "Defender" "Liverpool")
    ("Lewis Ferguson" "Midfielder" "Bologna")
    ("John McGinn" "Midfielder" "Aston Villa")
    ("Scott McTominay" "Midfielder" "Napoli")
    ("Ben Gannon-Doak" "Forward" "Bournemouth")
    ("Ryan Christie" "Forward" "Bournemouth")
    ("Lyndon Dykes" "Forward" "Birmingham"))
  "The starting lineup of Scotland's final qualifier match for the World Cup 2026.")

;; Here we go through a list of elements for side effects only while
;; relying on destructuring to create the bindings.
(pcase-dolist (`(,name ,position ,club) my-sample-data-for-pcase-dolist)
  (message "%s is a %s who plays for %s" name position club))

;; And here is the equivalent with a regular `dolist' and `let'.
(dolist (element my-sample-data-for-pcase-dolist)
  (let ((name (nth 0 element))
        (position (nth 1 element))
        (club (nth 2 element)))
    (message "%s is a %s who plays for %s" name position club)))
```

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. I do rely on them on occasion, though I am not fully convinced that destructuring is the superior paradigm. It makes the code harder to reason about if you are not well versed in its domain-specific language (Chapter 2 [Basics of how Lisp works], page 2). This is especially true once you work with real-world programs rather than the simple snippets I am showing you here.

21 Run some code or fall back to some other code

Your typical program will rely on `if`, `cond`, and the like for control flow (Chapter 17 [Basic control flow with `if`, `cond`, and others], page 45). Depending on your specific needs or stylistic considerations, it may even include `pcase` (Chapter 20 [Pattern match with `pcase` and related], page 55) as well as `if-let*` or related (Chapter 18 [Control flow with `if-let*` and friends], page 50). Whatever style you prefer there are some cases 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 so that any unwanted—yet temporarily necessary—side effects are undone (Chapter 5 [Side effect and return value], page 11). The logic is "do some computation with all the desired side effects, then once that transpires do some other computation to undo the side effects of the latter or perform whatever kind you reset you consider necessary". This is the concept of "unwinding", which is implemented via `unwind-protect`.

Here is some pseudo code:

```
(unwind-protect
    ;; First do this
    DOING-STUFF
  ;; Then do this
  UNDOING-STUFF)
```

With the above example in mind, '`DOING-STUFF`' will be evaluated first and then '`UNDOING-STUFF`' will follow. If '`DOING-STUFF`' does not encounter errors, '`UNDOING-STUFF`' will be evaluated in sequence and the return value of this entire expression will be what '`DOING-STUFF`' returns. If, by contrast, '`DOING-STUFF`' exits without doing what it was supposed to (a "non-local exit"), then the '`UNDOING-STUFF`' will still be evaluated and there will be no return value. A non-local exit occurs when the `signal` function is called to report on some error. What `signal` effectively does is abort the current computation and, depending on the type of the signal like with `error`, produce a backtrace. Some signals, like `quit`, do not produce a backtrace: they simply abort the current computation. You signal `quit` whenever you type *`C-g`* (`keyboard-quit`).

To illustrate what I just wrote:

```
;; It does not make sense to use `unwind-protect' if we are not
;; producing side effects, but focus on the return value for now.
(unwind-protect
    (+ 1 1)
  (+ 10 10))
;; => 2

(unwind-protect
    (progn
      (+ 1 1)
      (signal 'quit (list "We are aborting")))
  (+ 10 10))
;; => NO RETURN VALUE.
;; => Quit: "We are aborting"
```

In other words, the unwinding part of the logic, which is the '`(+ 10 10)`' in this case, is for side effects only: it does not return a value.

Let me now show you when `unwinding-protect` makes sense. We want to produce some side effect and then undo it. In the following code block, I define a function which creates a minibuffer prompt. It asks 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 waiting for your input, the function highlights all instances of the regular expression '`(message`' in the current buffer. Those highlights must go away after you are done with the minibuffer and its consequences. What is in the `progn` is the part we want to run and we guard against its side effects with the following `unhighlight-regexp`.

```
(defun my-prompt-with-temporary-highlight ()
  "Ask for confirmation and highlight all instances of a regexp while waiting."
  (let ((regexp "(message"))
    (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))))

;; Call the following with the above in context.  Check how the
;; regular expression is highlighted.
(my-prompt-with-temporary-highlight)
```

Try the above in your Emacs to get a feel for it. Whatever answer you provide at the prompt, the code still exits locally, meaning that the `progn` produces whatever value it is designed to return. As always, the `unhighlight-regexp` is evaluated no matter what. To get a non-local exit from the `my-prompt-with-temporary-highlight` wait for the prompt and while it is open type *C-g* (`keyboard-quit`). Whether you exit locally or not, the highlights will be undone, which is the desired outcome.

In practice, your code may encounter an error, which is produced by the `error` function (it technically is the `signal` function with the `error` type). When that happens, you are presented with a backtrace and Emacs enters a "recursive edit". The recursive edit means that unwinding forms are suspend until you either type *C-]* (`abort-recursive-edit`) or select the window with the '`*Backtrace*`' buffer and do *q* (`debugger-quit`) from there. Emacs can enter more than one such recursive edit. While you are inside one, no more backtrace buffers will be shown: you must first exit the recursive edit. With default settings, the mode line shows levels of recursive edits by placing square brackets around the indicator with the major and minor modes (Chapter 10 [What are major and minor modes], page 23). Always remember to exit the recursive edit state, otherwise the unwind forms will not be evaluated and you may end up with an Emacs session that is in an inconsistent state.

To help you get a feel for it, I modify the function I wrote earlier to now produce an error before it reaches the minibuffer prompt. This will cause a non-local exit and will create a backtrace if you are not already in a recursive edit—so make sure you exit that one first.

```
(defun my-prompt-with-temporary-highlight-try-with-error ()
  "Ask for confirmation and highlight all instances of a regexp while waiting."
  (let ((regexp "(message"))
    (unwind-protect
        (progn
          (highlight-regexp regexp)
          (error "This error makes no sense here; close the backtrace to test the unwi
          (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))))
```

Evaluate this in some buffer that has the '`(message`' text and notice how the unwinding does not happen outright. It happens after you exit the recursive edit. Again, remember to use `abort-recursive-edit`.

```
(my-prompt-with-temporary-highlight-try-with-error)
```

Taking a step back, you will figure out how `unwind-protect` is a more general form of specialists like `save-excursion` and `save-restriction` (Chapter 16 [Switching to another buffer, window, or narrowed state], page 44), while it underpins the `save-match-data` (Chapter 15 [The match data of the last search], page 42) 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 controlled fashion, such as to handle an error specially instead of what it normally does, 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 any kind of `signal`. Put differently, your Elisp will express the intent of "I want to do this, but if I get a certain signal 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 choose to define via `define-error`. 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 (Chapter 13 [Evaluation inside of a macro or special form], page 33). 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 "You prefer not to continue; but we need to return an `error' for t
      (:success
       (unhighlight-regexp regexp)
```

```
           (message "No errors, but still need to unwind what we did, plus whatever else w
          (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, i.e. a local exit. 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 bind the error data to the symbol `error-data-i-got`, 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 "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 w
       (message "The error is `%s' and its data is `%S'" (car error-data-i-got) (cdr e
      (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 e
      (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 e
      (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 er
```

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.

22 When to use a named function or a lambda function

The lambda is an anonymous function or, if you will, the form of a computation as such (Chapter 6 [Symbols, balanced expressions, and quoting], page 12). defun defines a function with a given name ("eponymous", if I may bring more of the Greek language into this): technically, it creates an alias for a lambda. When to use one versus the other is often a matter of style. Though there are some cases where a certain approach has clear advantages.

The rule of thumb is this: if you need to use the function more than once, then give it a name and call it by its name. This way, you have a symbol which provides an indirection to the actual computation: if you change what the function is doing, the symbol is still the same, so your changes apply across your code base. Otherwise, you will have to update all the lambda forms you have.

The lambda is anonymous in the sense that it has nothing else pointing to it: there is no indirection. As such, you must modify the lambda in place. Knowledge of this quality is of paramount importance, especially when working with hooks and the advice mechanism (Chapter 11 [Hooks and the advice mechanism], page 26). Unless your code is trivial, do not use a lambda in a hook or advice as the lack of indirection makes it harder for you to tweak how your function works. Rely on a lambda when what you want to do is either trivial and not subject to further revision or you wish to perform an ad-hoc computation that is not needed elsewhere.

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 (Chapter 14 [Mapping over a list of elements], page 38):

```
(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-fn (lambda (name)
                          (message "Hello %s" name)))
         (greet-team-and-names-fn (lambda (team)
                                    (message "Greeting the team of `%s'..." team)
                                    (mapc greet-name-fn team))))
    (mapcar greet-team-and-names-fn teams)))
```

```
(my-greet-teams
 '("Pelé" "Ronaldo")
 '("Maradona" "Messi")
 '("Beckenbauer" "Neuer")
 '("Platini" "Zidane")
 '("Baresi" "Maldini")
 '("Eusebio" "Cristiano Ronaldo")
 '("Xavi" "Iniesta")
 '("Charlton" "Shearer")
 '("Cruyff" "Van Basten")
 '("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.

Again, there is no right or wrong. Use a `lambda` for ad-hoc computations, where you do not care about the potential for future edits. Otherwise give your functions a name and you shall benefit from the indirection they provide.

23 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" (Chapter 19 [Autoloading symbols], page 52). 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. Unless they absolutely depend on user input, they will continue to work just fine if called directly from Lisp (Chapter 3 [Evaluate Emacs Lisp], page 6).

In its simplest form, the `interactive` specification is an unquoted list like `(interactive)`, which looks exactly like a function call (Chapter 2 [Basics of how Lisp works], page 2). Here is a trivial example that calls the function `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 directly 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))
```

For trivial computations, it is inconsequential whether you have multiple functions do the work interactively and non-interactively. But if you are doing something more involved, you do not want to duplicate it across different functions. It is more maintenance work for you as the implementation details might change and the subtle problems shall arise. Emacs gives you the tools to write one function that can accept arguments when called from Lisp while still working the way you want when invoked interactively. 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 programmatically or interactively. 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 mirrors 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"
```

The code you write inside of the `interactive` specification does not have some special restriction. Express yourself as you see fit with `let` bindings and control flow (Chapter 17 [Basic control flow with `if`, `cond`, and others], page 45). What matters though is that you return a list whose `length` is equal to that of the list of parameters specified by the `defun`.

Of interest in this context is the `current-prefix-arg` variable: it holds the value of the prefix argument passed to the current function in interactive use. Users input it by typing *`C-u`* (`universal-argument`) for a "universal prefix" or *`C-u`* and then a number (`digit-argument`) for a "numeric prefix" (something like *`C-1`* is equivalent to *`C-u`* followed by *`1`*). There is no predefined behaviour to the prefix argument: it will do whatever you decide within the `interactive` by reading the `current-prefix-arg` and later in the body of the function where you parse that value. Here is an example:

```
(defun my-greet-interactive-and-non-interactive-plus-prefix-arg (name &optional be-pol
  "Greet person with NAME.
With optional BE-POLITE as a prefix argument, use a formal greeting,
otherwise remain casual.

When called interactively, produce a minibuffer prompt asking for NAME.

When called from Lisp, NAME is a string, while BE-POLITE is a non-nil
value."
  (interactive
   (list
    (read-string "Whom to greet? ")
    current-prefix-arg))
  (if be-polite
      (message "Greetings %s" name)
    (message "Howdy %s" name)))
```

```
(my-greet-interactive-and-non-interactive-plus-prefix-arg "Prot")
;; => "Howdy Prot"

(my-greet-interactive-and-non-interactive-plus-prefix-arg "Protesilaos" t)
;; => "Greetings Protesilaos"
```

And here is how you can take it a step further:

```
(defun my-greet-interactive-and-non-interactive-plus-prefix-arg (name &optional title)
  "Greet person with NAME.
With optional TITLE as a prefix argument, prompt for a formal title and
prepend it to NAME.

When called from Lisp, NAME and TITLE are strings."
  (interactive
   (list
    (read-string "Whom to greet? ")
    (when current-prefix-arg
      (read-string "Which title to use? "))))
  (if title
      (message "Greetings %s %s" title name)
    (message "Howdy %s" name)))

(my-greet-interactive-and-non-interactive-plus-prefix-arg "Prot")
;; => "Howdy Prot"

(my-greet-interactive-and-non-interactive-plus-prefix-arg "Protesilaos" "Teacher")
;; => "Greetings Teacher Protesilaos"
```

Call `my-greet-interactive-and-non-interactive-plus-prefix-arg` with and without a prefix argument to get a feel for it. It should do the same as what it does in the Lisp calls included in this code block.

The `interactive` specification is a powerful mechanism when used correctly. Do it right and you will end up with a rich corpus of functions that are equally useful to programs and to users.

24 Emacs as a computing environment

When you start using Emacs, you experience a text editor. You move the cursor around and insert text. This is basically it. At the surface level, Emacs looks like every other generic editor you have seen before. Other actions, such as selecting another window or switching to a new buffer, are ancillary to text editing. Using Emacs in this capacity is perfectly fine.

What makes Emacs something far more powerful than a text editor is its ability to be reprogrammed, else "extended", live by evaluating some Emacs Lisp (Chapter 2 [Basics of how Lisp works], page 2). Once you install a few packages or, better, learn to program in Emacs Lisp, your text editor will become a text-centric interface to increasingly more parts of your computing life. For example, if you are a programmer, you will eventually incorporate into your Emacs-based workflow tools for compiling your project, debugging your code, viewing changes, and sending patches. If you only write prose, you get to streamline your work for taking notes, maintaining an agenda, writing technical documents, sending emails, and more. The best part is that you can have your integrated programming setup and your plain text writing platform plus, generally, everything you can use Emacs for, all in one.

With Emacs as your primary interface, you benefit from the consistency and predictability of a unified system. You have pieced it together so you understand how it works, more or less. This means you can extend it when needed. Since Emacs is free software, you retain the freedom to modify it as you see fit, such as in response to evolving work requirements. If you have something that does what you expect, you will rely on it long-term. Emacs will grow or shrink to fit your current needs.

The consistency is one of presentation and function. At the level of appearances, you get to use the same font and theme settings, as well as the standard Emacs paradigms of frames, windows, buffers, the minibuffer, and text editing. On the functionality front, you rely on one programming language—Emacs Lisp—for all the interfaces you will be working with. As such, commands you have written or received from a package will be available to you in whatever new interface you bring into your setup. For example, if you have a minor mode that blinks the current line after a few seconds of idleness (e.g. because you want to be reminded where you are), this feature is already available when you start writing emails from inside Emacs: you do not need to port it the way you would if you were introducing an entirely new program to your workflow.

Compare this potential for unencumbered extensibility with what you would otherwise be doing. You would be relying on disparate applications that (i) do not necessarily talk to each other, (ii) each has its own concepts for the user experience, and (iii) each uses its own programming or configuration language. You can still get something decent if you are determined. Though you will realise that Emacs offers a better return on investment. For instance, the programming skills you acquire from extending your text editing apply directly to any future project to further extend Emacs. This is the gift that keeps giving.

As such, you will benefit from thinking of Emacs as (i) the layer of interactivity on top of command-line tools like `ls` and `grep`, and (ii) an interpreter of a programming language that can draw linkages between such tools, plus everything else it can do in its own right. In effect, Emacs is a turbocharged command-line interpreter like Bash equipped with a fully realised model of user interaction. Your Emacs Lisp programs are the equivalent of Bash scripts, plus all the accoutrements of interactivity. The counterpart to '`#!/bin/bash`' (which tells the computer which interpreter is needed for the given script) is the current Emacs process.

Use this concept to better understand what the value proposition of Emacs is for you in present time and in terms of sheer potential. Have fun!

Appendix A GNU Free Documentation License

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

25 Indices

25.1 Function index

25.2 Variable index

25.3 Concept index