Hacking your way around in Emacs

Marcin Borkowski

Hacking your way around in Emacs

Marcin Borkowski

This book is for sale at http://leanpub.com/hacking-your-way-emacs

This version was published on 2021-11-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Marcin Borkowski by spreading the word about this book on Twitter!

The suggested hashtag for this book is #hackingyourwayemacs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#hackingyourwayemacs

Contents

Introduction

The goal of this book is really very simple: to teach you, dear reader, how to program Emacs, or rather, to help you teach it to yourself. Emacs – and Emacs Lisp – have grown to be quite complex, but that doesn't mean you need a Ph.D. to code in Elisp. In this book, we will start small, though not *very* small – I assume that you've read the late Robert J. Chassell's excellent An introduction to programming in Emacs Lisp[1]. It's well-written, informative, pretty light and free, so why not? I will repeat a thing from that book here and there, but mostly I will build upon what you already know from it.

Of course, the first question we might want to ask is: *why* would you want to program Emacs? If you are an Emacs user, you most probably know the answer – to make your editing experience better. Maybe there is some behavior of Emacs which annoys you and you'd like to change it. Maybe you have some repetitive action you want to automate. Maybe you use some application which can be used via a CLI or a Web API and you want to drive it from Emacs. There can be probably a ton of other reasons.

If you are not (yet) an Emacs user, this book is probably not for you. If you heard that Emacs is a fantastic, programmable text editor and you could benefit from using it and coding in Emacs Lisp, you should probably start with learning how to *use* Emacs. The first resource you might want to check out is the built-in Emacs tutorial, and you could proceed to the (very fine!) Emacs manual or the Mastering Emacs[2] blog and book by Mickey Petersen. Then, at some point in time you can read *An introduction to programming in Emacs Lisp*, and come back to this book afterwards.

Ok, so you're still here. Great! Let me explain a bit about what I wanted to achieve with this book.

I spent considerable time thinking what to include here and what to leave out, and in what order to present things. And really, there is no universal way of doing it. It was clear to me from the beginning that I should start easy and then move on to more sophisticated topics, but that opened more questions than it answered. After a lot of internal discussions with myself and a few drafts, I decided on a few guiding principles for myself to follow.

First of all, I try not to pull rabbits out of my hat – instead of telling the reader "well, this and that is a function which solves the issue at hand, what, you didn't know about it? how stupid!", I tried to assume that you haven't read the whole source code of Emacs, or even the whole 1300+ pages of the GNU Emacs Lisp Reference Manual[3] (called the "Elisp reference" from now on). (If you have, why would you need this book anyway?) So, at least in the beginning, whenever I introduce some useful function, I try to show how you, yourself, could find out about it. (Later on you'll already know most of the tricks to find functions doing what you want.) It is not always easy, but Emacs

[1] https://www.gnu.org/software/emacs/manual/html_node/eintr/index.html
[2] https://www.masteringemacs.org/
[3] https://www.gnu.org/software/emacs/manual/elisp.html

really *is* self-documenting and many things are more or less easily discoverable. (Although why some functions were *named* the way they were is still a mystery to me.)

The next thing is that I tried to show real-life examples (as much as possible). Of course, many, many possible enhancements to basic text editing are either present in core Emacs or as packages, so sometimes the code here may replicate what is already available – but nevertheless, I tried to choose ideas that are really useful, and for as wide as audience as possible. This means that the examples are not tied to any specific programming language. Also, I did not touch Org-mode, which is great – and programmable – but not every Emacs user is also an Org-mode user.

Last but not least, I tried to cover as many useful things as possible (even if sometimes superficially), so that you will learn about some common techniques useful in many different situations, like leveraging built-in editing features, using region in your custom commands, writing a minor mode and many others. Sometimes I only mention some feature, variable or function, and if you find it interesting or useful, you can always go to the Elisp reference, the relevant docstrings or even the source code.

The book is intended to be read in order. This means that even if you are interested in e.g. writing a minor mode, you should probably at least skim through the first chapter before diving in to the second one. After all, this is a *textbook*, not a *reference*, and so in later chapters I sometimes use ideas explained earlier without much commentary.

Here is a short breakdown of the chapters. In the first one, we build a function to move the current line up or down, which is quite useful when programming – but also when e.g. typing short itemized lists. The second chapter is devoted to a utility helping with editing texts – a tool to rearrange words in a sentence. It is the longest one in the book, since the tool is going to be quite complicated, and we will use many Emacs features. In the third one, we count lines of code (in a way that can be used for Elisp or other programming languages).

This is a good place to thank a few people who made this book possible. It is obvious that it could never exist without Richard Stallman who wrote the first version of Emacs and numerous people who contributed to it over the decades. It is written in Org-mode, first made by Carsten Dominik and then developed by many other people. Diego Zamboni wrote the Org exporter[4] to convert Org-mode syntax to Markua, expected by Leanpub. Christian Tietze provided a nice idea now incorporated in the sentence-reordering code. Last but not least, my wife and children put up with me writing instead of spending more time with them.

If you like my writing and want to learn more about Emacs and Emacs Lisp, you might consider reading some Emacs blogs. A good portion of my personal blog[5] is also devoted to Emacs.

Just in case, please remember that when I link to some website, mention some person or quote some piece of art, it does not mean that I fully endorse any of those things or share the values of the person mentioned – it means that I consider that information relevant, interesting or maybe just funny in the context.

[4] https://github.com/zzamboni/ox-leanpub
[5] http://mbork.pl

Important note: this book is still subject to change. What you are reading is version 1, but I hope to add at least two more chapters some day in the future, creating version 2. (When? I don't know yet. A Wednesday, perhaps. Maybe next Wednesday, or just one of the Wednesdays. Seriously though, some time in mid-2022 is the most probable time.) Nevertheless, I consider it "complete" in the sense that it is a certain whole, and even if I don't manage to add anything, it can stand on its own as it is now, and the chapters that do exist should not change too much.

Also, you may want to know that I consider releasing this book on some pretty permissive license in the future (most likely one of the Creative Commons ones), but not earlier than perhaps 2023. If you prefer not to pay for it, it might be enough to wait a few years. On the other hand, I owe it to my family that the time I spend *without* them is compensated, e.g. in the form of money needed to sustain our well-being – so I am only willing to release the book for free if and when I attain a reasonable hourly rate (that is, if/when the income it generates divided by the amount of time I spent on it reaches some minimum). This means that paying me *now* is a step towards making it more probable that the book will be available for free *in the future*.

So, let's get into business of programming Emacs!

Moving lines around

Introduction

Let us start simple. We will write a function to move the current line up or down. Of course, "moving up" means "swapping this line with the previous one", and "moving down" means "swapping this line with the next one". This is actually already a solved problem – Emacs has the `transpose-lines` function, but the way it works is conceptually different than "moving the current line up or down" – you need to put the point *between* the lines you want to transpose (or more precisely, on the second one), and after the transposing the point lands *after* the line that was moved down. Therefore, the code we are going to write actually extends Emacs – if only a bit. (Well, there *does* exist a package which allows to move a line or the region up or down, called move-text[6], but our solution is in fact a bit different – definitely simpler, but in some respects better.)

In this chapter, we will first encounter the most basic technique of Elisp programming, which is mimicking the actions of a human editor. In other words, our first version of the `move-line-down` command will be just invoking commands which perform actions the user might do when asked to move the line the point is on below the next one. Then, we are going to make things more complicated for the programmer, but simpler for the user – we are going to use variables to make the user experience less surprising, then we will learn to use prefix arguments, make our command play nice with the undo system, manipulate text in the buffer (deleting and inserting portions of it), and finally use conditionals to support negative prefix arguments.

All the techniques we use in this chapter are pretty basic (with the possible exception of the short section about undo), and are hardly new if you did your homework and read Chassell's *An introduction to programming in Emacs Lisp*. But the main point of this chapter is something else. While any seasoned Emacs programmer knows most of the things used here by heart, a beginner does not, and the recurring theme in every section of this chapter is how one can learn about functions and variables we need without reading over thousand pages of the Elisp reference. We will learn how to use that manual to quickly find what we need, and also to use the various `apropos-...` and `describe-...` commands Emacs has to offer to find out how to do stuff in Elisp.

The first attempt will just use `transpose-lines`. Saying `M-x transpose-lines` shows that this is not what we want – that function swaps the current line with the one above. So, we need to move *the point* one line down before calling it. How to do that? Well, in normal, interactive use of Emacs, we would just press the down arrow or `C-n`. Let's check how Emacs does that by pressing `C-h k C-n`. It tells us that `C-n` (or `<down>`) runs the command `next-line`, but before presenting us with the documentation string (or *docstring* for short) of the command, it tells us that this function should

[6] https://github.com/emacsfodder/move-text

only be used interactively, and that if we want to use it in Elisp code, we should probably use `forward-line` instead.

Note that when Emacs tells us that you shouldn't use `next-line` in Elisp code, you should probably heed that advice – unless you really know what you're doing. Also, notice that this is a very useful piece of information. One thing you should always do when coding Elisp is asking Emacs about various functions, variables etc. – it really *is* very helpful with that. Also, it is a *very* good idea to install Emacs *with sources* on your machine. This way, when you look at the description of a function provided by one of the `describe-...` commands, you'll also see the name of the file it is defined in, and clicking on that name – or pressing `RET` with point on it – will take you directly to the place in Emacs sources where you can see and study the source code of any function. While we are at that, remember that you can move between "buttons" in the `*Help*` buffer with `<tab>` and `S-<tab>` (that is, "shift" and `<tab>`) way faster than with the mouse. In fact, the `*Help*` buffer is always set up in `help-mode`, which contains more useful commands and keybindings – to learn them, you can e.g. press `C-h m`, switch to the `*Help*` buffer that pops up and press `C-h m` again. That way, you will see the docstring of `help-mode`, together with lots of other useful information, including the key bindings defined by that mode. The most useful (besides the ones I already mentioned) are probably `SPC`, `DEL` (i.e., the "backspace" key), `l` and `r`. Also, you might want to check out the Helpful[7] Emacs package (available on MELPA[8]), which contains much more powerful alternatives to many built-in `describe-...` commands.

The first, naive approach to moving a line down

So, we are going to use `forward-line` as we are told.

```
1  (defun move-line-down ()
2    "Move the current line down."
3    (interactive)
4    (forward-line 1)
5    (transpose-lines 1))
```

Note that I could just say `(forward-line)`, since – while it takes one argument (the number of lines to go forward) – it is optional and defaults to one.

Unfortunately, this command does not work very well – it would be nicer if it left the point in the line being moved. That way, calling it more times would move the same line further down. That is easy, it is enough to say `(forward-line -1)` at the end.

[7] https://github.com/Wilfred/helpful
[8] https://melpa.org/

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (forward-line 1)
  (transpose-lines 1)
  (forward-line -1))
```

It is much better now – you can call it a few times and move a line further down. (This is especially nice if you decided to bind it to same key, e.g. `M-<down>`. You can use the `global-set-key` command for that.)

Preserving the column

Of course, there is always (or almost always) room for improvement. Sometimes it is important to know where to stop – coding little editing utilities like this is about making your life easier and saving time, and it won't save you much time if you spend several hours on one little function. Here, however, we are learning, so let's explore some of the possibilities.

One thing is that it would really be nice if the point did not move to the beginning of the line moved. (This is because how `forward-line` works.) It turns out that it is a bit tricky. Using `next-line` (the interactive one we were explicitly told not to use) does not help at all, since `transpose-lines` *also* moves point to the beginning (in fact, it calls `forward-line` at the end). One thing we could do would be to store the position of the point in the line in a temporary variable and move that many characters to the right after performing the moving. The question is, how do we know the position of the point in the line? One way to learn that would be to recall that there is a command which displays that – `C-x =`, or `what-cursor-position`. Let's look at its code – press `C-h k C-x =`, go to the `*Help*` buffer and press the button saying `simple.el` (either with the mouse or with `RET`). The source code for `what-cursor-position` is pretty long and complicated, but near the end there is a call to `message`, and it turns out that the "column" information is taken from the `col` variable. Now, it is enough to go there and type `C-r col` to start looking for its definition, and we can quickly see that it is assigned the value returned by the function `current-column` in a `let`.

The only thing we don't know yet is how to move to the column whose number we store. A naive version would be to call `forward-char` with the number, and it would probably work – but there is a better way. You can say `M-x apropos-function column RET` and Emacs will show you all functions (and macros, for that matter) whose names match the regex `column` – since this regex only contains non-special characters, this means just "contain the word «column»". (One tip is that if you get too many results this way – for instance, in my Emacs it gives more than 140 hits – you may want to repeat it in a fresh Emacs session, started with `emacs -Q`. This way you will only get the results from stock Emacs – no your customizations, no packages etc. In my Emacs it gave 31 hits, which is much more manageable to look through manually.) And indeed, while there is no `goto-column` function, there is `move-to-column` which does exactly what we want (go figure – there is `goto-line`, but `move-to-column` – Emacs function names are sometimes a mess...)

Another way to find the `move-to-column` function would be to assume that most probably, Emacs has a *command* to move to a given column, and look for it in the Emacs manual. Indeed, it is mentioned in the "Changing the Location of Point" section.

Yet another way would be to recall that `M-g M-g` moves to a *line* with the given number, and hope that the `M-g` prefix has something to move to a column – then, pressing `M-g C-h` shows all bindings starting with `M-g`, and indeed `M-g TAB` is what we are looking for.

Knowing that, we can code yet another version of `move-line-down`.

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (let ((position-in-line (current-column)))
    (forward-line 1)
    (transpose-lines 1)
    (forward-line -1)
    (move-to-column position-in-line)))
```

Moving by more lines

Let us now extend this function and make it accept a prefix argument to move the current line *that many* lines down. For that to work, we are going to add an argument to our command, and modify the `interactive` clause accordingly. Thing is, providing arguments to commands is tricky business – they may be numbers (given via `C-u`, like in the case of numerous commands accepting a *count*), filenames (like in `C-x C-f`), buffer names and many, many more. And, there is more than one way to teach a command to get the values of its arguments from the user. The simplest one, however, is to provide `interactive` with a special string, describing (using special codes) where to get those values from. Basically, for each argument you should provide a one-letter code describing its "nature", optionally followed by a prompt (for arguments whose values are read using the minibuffer). In addition, that string may have one or more special characters at the beginning, giving Emacs additional information about the command. See the manual node Defining Commands[9] for the whole (pretty long) story – for now, we need only two pieces of information. To use the *numeric prefix argument*, we need the `p` code. Note that this is lower case "p" – upper case variant means the *raw prefix argument*, which we will discuss later. When using the "numeric" variant, bare `C-u` is translated to 4 to the power of n (where n is the number of times `C-u` was pressed), no prefix argument is translated to 1, and an isolated minus sign is translated to -1 – and this is precisely what we need to interpret the prefix argument as a "count"-type parameter. Also, we are going to begin the string with an asterisk (`*`), which makes Emacs signal an error if the buffer the command was called in is read-only (which makes sense for commands that change the buffer contents).

Of course, all this is only half of the story – apart from properly *getting* the argument, we still need to *use* it. Checking the docstring for `transpose-lines` again we learn that it also accepts a numeric

[9] https://www.gnu.org/software/emacs/manual/html_node/elisp/Defining-Commands.html

argument (and in fact, its `interactive` form is exactly the same as we will use for our command), so we are lucky and can do this.

```
(defun move-line-down (count)
  "Move the current line down."
  (interactive "*p")
  (let ((position-in-line (current-column)))
    (forward-line 1)
    (transpose-lines count)
    (forward-line -1)
    (move-to-column position-in-line)))
```

Making `move-line-down` work with undo

And that is almost the end of the story, at least with this version of `move-line-down`. Unfortunately, it is still not perfect. One thing that bothers me a bit is that our function behaves very strangely when provided a numeric argument of zero. (Try it out. Can you see what happens?) Another is that undoing works, but leaves the point in a strange place. This is easy to fix. Going to the "Undo" section in the Elisp reference tells us about the `buffer-undo-list` variable, which is a list containining information needed to perform, well, undo. One possible value that can be put (or `push`ed, really) onto it is an integer, which means that undo should move point to that position.

```
(defun move-line-down (count)
  "Move the current line down."
  (interactive "*p")
  (let ((position-in-line (current-column)))
    (push (point) buffer-undo-list)
    (forward-line 1)
    (transpose-lines count)
    (forward-line -1)
    (move-to-column position-in-line)))
```

To be fair, `buffer-undo-list` is a pretty advanced concept, and putting it into the first chapter is perhaps a bit controversial – but having an actually useful function is what we really want here. There is a certain trade-off when extending Emacs – oftentimes, either the functions we write are simple, or they are designed to work well with various Emacs "subsystems". Emacs has accumulated *a lot* of stuff over the years, and trying to make your functions play well with *all* of them is pretty daunting. Don't even try to do that – code something that works for you, and if you encounter an unexpected interference between your code and some more obscure feature, decide if you want to (and can) support it. The undo feature, however, while a bit complicated from the coding perspective, is such a basic part of the Emacs experience that making your code play well with it is probably *always* worth the effort.

And now for a completely different approach

Now, instead of fixing the zero-argument behavior, let us step back a bit and reconsider our approach to this function. About half of the issues we had with this approach was that we were so intent on using `transpose-lines` – but in order to do it, we had to move point to a suitable line, then move it to a suitable column, mess around with the undo list, and even then the zero-argument behavior is not what we want. Why not implement everything from scratch instead of relying on `transpose-lines`, which neither does *exactly* what we need nor is a simple function to work with?

So, back to square one. How would we move a line down as human editors, working in Emacs? (This is actually a very important question. Many Emacs extensions can start as some simple code replicating the actual actions of a human user.) One way to do it would be to kill the current line, go down and yank it. We could use the approach of "just coding in Elisp the action a human would perform" and do exactly this, but in fact this is not a good approach – that way, our command would "pollute" the kill ring with the line we're at. Remember, when a user issues a command like `move-line-down`, they will most probably *not* think about it as "killing a line and yanking it below". If they try to yank something later, having the line moved earlier would be pretty surprising.

However, this is not a problem. If we go to the chapter "Text" of the Elisp reference – specifically, the "Deletion" section, we are going to learn about a function which is (in a sense) a programmatic equivalent of killing. `delete-and-extract-region` takes the region between two given positions, deletes it, and returns the string it deleted – all that without touching the kill ring.

Note: if you look at the "Buffer Contents" section, you'll learn about other functions which could be useful here, the `buffer-substring-*` ones. And this is the place where you could start thinking. Should we use `buffer-substring` here? Or maybe `buffer-substring-no-properties`, and what are *text properties* anyway? Or maybe we should heed the warning in the description of `filter-buffer-substring` ("Lisp code should use this function instead of `buffer-substring`, `buffer-substring-no-properties`, or `delete-and-extract-region` (...)")? Here is my (unofficial, since I'm not part of the Emacs developer team) advice. Do. Not. Care. If the simplest function (like `delete-and-extract-region` in this case) works for you, go for it. If it's the right choice, then great. If it's the wrong choice, one day it won't work for you, and then you will have a minor moment of frustration, a need for undo (which works great in Emacs) and possibly a dive into the docs or a question on the help-gnu-emacs[10] mailing list. And if you share your code with other people, maybe it won't work for someone else, so you'll get a bug report or feature request – the rest of the cycle is the same. In either case, your code should converge over time to a "good enough" state.

Don't get me wrong – I'm not in favor of sloppy coding. But Emacs is a complicated beast, and it may be the case that there are two functions which seem to do the same thing, and you read about the difference in the docs and you can't make that out because it talks about some gibberish like "text properties". But if you don't even know what "text properties" are, chance is that you don't care for them, so either function would be good for you. And if it turns out that you *do* care for them after all, only you didn't know the name, but your code doesn't work as you expected – now

[10] https://mail.gnu.org/mailman/listinfo/help-gnu-emacs

that's even better, since you'll learn something new along the way. (The alternative would be to stop, study what text properties are, think carefully if they are what you need and code accordingly. This is a great idea if you have time for it, which is not always the case…)

So, coming back. As we said, instead of using `buffer-substring` to get some portion of the buffer and store it *somewhere* temporarily (only not in the kill ring, which is a feature more for human users and not Elisp code – we'll get to the "somewhere" shortly) and then deleting that exact same portion with `delete-region` (which, by the way, you can also find in the "Deletion" section of the same chapter), we can use `delete-and-extract-region` which combines both. So, what we want to do now is basically this: find out where the beginning and end of the current line are, `delete-and-extract-region` the text between them, store it somewhere, move to the next line and insert it back there.

We can almost write the code now, except for one thing. How do we get the position of the beginning of the current line? The natural thing would be to look for functions containing the strings "begin" and "line". This is easy: just say `M-x apropos-function RET begin.*line RET`. (If you don't know regular expressions – and `begin.*line` is a regular expression, or regex – go learn some basics about them now, seriously. They are really cool and *extremely* useful.) If you use some clever completion engine instead of Emacs bare-bones completion mechanism (I use Ivy, there exist others, like – in alphabetical order – Helm, Icicles and Ido), you'll probably have a similar thing baked into `C-h f`, too. Anyway, you can see all sorts of functions which *go* (as in: move the point) to the beginning of the current line (for various ways of understanding the words "beginning" and "current line"), but no functions for *getting the position of the beginning of the current line*. (Well, there is also `bolp`, which is not easy to find in this way because of its cryptic name, but it still isn't it.) So, what do we do? Well, there are two more ways of finding such a function, both being rather a long shot, but they shouldn't take us more than a minute, so why not? One is looking at help for some *other* function, doing a "similar" thing – e.g., `point` (which gives us the *current* position, not the position of the beginning of the *current line* – but it seems somehow related). In fact, `C-h f point RET` pops up a window which says – among others – something like "Other relevant functions are documented in the buffer group", and the word "buffer" is blue and underlined. Going to it with `TAB` and pressing `RET` shows yet another buffer, in `shortdoc-mode` (seemingly a bit underused in Emacs), which contains short information about various related functions, together with examples and links to documentation. There are a few position-related functions there apart from `point`, and one of them is… `line-beginning-position`, which wxactly what we need!

The second possible way is (now obviously) to not only look for functions matching `begin.*line`, but also `line.*begin`. Changing the order of "keywords" when looking for built-in functions is another useful tip.

So, let's write the next version of `move-line-down`. While storing something temporarily when *using* Emacs usually means the kill ring (or perhaps registers), in Elisp *code* it usually means a local variable – in other words, `let` (or `let*`).

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (let ((current-line (delete-and-extract-region
                       (line-beginning-position)
                       (line-end-position))))
    (forward-line 1)
    (insert current-line)))
```

Well, if you try it out, you'll see that it doesn't work. Of course it can't – we moved the line without the end-of-line character, i.e., we extracted one character too few. There are many ways to remedy that, but the simplest one is probably just to add one to `(line-end-position)`. There is a built-in function to add one to something, `1+`, or we could use the normal addition function, `+`. And while we are at that, let's reinstate the code moving the point to the right column.

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (let ((position-in-line (current-column))
        (current-line (delete-and-extract-region
                       (line-beginning-position)
                       (1+ (line-end-position)))))
    (forward-line 1)
    (insert current-line)
    (move-to-column position-in-line)))
```

Well, still not good – the point jumped to a seemingly random location, one line too far. When you think about it, it becomes obvious, too – after `inserting current-line`, which now contains the end-of-line character, the point is effectively in the *next* line and not in the one inserted. We could say `(forward-line -1)` to remedy that, but we can also tell Emacs to get back where we were after inserting `current-line`. In fact, "do something and get back where you were" is such a common thing to do when coding in Elisp that there is a special idiom for that: `(save-excursion ...)`. We will encounter it many times later, so let's just recall that it remembers the current point position and the current buffer, evaluates the code denoted by `...` above, and restores the buffer and point. This means that wrapping `(insert current-line)` in `(save-excursion ...)` is enough. (Note: sometimes you may want to restore more things, like e.g. the value of *mark*, from before your code was evaluated – there are other `save-...` constructs, like `save-mark-and-excursion`, `save-match-data`, `save-restriction` and a few more. They seem to be less frequently used, but they are useful, too. A nice exercise now would be to find out about all of them. The fact that they are named similarly to commands for actually *saving buffer to the disk* is not helpful, but there aren't *that* many of them anyway.)

And yet another approach

However, there is another way. Instead of remembering in what column the point was (and the point's position is obviously lost when the current line is deleted), we can flip the whole idea and delete the *next* line and reinsert it above. That way, we won't even have to move the point at all.

```
(defun move-line-down ()
  "Move the current line down."
  (interactive)
  (save-excursion
    (forward-line 1)
    (let ((region-to-move-up
           (delete-and-extract-region
            (point)
            (save-excursion
              (forward-line 1)
              (point)))))
      (forward-line -1)
      (insert region-to-move-up))))
```

And this seems the best idea we had so far – alas, it contains a serious bug. Can you spot it? I couldn't, I only discovered it by experimenting, and even then it was not obvious for me what was happening… This is a common theme in programming – it is often just complicated enough we humans have problems wrapping our heads around it. Here is the problem: it doesn't work when the point is *at the beginning of the line* when our command is invoked. And after a while it becomes fairly obvious why is that so – in such a case, we insert the *next* line basically *at point* (by which I mean at the position of the point before the command was invoked), so `save-excursion` is fooled and (more or less) thinks that instead of inserting *before* the current line, we are *prepending* to it, and lefts the point (as instructed) at the beginning of the current line – but now it thinks that the "current line" is two lines, the one that used to be the next one and the proper "current line".

There are a few ways to mitigate this, but the best one I found is this. Instead of `insert`, we can use `insert-before-markers`. This is a function which is probably much less known and much less often used than `insert`. It does exactly what it says on the tin – it inserts the text(s) given at point (like `insert`), but if some *marker* (not only point, but also e.g. mark and other remembered positions, including everything on the mark ring – you can read about markers in the Elisp Reference, we will not deal with them in this book) points at the very same position, it is moved along with point (unlike what `insert` does).

A legitimate question you may ask now is *how am I supposed to know* about `insert-before-markers`? The short answer is, well, you aren't, at least not necessarily *that* easily. While Emacs *is* very well-documented, not all functions are readily discoverable. You *could* say `M-x apropos-function RET insert RET` (about a hundred hits!), or even `M-x apropos-function`

`RET ^insert RET` (about 35 hits), but for that you should at least *suspect* that such a function exists. Probably a better idea (which I admit I didn't try, and now that I know Elisp pretty well it's too late for me;-)) is this: if you have spare 30 minutes, go through the Elisp reference, visit every chapter and read one or two paragraphs at the beginning. This should give you a rough idea about what's in there in general. If you then need something related to e.g. markers, you can go to the "Markers" chapter and do a similar exercise with its sections. You will then learn about "marker insertion types", and `insert-before-markers` is also mentioned there. (Of course, who knew it is there you should be looking at? Well, I said Emacs *is* complicated…) Another way to find out about that function is to study the manual section "Excursions", which talks about `save-excursion` and some related constructs. At the end of that section there is a warning about precisely the situation we ran into, with a link to the section about marker insertion types – which then gives the hint about `insert-before-markers`.

I guess that the moral of the story is that if some Elisp construct does not work as you expect (or as you wanted it to work!), you should go to the Elisp reference and read about it – chances are that the relevant section somehow deals with the issue you're having. And you can ask on the mailing list, too.

Changing `insert` to `insert-before-markers` is easy, so let's up the ante a bit and add three more things. First of all, we will reinstate the `undo` fix as we did previously. Then, we will generalize our function to moving by more than one line down (again). Finally, let us make it work nicer with code. When moving a line of code down, it may happen that it will end up on a different level in the syntax tree – it might land in a then-branch of a conditional, or fall out of a loop etc. This means that we might want to reindent it to match its new surroundings. This sounds pretty complicated – if we were to do it by hand, it could depend on the programming language used, and even individual style of a person using it. However, Emacs (obviously) has good support for both these things (at least in theory – in practice the support quality varies from language to language…), namely major modes (to work with different languages) and individual customizations (to align with the user's personal style). The only thing we need to know is what function indents the current line according to those.

In my Emacs, `M-x apropos-function RET indent RET` gives a whopping 255 hits (no kidding!). If I start it with `emacs -Q`, the number is down to 74 – still quite a lot. Instead of going through all of these and try to find the suitable one, let's go to the manual. Happily, the table of contents for the Elisp reference has something called "the detailed node listing", which apparently contains links to *all* the individual nodes of the whole book. This means that we might find what we need with the good ol' `C-s` – though probably a better way would be to use `m` (for `Info-menu`). Either way, we should arrive to the node called "Mode-Specific Indent" (possibly via the node "Auto-Indentation"), where we can find the function (actually, command) `indent-according-to-mode`.

Another tip is to use the index – when in an Info buffer with the Elisp reference, press `i` and type e.g. `indentation`. The command `Info-index` is smart enough to tell you if the entry you selected points to more than one place in the index, and if so, it gives you a nice tip: pressing `,` (the comma) moves you to the next one. Even better, there is the `Info-virtual-index` command (bound to capital `I`), which constructs a "virtual node" with a menu of all the places the given index entry points to. Either way, the first index entry for "indentation" is the "Indentation" section, which contains "Mode-specific

indent" as its second subsection.

And here is the code.

```
(defun move-line-down (count)
  "Move the current line down."
  (interactive "*p")
  (save-excursion
    (push (point) buffer-undo-list)
    (forward-line 1)
    (let ((region-to-move-up
           (delete-and-extract-region
            (point)
            (save-excursion
              (forward-line count)
              (point)))))
      (forward-line -1)
      (insert-before-markers region-to-move-up)
      (indent-according-to-mode))))
```

Supporting different prefix arguments

Works great, right? Well, not necessarily. Giving a numerical argument works fine *if the argument is positive*. If the argument is negative, moving the line works ok, but the point ends up in the wrong place.

This is easy to fix – instead of `(forward-line -1)`, we may need `(forward-line 1)` for negative arguments – but before we implement that, let us stop for a moment and discuss prefix arguments. If we write a command and decide that it should support them (and we should *always* consider supporting them if that makes sense!), it is a very good idea to consider a few cases.

The simplest case is using the prefix argument as a flag – then, `M-x our-command` does one thing, but `C-u M-x our-command` does something a bit different. For instance, one of the most important commands every Emacs user needs to learn at some point (usually at the very beginning!), `save-buffers-kill-terminal` (bound to `C-x C-c` by default) asks if it should save unsaved files, but does it without asking if prefixed by `C-u` (or, actually, *any other prefix argument*). This is very easy to support – you can just define your command to have a parameter and declare that it is going to use the "raw prefix argument" for that: `(interactive "P")` (note: that is an upper-case `P`). This means that it is nil if there was no prefix argument and some other value otherwise. (You can check the "Prefix Command Arguments" node in the Elisp reference for details about the possible "other values".)

A slightly more complicated case is when the prefix argument should be treated as an integer. This is usually the case when we want to treat it as a "count", making our command perform some action

more than one time if prefixed with an integer. In that case, say `(interactive "p")` (note: this is a lower-case `p`). Then – as we already know – if no prefix argument was supplied, the argument will get the value of one, if the prefix argument was `C-u -` (i.e., just a minus sign), it will be minus one, if it was `C-u` repeated `N` times, it will be four to the power of `N`, and in all other cases it will be the provided numeric argument.

The most involved case is when we want to actually distinguish between e.g. `C-u 4` and plain `C-u`, or between `C-u 16` and `C-u C-u`, or between `C-u` and `C-u C-u` etc. In that case we can use `(interactive "P")` again, but look carefully at what the argument value is. It can be nil (no argument), an integer (a numeric prefix argument), the symbol `-` (remember to write `'-` when you mean the symbol itself and not the value of a *variable* named "-"!) for `C-u -`, or a list like `(4)`, `(16)`, `(64)` etc. for one, two, three etc. `C-u`'s.

In our case, we need just a count (as we did previously), so `(interactive "p")` is just fine – and we are going to use the asterisk again so that the command is not even run in read-only buffers.

```
(defun move-line-down (count)
  "Move the current line down."
  (interactive "*p")
  (unless (zerop count)
    (save-excursion
      (push (point) buffer-undo-list)
      (forward-line (if (> count 0) 1 0))
      (let ((region-to-move-up
             (delete-and-extract-region
              (point)
              (save-excursion
                (forward-line count)
                (point)))))
        (forward-line (if (> count 0) -1 1))
        (insert-before-markers region-to-move-up)
        (indent-according-to-mode)))))

(defun move-line-up (count)
  "Move the current line up."
  (interactive "*p")
  (move-line-down (- count)))
```

Note that we check if `count` is equal to zero, which is kind of strange – our function does not make too much sense then – but the user *can* say `C-u 0 M-x move-line-down RET`, and we *should* do something reasonable then – like not changing anything. The `zerop` function just tests if its argument is equal to zero, and `unless` is more or less a shorthand for `(if (not ...) (progn ...))` – in other words, it evaluates the forms inside if the given condition is nil. (There is also `when`, which works the same but checks for non-nil-ness of the condition. We will meet it later.) Also, since we decided to support

negative arguments, it is useful to add another function, `move-line-up`, so that both can be bound to some keys easily.

Summary

In this chapter, we learned a whole lot of things. Probably the most important lesson, however, is how to learn about Emacs functions to perform various practical tasks.

Elisp, at its heart, is a very simple language (it is a Lisp, after all!). It *does* have advanced features and dark corners, but those are outside the scope of this book (and you can go a *long* way without ever looking at them). The most intimidating part when learning to code actually useful tools is learning the "library" – in other words, knowing what functions are available. Luckily, Emacs is more than happy to tell us that – we just have to ask nicely;-). The places to look are:

- the docstrings of commands you use to perform a similar task by hand (displayed with `C-h k` or `C-h f`),
- links in those docstrings (sometimes leading to other, similar functions and sometimes to a `shortdoc` buffers),
- output of commands like `M-x apropos-function` with a regex constructed from some keywords (this sometimes requires a bit of luck, since the naming of Emacs library functions can be a bit wobbly at times, and it is worth trying them out in various order), and
- Elisp reference, especially its detailed node list and index (for quick access of those, it's worth remembering about the `m`, `i` and `I` keys).

Reordering parts of a sentence

Introduction

Our second utility will be something useful for people who edit texts in a natural langauge. (I use Emacs to write emails, my blog, this book and other stuff, so I happen to do it a lot of the time. If you use a word processor like LibreOffice Writer to satisfy such a need, I strongly encourage you to consider Org-mode or the AUCTeX+LaTeX combo for a similar purpose.) One of the very common errors is having the *order* of things wrong – letters (like tihs) or words (there is no such thing[11] as a "wooden blue British box", only a "blue British wooden box", for instance). Stock Emacs has basic support for fixing those – you can press `C-t` (`transpose-chars`) with point between characters to swap them, and `M-t` (`transpose-words`) with point between words to swap them (well, both are in fact more complicated, but let's leave it at that). So, the former of our errors could be fixed by placing the point where the asterisk is: `ti*hs` and pressing `C-t`, and the latter with placing the point between "wooden" and "blue" and pressing `M-t` twice (or `M-2 M-t`). However, what if we want to fix a "wooden British blue box", and we are not interested in acrobatics involving setting the mark and calling `transpose-words` with a prefix argument of zero (see its docstring to learn what it does)? Well, read on to learn.

This chapter is quite long, and some people might find large parts of it boring because they can see no use for tools we use. If so, I can only say that I'm sorry – but please give it a try anyway. The reason it is so long is that I wanted to show a fairly feature-complete Emacs tool written from scratch, and the idea of the `reorder-sentence` command (and all other commands and functions working with it) enabled me to introduce *a lot* of concepts you might find useful – non-modal design typical to Emacs, writing a minor mode, employing `save-excursion` and Emacs movement functions to find places in the buffer, using other buffers, global variables, overlays and many more. Some of them I will only mention, some I will cover in more depth, but you should at least have a good starting point to learn more about every of them if a need arises.

Modal versus non-modal design

Before we write even a single line of code, let us stop and think for a moment about the *design* of our utility. For people coming from the outside of the Emacs world, perhaps a natural choice would be for the `reorder-sentence` command to display a temporary buffer, then (somehow) wait for the user to select the first, second and the following words, constructing the reordered sentence in that temporary buffer, and finally replace the original with the reconstructed one – not letting the user do anything else while doing that. This makes sense, but is not very Emacs-y. Whenever you want

[11] https://youtu.be/mTm1tJYr5_M?t=48

to write a command which performs a kind of "dialog" with the user, repeatedly waiting for them to perform some action and responding to it in a suitable way, it is probably best to reuse the tool Emacs has precisely for that: the main command loop. In other words, instead of having one large command using functions like `read-char` (which blocks Emacs from doing anything else, waits for the user to press some key and returns that key), it is much better to have some command to *initiate* the whole process (e.g., set up the buffer/window pair), then have one or more commands to perform all the possible user actions, and then dedicate one more command to finish the whole process (e.g., kill the temporary buffer). This way the user will be able to e.g. pause the whole process, go to another window, do something else, and get back. Sometimes you won't even need a command to "cancel" the whole thing – the user can just stop doing that and not finish. More importantly, it will not block things Emacs does in the background, like idle timers.

Another (minor) issue we might want to consider is how we *name* the buffer in which the sentence will be reordered. There are two main considerations here. One is whether the name will be *constant* or generated at runtime. The former is slightly simpler, but the latter makes it possible to work on reordering *two* sentences (possibly in different buffers!) at the same time. We will go for simplicity, and we will only allow one "reordering session" at a time. (I don't think anyone would *want* or *need* to pause reordering one sentence and do it with another one – at least *my* brain probably couldn't handle this. On the other hand, learning to support something like this might be useful, so we will briefly discuss how this could be done.) The other issue we need to decide on is if that buffer should be visible on the buffer list. If we begin a buffer's name with a space, it is an "ephemeral" buffer. Such buffers are not suggested for autocompletion when `switch-buffer` is used, they do not record undo information by default, and perhaps are treated specially in some other ways. One cold argue that we do not want the user to mess up with the buffer in which we reorder the sentence – but on the other hand, why would we want to forbid it, especially that it may be the case that the reordering involves some trickier manipulations our code won't support. In any case, when in doubt, I'd go with "letting the user do whatever they want" policy, so we will call that buffer `*Reorder sentence*` (without a space).

We will start with the set-up and tear-down commands, in their simplest possible form. Then, we are going to make things more and more complicated (for us) – and more and more straightforward (for the user).

The set-up command

In its simplest form, the set-up command needs to do four things – store the information about the position of the region whose words we want to reorder (so that when we finish the reordering, Emacs will know where the sentence we started with was – it will then need to delete it and replace with the new one!), remember the window configuration so that it can be restored later, set up a buffer to construct the reordered sentence and display it in some window. Do not underestimate the importance of the second step – we are going to show some buffer in some window, and it may be the case that the user has spent some time configuring the windows to display whatever they need. Breaking that would be *very* rude.

Let's begin with storing the information about the region being reordered. We will use global variables holding all the data we need. (If we decided to have a separate "reordering sessions" for various buffers, we could use so-called "buffer local variables". They are exactly what they say on the tin – variables which can keep different information in different buffers, but are otherwise like global variables, that is, they are not tied to any function or `let`-like expression. A classical example of such a variable is `major-mode`, which holds the current buffer's major mode, and – obviously – varies by buffer.)

```
(defvar reorder-sentence--begin nil
  "The beginning of the sentence to be reordered.")

(defvar reorder-sentence--end nil
  "The end of the sentence to be reordered.")

(defconst reorder-sentence--buffer-name "*Reorder sentence*"
  "The name of the buffer used to construct a reordered sentence.")

(defvar reorder-sentence--buffer nil
  "The buffer used to construct a reordered sentence.")

(defvar reorder-sentence--previous-window-configuration nil
  "The window configuration from before reordering.")
```

Note that when I declared variables with `defvar` (and it is a very good practice to declare every global variable you are going to use!), I provided them with docstrings, like functions. I tend to do that even for ones only used by my code and not supposed to be set by the user – after all, someone might want to study or even modify my code, and documenting it this way may help them.

Also, we have used `defconst` here. It is (almost) a synonym of `defvar` – nothing prevents Elisp code from changing the value of a variable defined with `defconst`. The "const" here is just a signal to the programmer that this *should* not be changed. (Also, contrary to what `defvar` does, `defconst` actually *requires* a value, which is quite reasonable when you think of it! There are also other, rather subtle differences we are not interested in.)

I also took care to *name* the variables according to a custom where every function, command or variable in some package gets prefixed with the name of that very package, and if the variable or function is an internal one (i.e., not meant for the user to tamper with), it gets *two* dashes between the prefix and the actual name. This is not *enforced* in any way (also, the user *still* has access to these functions and variables, of course), but the convention is to use those two dashes to signal "don't touch this, this is just used in the current implementation, but it can change without a warning in the next version" etc.

Having defined the places to store the information we need, let us write the set-up code.

An important thing we'll need is how to pass the *region* to the command. There are many ways to do it, one of them being the `r` code for the `interactive` clause. It is a bit exceptional in that it handles

two arguments and not one, and it causes Emacs to assign two integers to them: the positions of the beginning and of the end of the region. (It is not my favorite way of handling the region in interactive commands, but it'll do for now.)

There is also a small catch here. By default, the beginning and end of the region are *integers* – positions in the buffer (this is what `(interactive "r")` puts in the arguments, and this is also what the functions `region-beginning` and `region-end` return). However, storing these positions as integers may be a bit brittle. If we did this at the beginning of some function and then used them at the end, it's ok – the user most probably has no chance of messing up with the contents of the buffer in the meantime. (How would they pause evaluating a function and resume it later anyway? Well, Emacs *has* a way to do exactly that, but then the function must use a special provision like `recursive-edit`.) But here we will store them in one function and use later in another, and all sorts of things may happen in the meantime. While we won't *encourage* the user to edit the buffer while the sentence is being reordered, we won't *forbid* it, either. (Well, in a sense we will, since we will later see that the buffer will be marked as read-only for the time of reordering the sentence – but the user can change that at any time!) If anything is inserted or deleted *before* these positions in the buffer, they will start pointing at wrong places. One simple way to deal with that is to use *markers*, which store positions in the buffer, and if some text in inserted or deleted *before* a marker, the position it points to moves accordingly. (The way to "convert" a numerical position to a marker is the `copy-marker` function. It can also accept a marker and then creates a new one in the same place, which may be useful, since then these two marker are independent and the positions they point to may diverge in the future.)

This is not the whole story, though. As I mentioned in the first chapter, inserting things precisely *at* the marker may behave in two ways. If the user inserts something exactly at the `reorder-sentence--begin` marker, the marker should move after the inserted text, so that when the whole reordering thing is finished, the newly inserted text is not deleted together with the "old" version of the reordered region. In other words, the marker at the beginning should have the (non-default) insertion type of `t` (which means exactly that text inserted at the marker will move it forward).

At this point, the reader might start thinking "this whole Elisp thing is really crazy, I just want to record the position in the buffer, and that madman has spent two paragraphs of text talking about three ways to do that, two of which are apparently wrong in this context!" Well, this is a fair point. I warned that over time, Emacs grew to a complicated beast! But do not worry. If *you* were to code all this yourself, and you didn't think about all these things and just used numeric positions (or markers of the wrong type), most probably nothing bad would happen. After all, in a typical case hardly anyone would turn off the read-only mode while reordering the sentence and insert something before the region reordered, and even fewer people would insert some text precisely at its beginning. And even if someone did that, and the (slightly) faulty code did the wrong thing, they would simply undo and (maybe) report a bug – which would then be fairly easy to fix (since it would be pretty obvious what happened). So the real reason I'm spending so much time talking about all this is that markers (and their types) may happen to be even more useful in *other* contexts, and it's a nice opportunity to mention them here.

Someone might also ask why the `r interactive` code assigns numerical positions and not markers to the respective parameters. Well, I don't know, since it wasn't me who coded that, but I suspect the reason is that it would be unnecessary in most cases – usually, after the command operating on the region finishes its work, the values of the beginning and end of the region are no longer needed, and hence the user has no way of messing around with the buffer contents and interfering with its work. Also, if markers were used in this context, Emacs would have to allocate memory for them and free that memory when they are no longer needed – this happens automatically, but the so-called "garbage collection" (i.e., freeing memory that is no longer useful) is a time-consuming process which may even slow down Emacs noticeably in extreme cases (e.g., in the case of a sloppy written loop which involves allocating and freeing lots of memory many times), so it seems a good practice to use integers instead of markers whenever feasible.

Coming back to our code: yet another remark is that we don't need to create a new buffer *if* the sentence reordering buffer already exists – in that case, we are just going to reuse the existing one. The function `get-buffer-create` does exactly that. One note about that function is that it accepts an optional second argument, and passing a non-nil value to it may be a good idea for – as the docstring says – "internal or temporary buffers". (In our case, this doesn't make much difference – read said docstring to learn the details.)

By the way, if we wanted to have many simultaneous reordering sessions, we might want to use the neat `generate-new-buffer` function. It creates a buffer using the given string as the basis of its name, but if such a buffer already exists, it appends a suffix to make one that doesn't.

Last but not least, we should definitely deactivate the region after we do all the housekeeping – this is usually what the user expects from commands which "do something with the region".

```
(defun reorder-sentence (beg end)
  "Reorder the words in the region."
  (interactive "*r")
  (setq reorder-sentence--begin (copy-marker beg t)
        reorder-sentence--end (copy-marker end)
        reorder-sentence--previous-window-configuration (current-window-configuration)
        reorder-sentence--buffer (get-buffer-create reorder-sentence--buffer-name t))
  (with-current-buffer reorder-sentence--buffer
    (erase-buffer))
  (display-buffer reorder-sentence--buffer)
  (deactivate-mark))
```

We used the `deactivate-mark` function here, which does exactly what you would expect. As a side note, let me mention that there is also a *variable* called `deactivate-mark`. When a command exits (and Emacs returns to what is called "the main command loop", i.e., it awaits user input), the mark is deactivated if this variable is non-nil. A bit unexpectedly, its main use is to *disable* mark deactivation in your custom commands. The idea is that the built-in Emacs functions which modify the buffer set it to `t`, and if you use such a function in your command, the behavior I've just described will

make the mark inactive. If you surround these functions with `(let (deactivate-mark) ...)`, they will set the *local* version of the variable to `t`, and after `let` exists, the previous value (which is nil when the command starts) is restored – and hence the mark will *not* get deactivated.

Constructing the reordered sentence

Now that we have the buffer where we want to construct the reordered sentence, let us do the reordering itself. We will write a command which will copy the word the point is on to the "reorder sentence" buffer we created previously.

In order to do that, we need a few things. First of all, we need a way to *get* the "current word", i.e., the word the point is on. Then, we need to insert it into the proper buffer, preceding it with a space if necessary. Next thing, we need to decide how to treat *punctuation* (actually, we will save that for later).

So, let's get started. The first thing that comes to mind when approaching the problem of "getting the current word" is to use the `mark-word` function. We can do it, although it has some drawbacks. First of all, this function's name is a bit misleading: it does not really mark the "current word", but rather everything from the point to the end of the current word. In other words, we'd need to back up first to the beginning of the word to make it work. What's worse, "backing up" is not easy – again, the most natural thing would be to use `backward-word`, but if we *are* already at the beginning of a word, it moves to the beginning of the *preceding* one. (We could first use `forward-word`, store the point position, then call `backward-word`, and store the point position again to get the "boundaries" of the current word.) Then, it changes both the point *and* the mark, which is not necessarily the thing we want in Elisp code – mark is a user-level feature Elisp code should not tamper with unless really needed. (This could be remedied with the `save-mark-and-excursion` macro, though, which acts like `save-excursion`, but restores also the previous mark position.)

It turns out that there *is* an Emacs function which just returns the word the point is on. If you say `M-x apropos-function RET word RET` (in a fresh `emacs -Q` session), you get close to a hundred results, and near the top you can find `current-word`. It has two optional arguments which we will not need, and if you look at its code, you will see that indeed it does not use the mark (in fact, its code is very simple if we disregard the parts responsible for some edge cases).

As a side note, Emacs has a built-in package called `thingatpt` (short for "thing at point"), which has functions returning various possible "things" the point is on – words, sentences, urls, emails etc. Interestingly, `word-at-point` from that package is a lot more complicated if you drill down the source code. Also, it is not a simple equivalent of `current-word` and can actually yield different results in certain circumstances. We will hence use the simpler `current-word`. The `thingatpt` package will turn out to be quite useful for us later, though.

Coming back – we now need a way to copy the current word to the buffer we've created. This is actually easy – all we need to do is to get the current word and insert it there, remembering about checking whether a space needs to be inserted.

```
(defun reorder-sentence-copy-word-at-point ()
  "Copy the word at point to the sentence reordering buffer."
  (interactive)
  (let ((word (current-word)))
    (with-current-buffer reorder-sentence--buffer
      (goto-char (point-max))
      (when (not (bobp))
        (insert " "))
      (insert word))))
```

One thing that is worth mentioning here is the `when` form. (It is actually a macro, but it could be defined as a special form, much like `if` or `cond`.) It is basically an `if` without the `else` condition. In this case, we could replace `when` with `if` and nothing would change, but there are situations when `when` is actually better than `if`. Assume that we want to evaluate more than one expression if some condition is true (non-nil) and do nothing if it is nil. With `if`, we have to say

```
(if condition
    (progn
      (do-this)
      (and-that)))
```

The `progn` thing is a special form enabling us to put more than one expression where Emacs expects only one. If we omitted it and said just `(if condition (do-this) (and-that))`, we would change the logic – `do-this` would be evaluated if `condition` is non-nil and `and-that` otherwise. However, with `when` there is no `else`-part, so we can say just

```
(when condition
  (do-this)
  (and-that))
```

which is more convenient.

Well, this is how `when` works. And what does it do for us here? Quite obviously, it checks if we should insert a space before the new word. We should do it unless we are at the beginning of the buffer (no need for a space before the first word!). The functions `bobp` and `eobp` check if we are on the beginning or end of the current buffer. (There are also `bolp` and `eolp`, checking if we are on the beginning or end of the current line.)

Another thing which definitely warrants an explanation is the `(goto-char (point-max))` part. The bad news is that the above function doesn't work without it. The reasons are *very* subtle, and aparently not covered well in the docs. It turns out that not only does every *buffer* record the value of point, but so does every *window*. This makes perfect sense, since having two different places of *the same* buffer displayed in two different windows is a perfectly legitimate use case (and in fact, I

rely on it very often), but it makes the interaction between the "buffer point" and "window point" complicated. Consult the node "Window Point" in the Elisp reference for more details (see also the thread on the Emacs mailing list[12] where I asked about this exact code).

As you can see, I went for a simple solution and called `(goto-char (point-max))` before inserting anything. This worked – sort of – but had one problem: the point in the window displaying the `*Reorder sentence*` buffer was at the beginning all the time. This is not a problem for now, but it is going to be later – and when that time comes, we will implement a better way.

One moral of this story is that sometimes Emacs may behave in a way you can't understand and then it may be a good idea to ask the experts about it. But an even more important moral is that making your functions general is a good idea *in general*, but if it comes at a cost of making them overcomplicated, it is often ok to choose simplicity and not cover 0.01% of the cases.

Bringing the reordered sentence back

What we need now is a way to clean up after ourselves and bring the reordered sentence back to the buffer we started with. This is easy – we will just delete the portion that was selected when we called `reorder-sentence` (that is why we remembered the beginning and end of the region!), insert the whole contents of the temporary buffer there and finally restore the window configuration from before the changes.

```
(defun reorder-sentence-finish ()
  "Finish the reordering of the sentence.
Replace the selected region with the constructed sentence and restore
the window configuration."
  (interactive)
  (goto-char reorder-sentence--begin)
  (delete-region reorder-sentence--begin reorder-sentence--end)
  (insert-buffer-substring reorder-sentence--buffer)
  (set-window-configuration reorder-sentence--previous-window-configuration))
```

Go check this out – it works now! Well, sort of – it still has some issues. One is (obviously) the usability – with all hopping around with the point and saying `M-x reorder-sentence-copy-word-at-point` it is not really *that* convenient. We will fix this shortly. Another problem is that while it deals nicely with *words*, it completely disregards *punctuation*. Also, if you need to swap the *first* word of the sentence to something else, the *capitalization* gets wrong, too.

Before we start fixing all that stuff, let us do one more thing. It may happen that the user starts reordering the sentence and then changes their mind, so we need a way to *cancel* a started reordering session. This is very easy to do.

[12] https://lists.gnu.org/archive/html/help-gnu-emacs/2021-04/msg00003.html

```
(defun reorder-sentence-cancel ()
  "Cancel the reordering of the sentence."
  (interactive)
  (set-window-configuration
   reorder-sentence--previous-window-configuration))
```

Note that this function is pretty minimal – we do not kill the buffer keeping the sentence reordered, for instance. (We do not kill it in the `reorder-sentence-finish` command, either, for that matter.) That's ok – it doesn't hurt to have it open, it will be cleared when we start another reordering session, and it probably doesn't take too much memory (unless we do a reorder on a multi-megabyte region, which I think nobody should do anyway). Feel free to add a `(kill-buffer reorder-sentence--buffer)` if a useless buffer hanging around bothers you. (Beware that `kill-buffer` runs some hooks before the buffer gets actually killed. "Hooks" are basically user-defined actions that may be run at various points in time. Technically, a hook – in most cases – is a variable holding a list of functions to be run when something happens. For instance, `kill-buffer` runs the hook `kill-buffer-hook`, which may contain various clean-up functions. In our case, this hook would *not* be run, since we added `t` as the second argument to `get-buffer-create`.)

We will soon implement the simplest improvement – making the "current sentence" be the default if the region is not active. But first things first – our code so far is buggy. Can you spot the issue?

Fixing a bug with region boundaries inside words

Well, in normal circumstances the region will span either a complete sentence (we will make that the default in the next section) or just several words. But what would happen if the region beginning (or end) would be in the middle of a word? Of course, our code *should* not be used that way. But it *can*, so why not make it behave reasonably then?

Of course, the issue here is that we use `current-word`, which knows nothing about the region or mark – it just gets the word the point is on. In fact, since it relies on the *point* having been moved to the word it should operate on, it even no longer has access to the "original" region. We could code our own version of that function, accepting some `begin` and `end` parameters, but there is a much simpler way. Emacs has the tremendously useful *narrowing* feature – useful both for the human user *and* for Elisp code. With it, we can temporarily "focus Emacs' attention" to a portion of the buffer so that it "doesn't see" anything else. From a user's perspective, this is usually done with the `narrow-to-region` and `widen` commands. However, Elisp code should avoid using `widen`, since it might interfere with the user's manual narrowing. (For instance, the user might narrow down a buffer with a book to just one section, and then Elisp code might narrow further down to a sentence. Calling `widen` then would make Emacs forget the narrowing to the section.) Happily, Elisp has the `save-restriction` special form, which lets the code do whatever narrowing or widening it needs and restores the previous narrowing (or the lack of it). So, here is a better version of `reorder-sentence-copy-word-at-point`.

```
(defun reorder-sentence-copy-word-at-point ()
  "Copy the word at point to the sentence reordering buffer."
  (interactive)
  (save-restriction
    (narrow-to-region reorder-sentence--begin reorder-sentence--end)
    (let ((word (current-word)))
      (with-current-buffer reorder-sentence--buffer
        (goto-char (point-max))
        (when (not (bobp))
          (insert " "))
        (insert word)))))
```

This way, even if the user defines the region to start in the middle of a word, only the portion of the word in the region will take part in the reordering.

Defaulting to the current sentence

Quite a few Emacs commands work on the current region *if there is one* (and if it is active) and on something else if not. This is very useful – for instance, when reordering words for stylistic reasons, the natural unit would be the *sentence*, so why not use that if there is no active region?

The downside is that we can't use the handy `r` descriptor in our `interactive` call. (Well, technically we *could*, in a very convoluted way. We could write a command which just selects the current sentence if the region is inactive and *then* call the command we have already written, simulating an interactive call so that Emacs actually uses the `interactive` spec. This can be done with the `call-interactively` function. It is definitely not worth the effort, though.) Instead, we will provide `interactive` with an *expression*. In such a case this expression is evaluated and expected to provide the list containing all the arguments for the command.

In order to check if the region is active, we will use the `use-region-p` function. It is almost the same as `region-active-p`, which is almost the same as just looking at the `mark-active` variable, except these functions cover some edge cases. And by the way, you might want to know that the `-p` at the end of the name of the `region-active-p` function is a shorthand for "predicate" (which means "a function returning a true or false value" – of course, Elisp does not have a Boolean type, so this boils down to "non-nil – usually `t` – versus nil"). For the sake of completeness: the "p" added at the end of the function name is just a convention, Elisp does not rely on it in any way, and it is usually spelled without a dash if the function name is just one word – so we have e.g. `buffer-modified-p` but `stringp`. Of course, you may be wondering how you are supposed to *know* to use `use-region-p` and not `mark-active` or `region-active-p` – the answer being (of course) you can find it in the Elisp reference (it is even mentioned in two sections – the one about the mark and the one about the region). You can also say `M-x apropos-function RET region RET`, and look through several screens' worth of functions.

We also need a way to get the "current sentence", i.e., the sentence the point is in. This is surprisingly tricky. We could use the `sentence-at-point` function from the `thingatpt` package we mentioned earlier, but it only gives the actual *text* of the sentence, and we also need the *positions* of its beginning and ending. We could use the functions `forward-sentence` and `backward-sentence` (all within `save-excursion`) for that, but what if the point just happens to be at the very end of a sentence? Should we get the *preceding* or the *following* sentence then? I would incline to the preceding one, and here is the reason. One quite natural use-case of the `reorder-sentence` command is this: imagine you have just written a sentence, and you are not happy with it, so you just say `M-x reorder-sentence` and there you go.

So, for that to work, here is what we are going to do. First we go `backward-sentence` (so that if we are mid-sentence, we just go to its beginning, and if not, we go to the beginning of the preceding one), store the point value, then go `forward-sentence` and the point position is then the end of the sentence.

When we think of this, it's not even necessary to have `beg` and `end` as the parameters for our command now – after all, it can just use `region-beginning` and `region-end` if `use-region-p` evaluates to non-nil and the trick with the current sentence bounds otherwise. We will, however, leave these arguments in place, and here is why this is a good practice. It is perfectly imaginable that someone would like to write a function like `reorder-whatever`, working in a similar way but on some other unit of text – for instance, a *string* or a *comment* in some source code, or a *part* of a sentence between commas, or maybe something else entirely. Then, that someone could just write a command which (a) determines the starting and ending positions of the fragment to reorder and (b) calls `reorder-sentence` from its code giving these positions as arguments, without activating the region first (which would not be a good practice – Lisp code should only do things like activating the region or setting the mark *if* the user expects that to happen when the command finishes execution).

In order not to make `reorder-sentence` too long, let us factor out the "current sentence" code to an auxiliary function. Here is the first attempt. (Notice that however tempting it would be to call it just `sentence-bounds`, it is not a good idea – there may be other reasonable functions with such a name, so stealing the generic name `sentence-bounds` or `current-sentence-bounds` by our package would be a bit rude).

```
(defun reorder-sentence--current-sentence-bounds ()
  "Find the positions of the current sentence's beginning and ending.
Use the preceding one of the point is between sentences.  Return
a two-element list containing them."
  (let (beg)
    (save-excursion
      (backward-sentence)
      (setq beg (point))
      (forward-sentence)
      (list beg (point)))))
```

Now I am not really happy with this function. First of all, it is not very elegant, with the `setq` in the

middle. But honestly, I could not think of a better way – we want `forward-sentence` to start from the position `backward-sentence` sent us to, and somehow remember that position. One alternative could be to say

```
(list
 (save-excursion
   (backward-sentence)
   (point))
 (save-excursion
   (backward-sentence)
   (forward-sentence)
   (point)))
```

but frankly, this looks even worse. When in doubt which of the ones like these to choose, I try to keep in mind that it is usually not necessary to think about it in terms of *performance*, but rather *legibility*, and choose the one whose code is easier to understand. (Anyway, the second form would almost surely be slower, since we need to do the same thing – go `backward-sentence` – twice instead of once – but on my old laptop, `backward-sentence` takes about 50 microseconds to execute, and it's not like we are going to execute `reorder-sentence` a thousand time in a loop, right?)

So, here is the code of `reorder-sentence` again.

```
(defun reorder-sentence (beg end)
  "Reorder the words in the region."
  (interactive (if (use-region-p)
                   (list (region-beginning) (region-end))
                 (reorder-sentence--current-sentence-bounds)))
  (setq reorder-sentence--begin (copy-marker beg t)
        reorder-sentence--end (copy-marker end)
        reorder-sentence--previous-window-configuration (current-window-configuration)
        reorder-sentence--buffer (get-buffer-create reorder-sentence--buffer-name t))
  (with-current-buffer reorder-sentence--buffer
    (erase-buffer))
  (display-buffer reorder-sentence--buffer)
  (deactivate-mark))
```

This is nice, but unfortunately has a serious bug: it does not work at the first character of the sentence. While it makes sense that with the point *between* sentences, the *preceding one* is selected for reordering, it is not a good idea to do the same with the point *at the beginning* of a sentence.

What can we do about it? An easy patch would be to move forward one character at the very beginning of `reorder-sentence--current-sentence-bounds`. This is probably not the most elegant way to fix the issue, but should work. However, it has one problem apart from the general ugliness: the `forward-char` function signals and error when called with the point at the end of the buffer, so we would have to check for that explicitly. It is of course possible with a function like this:

```
(defun forward-char-if-possible ()
  "Move point forward one character unless at eob."
  (unless (eobp)
    (forward-char)))
```

or maybe use `condition-case` or `ignore-error` or something similar (we will not cover those in this book, see the Elisp reference for them!)

I think, however, that if we need to go to such lengths, this can no longer be classified as a "quick fix" – and if there is no "quick fix", why not fix it *properly* in the first place? Here is an idea: instead of cheating by moving one character forward, let's write a function which will explicitly check if we are at the beginning of a sentence. (Such a function could be useful outside the sentence reordering context, by the way.)

```
(defun reorder-sentence--bos-p ()
  "Return t if at the beginning of sentence."
  (save-excursion
    (let ((pos (point)))
      (forward-sentence)
      (backward-sentence)
      (= pos (point)))))
```

Alas, this is still not good – it still quits with a beep when on end of buffer. Not good. A careful inspection shows that deep inside `forward-sentence`, a `forward-char` is (sometimes) issued – so we get the worst of all worlds...

Never mind – we can fix it. It will still be a bit ugly, but at least we'll have a function checking for the beginning of sentence. Plus, it may be a good idea to learn how to code this, right? This is why we're here after all! In fact, there are a few ways of doing this. The easiest one, "if we are at eobp, return nil, otherwise perform the code above", is too wordy – if we have an `if` form with either the "then-part" or the "else-part" being nil, it means that we can use either the `when` or the `unless` form. Note that the latter is sometimes tricky to decipher, since we humans are not very good at handling negatives (especially multiple ones) in our heads, so it may take a few seconds to wrap your head around this one.

```
(defun reorder-sentence--bos-p ()
  "Return t if at the beginning of sentence."
  (unless (eobp)
    (save-excursion
      (let ((pos (point)))
        (forward-sentence)
        (backward-sentence)
        (= pos (point))))))
```

Notice that the only reason we leave it like this – with `forward-sentence` and `backward-sentence` calls back to back – is that this function is going to be called once in an interactive command – if this was going to be called in some loop, it could be not very efficient because moving by sentences is complicated (and involves searching for regular expressions). On the other hand, it took my laptop less than a microsecond to evaluate this function in the middle of a sentence, so maybe that's not really bad after all. (But still, it was more than a thousand times slower than at the end of the buffer!)

Having this, let's get back to getting the bounds of the current sentence.

```
(defun reorder-sentence--current-sentence-bounds ()
  "Find the positions of the current sentence's beginning and ending.
Use the preceding one of the point is between sentences.  Return
a two-element list containing them."
  (if (reorder-sentence--bos-p)
      (list (point)
            (save-excursion
              (forward-sentence)
              (point)))
    (let (beg)
      (save-excursion
        (backward-sentence)
        (setq beg (point))
        (forward-sentence)
        (list beg (point))))))
```

Let us pause for a second and look at the code of `reorder-sentence--current-sentence-bounds` (together with its helper function `reorder-sentence--bos-p`. What started as a pretty simple concept is now a pretty complicated thing, with two conditionals (one `if` and one `unless`), three `save-excursions` and two `lets`. This is often the case when coding in Emacs Lisp: what starts as a pretty simple tool, just a few lines of code, tends to grow because of edge cases (like the beginning-of-sentence one, and then the end-of-buffer one inside the previous one!). It may grow even more complicated if it turns out we need to support *more* special cases, or make it operate in various minor modes which can change Emacs behavior in surprising ways. This is why reading Emacs source code may be intimidating – over time it grew considerably to support quite a lot of

use cases, exceptions, modes etc. This is also why it is a *very* good idea to comment your code. Putting all jokes about commenting the code aside, let me state what many seasoned programmers know: comments should not explain *what* the code does (this is usually pretty obvious – you can just *read* the code), but *why* it does what it does. So, let me show the source code for the `reorder-sentence--current-sentence-bounds` again, this time commented a bit so that everyone reading this code can understand why it was done the way it was.

```
(defun reorder-sentence--bos-p ()
  "Return t if at the beginning of sentence."
  ;; because `forward-sentence' barfs at eobp
  (unless (eobp)
    (save-excursion
      (let ((pos (point)))
        (forward-sentence)
        (backward-sentence)
        (= pos (point))))))

(defun reorder-sentence--current-sentence-bounds ()
  "Find the positions of the current sentence's beginning and ending.
Use the preceding one of the point is between sentences.  Return
a two-element list containing them."
  ;; `backward-sentence' would move to previous one on bos
  (if (reorder-sentence--bos-p)
      (list (point)
            (save-excursion
              (forward-sentence)
              (point)))
    (let (beg)
      (save-excursion
        (backward-sentence)
        (setq beg (point))
        (forward-sentence)
        (list beg (point))))))
```

Introducing a mode

Now that we have the code for selecting the current sentence when the region is not active, let's continue our quest to make the sentence reordering a smooth user experience. Jumping manually from word to word and using `M-x reorder-sentence-copy-word-at-point` does not make sense – it's not really more convenient than just killing the words and yanking them in their correct places. What would be really great is this: after `reorder-sentence` is called, all words in the sentence could

be marked with a character (say, a letter), and pressing the letter corresponding to one of those words would copy that word to our temporary buffer.

Since we settled on a non-modal design, the first idea that comes to mind is to define a custom *minor mode*, redefining some keys to perform our word-copying. The first thought is that *all* self-inserting characters should be disabled in that mode, since it is not supposed to be used to actually *edit* anything by hand.

This is easier said than done. The way keybindings work in Emacs is pretty complicated, but the idea is quite simple. When the user presses some key, it is first looked up in the currently active minor modes' keymaps. If not present there, the `local-map` variable (which holds the current major mode's keymap) is consulted. If still not found, Emacs checks the `global-map` variable, which holds keybindings that should work everywhere unless explicitly asked not to – and in fact, many basic Emacs keybindings are defined there, like a lot of movement commands and most if not all self-insertion commands. (That is not the whole story, though – see the "Active keymaps" section of the Elisp reference for that, since I simplified the reality a bit.)

Here's the problem: if we want to disable all self-inserting commands (since they would not make sense while reordering the sentence!), we would have to redefine *all* of them in our minor mode's keymap. This is because Emacs has no way to "temporarily disable" the global keymap – which is probably a good thing, because disabling e.g. `M-x` could make it stuck without a way out. (Incidentally, it *has* a way to temporarily disable all the rest of the keymaps via the `overriding-local-map` variable.) Worse still, we'd have to do the same with other keybindings which normally run commands that change the buffer. That would be very tedious, and wouldn't even work, since we cannot know what modes may be active when the user calls `reorder-sentence` and what keybindings they define.

We could, of course, temporarily change the global keymap – but the Elisp reference advises against that, and it is probably a good advice. So here is another idea we can use. Instead of redefining all the keybindings normally bound to commands changing the buffer, we can turn on the *read-only* mode. In fact, we will do something slightly different. If you consult the docstring of the `read-only-mode` command, you will learn that what we really want to do is to set the `buffer-read-only` variable to a non-nil value.

So, what we are going to need is a minor mode which sets the `buffer-read-only` variable to `t` when activated and resets it to nil when deactivated. (Note: one could argue that instead of resetting it to nil when the mode is deactivated, we should store its value when the mode was activated and restore the previous value – after all, we don't know whether the buffer was read-only when we started. This may be a good idea in other situations, but we actually *do* know that the buffer was *not* read-only in the beginning, since we use the `*` flag with `interactive`.) The best way to define a minor mode is via the `define-minor-mode` macro. Here is the first version of our mode.

```
(defconst reorder-sentence-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "C-c C-c") #'reorder-sentence-finish)
    (define-key map (kbd "C-c C-k") #'reorder-sentence-cancel)
    map))

(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap reorder-sentence-mode-map
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

(Note that we stick to the convention of saying `#'function-name` when we want to quote it. We could as well say `'function-name`, but using `#'` is a hint to the human reader that this is a function and not just some symbol. Also, it is a hint for Emacs, and it actually makes a difference in some circumstances.)

There are three main things you should probably define for every minor mode. The "lighter" is some text shown in the modeline when the minor mode is active. Since the lighters for all active minor modes are concatenated, the convention is to start it with a space. The keymap is, well, the keymap – most (though not all) minor modes need to bind some keys to some commands. (Actually, it is more complicated than that, we'll get to it in a moment.) The *body* of the mode (which is what comes after all the keyword parameters) is a list of forms evaluated when the mode command is called. This means the body forms are evaluated both when the mode is being turned on and turned off, hence the `if` – in fact, setting the mode body to `(if MODE-NAME ...)` is a common idiom. For now, our mode does nothing except making the buffer read-only.

Defining a minor mode keymap is slightly complicated. In fact, defining any keymap is slightly complicated. Happily, in the most common case we can just use the most common way and not think about it too much. The point is, there are *many* ways to do it. Probably the most usual is what we used above, where we first define a keymap using `defconst` or `defvar`, then use `make-sparse-keymap` to generate a new "sparse keymap" (which is basically always what we want unless we define some very atypical major mode), and finally we use `define-key` to put some keybindings into our newly defined keymap.

Note how we used the extremely useful `kbd` function to translate human-readable strings describing key sequences to the internal representation Emacs uses. For simple keys it is not necessary – for example, `(kbd "a")` evaluates to just `"a"` – but if we have anything fancy, `kbd` provides an easy way to describe the keys. For example, `(kbd "C-c M-a")` translates to `[3 134217825]` – definitely not something one would come up with easily, `(kbd "C-c C-k")` becomes `"^C^K"` (which is just the way Emacs displays the actual control characters – it is a two-character string, not a four-character one). It is even better – if you want to bind some more exotic key, like `PageDown` or `Scroll Lock`, you can

press `C-h c`, hit that key to learn that Emacs calls it `<next>` or `<Scroll_Lock>`, and then do `(kbd "<next>")` or `(kbd "<Scroll_Lock>")` to produce a representation suitable to use with `define-key` etc.

We could, however, do something a bit shorter. Instead of defining the keymap as a separate variable and giving the *name* of that variable as the `:keymap` parameter (unqouted – the macro takes care of that), we could just pass the whole `let` form to `:keymap`; the `define-minor-mode` macro would then create a variable holding that keymap for us. This means that we could do this.

```
(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap (let ((map (make-sparse-keymap)))
            (define-key map (kbd "C-c C-c") #'reorder-sentence-finish)
            (define-key map (kbd "C-c C-k") #'reorder-sentence-cancel)
            map)
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

In fact, there is an even shorter way, but it requires us to understand a few (very) important new concepts, so we will postpone it to the next section.

The next part of our mode definition is probably the most atypical. Most minor modes are also interactive commands you use to enable or disable them. For example, you can say `M-x auto-fill-mode` to toggle the auto-fill mode, `M-x highlight-changes-mode` to toggle change highlighting etc. In this case, I decided *not* to make `reorder-sentence-mode` an interactive command, and still use `reorder-sentence` as the "entry point". Of course, we *could* just drop the `reorder-sentence` command and perform all initialization actions we did there in the `reorder-sentence-mode` command instead, but I decided that since reordering the sentence involves creating a new (even if temporary) buffer and changing the window configuration, it *feels* too heavyweight to be just a minor mode in one buffer. More importantly, the convention is that a minor mode command accepts one (optional) argument which decides if the mode should be enabled or disabled (and interpreting the value of this argument is slightly complicated and depends on whether the mode was activated interactively or via Lisp code (see "Minor mode conventions" in the Elisp reference). Our `reorder-sentence` command, however, accepts *two* arguments, the bounds of the region to be reordered. This is why I decided to make `reorder-sentence-mode` non-interactive (because enabling that mode without running all the set-up code doesn't really make much sense) and call it from `reorder-sentence` instead.

Speaking of which, here are the updated set-up and tear-down commands, using the newly defined minor mode.

```
(defun reorder-sentence (beg end)
  "Reorder the words in the region."
  (interactive (if (use-region-p)
                   (list (region-beginning) (region-end))
                 (reorder-sentence--current-sentence-bounds)))
  (setq reorder-sentence--begin (copy-marker beg t)
        reorder-sentence--end (copy-marker end)
        reorder-sentence--previous-window-configuration (current-window-configuration)
        reorder-sentence--buffer (get-buffer-create reorder-sentence--buffer-name t))
  (with-current-buffer reorder-sentence--buffer
    (erase-buffer))
  (display-buffer reorder-sentence--buffer)
  (deactivate-mark)
  (reorder-sentence-mode 1))

(defun reorder-sentence-finish ()
  "Finish the reordering of the sentence.
Replace the selected region with the constructed sentence and restore
the window configuration."
  (interactive)
  (reorder-sentence-mode -1)
  (goto-char reorder-sentence--begin)
  (delete-region reorder-sentence--begin reorder-sentence--end)
  (insert-buffer-substring reorder-sentence--buffer)
  (set-window-configuration reorder-sentence--previous-window-configuration))

(defun reorder-sentence-cancel ()
  "Cancel the reordering of the sentence."
  (interactive)
  (reorder-sentence-mode -1)
  (set-window-configuration reorder-sentence--previous-window-configuration))
```

Notice that we used an explicit, positive or negative argument to `reorder-sentence-mode` so that our intention is more clearly stated (calling a minor mode function from Elisp with no argument is equivalent to enabling it, so we could omit the `1` argument, but since calling a minor mode *command* interactively with no prefix argument *toggles* the mode, this could be misleading for someone reading the code). Also, we disabled the minor mode at the very beginning of the `reorder-sentence-finish` command – this is reasonable, since we need the buffer to be editable (and turning the mode off disables its "read-only" status), and we won't be needing any "interactive" stuff (like the keymap) inside `reorder-sentence-finish` anyway.

We have our minor mode, which – from the user's perspective – is the skeleton of the whole thing. We will soon put some meat on its bones – but let's fulfill our promise first.

A shorter definition of the mode's keymap

As promised, we will first show how to make our minor mode's keymap definition even shorter. First, we need to understand a few new things, which are so important in Elisp coding that they deserve a section of their own.

First of all, we need to talk on *equality*, in a very specific sense. The question is, what does it mean that two things are *equal*? People keep using that word, saying e.g. that "$\frac{1}{2}$ is equal to `0.5`", or "all people are equal", and I'm not entirely sure it always means what they think it means. For instance, when we say that "$\frac{1}{2}$ is equal to `0.5`", it means that both represent the *exact same number*. When we say that all people are equal, we (obviously) don't mean that every person is the exact same person (and that humankind is, in fact, a one-element set), nor even that they *look* or *behave* in the same way, but that all people are similar in some particular respect – they have the same dignity, or they share the same nature. (More mathematically inclined readers would probably recognize the idea of an equivalence relation here.)

Likewise, Elisp has more than one concept of "equality". One is the `eq` function, which accepts two objects and determines if they are both, in fact, the very same object. For example, `(eq '() nil)` returns `t`, since the empty list and nil are exactly the same object (in fact, even `(eq () nil)` returns `t`, since the empty list evaluates to itself). Likewise, `(eq 'symbol 'symbol)` evaluates to `t`, since one of the features of Elisp symbols is that if you have two symbols with the same name, they can never be distinct objects that just happen to have the same name. (Well, actually, this is a lie, but the concepts of an "obarray" and "interning" are too advanced for this book; look for them in the Elisp reference if you are curious.) However, comparing things other than symbols with `eq` is tricky. Here is one example:

```
(setq a '(this is a list))
(setq b '(this is a list))
(eq a b)
```

and the result is nil, but a very similar code:

```
(setq a '(this is a list))
(setq b a)
(eq a b)
```

returns `t`. Why is that so? Well, in the first case, we constructed two lists (each containing four symbols) *separately*, so – even though these lists look identical (and in fact, have `eq` elements), they are *not the same list*. (Technically, they are stored in distinct places in the computer's memory.) In the second case, however, the lists bound to the symbols `a` and `b` are *the very same list*, occupying the same place in memory, so they are `eq` to each other. (Note that this means that if we change the value of `a`, the same change is reflected in `b` – we will talk later about so-called "destructive functions" which are a related concept.)

A more lax version is `equal`, which checks if the two objects are the same (if they are symbols), if they are of the same type and value (if they are numbers), or if they have `equal` components (in case of lists or strings). (Again, this is not the whole story, but it is enough for us.) This means that all of these forms return `t`:

```
(equal 1.0 1.0)
(equal "string" "string")
(equal '(symbol "string" (and a list)) '(symbol "string" (and a list)))
```

but every one of them would return nil if we used `eq`. On the other hand, `(equal 1.0 1)` returns nil, since `1.0` is a floating point number and `1` is an integer.

The best way to compare numbers (well, also markers) is to use yet another comparison function, `=` – for instance, `(= 1.0 1)` returns `t`. The standard way to compare strings is the `string=` function, although `equal` would do the job as well (the difference is subtle and manifests itself only when the comparison involves values which are *not* strings – `(equal 'this "this")` yields nil, but `(string= 'this "this")` first converts the symbol to a string and hence yields `t`).

Having learned about `eq` and `equal` (and also `=` and `string=` – there are in fact more equality-like functions in Elisp, but they are a bit more, let's say, exotic, and we won't need them here), we can talk about *alists*, or *association lists*. It happens quite often (both in programming and in real life!) that we need to make, well, a *list* of *associations*. For instance, we might have a bunch of friends (well, hopefully we do!), and we want to record when every one of them has their birthday. There are plenty of ways you could record such a thing in Emacs. You could create a buffer containing contents of a csv file, for instance:

```
Jack,1996-04-12
Jill,2001-11-04
Steve,1998-12-31
```

which is perfectly reasonable, but not very well-suited to be used in Elisp programs. For them, you might want to record these data in a list, like

```
'("Jack" "1996-04-12" "Jill" "2001-11-04" "Steve" "1998-12-31")
```

Actually, it would be more Lispy to use *symbols* instead of strings:

```
'(Jack "1996-04-12" Jill "2001-11-04" Steve "1998-12-31")
```

and that is because the "standard" equality comparison used under the hood in many Elisp function is `eq`, which is guaranteed to return `t` for two symbols with the same name (again, under normal circumstances), but not for two strings with the same contents. Anyway, while this is a perfectly valid way of storing such an association (and in fact is used in several places in Emacs – it even has a name, "plist", which is an abbreviation for a "property list"), the most usual way is the *alist*:

```
'((Jack . "1996-04-12") (Jill . "2001-11-04") (Steve . "1998-12-31"))
```

Note that this is a list consisting of cons cells, and every cons cell consists of the "key" (the thing we use to distinguish between different values, or search for the value we need) in its car and the value (the thing associated with the key) in the cdr. If we have a variable `alist` bound to the list above, we can easily retrieve the birthday of Jack, saying `(alist-get 'Jack alist)`. (There are more ways to work with alists, also ones that use `equal` instead of `eq` to compare the keys, but `alist-get` is one of the simplest ones.) Association lists are very useful both in Elisp programming and configuring Emacs, and they appear in lots of places.

Knowing all that, we can now understand the following: we can just pass an alist of pairs consisting of key sequences and their corresponding definitions to the `:keymap` key in the minor mode definition. The `define-minor-mode` macro will then take care of properly defining the variable and the keymap. This, however, is a bit tricky to do properly. We cannot just write `(...)`, since Emacs would treat it as a form to evaluate and not a list. We don't want to just write `'(...)`, since then *nothing* in the list would be evaluated (and we would like to be able to use the `kbd` function in there). We could construct this list programmatically, using functions such as `list` and `cons`, like this:

```
(list (cons (kbd "C-c C-c") #'reorder-sentence-finish)
      (cons (kbd "C-c C-k") #'reorder-sentence-cancel))
```

This, however, is rather verbose, and completely defeats the purpose of keeping the `define-minor-mode` definition short. There is, however, a way out of this dilemma (trilemma?), the (in)famous *backquote*. If you quote a list with a backquote (also called a *backtick* or *grave accent*, you can "escape quoting" inside the list by preceding what you actually want to evaluate with a comma. For example, `` `(+ (+ 1 2) ,(+ 1 2)) `` yields `(+ (+ 1 2) 3)` – the former `(+ 1 2)` will not be evaluated because the whole list is preceded by a backquote, but the latter one is evaluated because of the comma. (There is more to backquoted lists than this, but this is enough for us now – we will see a bit more soon.) So, if we give the `:keymap` parameter a list of

```
`((,(kbd "C-c C-c") . reorder-sentence-finish)
  (,(kbd "C-c C-k") . reorder-sentence-cancel))
```

we get the best of both worlds – the shortest version of `:keymap`, just the alist of bindings, and the usefulness of the `kbd` function.

```
(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
            (,(kbd "C-c C-k") . reorder-sentence-cancel))
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

Showing the key for a word

Besides all these technicalities about defining minor mode maps, we need to take the more high-level design question into account. What keys are we actually going to define for our mode keymap so that it is convenient to use? The obvious part is what we did above – `C-c C-c` is a widely-spread Emacs convention for "what I'm doing is done, confirm", and `C-c C-k` is a similar convention for "no, I don't want to do this after all, cancel". Of course, we could do something else (or something more). In fact, we *will* do a few more things in the following sections. However, `reorder-sentence-copy-word-at-point` being the workhorse of our tool, we will need to make it work with some keys – but what ones?

The whole point of the sentence reordering feature we are working on is to make it work fast and convenient. Even if we bound `reorder-sentence-copy-word-at-point` to some easy to reach key (`SPC` might be a good candidate), having to move the point to every word we want to copy to our temporary buffer would be no fun at all. Here is the idea. We will arrange things so that pressing `a` will copy the first word, pressing `b` the second one etc.

This is easier said than done, however. Before we write a single line of code, we need to decide on a few things. First of all, what if there are more than 26 words in the region to be reordered? Let's use the digits for the next 10, then capital letters. This way we'll be able to manage 62 words, which should be more than enough in practice – of course, a longer region/sentence should not cause an error, in case the user is e.g. marks the whole buffer and calls `reorder-sentence` just to see what happens. Since we assume this shouldn't normally happen, "supporting" may as well mean "doing something much less convenient as a fallback" – and it seems that letting the user just say `M-x reorder-sentence-copy-word-at-point` is good enough.

So, this is our plan for now. First, we are going to design a way to *show* which letter/digit is associated with which word (otherwise the user would have to count words and letters manually, which would be insane!). Then we are going to find a way in which key presses will be "translated" to positions of the corresponding words in the buffer. Finally, we are going to write a command which will find the suitable word and act on it accordingly.

I can think of three ways of showing the user which key corresponds to which word: a lousy one, a "meh" one, and a good one. The lousy one would be just to insert some pieces of text before every

word, for example `<a>`, `<b>` etc. This is a very bad idea – we modify the buffer, we pollute the undo list unless we are very careful, we change the buffer modification status, we have to remember to delete them if we call `reorder-sentence-cancel` – a lot of problems.

A slightly better one would be to use *text properties*. In Emacs, every character in a buffer (or even in a string!) may be assigned one or more "properties", which influence quite a few of things in Emacs. Each property has a *name* (usually a symbol, much like function or variable names) and a *value*. This means that text properties are in fact kind of "hash tables" known from other programming languages – for every character we can ask "what is the value of the property named X at this position in the buffer or string". You can use any property name you want for any purpose you want, but some property names are special in that they are treated by Emacs in specific, predefined ways. For instance, there is the `face` property which decides what font, color etc. Emacs uses to display that character. Some of these properties have self-explanatory names, like `invisible` or `read-only`. Some of them are pretty clever – for example, there is the `help-echo` property, whose value can be some string, and that string is displayed in the echo area or a tooltip if you point your mouse to the character possessing that property. Another example is `keymap`, which can define a keymap active when the point is on a character possessing that property.

One of the special properties is `display`, which governs how the character (or characters) are displayed. It may be used to display some image or string *instead* of whatever character(s) have this property. We could use it to display e.g. the string `<a>word` instead of just `word` – but this is still far from ideal, since there is no way to make this `<a>` stand out, e.g., by using a different color.

The best way to temporarily display such things is to use *overlays*. In a sense, they are a way to assign something much like text properties to some portion of buffer, but in a way that can be removed very easily (in fact, also *moved* to some other portion of the buffer, which we won't need). In fact, an overlay consists of four things: the buffer (an overlay may belong to some buffer – if it doesn't, it does not influence the display in any way), the beginning and the end (which are markers, so inserting or deleting text *before* an overlay moves it correspondingly), and the property list.

Many overlay properties have the same meaning as the corresponding text properties, so we have e.g. `face`, `help-echo` or `keymap` overlay properties. There are, however, some overlay properties which do not have corresponding text properties. One of them we are interested in now is the `before-string` property. (It also has a twin called `after-string`.) If we set this property to a string, that string will be displayed right before the beginning of the overlay. Even better – as we will see shortly, we may set the *text properties* of that string to e.g. change the face used to display the `before-string`!

One caveat with overlays is that having a lot of them in a buffer may slow Emacs down. This is not something we will be worried about, since we are going to create at most several dozen of them, and we will take care to delete them afterwards.

Let's use overlays to show which keys are going to copy which words to the temporary buffer. Now, one issue is that while Emacs has the function `current-word` which returns, well, the current word, we don't have functions like `word-beginning` or `word-end`, returning the positions where the current word starts or ends. (There is `mark-word`, but we already know it is not what we need.) It turns out that we can return to the idea of using the `thingatpt` package, which has a function well-suited

to our current need: `bounds-of-thing-at-point`. If we run it with the point at (or right after) some word with a parameter of `'word`, we will get a cons whose car is the beginning position of the current word and cdr is the end position.

Well, the last sentence probably deserves some explanation, just in case you don't remember how Elisp treats symbols and how lists are made. First of all, the `bounds-of-thing-at-point` expects one argument – the type of whatever we want to get the start/end positions of. (It can be a sentence, a defun, but also e.g. an email address and several other things.) We cannot write just `(bounds-of-thing-at-point word)` – if we did that, the Elisp interpreter would try to look up the *variable* `word` to see if it contains the name of the entity we are looking for. The quoted form, `'word`, means "take the actual *symbol* `word`, not the value it is pointing at" – in other words, it is the way to distinguish between the *value* of the variable `word` and the *name* `word`. The second important thing is the idea of *cons*, which is a *pair* of values. The most common use of conses in Elisp is how lists are made up of them, which is nicely explained in *An Introduction to programming in Emacs Lisp* in the *Lists diagrammed* section. A cons, however, can just contain two things (numbers, strings, symbols – you name it), and in such a case it is often written down using the "dotted pair" notation. For example, if I say `M-: (bounds-of-thing-at-point 'word)` with the point at he beginning of this sentence, Emacs says `(112935 . 112938)`, which means that (at the moment I write this) the word "For" starts at position 112935 in the buffer and ends at 112938. (A good exercise would be to guess what – and why – will happen when you say `M-: '(1 . (2 . nil))` – try it, but think about it first!)

So, let us write a pretty simple function which will "mark" (not in the sense of putting the region around it, but in the sense of putting an overlay on it) a word to "prepare" it to be copied to our temporary buffer. Words will be "marked" with *characters*, which are similar to strings of length one, but internally they are just integers – more or less, ASCII codes or Unicode codepoints. Since we are only going to use English letters and digits, we just need the ranges 97–122 (lowercase letters), 48–57 (digits) and 65–90 (uppercase letters), all falling within ASCII.

```
1  (defun reorder-sentence--put-overlay-on-word-at-point (key)
2    "Put an overlay showing KEY over word at point and return it.
3  The overlay will display the KEY (a character) to the left."
4    (let ((bounds (bounds-of-thing-at-point 'word))
5          overlay)
6      (when bounds
7        (setq overlay (make-overlay (car bounds) (cdr bounds)))
8        (overlay-put overlay 'before-string (format "<%c>" key)))))
```

Notice that this function does not return the newly created overlay for further use. We could do that in order to create a list of all the overlays we created, but it turns out we won't need such a list now – Emacs already keeps a list of all overlays in the buffer and has ways to find the one we need. We will change this function later to make it possible.

Also, notice that `bounds-of-thing-at-point` may return nil if the point is not on any word. (This will actually happen if the beginning or end of the region to reorder is not on any word.) This means

that we cannot set `overlay` directly in the `let` – instead, we only set up a local binding and then use `setq` to actually create the overlay if `bounds` is non-nil. (If we tried to say `(make-overlay (car bounds) (cdr bounds))` with `bounds` equal to nil, Emacs would complain about nil being neither an integer nor a marker. Interestingly, you *can* say both `(car nil)` and `(cdr nil)`, getting nil in both cases.)

This function has one drawback, though. The key is not visually distinguished enough from the rest of the text – it is just a character in angle brackets, after all. Actually, this might make sense on terminal displays, but people using Emacs with a GUI (which I assume is the majority) could use something better. (And even terminal displays often support colors.) Let us write a very simple function accepting a character and returning a string ready to be put in the `before-string` property of our overlay. This way we'll be able to put this function in a customizable variable so that every user can decide what these keys are going to look like.

```
(defface reorder-sentence-key
  '((t :foreground "chocolate" :underline t :height 0.9))
  "Face to display word keys for sentence reordering.")

(defvar reorder-sentence-key-format "%c"
  "Format string to display the key for sentence reordering.")

(defun reorder-sentence-prepare-key-for-display (key)
  "Format a string to mark a word using the KEY character."
  (propertize (format reorder-sentence-key-format key)
              'face
              'reorder-sentence-key))

(defvar reorder-sentence-prepare-key-for-display-function
  #'reorder-sentence-prepare-key-for-display
  "A function used to highlight the keys for reordering the sentence.
It should accept a character and return a string.")

(defun reorder-sentence--put-overlay-on-word-at-point (key)
  "Put an overlay showing KEY over word at point and return it.
The overlay will display the KEY (a character) to the left."
  (let ((bounds (bounds-of-thing-at-point 'word))
        overlay)
    (when bounds
      (setq overlay (make-overlay (car bounds) (cdr bounds)))
      (overlay-put overlay 'before-string
                   (funcall reorder-sentence-prepare-key-for-display-function
                            key)))))
```

Now be warned that this only serves as an illustration – actually, this is a rare example

of making things *too* customizable. Once we have the `reorder-sentence-key` face *and* the `reorder-sentence-key-format` variable, it's hard to imagine a case when anything else than `reorder-sentence-prepare-key-for-display` to display the keys could be needed. In fact, the only reason I decided to introduce the `reorder-sentence-prepare-key-for-display-function` variable is to show how this is done. By convention, if a variable name ends in `-function`, it is expected to be bound to, well, a *function*. (By a similar convention, if a variable name ends in `-hook`, it is expected to contain a *list of functions*, each of which will be called without any arguments. You can read more about these conventions in the "Hooks" section in the Elisp reference.) Providing the user with such variables is an ultimate way of making things customizable. However, if possible, simpler ways are preferable – it may be easier for a user not knowing Elisp to provide a format string or a face than to code a function. Thus in our case it would be a better design to just stick to the face and the format variable.

The other side of the coin is of course the question how to *use* such a variable. The easiest – and used above – way is to use the `funcall` function, which accepts a *function* to call and the *arguments* to give it. If the first argument is a literal, like this: `(funcall #'message "Hello, sweetie!")`, this is the same as calling the function directly: `(message "Hello, sweetie!")`. The point of the `funcall` function is that it can accept a *variable* bound to a function (like we did above), or – more generally – any expression which evaluates to, well, a function. For instance, here is a (slightly contrived) example of using `funcall` to insert a string at point if a buffer is not read-only or show it in a minibuffer otherwise:

```
(funcall (if buffer-read-only #'message #'insert) "Hello, sweetie!")
```

This is marginally better than saying

```
(if buffer-read-only
    (message "Hello, sweetie!")
  (insert "Hello, sweetie!"))
```

since we do not have to repeat the string. (Of course, we could also say

```
(let ((string "Hello, sweetie!"))
  (if buffer-read-only
      (message string)
    (insert string)))
```

but this is considerably longer than the `funcall` version. On the other hand, it may be easier to read.) Besides `funcall`, there is also `apply`, which – instead of as many arguments after the first one (providing the function to call) as the function needs – accepts a *list* of arguments to give the callee. (Well, it's not the whole story – check its docstring or the relevant section in the Elisp reference to learn more.)

Generating the list of keys to use

We will now learn how to generate the list of keys to use. We have already decided that we want to use the lowercase letters first, the digits next and finally the uppercase letters.

First of all, we are going to construct the key sequence iteratively, using `push` to add subsequent element to the beginning constructed list. This means that we will generate the list backwards. Actually, we *could* start with the capital letters from `Z` to `A`, followed by digits from `9` to `0` and finally lowercase letters from `z` to `a` – but the way we'll do it feels slightly more natural. More importantly, we'll use this approach to introduce a very common Elisp idiom of constructing the list from back to beginning and then reversing it in one go. More precisely, we will first `let`-bind some variable to an empty list, `push` some values to it, and then `nreverse` the variable to put them in the right order. An important property of `nreverse` is that it *modifies* the list it was given. This means that it doesn't make much sense to say something `(nreverse '(3 2 1))` (well, you *can* say that, but you really *shouldn't*). You should be even careful not to give `nreverse` a *variable* which was bound to a list literal (i.e., a "constant" list specified in the code, as in `'(3 2 1)` or its equivalent `(quote 3 2 1)`) like this.

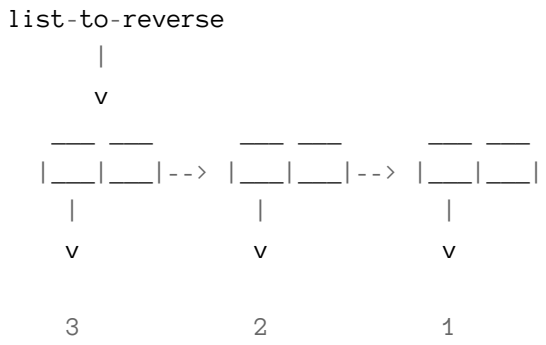
Let us pause for a few moments, because this is actually important. The reason is a bit complicated – `nreverse` is one of many so-called *destructive* Elisp functions, meaning that it *modifies* its argument. (Modifying its own arguments without warning would be an *extremely* rude behavior, so any function that does that *must* be clearly designed as "destructive" or some synonym in its docstring and in the Elisp reference. For instance, the docstring of `nreverse` clearly says: "This function may destructively modify SEQ to produce the value", and the reference says: "the original SEQUENCE may be modified.") This means that if you give it a list literal, very bad things will happen.

When Emacs reads such code, it first stores this list somewhere in memory, and then `nreverse` modifies *that place* in memory. This can lead to really unexpected behaviors. Consider this function:

```
1  (defun destructive-havoc ()
2    "Example of destructive havoc."
3    (setq list-to-reverse '(3 2 1))
4    (message "before: %s" list-to-reverse)
5    (message "after: %s" (nreverse list-to-reverse)))
```

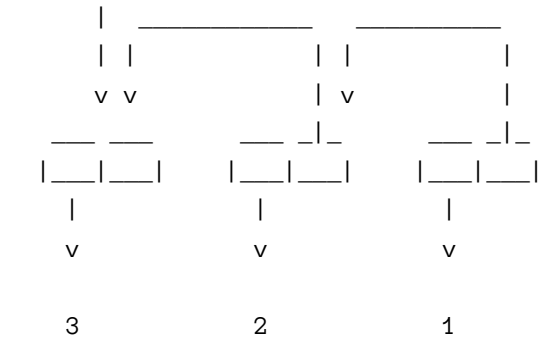
If you call this function, everything seems alright – you get `before: (3 2 1)` and then `after: (1 2 3)`. (You may want to open an Emacs window showing the `*Messages*` buffer to see what is printed there.) But call it again, and really weird stuff happens: `before: (3)` and `after: (3)`. What is going on here? Well, the (second) `nreverse` *returned* the reversed list alright, but it was *given* the thing after it was "destroyed" by the (first) `nreverse`.

Here is the same thing "in slow motion", so to speak. When Emacs evaluated the `setq`, it made a list, which, as explained in *An introduction to Emacs Lisp* – which you have read, haven't you? – is made up of cons cells (or conses), and each cons is a "box" with two "segments", the car and the cdr. Here is a diagram showing the relevant portion of Emacs memory:



This diagram means that the `list-to-reverse` variable "points" (or "is bound to") to the first cons, so the first element of the list is its car (that is, `3`). The cdr of that first element points to the second cons, whose car is `2` – and hence `2` is the second element of the list. Lastly, the cdr of the third cons is nil (so it's left empty in the diagram above), meaning that its car (`1`) is the last element of the list.

Now calling `nreverse` means that Emacs is going to change the bindings of the cdrs so that they "point in the opposite direction". This means that it won't even touch the cars of these three cells, but will modify their cdrs so that they look like this:



The return value of the `nreverse` is the third cons cell, which means that effectively `nreverse` returned the list `(1 2 3)` – which is correct. However, it knew *nothing* about even the existence of the `list-to-reverse` variable (note that it was given the *value* of that variable, not the variable itself – this is how function evaluation in Elisp works, the arguments are evaluated before being given to the function!), and hence that variable is still pointing to the first cons cell in the picture – meaning that its value is now the list `(3)`!

But it's even worse than that. When we ran the same function for the second time, it said: `before: (3)` and `after: (3)`. Here is what happened. When the function was defined (that is, when the form `(defun destructive-havoc ...)` was evaluated), Emacs read in the list `'(3 2 1)` and stored it somewhere in the memory. It also remembered that the `(setq list-to-reverse ...)` form, when evaluated (during the execution of our function), should make `list-to-reverse` point at that particular place in memory. This means that the `list-to-reverse` variable points then at the cons cell whose car is `3` and whose cdr is the next cons cell in the chain. But after evaluating the `nreverse`,

this place becomes a cons cell whose car is `3` and whose cdr is nil, so the next time after the `(setq ...)` is evaluated, `list-to-reverse` is just the list `(3)`! How confusing…

One way to be sure that `nreverse` (or other destructive functions, like `sort`) will not do things like that is to give them some *local* variable, i.e., something "made specially for them". The common Elisp idiom mentioned a few paragraphs ago is exactly that – we `let` a new variable point to a (newly created) empty list (and nil is not stored in a cons cell, it's just a value of its own, so `(let ((var nil)) ...)` will create `var` anew every time it's evaluated!), then `push` to the list as many times as we want, and finally return `(nreverse var)`. Here is a simple example, generating a list of integers starting from `a` and ending in `b - 1` (or `(1- b)`, as it's spelled out in Elisp):

```
(defun range (a b)
  "Return a list of numbers starting with A and ending before B."
  (let ((list ())
        (counter a))
    (while (< counter b)
      (push counter list)
      (setq counter (1+ counter)))
    (nreverse list)))
```

There are in fact four interesting things I'd like to say about this function before we go on to (finally!) producing the keys for our overlays. First of all, Elisp has a function called `list`, but we can also have a variable called `list` – this is perfectly fine (and in fact used pretty often), and it's even mentioned in *An introduction to Emacs Lisp* in the *Symbols as Chest* section. (The fact that functions and variables are "kept separate" this way is sometimes expressed by saying that Emacs Lisp is a "Lisp-2".) Then, instead of saying `(let ((list ())) ...)`, we could say just `(let ((list)))`, since nil (which is the same thing as the empty list) is the default value of any variable in `let`. (In fact, `(let (list) ...)` would also work, since if a binding in `let` is a symbol and not a two-element list, it just binds that symbol to nil.)

The third one is longer. Because the arguments a function receives become effectively local variables within that function, we could use the argument `a` instead of creating a local variable called `counter` and increment that instead:

```
(defun range (a b)
  "Return a list of numbers starting with A and ending before B."
  (let (list)
    (while (< a b)
      (push a list)
      (setq a (1+ a)))
    (nreverse list)))
```

Of course, using `a` instead of defining `counter` is shorter and most probably a very tiny bit faster, but may be confusing, so I slightly prefer the longer form involving an explicit local variable. Also, this

trick is only possible because numbers in Elisp are *immutable* – unlike lists, we cannot "destruct" them, or e.g. change a variable given as an argument to a function from within that function. Consider this snippet of code:

```
(setq c 1)
(range c 10)
c
```

We first assign the number `1` to the (global) variable `c`, then call our `range` function using the newly assigned variable as the first argument, and finally evaluate `c` again. Even with the latter, shorter version of the `range` function, `c` will still be `1` at the end – in other words, Emacs will use a "copy" inside `range` and not modify the global value of `c`. Contrast this with this code, where the situation is similar to the `destructive-havoc` we discussed some time ago:

```
(defun make-car-a (list)
  "Change the car of LIST to the symbol `a'."
  (setcar list 'a))

(setq l '(1 2 3))
(make-car-a l)
l
```

Here again we define a function which changes its argument, but we assume that this argument is a list. Since Elisp lists are mutable (which is basically another way of saying that a variable holding a list really holds the *address* of that list – more precisely, its first element – in memory, not the list itself, and that if we assign another variable to it, they now both point to *the same* place in memeory), the change we have made to the car of that list persists even after the function finishes its job. Again – the difference is caused by the fact that while a variable assigned a number "contains" precisely that number, a variable assigned a list really "contains" an *address in the memory* where the list (more precisely, its first cons cell) is stored. So, when we `setcar` of that list to something else, the *address* (which is the "real value" of the variable) is still the same (and is not even touched by `setcar`), but the list itself – somewhere in the computer's memory – *is* changed. If we were to first change the *address* the variable points to, and then `setcar` of the list to something else, the global value would remain unaffected:

```
(defun reassign-and-mutate (list)
  "Reassign LIST to something else and then mutate it."
  (setq list (list 1 2 3))
  (setcar list 'a)
  list)

(setq l '(1 2 3))
(reassign-and-mutate l)
l
```

(Notice that we say `(list 1 2 3)` in the body of that function – effectively creating a new list every time – instead of `'(1 2 3)`, since we then mutate that list. As seen with the `destructive-havoc` function earlier, mutating lists defined as literals is a very bad idea.)

You may ask which kinds of values can be changed, or *mutated*, as it's often called, and which cannot. This is sort of a grey area. Basically, numbers (and hence also characters, which are just integers) are immutable, nil (which is the same as the empty list) is immutable, lists, strings and other "composite" types we do not discuss in this book (like arrays or vectors) are mutable (although it's safer to treat especially strings as immutable, since they are often defined using literals), symbols are mutable (i.e., you can change the contents of the "drawers" they possess, like the variable or function value associated with the symbol – but the *name* of the symbol cannot be changed), and most "editing" types (like buffers, markers and many others) are mutable.

Coming back – the last, fourth thing about our `range` is that it is basically a simplified version of Emacs' own `number-sequence` function, which you might want to check out (and study both its docstring and source code).

Ok, that was pretty long. Let's get back to generating our key list. We already know how to construct the list piecewise and `nreverse` it at the end. Now we need to take into account the fact that what we want is not a contiguous sequence of ASCII characters, but three "runs" – lowercase letters first, digits next and then uppercase letters. A simple way to code this would be as follows:

```
(defun reorder-sentence--generate-keys ()
  "Return a list of characters to use for sentence reordering."
  (let ((list)
        (counter 97))
    (while (/= counter 91)
      (push counter list)
      (setq counter (1+ counter))
      (cond
       ((= counter 123) (setq counter 48))
       ((= counter 58) (setq counter 65))))
    (nreverse list)))
```

We start at `97` (lowercase `a`), jump to `48` (the digit `0`) when we pass through the entire alphabet, jump again to `65` (uppercase `A`) when we use up all digits, and continue until we add `90` (uppercase `Z`). (Note that we check for the "jumping points" *after* incrementing `counter`, so instead of checking for `122`, `57` and `90` we need to check for `123`, `58` and `91`. Note also how we use the `cond` form, which is much like `if`, but can have more than one condition. It is very similar to a `switch` statement found in some languages or a chain of `if ... else if ... else if ...`, which would be a bit unwieldy in Lisp because of the nesting of the parentheses.) This, however, has one (very minor) drawback – the logic of "when to jump to another place and when to stop" is split between two places – the condition in the `while` loop and the `cond` form. Wouldn't it be nicer if we could handle all this in one place? In langauges descending from C, you can use a `break` statement anywhere in a loop to, well, break out of it before the condition is normally checked. This would enable us to put the code checking if we should finish within the `cond`. It turns out that while Elisp does not have a `break` statement (in fact, it has no statements at all, only expressions!), we can use a more general `goto`-like mechanism called `catch`/`throw`. We can wrap an Elisp form (or forms) within a `catch` special form, and jump out of it from any point inside using `throw`.

The `catch` special form needs a "tag" (which is normally a symbol, but technically could be anything but nil), used as a "label" to distinguish it from other `catch` forms. Anywhere inside the `catch` (and here, "inside" means not "inside the textual contents of the `catch` form", but rather "during the evaluation of code inside the `catch` or called from such code" – we can have `catch` in one function and `throw` in another one, provided that it is called from the former one) you can say `(throw tag value)`, where `tag` is the tag of the corresponding `catch` and `value` is what becomes the value of *the whole* `catch`. So, we can rewrite our function like this:

```
(defun reorder-sentence--generate-keys ()
  "Return a list of characters to use for sentence reordering."
  (let ((list)
        (counter 97))
    (catch 'exit
      (while t
        (push counter list)
        (setq counter (1+ counter))
        (cond
         ((= counter 123) (setq counter 48))
         ((= counter 58) (setq counter 65))
         ((= counter 91) (throw 'exit list)))))
    (nreverse list)))
```

I'm not sure if it's really worth the effort, but at least all the logic describing the "jumps" in the sequence and its end is brought together.

In fact, there is one more thing you could do with this function, although it is kind of an "nuclear option". There is an Elisp package called `cl`, which implements lots of features from Common Lisp in Elisp. Common Lisp is Elisp's big brother, so to speak – and has many features not present in

Elisp. The `cl` package (which comes installed with Emacs, so you don't have to download anything to use it) has a lot of very useful stuff. One of them is the `cl-loop` macro, which is a pretty general iteration device. We will not describe `cl-loop` here – see the documentation of `cl` for that – but let me just mention that you can use `(cl-return)` inside it, which is quite similar to `throw` (although with notable differences, too – see the manual for the details). Another possible reason to use `cl` here would be the `cl-incf` macro, which increments a given variable (in fact, it is more general than that, but let us put that aside), so instead of saying `(setq counter (1+ counter))`, you could just say `(cl-incf counter)`. We will not, however, use `cl` in this book – I just mention it here in case anyone needs it for some other projects.

Anyway, now that we have our code generating the list of keys, we can move on and actually use it.

Showing the keys for all words in the region

We will now write a function creating overlays for every of the subsequent words in the region. This is fairly easy now – we need to go to the region's beginning and call `reorder-sentence--put-overlay-on-word-at-point` followed by `forward-word` until we reach (or more accurately, get past) the end.

```
(defun reorder-sentence--create-overlays (begin end)
  "Create overlays for reordering region from BEGIN to END."
  (save-excursion
    (save-restriction
      (narrow-to-region begin end)
      (goto-char begin)
      (let ((keys (reorder-sentence--generate-keys)))
        (while (and keys
                    (< (point) end))
          (forward-word)
          (reorder-sentence--put-overlay-on-word-at-point (car keys))
          (setq keys (cdr keys)))))))
```

Try it out e.g. by activating the region and saying `M-: (reorder-sentence--create-overlays (region-beginning) (region-end))`. It's better to do this in a temporary buffer which you can kill afterwards, since we don't yet have an easy way to remove all the overlays we created (although since we do not store any references to them, killing the buffer will dispose of them).

Actually, let us fix that right now. Elisp has the `remove-overlays` function, which can remove all the overlays in a region (or buffer) or some of it. This is of course way too much – there might be other overlays here, created by some other tools, and we should not interfere with that. The easiest way to remove only some of the overlays is to give them some property-value pair. We can then tell `remove-overlays` to only remove overlays which have that particular property set to that particular value.

```
(defun reorder-sentence--put-overlay-on-word-at-point (key)
  "Put an overlay showing KEY over word at point and return it.
The overlay will display the KEY (a character) to the left."
  (let ((bounds (bounds-of-thing-at-point 'word))
        overlay)
    (when bounds
      (setq overlay (make-overlay (car bounds) (cdr bounds)))
      (overlay-put overlay 'before-string
                   (funcall reorder-sentence-prepare-key-for-display-function
                            key))
      (overlay-put overlay 'reorder-sentence-key key)
      (overlay-put overlay 'reorder-sentence t))))

(defun reorder-sentence--remove-overlays ()
  "Remove all reorder-sentence-related overlays from the buffer."
  (remove-overlays (point-min) (point-max) 'reorder-sentence t))
```

Now that we have a way to both set up and tear down the overlays, we can use it in our main set-up and tear-down commands, too.

```
(defun reorder-sentence (beg end)
  "Reorder the words in the region."
  (interactive (if (use-region-p)
                   (list (region-beginning) (region-end))
                 (reorder-sentence--current-sentence-bounds)))
  (setq reorder-sentence--begin (copy-marker beg t)
        reorder-sentence--end (copy-marker end)
        reorder-sentence--previous-window-configuration (current-window-configuration)
        reorder-sentence--buffer (get-buffer-create reorder-sentence--buffer-name t))
  (reorder-sentence--create-overlays reorder-sentence--begin reorder-sentence--end)
  (with-current-buffer reorder-sentence--buffer
    (erase-buffer))
  (display-buffer reorder-sentence--buffer)
  (deactivate-mark)
  (reorder-sentence-mode 1))

(defun reorder-sentence-finish ()
  "Finish the reordering of the sentence.
Replace the selected region with the constructed sentence and restore
the window configuration."
  (interactive)
  (reorder-sentence-mode -1)
  (reorder-sentence--remove-overlays)
```

```
24    (goto-char reorder-sentence--begin)
25    (delete-region reorder-sentence--begin reorder-sentence--end)
26    (insert-buffer-substring reorder-sentence--buffer)
27    (set-window-configuration
28     reorder-sentence--previous-window-configuration))
29
30  (defun reorder-sentence-cancel ()
31    "Cancel the reordering of the sentence."
32    (interactive)
33    (reorder-sentence-mode -1)
34    (reorder-sentence--remove-overlays)
35    (set-window-configuration
36     reorder-sentence--previous-window-configuration))
```

One thing which may not be optimal here is the fact that we need to call `reorder-sentence--remove-overlays` twice – once in `reorder-sentence-finish` and once in `reorder-sentence-cancel`. One way to avoid this repetition would be to put it in the code for `reorder-sentence-mode`, together with the `(setq buffer-read-only nil)` part. I didn't do it because I just prefer to have that code this way.

A careful reader will also notice that we actually added *two* new properties to every overlay we create. Indeed, while the `reorder-sentence` property is used to remove all overlays we have created, we will use the `reorder-sentence-key` one to find the right overlay when a key is pressed. But that is the topic of the next section.

Making the word selection easier – handling the keys

We are now going to make suitable keys actually perform the right thing, i.e., copy the words to the temporary buffer. Previously, we wrote the `reorder-sentence-copy-word-at-point` for that. We will leave it as a command (just so that our mode will still be usable – even if in a very inconvenient way – when the region contains more than 62 words). We will, however, write another command and bind it to every key corresponding to a word with an overlay. This means that – first of all – it will need to know what key was pressed to run it. This is of course similar to what `self-insert-command` (normally bound to most "normal" keys, like letters, digits etc.) does. It would be great to analyze its source code to learn how it does its trick, but here is the issue: it is not written in Elisp, but in C, and we are not learning C in this book. It turns out, however, that even without C knowledge, it is easy to guess how to do a similar thing in Elisp. Here is a snippet of the source of `self-insert-command`:

```
1  if (NILP (c))
2    c = last_command_event;
```

You don't need a deep C understanding to guess that Elisp has a corresponding `last-command-event` variable. If the last command was invoked with a character or a character sequence, this variable contains the last character typed to invoke it.

So, our `reorder-sentence-copy-word` should iterate over all `reorder-sentence`-related overlays until it finds the one with the matching key, move the point there and run `reorder-sentence-copy-word-at-point`. Unsurprisingly, "iterating over a list to find a matching element" is such a common operation that Elisp has a function to do exactly that: `seq-find`. If you look up its docstring, you will see that its first argument is a so-called *predicate* – recall that this means a function that accepts something (in this case, an element of the searched list) and returns a Boolean value. Our predicate should accept an overlay (since we will feed `seq-find` with the list of all overlays within the reordered region) and check if its `reorder-sentence-key` property is equal to `last-command-event`. Instead of defining a separate function for that with `defun`, we are going to use an *anonymous function* for that. An anonymous function is exactly what is says on the tin – a function that is not *named*. Such functions are created using the `lambda` special form (actually, it is a macro using the more primitive `function` special form, but we don't need to know that). For instance, here is an anonymous function which accepts one argument – a number – and decides if it is even:

```
(lambda (integer)
  (zerop (mod integer 2)))
```

(The reasons this is called `lambda` and not e.g. `function` or anything else are fascinating[13], but we will not spend time discussing them here.)

You can use this function to find the first even number on a list:

```
(seq-find (lambda (integer)
            (zerop (mod integer 2)))
          '(1 3 4 6 8))
```

Of course, this example is a bit contrived – you could just as well call this function `evenp`:

```
(defun evenp (integer)
  "Return t if INTEGER is zero."
  (zerop (mod integer 2)))
```

and say `(seq-find #'evenp '(1 3 4 6 8))`, but there are cases when anonymous functions (or *lambda expressions*, as they may be called) are genuinely useful. (We will encounter such a situation in the next chapter.)

[13] https://en.wikipedia.org/wiki/Lambda_calculus#Origin_of_the_lambda_symbol

```
(defun reorder-sentence-copy-word ()
  "Copy word designated by the key used to invoke this function."
  (interactive)
  (let ((key last-command-event))
    (when (characterp key)
      (save-excursion
        (goto-char
         (overlay-start
          (seq-find (lambda (overlay)
                      (eq (overlay-get overlay 'reorder-sentence-key)
                          key))
                    (overlays-in reorder-sentence--begin reorder-sentence--end))))
        (reorder-sentence-copy-word-at-point)))))
```

What we want to do now is to bind all the keys we have listed with `(reorder-sentence--generate-keys)` to `reorder-sentence-copy-word`. There are quite a few ways to do this. Probably the most natural one would be in the definition of `reorder-sentence-mode` – this would mean that we add these keys to `reorder-sentence-mode-map` *once* instead of doing it every time we start reordering stuff. This means that we need to add 62 items to the `:keymap` alist. Of course, we won't do it by hand – that would be tedious, error-prone and not smart at all. However, I really like the shorthand way of defining the keymap with the backticks and unqouted `kbd` function. Is there any way we could still use it? It turns out that there is indeed. First of all, let us think how to convert the list returned by `(reorder-sentence--generate-keys)` to an alist suitable for inclusion in `:keymap`. Converting some list to another list of the same length, but with every item transformed in some uniform way is a *very* common task, and so Elisp has a very useful function designed precisely for that: `mapcar`. It accepts two arguments: a function to transform every element on the list and the list itself. (A programming style where we have a general function doing "something" e.g. with all the elements of a list, and we give it a function argument to say what the "something" is, is called "functional programming". Also, the description in the previous sentence is a vast oversimplification of what "functional programming" really is.) A classic example shown in numerous introductory texts on functional programming shows how to e.g. increase every number on the list by one:

```
(mapcar #'1+ '(1 2 3 4))
```

or double every number on the list:

```
(mapcar (lambda (arg) (* 2 arg)) '(1 2 3 4))
```

In our case we want to take every key on the list (say, `97`) and convert it to a cons like `(97 . reorder-sentence-copy-word)`. Easy enough:

```
(mapcar (lambda (key)
          (cons (kbd (string key))
                #'reorder-sentence-copy-word))
        (reorder-sentence--generate-keys))
```

(the `string` function takes a character – or more of them – and makes a string out of them).

Here, however, comes the trickier part. We have to somehow include this long alist of key-function pairs in the argument to `:keymap`. We cannot just precede the above expression with a comma like this:

```
`((,(kbd "C-c C-c") . reorder-sentence-finish)
  (,(kbd "C-c C-k") . reorder-sentence-cancel)
  ,(mapcar (lambda (key)
             (cons (kbd (string key))
                   #'reorder-sentence-copy-word))
           (reorder-sentence--generate-keys)))
```

(try this and see what happens!). This would create a three-element list: the first two elements would be the conses corresponding to `C-c C-k` and `C-c C-c`, and the last, third one would be the whole freshly generated alist. What we want instead is a long list with 64 elements – the two explicit conses first and the remaining 62 generated ones next. In other words, instead of what a comma does ("evaluate this expression and put the result here") we need somehow to tell Emacs to evaluate this expression (assuming that the result is a *list*) and put *all its elements here* (as opposed to the list as a whole). Luckily, this is another common need in Elisp programs, and Elisp has a way to do it very conveniently, using so-called *splicing*. The only thing we have to change in the above code to fix it is to use `,@` instead of just `,`:

```
(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
            (,(kbd "C-c C-k") . reorder-sentence-cancel)
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              #'reorder-sentence-copy-word))
                      (reorder-sentence--generate-keys)))
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

Note that for this to work, we have to make sure that the `reorder-sentence--generate-keys` is defined *before* the definition of the mode in the source file. This is because it is going to

be evaluated when the mode is defined, not when it is actually turned on. This is sometimes tricky: `define-minor-mode` is a macro, and macros can accept Elisp code and either immediately evaluate it (like what happens with the `:keymap` argument above), or do something else with it (like what happens with the "body" if the mode – in our case the `if` expression – which is stored in the command connected with the mode and evaluated when the mode is turned on or off). Forgetting about the moment when something is evaluated is pretty easy. In this case, at first I put the definition of `reorder-sentence--generate-keys` later in the file than the mode definition. Everything was ok during the development: I evaluated the `defun` form first so that I could play around with key generation, and *then* I evaluated the new version of the mode definition, with the key generation function already defined. When evaluating the whole file in a fresh Emacs session, however, I got an error with the message `mapcar: Symbol's function definition is void: reorder-sentence--generate-keys` – no wonder, given that `mapcar` wanted an argument computed by a not yet existing function! You should always test your code in a fresh Emacs session.

Try to reorder some sentence now, and you can see that we're getting close – it *almost* works! You can press all these keys and the words get correctly copied to the temporary buffer, `C-c C-k` cancels everything and `C-c C-c` installs the changes in the buffer we started in. However, there are still a few problems. The most prominent one is that if we press a key which is *not* assigned to any word, we get a `Wrong type argument: overlayp, nil` error. The reason is pretty obvious – Emacs looks for the non-existent overlay, so `seq-find` returns nil (which is sensible), and trying to call `(overlay-start nil)` doesn't make sense. The remedy is also pretty obvious – we should check if `seq-find` actually returns an overlay, and only perform the rest of the code if it does. This, however, raises two more questions. First of all – does it make sense to have all the keys bound to `reorder-sentence-copy-word` even though the actual sentence (or region) may have much fewer words? I'd argue that yes, for the reasons I've already explained – it seems better for me to define the keymap once instead of every time we start to reorder words. Also, if we did it the other way, it would mean that any key not bound to `reorder-sentence-copy-word` would trigger a `Buffer is read-only` error, which is not good, either.

This, however, means that we need to answer the second question – what should happen in such a circumstance? I can imagine three things. One would be just "do nothing". That makes sense, but provides no feedback to the user that something was wrong. If the user accidentally presses `3` instead of `e` and does not realize their mistake immediately, it basically means that they need to press `C-c C-k` and start anew. Not good. This leads to the second choice – just call the `ding` function, which makes a "beep" sound. Much better, but there is an even better way. We can use the `error` function. It takes the same arguments as the `message` function (a string first, with possible placeholders like `%s`, and the arguments to print in place of the placeholders), and signals an error. This basically means exiting whatever code is currently being evaluated and going back to the main command loop (i.e., making Emacs wait for the user's actions), with an error message *and* a beep – exactly what we need here.

```
(defun reorder-sentence-copy-word ()
  "Copy word designated by the key used to invoke this function."
  (interactive)
  (let ((key last-command-event))
    (when (characterp key)
      (let ((overlay
             (seq-find
              (lambda (overlay)
                (eq (overlay-get overlay 'reorder-sentence-key)
                    key))
              (overlays-in
               reorder-sentence--begin reorder-sentence--end))))
        (if overlay
            (save-excursion
              (goto-char
               (overlay-start overlay))
              (reorder-sentence-copy-word-at-point))
          (error "Key `%c' does not correspond to any word" key))))))
```

Notice how we put `save-excursion` inside the then-branch of the `if` and not outside. This is deliberate – first of all, there is no point in even remembering the point position if we are not going to change it, and – perhaps more importantly – we have two functions to evaluate within the then-branch (`goto-char` and `reorder-sentence-copy-word-at-point`), so we need a way to wrap them in one form (like with `progn`), and `save-excursion` fulfills this purpose perfectly.

Dealing with punctuation

Well, we finally have code which starts to be actually useful. Great, isn't it? Still, there are things we could do to make it even better. The first one we'll tackle is dealing with punctuation. Right now, the reordered sentence (or region) just doesn't have any, and reinserting it manually defeats the purpose of a smooth editing experience. How are we going to deal with that? For starters, let us write a command which inserts whatever was pressed to invoke it in the `*Reorder sentence*` buffer and bind it to some common punctuation characters.

```
(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
            (,(kbd "C-c C-k") . reorder-sentence-cancel)
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              #'reorder-sentence-copy-word))
                      (reorder-sentence--generate-keys))
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              #'reorder-sentence-insert-punctuation))
                      reorder-sentence-punctuation-characters))
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))

(defun reorder-sentence-insert-punctuation ()
  "Insert the key typed in the sentence-reordering buffer."
  (interactive)
  (save-restriction
    (with-current-buffer reorder-sentence--buffer
      (goto-char (point-max))
      (insert last-command-event))))

(defvar reorder-sentence-punctuation-characters "!\"'(),-.:;?"
  "Characters which should self-insert.")
```

One (very minor) issue with the above code is that in some languages (lke my native Polish), some punctuation requires spaces around it (like a dash), and the code above does not support it. Adding a space to `reorder-sentence-punctuation-characters` won't work (in fact, this was the first thing I tried!), since `(kbd " ")` yields an empty string. This is no surprise, since space is used to separate keys in sequences described as keys and handed to `kbd`. As I mentioned when introducing `kbd`, it is easiest to type `C-h c` followed by a space to learn that the *name* of the space character suitable to give to `kbd` is `SPC`. On the other hand, `(kbd "SPC")` evaluated to the string containing just the literal space, so we can also use `" "` directly, without `kbd`. Personally, I prefer `kbd` anyway, since I find `(kbd "SPC")` more explicit.

```
(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
            (,(kbd "C-c C-k") . reorder-sentence-cancel)
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              #'reorder-sentence-copy-word))
                      (reorder-sentence--generate-keys))
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              #'reorder-sentence-insert-punctuation))
                      reorder-sentence-punctuation-characters)
            (,(kbd "SPC") . reorder-sentence-insert-punctuation))
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

Perhaps a better way would be to let `reorder-sentence-punctuation-characters` contain a literal space, but use some custom function instead of `kbd` to recognize a space and use it literally instead of giving it to `kbd` first. That way allowing space or not as a "punctuation" character would be configurable. This is left as an exercise to the reader.

A completely different way to deal with binding `reorder-sentence-insert-punctuation` would be to bind it to every printable ASCII character not bound to `reorder-sentence-copy-word`. Of course, iterating over all printable characters and using `seq-find` to check if a particular character belongs to `(reorder-sentence--generate-keys)` would be extremely inefficient – we would generate the whole list of "copying" keys for every character checked, and `seq-find` would have to walk through this list until it finds the character being checked in every iteration of the loop. One way to do it would be to iterate over *all* printable ASCII characters when defining the keymap and use another function, checking if the given character falls into the "copying" range or not. We could for example use this function:

```
(defun reorder-sentence--copying-key-p (key)
  "Return t if KEY should copy the corresponding word."
  (or (<= 48 key 57)
      (<= 65 key 90)
      (<= 97 key 122)))
```

Notice how we used the convention of attaching `-p` to the function name if it returns a Boolean value (if the function name is one word, the hyphen is usually dropped and a single letter `p` is attached). Also, the fact that the `<=` function can take an arbitrary number of arguments (even one, by the way!) turns out to be quite useful here.

Finally, let us use the `range` function described earlier and rewrite the code generating the keymap:

```
(defun reorder-sentence--range (a b)
  "Return a list of numbers starting with A and ending before B."
  (let ((list ())
        (counter a))
    (while (< counter b)
      (push counter list)
      (setq counter (1+ counter)))
    (nreverse list)))

(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
            (,(kbd "C-c C-k") . reorder-sentence-cancel)
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              (if (reorder-sentence--copying-key-p key)
                                  #'reorder-sentence-copy-word
                                #'reorder-sentence-insert-punctuation)))
                      (reorder-sentence--range 32 127)))
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

One thing worth mentioning here is the fact that we prefixed the `range` function with `reorder-sentence--`. One might think that `range` is a general-purpose function, potentially useful elsewhere, so why not use the name `range` instead? Well, we could, of course, but that is not a good practice. What if the user defined their own `range` function working differently, e.g., generating numbers up to *and including* its second argument? Or having an optional third `step` argument, enabling to construct arbitrary arithmetic sequences? Since Elisp does not have proper "private" functions (i.e., ones only visible to the code of some package, for instance), we need to rely on prefixes (like `reorder-sentence-`) to make sure there are no conflicting functions with the same name defined in different packages (or in a package and the user's `init.el` file).

Someone may also ask how did I know that *every* single printable character works as the car of the alist given to `:keymap`. Well, this is indeed not obvious – I didn't. (The manual doesn't say that explicitly.) But here is how one could learn that.

```
(dolist (key (reorder-sentence--range 33 127))
        (when (not (string= (string key) (kbd (string key))))
          (message "Problem with key `%c'." key)))
```

This made me sure that for every character in the ASCII range from `!` (code 33) to ~ (code 126), the `kbd` function taking a one-element string made of that character yields the very same string. Here, I used a quite convenient `dolist` macro, which – given a symbol (`key`) and a list of values (produced by our `reorder-sentence--range` function) executes the given body once for every value in the list and arranges it so that the given symbol is bound to values from the list during the subsequent iterations. (It also returns nil, although that can be overriden – see the Elisp reference for an example of that.)

And by the way, the moment has come when the "simple solution" of just going to `(point-max)` after inserting anything bites us back. It turns out that the window holding the reordering buffer always shows the point at the very beginning. Until we could insert literal spaces there, this – while maybe not *elegant* – was not very bothering, since it was pretty obvious that everything gets added at the end of that buffer. Now that we can add a space (which is useful before a punctuation character that needs a space before, like a dash), it would be really nice to *see* if a space was added or not. This means that we need to move the window point after all. There are a few changes in the code needed. First of all, we need access to that very window. This is fairly easy – `M-x apropos-function RET get.*window RET` tells us about the `get-buffer-window` function. Having that, we only need to delete the two `(goto-char (point-max))` instances and call `set-window-point` right after inserting a word (or a punctuation character).

```
(defun reorder-sentence-copy-word-at-point ()
  "Copy the word at point to the sentence reordering buffer."
  (interactive)
  (save-restriction
    (narrow-to-region reorder-sentence--begin reorder-sentence--end)
    (let ((word (current-word)))
      (with-current-buffer reorder-sentence--buffer
        (when (not (bobp))
          (insert " "))
        (insert word)
        (set-window-point (get-buffer-window reorder-sentence--buffer)
                          (point-max))))))

(defun reorder-sentence-insert-punctuation ()
  "Insert the key typed in the sentence-reordering buffer."
  (interactive)
  (save-restriction
    (with-current-buffer reorder-sentence--buffer
      (insert last-command-event)
```

```
20        (set-window-point (get-buffer-window reorder-sentence--buffer)
21                          (point-max)))))
```

Avoiding duplication in code

Now, this certainly works, but there is still one thing that bugs me a lot. We now have two functions – `reorder-sentence--generate-keys` and `reorder-sentence--copying-key-p` – that contain the logic for deciding which characters should correspond to words and which ones should self-insert. This is very bad. What if we decided to add something to the sequence of lower-case letters, digits and upper-case letters, for instance? We would have to remember to make a similar change in two separate places. This violates a famous programming principle called "DRY" ("don't repeat yourself") – we shouldn't have two places in the code duplicating each other.

The problem is, we want both of them – one (`reorder-sentence--generate-keys`) to encode the order in which subsequent keys are associated with subsequent words, and the other (`reorder-sentence--copying-key-p`) to allow for fast deciding if a given key is associated with a word or not. So, here is my idea: let's encode the actual sequence (which is precisely the thing that appears in both these functions) in a dedicated place and make these two functions use the same data.

```
1  (defvar reorder-sentence--word-copying-key-ranges
2    '((97 . 122)
3      (48 . 57)
4      (65 . 90)))
```

We used conses ("dotted pairs") since it makes extracting both endpoints of every range very easy – you just take the `car` or the `cdr` of the pair – but we could have used two-element lists as well (and extract the endpoints with `car` and `cadr`, which is a shorthand for taking the `car` of the `cdr` of its argument: `(cadr '(1 2))` is the same as `(car (cdr '(1 2)))`). Now, we need to be able to "expand" the list above to an union of ranges. This looks pretty general, so let's write a function to do just that. We will use the `dolist` macro again and start with generating a list of sequences of numbers for further concatenation. Here is this first step:

```
1  (defun reorder-sentence--generate-sequence (ranges)
2    "Generate a sequence of numbers from RANGES.
3  It is a list of dotted pairs containing the first and last
4  element of a range."
5    (let ((result))
6      (dolist (range ranges)
7        (push (reorder-sentence--range (car range) (1+ (cdr range)))
8              result))
9      (nreverse result)))
```

This is not yet what we want – for example, `(reorder-sentence--generate-sequence '((1 . 2) (5 . 7)))` yields `((1 2) (5 6 7))` instead of `(1 2 5 6 7)`. There are at least two ways to make the former into the latter. One is the `flatten-list` function (which incidentally is an alias for `flatten-tree`, since nested lists are sometimes used to represent so-called "trees"). This function returns a list of all "leaves" (non-list elements, and elements of elements, and elements of elements of elements etc. of the list given as argument) – in other words, it "deletes all parens except the outermost ones", so to speak. Another way is to use the `append` function, which – given several lists as arguments – returns a list resulting from concatenating them. The latter approach seems simpler (in fact, a few experiments I performed showed that in our case it is also faster by an order of magnitude!), but there is one catch: we cannot just say `(append (nreverse result))` instead of `(nreverse result)` in the definition above, since `append` does not expect *one* argument being a list of lists to concatenate, but rather *several* arguments. Luckily, we know the solution to that – the `apply` function. So, here is the working `reorder-sentence--generate-sequence` function and the new version of `reorder-sentence--generate-keys`.

```
(defun reorder-sentence--generate-sequence (ranges)
  "Generate a sequence of numbers from RANGES.
It is a list of dotted pairs containing the first and last
element of a range."
  (let ((result))
    (dolist (range ranges)
      (push (reorder-sentence--range (car range) (1+ (cdr range)))
            result))
    (apply #'append (nreverse result))))

(defun reorder-sentence--generate-keys ()
  "Return a list of characters to use for sentence reordering."
  (reorder-sentence--generate-sequence reorder-sentence--word-copying-key-ranges))
```

Now we need to teach `reorder-sentence--copying-key-p` to use `reorder-sentence--word-copying-key-ranges`. Given a number and a list of conses, we want to know if the number belongs to any of the (closed, i.e., including endpoints) intervals defined by these conses. Previously, we used the `or` special form and just hard-coded the intervals in the function. This had the advantage that if the number fell in any of the intervals, the subsequent ones were not even checked (this is how `or` works: if any of its arguments is non-nil, the subsequent ones are not even evaluated). It would be nice to preserve that feature (of course, this doesn't really affect performance in any visible way, since we only have three intervals – it's rather a question of good design and learning how to implement this). That's why we won't use the `dolist` macro, which does not allow an "early exit" (i.e., exiting before all the elements are processed). Well, we could use `catch`/`throw` again, but this is far from elegant. We could also use `cl`'s loops which actually have provisions for early exits. As mentioned earlier, we will prefer to avoid depending on `cl` (not because it is bad, but because we want to show some other technique!) and show a recursive solution instead. (One can argue that recursion is something we should strive to avoid, since it is often memory-expensive, i.e., it can use up large amounts of

memory. If I expected using this function with lists having dozens or hundreds of intervals, I would certainly go for a solution using `cl-dolist` to iterate over the list and `cl-return` to stop iterating once the interval containing the given number is found. Here I want to show an elegant – even if not optimal – recursive trick.)

```
(defun reorder-sentence--number-in-any-interval-p (number intervals)
  "Return t if NUMBER is in any of the INTERVALS and nil otherwise.
INTERVALS is a list of conses, each describing a closed interval."
  (when intervals
    (or (<= (caar intervals) number (cdar intervals))
        (reorder-sentence--number-in-any-interval-p
         number
         (cdr intervals)))))

(defun reorder-sentence--copying-key-p (key)
  "Return t if KEY should copy the corresponding word."
  (reorder-sentence--number-in-any-interval-p
   key
   reorder-sentence--word-copying-key-ranges))
```

If `intervals` is nil, `reorder-sentence--number-in-any-interval-p` returns nil (this ensures that the recursion will stop). Otherwise, the number is in any of the intervals if it is in the first interval (notice how we use `(caar intervals)` instead of a longer `(car (car intervals))` and similarly with `cdar`) or in any of the *rest* of the intervals (i.e., we recursively call the same function with `(cdr intervals)` in place of `intervals`).

Dealing with capitalization

Of course, there are still a few things we could do better. One of them is the capitalization. If we reorder a whole sentence, and we decide to change the *first* word of it, we probably want some other word to start with an uppercase letter, and the former first word to start with a lowercase one. While not very difficult to *code*, the hardest part here is the UI – how is the user going to tell Emacs to do that? Well, we could make the first word uppercase automatically, but making all the subsequent ones lowercase could be risky – after all, many words are capitalized even in the middle of a sentence. My first idea was to use the prefix argument (`C-u`) to `reorder-sentence-copy-word` to tell Emacs to change the capitalization of the first letter. That could work, but people might find it a bit inconvenient. Then a nice idea occurred to me. Emacs already has a command to make the first letter of a word uppercase – `capitalize-word`, bound to `M-c` by default. My muscle memory knows that, and it also knows to press `M-- M-c` to capitalize the *previous* word when I somehow forget to capitalize it when I type it. Of course, we can't just use that command without any provisions, since during reordering we are in a read-only buffer, but we can reuse the *keybinding* to make *our* case-fiddling commands a bit more intuitive. Here is my idea: if a user presses `M-c` (or `M-l`) while

reordering, the *next* word copied should be capitalized (or "downcased"). If the user presses `M-- M-c` (or `M-- M-l`), the previously copied word should. This gives a way of both telling Emacs to do the right thing with the next word *and* correcting a mistake with the previous one. Actually, I can see no reason not to support an arbitrary numeric argument with these commands – this could be especially useful with negative arguments. (Indeed, if I forgot to capitalize a word three words ago, I could say `M-- M-3 M-c M-- M-2 M-l` to first capitalize the last three words and then downcase the last two ones.)

Implementing this feature for the previous `N` words should be fairly easy – we can just delegate the actual work to the "real" `capitalize-word` and `downcase-word` (making a short excursion to the `*Reorder sentence*` buffer first), but "marking the next `N` words for case change" is more involved. The reason is obvious – when the user presses the corresponding key, we need to store the information for the future. Since we decided that only one reordering session can be active at the same time, we will simply use yet another global variable for that.

I know, I know. Global variables are bad and so on. (Indeed, too many of them can make the program harder to understand and maintain. Even one may introduce complexity if it is accessed – and possibly changed – by many functions.) Seriously though, of course, using – or actually, abusing – global variables is a *very* bad practice in general, but let's think about it. In this case, we need to have some state – the number of words we need to capitalize *in the future* – which has to be carried over between totally separate commands. I know two good ways of handling that. One of them is using *closures* – something we haven't talked about yet – and the other is what so-called object-oriented languages would call *instance variables*. Imagine we allowed more than one reordering session at once (an idea we explicitly dropped) – then, every such session would have its own "instance" of the counter saying how many of the next words should be capitalized. In our case we only need one such counter. This means that what we really want is a "package-local variable" – only visible to functions defined within the "reorder-sentence" package. Since Emacs doesn't have such a notion, the next closest thing is to define a variable whose name starts with `reorder-sentence--` (notice the two dashes, meaning this is not something the end-user should care about). So, we will use (technically) a global variable, but it will be closer in spirit to a variable local to our package (which doesn't sound *that* bad).

Before we actually write the capitalizing code, we need to learn one thing. Let's assume that we know the `capitalize-word` function (if not, check out `C-h k M-c`!), and – as mentioned above – we will actually use it, but we will also need to capitalize a *string* (i.e., a word we are about to insert if the user said just `M-c` in `reorder-sentence-mode`, possibly with an argument greater than one). Of course, Emacs' self-documenting nature helps here. Pressing `C-h f capitalize TAB` tells us about a few functions with the word `capitalize` in them, one of them being – well – just `capitalize`, which – given a (possibly multi-word) string – turns the first letter of every word in it into a capital version, and the remaining letters into a lower-case version. We can use that (actually, we won't even use its full potential, since – by design – our string will be one word only).

Also, we have already decided what to do if the prefix argument to `M-c` (which we are going to bind to `reorder-sentence-capitalize-word`) is negative (capitalize the last `-N` words inserted into the `*Reorder sentence*` buffer) or positive (capitalize the next `N` words to be inserted there) – but what

if the argument is zero? Is there anything reasonable we could do then? Well, of course there is – we can just cancel the "capitalize the next `N` words" feature. (It is especially nice that it doesn't require any additional code – in fact, it is just a special case of setting that counter.)

And here's the code.

```
(defvar reorder-sentence--capitalize-count 0
  "How many next words should be capitalized.")

(defun reorder-sentence-capitalize-word (count)
  "Capitalize next COUNT or previous -COUNT words."
  (interactive "p")
  (cond ((>= count 0)
         (setq reorder-sentence--capitalize-count count))
        ((< count 0)
         (with-current-buffer reorder-sentence--buffer
           (goto-char (point-max))
           (capitalize-word count)))))
```

Also, we need to add a line saying

```
(,(kbd "M-c") . reorder-sentence-capitalize-word)
```

to the keymap definition in the `define-minor-mode` form. And of course, to actually *use* the counter we defined, we need this right at the beginning of the `(let ((word (current-word))) ...)` form:

```
(when (> reorder-sentence--capitalize-count 0)
  (setq word (capitalize word))
  (setq reorder-sentence--capitalize-count (1- reorder-sentence--capitalize-count)))
```

(Note: I don't like showing only snippets of code instead of complete functions in this book, but I made an exception here, since in a few minutes we will rework these parts, and I wanted to avoid wasting space for two pieces of very similar code next to each other. Remember that the Git repository accompanying this book contains all the code.)

And finally, let's not forget about adding `(setq reorder-sentence--capitalize-count 0)` to both `reorder-sentence-finish` and `reorder-sentence-cancel` – otherwise the capitalizing feature could "spill over" to the next reordering session, which is definitely not what we want. (If you are worried that you'd never think about it – don't. I didn't, either, when I first wrote the code, until later. Think about it this way: if we'd forgot about it, and the bug manifested itself, it would be pretty obvious what's wrong, and we would fix it then. Also, this isn't the kind of bug that leads to permanent loss of someone's data, so it's definitely far from catastrophic.)

Now of course there is no symmetric feature for downcasing the next (or previous) `N` words, so let's introduce it now. There is one thing we need to think about for a second: do we need another counter

for downcasing words or could we just use the same one but with the opposite sign? I see arguments both ways. On the one hand, it seems that we shouldn't use the variable with `capitalize` in its name to hold information about *downcasing*. On the other hand, introducing two separate variables would mean that whenever we set one of them to a positive value, we would have to remember to set the other one to zero, and a state when *both* of them are non-zero wouldn't make any sense. After a short consideration, I went for the easy route of using the same variable with the convention that if its *negative*, we should downcase the next few words. One drawback is that if we ever wanted to add a feature to make a word all-caps, there would be no way to incorporate that information into our variable. I've decided, though, that this is pretty unlikely – the main use of this feature is changing the case of the first letter when some word becomes the first word of the reordered sentence (or ceases to be te first one). I'd assume that if something should be written in all caps, it doesn't depend on its position in the sentence. Another issue I have with this idea is that it feels "hackish" or "just a bit too clever" – and I'm usually afraid of such ideas. I mean, using "clever" tricks like this does not help anyone *reading* such code later. It's often the case that such shortcuts will come back and bite us later, and even if not, we'll always feel a crippling sense of brittleness of code written this way. Well, this time, it's a risk I'm willing to take.

So again, the code, finally in the form of complete functions.

```
(defun reorder-sentence-copy-word-at-point ()
  "Copy the word at point to the sentence reordering buffer."
  (interactive)
  (save-restriction
    (narrow-to-region reorder-sentence--begin reorder-sentence--end)
    (let ((word (current-word)))
      (cond ((> reorder-sentence--capitalize-count 0)
             (setq word (capitalize word))
             (setq reorder-sentence--capitalize-count (1- reorder-sentence--capitalize-coun\
t)))
            ((< reorder-sentence--capitalize-count 0)
             (setq word (downcase word))
             (setq reorder-sentence--capitalize-count (1+ reorder-sentence--capitalize-coun\
t))))
      (with-current-buffer reorder-sentence--buffer
        (when (not (bobp))
          (insert " "))
        (insert word)
        (set-window-point (get-buffer-window reorder-sentence--buffer)
                          (point-max))))))

(defun reorder-sentence-finish ()
  "Finish the reordering of the sentence.
Replace the selected region with the constructed sentence and restore
```

```
25  the window configuration."
26    (interactive)
27    (reorder-sentence-mode -1)
28    (reorder-sentence--remove-overlays)
29    (setq reorder-sentence--capitalize-count 0)
30    (goto-char reorder-sentence--begin)
31    (delete-region reorder-sentence--begin reorder-sentence--end)
32    (insert-buffer-substring reorder-sentence--buffer)
33    (set-window-configuration
34     reorder-sentence--previous-window-configuration))
35
36  (defun reorder-sentence-cancel ()
37    "Cancel the reordering of the sentence."
38    (interactive)
39    (reorder-sentence-mode -1)
40    (reorder-sentence--remove-overlays)
41    (setq reorder-sentence--capitalize-count 0)
42    (set-window-configuration
43     reorder-sentence--previous-window-configuration))
44
45  (define-minor-mode reorder-sentence-mode
46    "Easily reorder a sentence or region."
47    :lighter " Reorder sentence"
48    :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
49              (,(kbd "C-c C-k") . reorder-sentence-cancel)
50              (,(kbd "M-c") . reorder-sentence-capitalize-word)
51              (,(kbd "M-l") . reorder-sentence-downcase-word)
52              ,@(mapcar (lambda (key)
53                          (cons (kbd (string key))
54                                (if (reorder-sentence--copying-key-p key)
55                                    #'reorder-sentence-copy-word
56                                  #'reorder-sentence-insert-punctuation)))
57                        (reorder-sentence--range 32 127)))
58    :interactive nil
59    (if reorder-sentence-mode
60        (setq buffer-read-only t)
61      (setq buffer-read-only nil)))
62
63  (defvar reorder-sentence--capitalize-count 0
64    "How many next words should be capitalized.
65  If negative, downcase the next words.")
66
67  (defun reorder-sentence-capitalize-word (count)
```

```
  "Capitalize next COUNT or previous -COUNT words."
  (interactive "p")
  (cond ((>= count 0)
         (setq reorder-sentence--capitalize-count count))
        ((< count 0)
         (with-current-buffer reorder-sentence--buffer
           (goto-char (point-max))
           (capitalize-word count)))))

(defun reorder-sentence-downcase-word (count)
  "Downcase next COUNT or previous -COUNT words."
  (interactive "p")
  (cond ((>= count 0)
         (setq reorder-sentence--capitalize-count (- count)))
        ((< count 0)
         (with-current-buffer reorder-sentence--buffer
           (goto-char (point-max))
           (downcase-word count)))))
```

Undoing

This way we've arrived at the moment that our code is (almost) production-ready. It is fully usable (and I'd argue that it's quite useful!) – but that does not mean that it can be improved. There is a saying among programmers that "finished software" is like a "mown lawn". (Ponder on that!) We will resist the temptation to add more and more features, but we still need a few more things – the first of them pretty essential and the rest pretty useful. The essential one is undoing. Emacs has a great (even if atypical by today's standards) undo system, and it's a pity `reorder-sentence` doesn't support it. When we copy the wrong word to the `*Reorder sentence*` buffer, we can't easily undo that action – we can either `C-c C-k` and start anew or switch to that buffer, undo there and go back again. Invoking the `undo` command in the original buffer will cause an error – and rightly so, since we made it read-only.

It's pretty obvious what we could – and probably should – do at this point. Let's define a command which will work like `undo`, but in the `reorder-sentence-mode`.

```
(defun reorder-sentence-undo (count)
  "Undo COUNT changes while reordering a sentence."
  (interactive "p")
  (with-current-buffer reorder-sentence--buffer
    (undo count)))
```

However, we would also like to bind this to some key, and that's when we encounter a problem. By default, the undo command is bound to quite a few keys: `C-_`, `C-/`, and `C-x u`. It would be a

nuisance to rebind all these keys. Luckily, there's a nice way around, called *command remapping*. We can define a key with a special syntax `[remap <command-name>]` (this is actually something called a *vector*, a bit similar to a list, but different – and it is used, among others, to represent key sequences internally). This means that if any key is bound to `undo` in our keymap, pressing it will call `reorder-sentence-undo` instead. Moreover, this will happen even if some key is bound to `undo` *after* the `[remap undo]` part is put into `reorder-sentence-mode-map`. (Note that command remapping is not without its quirks – as usual, the Elisp reference is the good source to learn about them.)

One more thing worth doing is binding `reorder-sentence-undo` to the "backspace" key – since it would be useless anyway, and the user can press at least *some* keys just to insert them (even if not in the current buffer), it seems a very natural thing to do. (Notice the pattern here. What we are trying to do is to make our package play nicely with the existing Emacs infrastructure. The way we reused `M-c` and `M-l` makes it almost possible to *guess* how to capitalize the next or last `N` words. The ideal Emacs package does "the right thing" when the user presses a key which performs some action in stock Emacs, even if that action itself is impossible and the package substitutes some kind of an analogue or equivalent of it.)

```
(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
            (,(kbd "C-c C-k") . reorder-sentence-cancel)
            (,(kbd "M-c") . reorder-sentence-capitalize-word)
            (,(kbd "M-l") . reorder-sentence-downcase-word)
            ([remap undo] . reorder-sentence-undo)
            (,(kbd "<backspace>") . reorder-sentence-undo)
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              (if (reorder-sentence--copying-key-p key)
                                  #'reorder-sentence-copy-word
                                #'reorder-sentence-insert-punctuation)))
                      (reorder-sentence--range 32 127)))
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

Marking words already copied and making copying faster

One thing I don't like a lot with the current state of affairs is that it is not easy to see which words were already copied. If I want to reorder 3 or 4 words, that's not really needed, but when I need to reorder a list of 8 words, this could be pretty useful. Also, it is very easy to implement.

```
(defface reorder-sentence-inactive
  '((t :inherit 'shadow :strike-through t))
  "Face to display already copied words.")

(defun reorder-sentence-copy-word ()
  "Copy word designated by the key used to invoke this function."
  (interactive)
  (let ((key last-command-event))
    (when (characterp key)
      (let ((overlay
             (seq-find
              (lambda (overlay)
                (eq (overlay-get overlay 'reorder-sentence-key)
                    key))
              (overlays-in
               reorder-sentence--begin reorder-sentence--end))))
        (if overlay
            (save-excursion
              (goto-char
               (overlay-start overlay))
              (reorder-sentence-copy-word-at-point)
              (overlay-put overlay 'face 'reorder-sentence-inactive))
          (error "Key `%c' does not correspond to any word" key))))))
```

As you can see, all I did was defining a new face (almost completely identical to the `shadow` face defined by Emacs, but with the addition of a strike-through effect – thanks to Christian Tietze for this suggestion!) and adding `(overlay-put overlay 'face 'reorder-sentence-inactive)` in the right place. Of course, this is very simple – in particular, we don't bother to remove the key "markers" from the overlays. I would argue that it is a good thing, for two reasons. First of all, removing them would make the whole text on the right of the removed "marker" to "jump" (i.e., move to the left, which would be very uncomfortable for the eyes). More importantly, we can use the fact that Emacs knows which words are "used up" to do something smart – make pressing the *same* key more than once copy subseqeuent words. This is a recurring theme here – we think about what user *could* do (for example, press a key which is not a letter nor a digit, press a key twice) and decide what Emacs *should* do in such a case to make the whole experience more intuitive and smooth.

This is easier said than done, though. Here is the issue. Up till now, we searched for the word to copy using the key pressed. To get what we want now we would need to do that first and then – if that word is "inactive" – step over the following words until we find the first one which is not "inactive". In particular, instead of finding the one overlay in the list which correspond to "our" key, we will need a list *starting* with that overlay (so that we have easy access to the subsequent ones). This means we can't use `seq-find` anymore. (Well, technically we *could*, but this would be not great. The correct predicate to use would be "the word must be active *and* its key must be greater or equal than

the key pressed", where "greater or equal" is understood with respect to the order defined by the `reorder-sentence--word-copying-key-ranges` variable. Writing a function corresponding to such "inequality" is a nice exercise, but it is a bit complicated. Also, it is much easier to find the word corresponding to the correct key first and then – if necessary – to iterate until we find one that is active.)

Let's start to code, then. First of all, we will now need our list of overlays sorted – because we need to be able to go from one overlay to the next one. One could suspect that `overlays-in` already returns a sorted list of overlays. Well it does, but… in the reverse order! Well, we could define them from the last one to the first one, but that still moght not be correct. The documentation of `overlays-in` says nothing about the order of the result, so we can't really rely on this. Maybe the list will be sorted in the normal order in a future version of Emacs and our code would break? It is much safer not to rely on things you found by experimentation or even by looking at the source code, but not present in the documentation – in fact, it would be wrong to rely on them. One thing we *could* do, though, would be to do both – sort the list *and* define the overlays in the reverse order. This way the list we would be going to sort would – usually at least – be already sorted, which makes sorting *faster*. Let's not bother with that, though – it would be a classic case of "premature optimization", i.e., spending time and effort on optimizing things which are probably not an issue. It isn't worth it to spend a lot of time optimizing something which will cut down the execution time of an interactive command from, say, half a millisecond to a third of a millisecond – nobody would ever notice the difference anyway.

Now here comes a delicate point. At first I thought that since the docs of `overlays-in` do not say that the list is created anew every time (of course, in general it must be, given that it depends on the arguments to `overlays-in`, but it would be conceivable that e.g. `(overlays-in (point-min) (point-max))` returns the exact same list – i.e., the address of the same place in memory – every time, e.g., because of some optimization), I didn't want to use `sort`, which is a destructive function. Emacs has `seq-sort`, which is very similar, with two differences – it copies its argument before sorting (so that the original is left intact), and its arguments are in reversed order (predicate first, list next) – go figure. (Also, if you check the docstring of `seq-sort`, it contains some mumbo-jumbo about "generic functions" and "implementations". Disregard that, it is beyond of the scope of this book, but you can read about it in the Elisp reference.) I asked about it on the Emacs mailing list[14], and it turned out that it is safe to use the built-in `sort`, so that's what we will do.

The next thing is marking words as already copied. This is actually pretty easy – we only need to give their overlays the right face. (We will determine if a given overlay is "inactive" based on whether it has this face.)

The gist is – of course – finding the right word to copy. As mentioned before, we will do it in two steps. First we find the overlay corresponding to the key pressed. We do it with a simple `while` loop. Since we may be going to need the following overlays, too, we do it by iterating over the (sorted) list and cutting the first element until we get to the right one, but then keeping the entire rest. (This is easy, using the `(setq overlays (cdr overlays))` idiom.) Note that since we check `(overlay-get (car overlays) 'reorder-sentence-key)` in each iteration, we must explicitly make

[14] https://lists.gnu.org/archive/html/help-gnu-emacs/2021-07/msg00756.html

sure that the `overlays` list is not empty. Once we have the right word, we start iterating again, this time to find the first *active* overlay. (In both loops it is possible to stop before the first `(setq overlays (cdr overlays))`, which is correct – after all, the first word can also be copied!) And that's it – it's actually not *that* difficult.

```
(defun reorder-sentence-copy-word ()
  "Copy word designated by the key used to invoke this function."
  (interactive)
  (let ((key last-command-event))
    (when (characterp key)
      (let* ((overlays
              (sort (overlays-in
                     reorder-sentence--begin reorder-sentence--end)
                    (lambda (o1 o2)
                      (< (overlay-start o1) (overlay-start o2)))))
             overlay)
        ;; find the overlay corresponding to `key'
        (while (and overlays
                    (not (eq (overlay-get (car overlays) 'reorder-sentence-key)
                             key)))
          (setq overlays (cdr overlays)))
        ;; find the first active overlay
        (while (and overlays
                    (eq (overlay-get (car overlays) 'face)
                        'reorder-sentence-inactive))
          (setq overlays (cdr overlays)))
        ;; copy the word
        (setq overlay (car overlays))
        (if overlay
            (save-excursion
              (goto-char
               (overlay-start overlay))
              (reorder-sentence-copy-word-at-point)
              (overlay-put overlay 'face 'reorder-sentence-inactive))
          (error "Key `%c' does not correspond to any word" key))))))
```

This looks pretty good, and copying subsequent words works fantastic, with one exception – the undo system. Now that we mark words as copied/inactive, undoing should unmark them as such. The trouble is, how do we know which one was the last word to copy? And, for that matter, which one was the second to last etc., in case the user wants to undo more of them? It seems we have no choice but to remember all copied words (and in the correct order!) so that we can back up from copying them. The obvious way is to use a *stack*. A *stack* is a data structure which can hold zero or more things and is capable of two operations: accepting one more thing ("pushing" it to the stack)

or giving out one thing ("popping" it from the stack). The gist is that a stack works on a "last in, first out" basis: if you push something to a stack and then pop something from it, you get the *last* thing you pushed. This is ideal for implementing an undo-like feature, since performing an action means pushing information about it to a stack (which then remembers it) and undoing it means popping the information needed to perform the "undo" operation from the stack. This way the first "undo" will undo the last operation, and subsequent ones will undo the second to last etc.

A natural way to implement a stack in Elisp is by using a list, of course. There are even two forms, appropriately called `push` and `pop` (note: they are macros and not functions, but this is not important now). For instance, we can do this:

```
(setq stack ())
(push 1 stack)
(push 2 stack)
```

and the variable `stack` will be bound to a list `(2 1)`. If we say then `(setq var (pop stack))`, `var` will be bound to `2` and `stack` to `(1)`; one `(setq var (pop stack))` more and `var` is `1` and `stack` is the empty list again.

Our task now is pretty simple: we need some variable holding our undo stack for "deactivated" overlays, every time we copy a word we push one of them to the stack, and every time we undo such a change we pop one. There is one caveat, though – an undo command might undo an insertion, and in such a case we need to push some special kind of marker to our stack so that we don't accidentally reactivate some overlay when undoing something else than word-copying.

```
(defvar reorder-sentence--undo-stack nil
  "The stack holding overlays corresponding to copied words.")

(defun reorder-sentence (beg end)
  "Reorder the words in the region."
  (interactive (if (use-region-p)
                   (list (region-beginning) (region-end))
                 (reorder-sentence--current-sentence-bounds)))
  (setq reorder-sentence--begin (copy-marker beg t)
        reorder-sentence--end (copy-marker end)
        reorder-sentence--previous-window-configuration (current-window-configuration)
        reorder-sentence--buffer (get-buffer-create reorder-sentence--buffer-name t))
  (reorder-sentence--create-overlays reorder-sentence--begin reorder-sentence--end)
  (setq reorder-sentence--undo-stack ())
  (with-current-buffer reorder-sentence--buffer
    (erase-buffer))
  (display-buffer reorder-sentence--buffer)
  (deactivate-mark)
  (reorder-sentence-mode 1))
```

```
20
21 (defun reorder-sentence-finish ()
22   "Finish the reordering of the sentence.
23 Replace the selected region with the constructed sentence and restore
24 the window configuration."
25   (interactive)
26   (reorder-sentence-mode -1)
27   (reorder-sentence--remove-overlays)
28   (setq reorder-sentence--capitalize-count 0)
29   (goto-char reorder-sentence--begin)
30   (delete-region reorder-sentence--begin reorder-sentence--end)
31   (insert-buffer-substring reorder-sentence--buffer)
32   (setq reorder-sentence--undo-stack ())
33   (set-window-configuration
34    reorder-sentence--previous-window-configuration))
35
36 (defun reorder-sentence-cancel ()
37   "Cancel the reordering of the sentence."
38   (interactive)
39   (reorder-sentence-mode -1)
40   (reorder-sentence--remove-overlays)
41   (setq reorder-sentence--capitalize-count 0)
42   (setq reorder-sentence--undo-stack ())
43   (set-window-configuration
44    reorder-sentence--previous-window-configuration))
45
46 (defun reorder-sentence-copy-word ()
47   "Copy word designated by the key used to invoke this function."
48   (interactive)
49   (let ((key last-command-event))
50     (when (characterp key)
51       (let ((overlays
52              (sort (overlays-in
53                     reorder-sentence--begin reorder-sentence--end)
54                    (lambda (o1 o2)
55                      (< (overlay-start o1) (overlay-start o2)))))
56             overlay)
57         ;; find the overlay corresponding to `key'
58         (while (and overlays
59                     (not (eq (overlay-get (car overlays) 'reorder-sentence-key)
60                              key)))
61           (setq overlays (cdr overlays)))
62         ;; find the first active overlay
```

```
        (while (and overlays
                    (eq (overlay-get (car overlays) 'face)
                        'reorder-sentence-inactive))
          (setq overlays (cdr overlays)))
        ;; copy the word
        (setq overlay (car overlays))
        (if overlay
            (save-excursion
              (goto-char
               (overlay-start overlay))
              (reorder-sentence-copy-word-at-point)
              (overlay-put overlay 'face 'reorder-sentence-inactive)
              (push overlay reorder-sentence--undo-stack))
          (error "Key `%c' does not correspond to any word" key))))))

(defun reorder-sentence-insert-punctuation ()
  "Insert the key typed in the sentence-reordering buffer."
  (interactive)
  (save-restriction
    (with-current-buffer reorder-sentence--buffer
      (insert last-command-event)
      (set-window-point (get-buffer-window reorder-sentence--buffer)
                        (point-max))
      (push nil reorder-sentence--undo-stack))))

(defun reorder-sentence-undo (count)
  "Undo COUNT changes while reordering a sentence."
  (interactive "p")
  (let (overlay)
    (dotimes (_ count)
      (setq overlay (pop reorder-sentence--undo-stack))
      (when (overlayp overlay)
        (overlay-put overlay 'face nil))))
  (with-current-buffer reorder-sentence--buffer
    (undo count)))
```

Look carefully at what was changed – the actual changes are pretty small, but I wanted to avoid listing fragments of functions only. First of all, we defined the variable holding our stack. Since it is not supposed to be used by the user, we used the two-dash convention. Then, we initialize it to the empty list whenever we start reordering the sentence, and also after we finished it (either by "committing" the changes to the original buffer or by canceling the reorder) – this way the overlays are not referenced anymore by any variable and can be garbage-collected by Emacs. (If we hadn't done that, Emacs garbage collector would see the `reorder-sentence--undo-stack` list holding the

used-up overlays and inferred that they can't be garbage-collected since our Elisp code could find them, and only objects that may not be found by any Elisp code are eligible for garbage collection.) Notice that we explicitly use the `()` notation instead of `nil`, just to emphasize that this variable is going to hold a list – of course, using `nil` here is perfectly valid, too. Next thing is pushing the overlay to the stack whenever we mark it as inactive, and pushing nil if we insert punctuation (this way `reorder-sentence-undo` will know that when it undoes a punctuation insertion, it shouldn't reactivate any overlay!). Finally, `reorder-sentence-undo` pops the right number of overlays from the stack and reactivates them.

You might wonder what would happen if a malicious user said something like `C-u -2 C-/`, i.e., gave a negative or zero argument to `reorder-sentence-undo`. Well, it turns out that nothing wrong would happen, since `dotimes` just does nothing then, but it's good to think about such weird scenarios and make sure they don't break anything. Actually, our undo system is a bit simplistic and indeed *can* be broken if you know how to do it. Can you see it? Do you know how to prevent it?

Implement a more robust undo feature

Well, here is the thing: the "regular" undo stops its undoing work when you break the chain of `undo` commands, and our undo knows nothing about that chain, so if we do something like "undo, move the point, undo", they will fall out of sync – the "regular" undo will *redo* the last change and our undo will undo one more change. (Try it yourself!) There is more than one way to fix that. Again, we'll go for the simple one, where we just give up on the "regular" undo and implement the whole undo system ourselves. It's actually easier than it sounds, since the changes we introduce in the reordering buffer are very simple – they are just additions of some text (either a word or a character) at the end of the buffer. Every time we make such an addition we can record (in the `reorder-sentence--undo-stack` variable) the position of the point before we insert anything there, and delete everything from that position to the end of the buffer when undoing. While at that, let's disable the regular undo in the reordering buffer – this can be accomplished by setting `buffer-undo-list` to `t` in it.

```
(defun reorder-sentence (beg end)
  "Reorder the words in the region."
  (interactive (if (use-region-p)
                   (list (region-beginning) (region-end))
                 (reorder-sentence--current-sentence-bounds)))
  (setq reorder-sentence--begin (copy-marker beg t)
        reorder-sentence--end (copy-marker end)
        reorder-sentence--previous-window-configuration (current-window-configuration)
        reorder-sentence--buffer (get-buffer-create reorder-sentence--buffer-name t))
  (reorder-sentence--create-overlays reorder-sentence--begin reorder-sentence--end)
  (setq reorder-sentence--undo-stack ())
  (with-current-buffer reorder-sentence--buffer
```

```
13      (setq buffer-undo-list t)
14      (erase-buffer))
15    (display-buffer reorder-sentence--buffer)
16    (deactivate-mark)
17    (reorder-sentence-mode 1))
18
19  (defun reorder-sentence-copy-word-at-point ()
20    "Copy the word at point to the sentence reordering buffer."
21    (interactive)
22    (save-restriction
23      (narrow-to-region reorder-sentence--begin reorder-sentence--end)
24      (let ((word (current-word))
25            position)
26        (cond ((> reorder-sentence--capitalize-count 0)
27               (setq word (capitalize word))
28               (setq reorder-sentence--capitalize-count (1- reorder-sentence--capitalize-coun\
29  t)))
30              ((< reorder-sentence--capitalize-count 0)
31               (setq word (downcase word))
32               (setq reorder-sentence--capitalize-count (1+ reorder-sentence--capitalize-coun\
33  t))))
34        (with-current-buffer reorder-sentence--buffer
35          (setq position (point))
36          (when (not (bobp))
37            (insert " "))
38          (insert word)
39          (set-window-point (get-buffer-window reorder-sentence--buffer)
40                            (point-max))
41          position))))
42
43  (defun reorder-sentence-copy-word ()
44    "Copy word designated by the key used to invoke this function."
45    (interactive)
46    (let ((key last-command-event))
47      (when (characterp key)
48        (let ((overlays
49               (sort (overlays-in
50                      reorder-sentence--begin reorder-sentence--end)
51                     (lambda (o1 o2)
52                       (< (overlay-start o1) (overlay-start o2)))))
53              overlay
54              position)
55          ;; find the overlay corresponding to `key'
```

```
56          (while (and overlays
57                      (not (eq (overlay-get (car overlays) 'reorder-sentence-key)
58                               key)))
59            (setq overlays (cdr overlays)))
60          ;; find the first active overlay
61          (while (and overlays
62                      (eq (overlay-get (car overlays) 'face)
63                          'reorder-sentence-inactive))
64            (setq overlays (cdr overlays)))
65          ;; copy the word
66          (setq overlay (car overlays))
67          (if overlay
68              (save-excursion
69                (goto-char
70                 (overlay-start overlay))
71                (setq position (reorder-sentence-copy-word-at-point))
72                (overlay-put overlay 'face 'reorder-sentence-inactive)
73                (push (cons position overlay) reorder-sentence--undo-stack))
74            (error "Key `%c' does not correspond to any word" key))))))
75
76  (defun reorder-sentence-insert-punctuation ()
77    "Insert the key typed in the sentence-reordering buffer."
78    (interactive)
79    (save-restriction
80      (with-current-buffer reorder-sentence--buffer
81        (push (cons (point) nil) reorder-sentence--undo-stack)
82        (insert last-command-event)
83        (set-window-point (get-buffer-window reorder-sentence--buffer)
84                          (point-max)))))
85
86  (defun reorder-sentence-undo (count)
87    "Undo COUNT changes while reordering a sentence."
88    (interactive "p")
89    (let (undo-entry overlay position)
90      (dotimes (_ count)
91        (setq undo-entry (pop reorder-sentence--undo-stack))
92        (when undo-entry
93          (setq position (car undo-entry))
94          (setq overlay (cdr undo-entry))
95          (when (overlayp overlay)
96            (overlay-put overlay 'face nil))
97          (with-current-buffer reorder-sentence--buffer
98            (delete-region position (point-max)))))))
```

Final touches

While we are streamlining the overall experience of using our code, let's add a few more things. As of now, the `RET` key (i.e., `Enter`) does nothing in our mode, which doesn't make sense – it is a perfect candidate for `reorder-sentence-finish` (along with `C-c C-c`). On the other hand, `q` could be a good key to cancel. In many Emacs modes it kind of "cancels the current action" in a sense – usually, this means burying the current buffer. This works e.g. in `dired-mode`, `image-mode` and many others – basically, modes derived from `special-mode`. Of course, doing this would conflict with using `q` as one of the word-copying keys – we would need to change `reorder-sentence--word-copying-key-ranges` accordingly. I'm kind of on the fence with regard to this idea, and for me, the main argument against it is that it would be too easy to press `q` accidentally while reordering, and this would be a not-undoable action. Let's implement only the `RET` idea then.

```
(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region."
  :lighter " Reorder sentence"
  :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
            (,(kbd "RET") . reorder-sentence-finish)
            (,(kbd "C-c C-k") . reorder-sentence-cancel)
            (,(kbd "M-c") . reorder-sentence-capitalize-word)
            (,(kbd "M-l") . reorder-sentence-downcase-word)
            ([remap undo] . reorder-sentence-undo)
            (,(kbd "<backspace>") . reorder-sentence-undo)
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              (if (reorder-sentence--copying-key-p key)
                                  #'reorder-sentence-copy-word
                                #'reorder-sentence-insert-punctuation)))
                      (reorder-sentence--range 32 127)))
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

Also, let's make the package slightly more user-friendly and display a "usage information" in the echo area when the user starts a reordering session.

```
(defun reorder-sentence (beg end)
  "Reorder the words in the region."
  (interactive (if (use-region-p)
                   (list (region-beginning) (region-end))
                 (reorder-sentence--current-sentence-bounds)))
  (setq reorder-sentence--begin (copy-marker beg t)
        reorder-sentence--end (copy-marker end)
        reorder-sentence--previous-window-configuration (current-window-configuration)
        reorder-sentence--buffer (get-buffer-create reorder-sentence--buffer-name t))
  (reorder-sentence--create-overlays reorder-sentence--begin reorder-sentence--end)
  (setq reorder-sentence--undo-stack ())
  (with-current-buffer reorder-sentence--buffer
    (setq buffer-undo-list t)
    (erase-buffer))
  (display-buffer reorder-sentence--buffer)
  (deactivate-mark)
  (reorder-sentence-mode 1)
  (message
   (substitute-command-keys
    "Finish reordering with \\[reorder-sentence-finish] and cancel with \\[reorder-s\
entence-cancel].")))
```

Note how we used the `substitute-command-keys` function to show the user the keybindings for the two commands instead of hard-coding them in the displayed message. This way, if we (or the user!) change these bindings, the message will still display the correct ones.

Next thing, let's expand the docstring of `reorder-sentence-mode` so that it contains some actually useful information about how to use the mode.

```
(define-minor-mode reorder-sentence-mode
  "Easily reorder a sentence or region.
\\<reorder-sentence-mode-map>
Use \\[reorder-sentence] to start a reordering session, press the
keys displayed before subsequent words to copy them to a temporary buffer,
press non-alphanumeric characters to insert themselves into the temporary
buffer, and press \\[reorder-sentence-finish] to finish.  See below for more keybind\
ings.
Note: do not use \\[reorder-sentence-mode] directly.

\\{reorder-sentence-mode-map}"
  :lighter " Reorder sentence"
  :keymap `((,(kbd "C-c C-c") . reorder-sentence-finish)
            (,(kbd "RET") . reorder-sentence-finish)
```

```
            (,(kbd "C-c C-k") . reorder-sentence-cancel)
            (,(kbd "M-c") . reorder-sentence-capitalize-word)
            (,(kbd "M-l") . reorder-sentence-downcase-word)
            ([remap undo] . reorder-sentence-undo)
            (,(kbd "<backspace>") . reorder-sentence-undo)
            ,@(mapcar (lambda (key)
                        (cons (kbd (string key))
                              (if (reorder-sentence--copying-key-p key)
                                  #'reorder-sentence-copy-word
                                #'reorder-sentence-insert-punctuation)))
                      (reorder-sentence--range 32 127)))
  :interactive nil
  (if reorder-sentence-mode
      (setq buffer-read-only t)
    (setq buffer-read-only nil)))
```

Here again we used the special sequences to tell the user how to call various commands. The `\\<reorder-sentence-mode-map>` makes sure that `substitute-command-keys` will use bindings from that keymap, even if `reorder-sentence-mode` is not active when showing its docstring. The `\\{reorder-sentence-mode-map}` sequence displays the summary of the whole mode's keymap.

Finally, let's make life easier for people who use the "customize" feature of Emacs to define settings for various features. This means that we should define a "customization group", gathering all the settings of our package together, use `defcustom` instead of `defvar` for all variables that are in fact "user options" (i.e., variables that the user can set to make the package behave in the way they want), and give them all the `:group` keyword. The same goes for faces we define.

```
(defgroup reorder-sentence nil
  "Reordering words."
  :group 'convenience)

(defface reorder-sentence-key
  '((t :foreground "chocolate" :underline t :height 0.9))
  "Face to display word keys for sentence reordering."
  :group 'reorder-sentence)

(defcustom reorder-sentence-key-format "%c"
  "Format string to display the key for sentence reordering."
  :type 'string
  :group 'reorder-sentence)

(defcustom reorder-sentence-prepare-key-for-display-function
  #'reorder-sentence-prepare-key-for-display
```

```
  "A function used to highlight the keys for reordering the sentence.
It should accept a character and return a string."
  :type 'function
  :group 'reorder-sentence)

(defface reorder-sentence-inactive
  '((t :inherit 'shadow :strike-through t))
  "Face to display already copied words."
  :group 'reorder-sentence)
```

Package information

And that's pretty much it! We could (of course) tinker with this code a lot more, but let's leave it as it is and do something else instead. If we decided to distribute our code, we should add some comments (mainly at the top of the file), saying who the author is, what is the license, what the package is for etc. Every package distributed with Emacs has something like this, and it turns out that Emacs already has a way to insert that automatically – called, of all things, `auto-insert`. Note that running this command is meant to be used in an empty file, and while it can be used in a non-empty one, the results are not optimal and require some manual killing-and-yanking. It first asks for a "short description" (basically, a few words) and then for one or more "keywords", which are kind of predefined "tags" or "categories" for Emacs packages.

The `auto-insert` command inserts two things – a header (which is rather self-explanatory) and a footer, consisting of two lines – a comment at the very end (which indicates the end of the file, most probably a remnant from ancient times when people downloaded uncomressed files from a network, and the lack of the final comment saying that this is the end of file meant that the download was interrupted) and a call to `provide`. This is something that prevents Emacs from loading the same file twice using `require`. (The details of how this works and what else is done by `provide` can be found in the Elisp reference, we are not *that* interested in those.)

The header (inserted by `auto-insert` and then expanded manually a bit) looks like this:

```
;;; reorder-sentence.el --- reordering words in a sentence or region  -*- lexical-bi\
nding: t; -*-

;; Copyright (C) 2021  Marcin Borkowski

;; Author: Marcin Borkowski <mbork@mbork.pl>
;; Keywords: convenience

;; This program is free software; you can redistribute it and/or modify
;; it under the terms of the GNU General Public License as published by
;; the Free Software Foundation, either version 3 of the License, or
```

```
12 ;; (at your option) any later version.
13
14 ;; This program is distributed in the hope that it will be useful,
15 ;; but WITHOUT ANY WARRANTY; without even the implied warranty of
16 ;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 ;; GNU General Public License for more details.
18
19 ;; You should have received a copy of the GNU General Public License
20 ;; along with this program.  If not, see <https://www.gnu.org/licenses/>.
21
22 ;;; Commentary:
23
24 ;; This package implements a way to arbitrarily reorder words in
25 ;; a sentence (or region).  It was written primarily as an
26 ;; illustration for the book "Hacking your way around in Emacs" by
27 ;; Marcin Borkowski (http://mbork.pl), available at
28 ;; https://leanpub.com/hacking-your-way-emacs (see also the blog posts
29 ;; about the book at http://mbork.pl/CategoryEmacsBook).  It is,
30 ;; however, a fully usable (and useful) package in and of itself.  The
31 ;; entry point is the `reorder-sentence' command.  It starts a minor
32 ;; mode where every word in the region (or current sentence by
33 ;; default) is marked with a letter or digit, and pressing that letter
34 ;; or digit copies that word to a temporary buffer.  Other printing
35 ;; characters (like space or punctuation) insert themselves in the
36 ;; temporary buffer.  Pressing `C-c C-c' or `RET' commits the changes
37 ;; to the original buffer, and pressing `C-c C-k' cancels the whole
38 ;; operation.  During reordering, `backspace' (or any key normally
39 ;; bound to `undo') undoes the last operation and `M-c' and `M-l'
40 ;; change the case of the following or preceding word(s) (depending on
41 ;; the prefix argument).
42
43 ;;; Code:
```

By the way, if you noticed the strange thing with `lexical-binding` at the top, this is what `auto-insert` does by default. In case you've never seen anything like that before, it makes Emacs set the `lexical-binding` variable to `t` when visiting the file. We will not discuss what this variable *does* now, however – this will wait until the next chapter. For now, accept that setting it to `t` is a reasonable default.

The footer is much shorter:

```
(provide 'reorder-sentence)
;;; reorder-sentence.el ends here
```

Summary

So, this was a *really* long chapter. I won't even try to list all things we've learned here – in fact, `reorder-sentence-mode` was meant to teach you, dear reader, *a lot* of things every Elisper should know. Probably the most important one here was that you can develop even pretty complicated Emacs tools in small, manageable steps. The whole `reorder-sentence.el` file is almost 400 lines long (well, if you count blank lines, comments etc., at least – the actual code is a bit shorter than 300 lines). Is it much? Well, it depends. For a piece of software which is actually useful, I don't think so. This is thanks to the way Emacs is designed – to have a complete package, you don't have to code everything from scratch. The basics of *text editing* are already there. In the simplest case (like in the previous chapter) you only need to code one or two commands and it's enough. In a more complex one (like here) you may also need to define a mode, assign keybindings, configure user options etc. Still, it is just adding a small piece to an existing construction. Even if you plan to build a whole application on top of Emacs (like the famous Org-mode, or an email client), you can often write just a bunch of commands and maybe a minor mode to get something working.

Counting lines of code

Introduction

In this chapter, we are going to learn to count. For various reasons someone might want to know how many lines comprise some piece of software (not that it means that "more is better"). Emacs has the `count-words-region` command (bound to `M-=`) which can count lines, words and characters (i.e., it is the exact counterpart of the `wc` tool from the GNU Coreutils package), but we want to have a command which is smarter and does not count e.g. blank lines. We will also want to exclude comments, which may be controversial – they *are* an important part of the code – but let's say that we want to know the number of lines containing actual instructions for the machine to *do* something. Anyway, putting aside he discussion of whether SLOC tell us anything useful about code, let's write a `count-sloc` function.

This chapter has several goals. As every chapter here, it showcases a few functions Emacs has for various tasks. We will again see a technique coming from the field of *functional programming*, where we supply a function as another function's argument. We will briefly touch a much more advanced concept which is Emacs' notion of *syntax*. Perhaps most importantly, we are going to encounter the very important concepts of *lexical scoping* and *closures*. While their details and more advanced examples are out of the scope of this book, the example we will see is something you might bump into yourself, and not knowing what happens could be extremely confusing. So, let's start!

The skeleton of the counting command

We will start easy and count just the non-blank lines in the region. And before we do that, let's write a basic skeleton for this command, which may be actually useful for various other commands you might want to write one day – doing something on a region or buffer, and displaying the result in the echo area if called interactively. (Later, we will do something better – turn our "skeleton" into a function so that instead of typing – or yanking – the same code over and over again, we will be able just to call a function.)

We already know how to implement operating on the region if it is active and in the whole buffer otherwise, so let's copy the code responsible for that here.

```
(defun count-sloc (begin end)
  "Count non-blank lines from BEGIN to END.
If called interactively with no active region, count in the whole
buffer."
  (interactive
   (if (use-region-p)
       (list (region-beginning) (region-end))
     (list (point-min) (point-max))))
  (save-excursion
    (save-restriction
      (narrow-to-region begin end)
      (goto-char (point-min))
      (let ((count 0))
        ;; here we count
        count))))
```

We are now ready to move the point line by line, counting lines along the way. Before we start to actually do that, let's stop for one more thing. As I mentioned, it would be pretty nice if our function *returned* the count if called from Elisp and *printed* it if called interactively. There is a function `called-interactively-p` which returns `t` if the function containing it was, well, called interactively, but its docstring strongly discourages its use and hints at a better way: to use a special optional argument. It will be non-nil when the function is called interactively (and it will be the job of the `interactive` clause to make sure about it) *or* when explicitly set in an Elisp code.

```
(defun count-sloc (begin end &optional print-message)
  "Count non-blank lines from BEGIN to END.
Print a message if PRINT-MESSAGE is non-nil.  If called
interactively with no active region, count in the whole buffer."
  (interactive
   (if (use-region-p)
       (list (region-beginning) (region-end) t)
     (list (point-min) (point-max) t)))
  (save-excursion
    (save-restriction
      (narrow-to-region begin end)
      (goto-char (point-min))
      (let ((count 0))
        ;; here we count
        (when print-message
          (message "Non-blank lines: %s" count))
        count))))
```

Notice that we still return `count` if `print-message` is non-nil. In interactive use, this doesn't matter,

since the return value of a command is discarded anyway, but if `count-non-blank-lines` is called from Elisp – even with `print-message` set to a non-nil value – it probably makes sense to still return the count.

Knowing this technique, we can get quite fancy and tell the user if we counted the lines in a region or the whole buffer, for example.

```
(defun count-sloc (begin end &optional print-message where)
  "Count non-blank lines from BEGIN to END.
Print a message if PRINT-MESSAGE is non-nil.  WHERE should be
`\"region\"' or `\"buffer\"' and is printed within the message.  If
called interactively with no active region, count in the whole
buffer."
  (interactive
   (if (use-region-p)
       (list (region-beginning) (region-end) t "region")
     (list (point-min) (point-max) t "buffer")))
  (save-excursion
    (save-restriction
      (narrow-to-region begin end)
      (goto-char (point-min))
      (let ((count 0))
        ;; here we count
        (when print-message
          (message "Non-blank lines in %s: %s" where count))
        count))))
```

This seems not that good of an idea, though, since it requires the programmer to supply the `where` argument whenever `print-message` is supplied, too. Also, it is a bit misleading – for instance, it would say "buffer" if we had no active region, but had narrowing in place. The former issue could be corrected by using another local variable instead of an argument whose value is supplied by the `interactive` clause (but that would make the code longer and more complicated for little gain), and the latter one by introducing some conditional (perhaps using the `buffer-narrowed-p` function, which see – but that would *also* make the code longer and more complicated for little gain). Let's then drop this idea and do something more important: actually learn to count.

Counting non-blank lines

We will start with a simple counter of *non-blank* lines in the buffer. This is easy – we check if this is not the end of buffer (the aptly named `eobp` function checks for this, but it also returns `t` if the point is at the end of the *narrowed portion* of the buffer), and if not, we increase the counter – but only if the current line is not empty. The only non-obvious thing is how we check for empty lines. We

could use the `eolp` function to check if the point is at the end of line – remember that `forward-line` (obviously, also `(goto-char (point-min))`) will put it at the *beginning* of the line, and the line is empty if its beginning and end are in the same place. Let's be a bit fancy, however, and treat lines containing only whitespace as "empty" (i.e., not count them). How to check for this? There are a few ways, but the one that comes to my mind as the first is to test whether it matches a specially constructed *regular expression*. Regular expressions (or *regexen*, as they are sometimes called) are a bit outside the scope of this book, so instead of providing a general intro to them, let's just use this one: `"\\s-*$"`. The double backslash is there because a backslash needs to be escaped in Elisp strings – `"\\"` is a one-character string containing the backslash. This may look strange, but it makes it possible to do things like `"\n"` (this is another one-character string containing a newline) and put other non-printable characters in strings. Technically, the last one is not necessary, since you can embed a "real" newline in a string – this form evaluates to `t`:

```
(string= "hello\nworld" "hello
world")
```

However, escaping e.g. a double quote with a backslash (`"\""`) is the easiest way to include it in a string, so the backslash "escaping" is quite useful.

Coming back, `\s-` is the Emacs regex way to say "any character belonging to the whitespace syntax class". And what are "syntax classes"? Well, glad you asked. Every major mode can treat various characters as playing various syntactic roles. For instance, in Elisp, `-` (the "minus" character) has syntax class "symbol", since it can be part of a symbol name. The same character has syntax class "punctuation" in JavaScript, for instance. This makes it possible for Emacs to treat `count-sloc` as one symbol in Elisp and as two variable names with the minus operator between them in JavaScript. Syntax classes influence several mechanisms in Emacs, like font-lock (aka syntax highlighting), movement by symbols etc. The `\s` in a regex (spelled out `\\s` when a regex is defined by a string in Elisp code for reasons discussed earlier) means "any character of syntax class defined by the following character". For instance, `\s-` is any whitespace character, `\s.` is any punctuation character etc. (You can find the whole table in the Elisp reference, of course.)

Note that we could say `[[:blank:]]` instead of `\s-`, but this has a slightly different meaning – namely, it matches every character categorized as "horizontal whitespace" by the Unicode regex standard. This means that what `\s-` matches may differ in various major modes, but `[[:blank:]]` means the same thing in every mode. (I do not know, however, of any real mode where they actually *do* mean something different – but the possibility is there.)

The rest is easy: `*` means "zero or more", so `\s-*` matches an empty string or any number of consecutive horizontal whitespace characters (note that they do not need to be the *same* characters – a space followed by a tab matches, too), and `$` matches the end of the line (so that the entire regex matches at the beginning of a line containing only whitespace, but not at the beginning of a line which *starts* with whitespace but has something else then). Note that we could have started our regex with `^` ("match the beginning of line"), but we don't have to, since – as I mentioned – we are guaranteed to be at the beginning of line anyway.

That was quite a mouthful, wasn't it? Well, we have only touched the topic of regular expressions. They are incredibly useful and in fact many parts of Emacs use them. (There are problems, though, regexen can't solve, most notably matching strings depending on whether they have correctly paired delimiters of various kinds, like parentheses, brackets, braces or HTML tags. There is a solid theory justifying that, which is completely outside the scope of this book.) The only thing left is actually *checking* if the current line matches the constructed regular expression. This is simple, though: not only does Emacs have functions to *search* for regular expressions (like `re-search-forward`), but also to check if the text at point (more precisely, *after* point) matches a given regex. There are in fact two functions doing that: `looking-at` and `looking-at-p`. The difference between them is rather subtle. Every time Emacs performs a regex match, it saves its results as so-called *match data*, which are much like a global variable, only accessible via several functions (`match-beginning`, `match-end`, `match-data` and a few others). This looks like an *extremely* bad practice, and I have a strong suspicion that it is one of the remnants of the times when a megabyte was a lot of RAM and a 1 MHz processor was considered fast… In any case, `looking-at-p` is a more "functional" version, i.e., it doesn't touch the match data, which is perfect for any case when we just want to know if the text after point matches a regex or not, and we do not intend to do anything more than checking for a Boolean value (hence the name, I suppose). In general, I prefer `looking-at` unless I actually *need* `looking-at-p`, so we'll go with that.

```
(defun count-sloc (begin end &optional print-message)
  "Count non-blank lines from BEGIN to END.
Print a message if PRINT-MESSAGE is non-nil.  If called
interactively with no active region, count in the whole buffer."
  (interactive
   (if (use-region-p)
       (list (region-beginning) (region-end) t)
     (list (point-min) (point-max) t)))
  (save-excursion
    (save-restriction
      (narrow-to-region begin end)
      (goto-char (point-min))
      (let ((count 0))
        (while (not (eobp))
          (unless (looking-at "\\s-*$")
            (setq count (1+ count)))
          (forward-line))
        (when print-message
          (message "Non-blank lines: %s" count))
        count))))
```

Let's get more abstract...

Now, one might think that this is good enough – but this is never true. We are now going to use some more rules to decide if a line counts as a "line of code" or not. Thinking of it, it may be useful to have a general mechanism for counting lines *satisfying some condition*. Maybe we'd like to count only lines longer than some limit? Or lines containing some string? Or lines which do not begin with whitespace? Every one of these conditions can make sense in some particular context. Let's write a general `count-lines-if` function, which – given the condition and the range – counts the lines in that range satisfying that condition. (A careful reader may remember from the previous chapter that we can give a function an argument which is also a function. We will use this method here, too.)

```
(defun count-sloc--count-lines-if (predicate begin end)
  "Count lines satisfying PREDICATE from BEGIN to END.
PREDICATE should accept no arguments."
  (save-excursion
    (save-restriction
      (narrow-to-region begin end)
      (goto-char begin)
      (let ((count 0))
        (while (not (eobp))
          (when (funcall predicate)
            (setq count (1+ count)))
          (forward-line))
        count))))
```

Of course, this is no more a command. In fact, making it into one is non trivial – how is the user going to provide `predicate` interactively? There are some possibilities of doing that, but we will do something else instead. We are going to split our `count-sloc` function into several reusable pieces. One is the (non-interactive) function above, which will actually perform the counting. Another one is going to handle the interactive stuff – checking if there is an active region and using it if yes (and the whole buffer of not), and printing the result in the echo area. This, by the way, means that instead of a "template" we are going to have a proper abstraction – a function. Finally, we will write a short function which is going to act as a "glue" between those two pieces.

```
(defun count-sloc--act-on-region-or-buffer (func message)
  "Perform FUNC on region or buffer and print MESSAGE.
FUNC should accept two arguments, the beginning and end of the
range it operates on.  If MESSAGE contains
a placeholder (e.g. \"%s\"), the return value of FUNC is
substituted for it."
  (let (begin end)
    (if (use-region-p)
        (setq begin (region-beginning) end (region-end))
      (setq begin (point-min) end (point-max)))
    (message message (funcall func begin end))))

(defun count-sloc ()
  "Count non-blank lines in the region or buffer."
  (interactive)
  (count-sloc--act-on-region-or-buffer
   (lambda (begin end)
     (count-sloc--count-lines-if (lambda () (not (looking-at "\\s-*$"))) begin end))
   "Non-blank lines: %s"))
```

Let me make two remarks here. The `message message` may look strange, but it makes perfect sense: the former is a function and the latter is a variable (well, actually a function parameter, which is almost the same). We already know that Emacs Lisp is a Lisp-2 and allows that.

More important is the `lambda` expression in `count-sloc`. We cannot hand the `count-lines-if` function to `act-on-region-or-buffer`, since it accepts three arguments and not two. We *could* define one more function, call it `count-lines-if-not-blank` or something similar, but I chose to use an anonymous one instead.

In fact, this is a very specific case of an anonymous function. The idea here is that it takes some arguments and calls some other function, giving it first an argument which is always the same and then the arguments it received. It turns out that having a function of two or more arguments and making it into a function of fewer arguments by "fixing" some of the arguments is a known mathematical concept, called *partial application*, and it is supported in Elisp (assuming that the fixed arguments come first). Here is a simple example. As we know, the `+` function can add two numbers. (It can add more, but let's focus on two for now.) Now, the value of the expression

```
(apply-partially #'+ 1)
```

is a function which takes an argument and increments it by one (so it's basically a variant of `1+`):

```
(funcall (apply-partially #'+ 1) 2)
```

returns 3. And this is precisely what we can use in our case – we fix the first argument of `count-lines-if` to be the predicate checking if a line is blank. (And while we're at it, let's actually define a *named* function so that we don't need a `lambda` expression anymore. This is good idea for legibility reasons – it's much easier to read and think about things that are *named*.)

```
(defun count-sloc--non-blank-line-p ()
  "Return t if there is a non-blank character in the line.
Assume that the point is at the beginning of line."
  (not (looking-at "\\s-*$")))

(defun count-sloc ()
  "Count non-blank lines in the region or buffer."
  (interactive)
  (count-sloc--act-on-region-or-buffer
   (apply-partially #'count-sloc--count-lines-if #'count-sloc--non-blank-line-p)
   "Non-blank lines: %s"))
```

The approach we employed has a few advantages. Granted, the code becomes a bit more complex, but the logic for counting and handling the user interface are separated. The drawback is that we need to perform a bit more work if we want to count lines of code in an Elisp program – the `count-sloc` command does not accept any parameters anymore, since we moved the code handling them elsewhere.

Now that we can, we may want to count lines based on other properties. Instead of all non-blank lines, we may decide to skip lines containing only comments. How to do that?

Skipping comments

Having written our general mechanism, we now need to determine if the line we're in (and we may even assume the point is at its beginning) contains only comments. There are two ways to do this: a simple one and a correct one. Now of course we usually want the correct one, but it turns out that sometimes going for a simple solution is fine, even if it doesn't work in 100% cases. Let's look at the simple way first.

```
(defun comment-line-p ()
  "Return t if the point is at a comment line.
Assume that the point is at the beginning of line."
  (eq (char-after) ?\;))
```

We could use `looking-at` again, but here we only want to check if the character right after the point is the commenting character, i.e., `;`. We can thus use the simpler `char-after` function, returning – obviously – the character after the point.

The usual Elisp syntax for a single character is a question mark followed by the character itself, and if the character has special meaning in Elisp syntax, you should put a backslash between the question mark and the character. Since the semicolon is the comment character, we have to do it. Note that there is more ways to enter a character in Elisp code (look up the Elisp reference for the details), but we won't need them. Also, notice that characters are just integers under the hood, so `(eq 59 ?\;)` returns `t` – but writing `?\;` instead of `59` is better, since the human reader will immediately see which character we mean.

Anyway, the `comment-line-p` function above, while looking simple, is not correct, and for two reasons. We already know that Elisp strings can contain newlines, so

```
"Hello
; world"
```

is a valid string – but our function would think that the second line is a comment! Of course, this would probably never (or almost never) happen, but there's a much worse issue with our function: it only works in Elisp (and possibly other languages where the semicolon introduces a comment). Why don't we write it in a way that supports any programming language Emacs has support for?

Let that sink in. We want to support even languages we don't know, and even the ones *not yet supported* at the moment of writing our code – but if Emacs supports (or will support) them, we will too. How cool is that? Well, pretty cool, but how to do that?

Actually creating a major mode supporting a new language is way out of the scope of this book, but it should be fairly obvious that among the many things it involves is teaching Emacs the *syntax* of the new language. What do expressions look like? What do comments look like? These kind of questions. Emacs has a general notion of *syntax* for that. Since the mechanism should be general enough to support a wide array of languages people use, the details are very complicated, but we need only one thing – we need to know whether the current line contains a comment (and nothing else!) or not. (Still, even that is pretty complicated.) If you read the chapter about the "syntax tables" (which are the Emacs way of classifying characters according to their role in the language syntax) in the Elisp reference, you will gain – apart from a headache – an appreciation for ingenuity of people who came up with a general way which makes it possible to describe comments that start with a character and last till the end of line, comments that start and end with an "opening" and "closing" character, and also C-like comments that start with `/*` and end with `*/`.

So, how do we check if the line the point is at is a comment-only line? A bit surprisingly, this is fairly difficult. Emacs has a `syntax-ppss` function ("ppss" stands for "parse partial sexp state"), which returns a pretty detailed information about the given position – including whether it belongs to a comment or string. Unfortunately, if the point is *at* the character starting a comment (or, more precisely, *before* that character – remember that conceptually, the point is always *between* two characters!), `syntax-ppss` tells that it is *not* in a comment (which makes sense, but is not that useful for us). A hackish solution where we move forward by one character and check there doesn't work, either – if a point is between two characters introducing a comment (like `//` in JavaScript), `syntax-ppss` still doesn't consider it as being inside a comment. False start, then.

One thing that I initially thought might help here is the `skip-syntax-forward` function. Every character in Emacs has a "syntax class", which can be one of several things, like "whitespace", "word", "symbol", "open paren", "close paren", "comment starter", "comment ender" etc. (of course, the syntax class of the same character may be different in different major modes). The `skip-syntax-forward` function does exactly what it says on the tin – it accepts a set of syntax classes and moves the point forward across every character belonging to one of those classes. Unfortunately, due to the wild differences in comment syntax among languages, many characters which *can* start a comment do not have a "comment starter" syntax (for example, `/` is not a "comment starter", but "punctuation" in JavaScript – instead, its syntax has a flag saying that it's both a *first* and a *second* character of a two-character comment-starting sequence). This means that `skip-syntax-forward` won't help us, either.

Of course, Emacs being Emacs, there *is* a way to find out if a line is all comment – actually, I can think of two ways. One is an *extremely* dirty hack, which I do not recommend at all. We could check the *text properties* of the whole line, and check if the property `font-lock-face` is present and equal to `font-lock-comment-face`. (Of course, this assumes that the user has font locking turned on – but seriously, who hasn't?) Actually, it is not as bad as it might seem, especially since Emacs has some functions which could help with that. There is `next-single-property-change`, which finds the first position (starting with a given one) when some text property changes its value. Even better, there is `text-property-not-all` which checks if at least one character between given two positions does *not* have a given property name/value pair. (There's also `text-property-any` with the opposite meaning, and quite a few others – look up "property search" in the Elisp reference.) I am not sure *why* these functions are there, but it seems that they are used quite a lot in Emacs sources. (I wasn't brave enough to see what they are used *for*…)

The last idea I had for this is the one we will actually implement. It is more complicated than the trick with text properties and font lock, but seems cleaner to me (and does not depend on the user having font lock turned on). On the other hand, I am fairly sure it is much slower – but let's not worry about it until it is really a problem (which it won't). Emacs has the `forward-comment` function, which – guess what – moves forward by a comment. Actually, it can move by more comments, assuming that the only thing between them is whitespace. (It also moves past any witespace encountered, which is *extremely* convenient for us, since if a line contains only comments, it will move past the newline that ends it.) So, to move past any comments the point is on, one can say `(forward-comment 1000)`. (The 1000 constant is arbitrary and doesn't look very nice, but having more than 1000 comments in a row is not very probable. If using an arbitrary constant like this bothers you, you might want to use something like the `most-positive-fixnum` variable – it is actually a constant holding the greatest integer Emacs can store as a "fixnum". Fixnums are Emacs way of storing integers which are "not too big", and handling bigger ones is possible, but may be problematic. See the Elisp reference if you care, or just don't care – on my machine, `most-positive-fixnum` is 2305843009213693951. I would be rather surprised if I encountered a file with that many comments in a row. In fact, such a file would have to be larger that all the data stored in all the computers in the world.) One issue with `forward-comment` is that if the point is inside a string, it won't notice and still move past what might *look* like comments. This, however, is easy to avoid, using the `syntax-ppss` function we discussed earlier – it turns out that `(nth 3 (syntax-ppss))` is non-nil if the point is inside a string.

This means that we have a way to check if the current line contains only comments. First, if we are in a string, it certainly does not. Otherwise, we can remember the line number we are in, and move past any comments and whitespace, and if the line number changed, the line we began with must have contained only comment(s). Actually, we need to take care to check if we are at the end of the buffer, too – since if we are, we are still in the same line, but then it only contains comments anyway. (By the way, since `forward-comment` moves past whitespace, too, even if there are no comments involved, we get a check for a blank line for free!)

```
(defun count-sloc--comment-or-blank-line-p ()
  "Return t if the point is at a comment line.
Assume that the point is at the beginning of line."
  (unless (nth 3 (syntax-ppss))
    (save-excursion
      (let ((orig-line-number (line-number-at-pos)))
        (forward-comment 1000)
        (or (not (eq orig-line-number (line-number-at-pos)))
            (eobp))))))
```

Note that this is not really efficient – if we have many comment lines in a row, we will move past all those comments repeatedly, starting at the beginnings of subsequent lines (so that we will ensure that "this line only contains comments" for every one of those lines). If this is ever a problem, we might basically remove the `save-excursion` – we could safely move past all lines containing only comments and blanks without having our line counter tick. One reason I am not doing this is aesthetical – the `count-sloc--comment-or-blank-line-p` would then have the side effect of moving point. (A "side effect" is basically everything the function does *apart* from returning a value which is visible to the observer, e.g., changing values of global variables, match data, moving point, switching buffers, playing sounds etc.) This means that this function would then be useless outside the "line counting" context – instead of just telling us if the line the point is in (assuming it's in the beginning, of course) is a blank and/or comment line, it would do something else, too (moving point).

Now we want to plug this into our `count-lines-if` machinery. This turns out to be not so easy. The reason is that we need to "negate" the `count-sloc--comment-or-blank-line-p` – we actually need a function returning `t` if we *aren't* on a comment-only or blank line. Of course, we could write such a function in a minute or two, but let's do something different instead, so that we learn one very important concept. We are going to write a function which accepts one argument (some function) and returns another function, which performs the same computations as the given one, but returns the opposite logical value. Effectively, this function will "add a `not` in front of the given function".

Here is the first attempt at such a function.

```
(defun count-sloc--negate (fun)
  "Return a function returning the logical opposite of FUN."
  (lambda (&rest args)
    (not (apply fun args))))
```

It looks perfectly fine, but if you try and evaluate e.g. `(funcall (count-sloc--negate #'<) 1 2)`, it *won't work*, saying `Symbol's value as variable is void: fun`. Why is that?

Well, take a deep breath.

The expression `(count-sloc--negate #'<)` evaluates to

```
(lambda (&rest args)
  (not (apply fun args)))
```

The `lambda` macro *does not evaluate* its arguments, so it "doesn't see" the `fun` inside, nor does anything like substituting the actual *value* of the *argument* `fun` for it. It just creates an anonymous function with the body given. Since there is no global variable `fun`, it has no idea where to get `fun`'s value from. What we need instead is an anonymous function, yes, but with the actual parameter *subsituted* for `fun` in the body.

There is more than one way to do that, but apparently only one way to do it *cleanly*. Here is what we would like to have: an (anonymous) function, using the `fun` variable (well, actually a parameter, but it's more or less the same – a parameter behaves much like a variable local to the function) which "remembers" the value of that variable/parameter from the moment the function was *defined* instead of using its value at the moment it is *invoked*. It turns out that there exist a notion like that: it's called a "closure", and it is defined exactly like a regular function (anonymous or not). The only thing we need to do differently is tell Emacs that we want something called "lexical scoping" (or "lexical binding"). We won't analyze it too deeply apart from the short description above – once you encounter more situations when the distinction between dynamical scope and lexical scope really matters, you'll probably be experienced enough to study the relevant parts of the Elisp reference (look up "lexical scope"). The only thing we need to know now is how to enable this. The way to do it is via the variable `lexical-binding`. It is a good practice to set it as a "file local variable" in a comment in the first line of the file. (We have seen this before when we ran `auto-insert`, and in fact it's agood idea to *always* set it like this. The reason `t` is not the default value of that variable is that it has been introduced to Emacs pretty recently, and a lot of existing code may depend on lexical binding being *not* activated.)

So, let's put this in the first line of our file.

```
; -*- lexical-binding: t; -*-
```

Then the "negating" function above just starts to work correctly, and we can write the final form of the `count-sloc` function.

```
(defun count-sloc ()
  "Count non-blank lines in the region or buffer."
  (interactive)
  (count-sloc--act-on-region-or-buffer
   (apply-partially #'count-sloc--count-lines-if
                    (count-sloc--negate
                     #'count-sloc--comment-or-blank-line-p))
   "Non-blank lines: %s"))
```

By the way, let me also mention that even with `lexical-binding` set to `t`, there *do* exist variables working "the old way", i.e., bound dynamically. (In other words, when they are used in a function, the value used is the one from the moment of function *evaluation*, not *definition*.) In fact, Emacs depends heavily on existence of such variables – this is how user options work. If you set some option to some value – say, `confirm-kill-emacs` to `#'yes-or-no-p` (which, by the way, is pretty reasonable), you expect `C-x C-c` to honor your setting, not whatever was the value of `confirm-kill-emacs` when the corresponding command was defined. In fact, all variables declared with `defvar` or `defcustom` are so-called "special variables", which means they are always bound dynamically.

Summary

So, this was quite a short chapter. We could do more – initally, I wanted to show how to omit lines containing only docstrings when counting, for instance. (I decided to leave that out because (a) it only applies to Elisp, (b) it wasn't 100% reliable (that would be pretty difficult to do!), and (c) I decided that including it would take too much time. It is quite probable that I will include such a section in the future, though.) Still, the main takeaways here are: an introduction to regular expressions, more practice with handing functions as arguments to other functions, the idea of lexical binding and closures. As it happens, it was also the last chapter of the book – at least of version 1. If you want more... well, that's nice of you, and there's one more thing left – the afterword.

Afterword

So, this is it. Thank you for staying with me until the end! This book would not be possible without all the readers who believed that I could write it. Thank you, thank you, thank you! You were fantastic – absolutely fantastic! (And you know what? So was I;-)!)

Now look back for a second. We've spent about 100 pages to learn how to write a 20-line set of two simple functions, a useful package with less than 200 lines of code, and a 45-line tool to count lines of code. It took us quite some time, no? That may sound intimidating (how on Earth would anyone expect people to develop software whose line count is in thousands or millions?), but bear in mind that what we were doing was *learning*. With practice, a lot of the process becomes second nature and happens much faster – and I *do* encourage you to practice.

On the other hand, you can look at it another way – Emacs Lisp is a very concise language, and you can have a *useful* piece of code with 20 lines – and a pretty much complete, production-ready package with under 300 lines. This would be very difficult with some other programming languages! This also means that you can develop little tools and utilities helping you in your everyday Emacs usage literally within minutes. And I'd really like to see what you are going to code so that your (Emacs) life is easier!

This is not the end of the story, of course. If you think I missed something, or you find a typo, error, an unclear place etc. – don't hesitate to contact me. The best way is probably via email (mbork@mbork.pl). I plan to return to this book in about half a year and perhaps add some new material, so your idea might actually make it into the next version! Also, if you liked my book, I'd be very grateful for spreading the word – on your blog, on your Twitter account (please use the hashtag #hackingyourwayemacs then) etc.

Now – close the book, close your eyes and think. What will be your next addition to Emacs?