Rust Power Tools

Sam Van Overmeire

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Rust Power Tools

Version 1

For more information on this and other Manning titles go to
manning.com

welcome

Abandon all hope ye who enter he… I mean, thank you for purchasing the MEAP for *Rust Power Tools*! I wrote this book hoping to be of help to Rust enthusiasts who want to expand their knowledge of the language, particularly when it comes to powerful - but difficult to master - macros. And so the book focuses on macros, particularly the procedural kind, though we will be exploring other topics as well.

TAfter an introduction and overview - with examples - of declarative macros, we start our procedural journey with a 'Hello World' example using a derive macro. In the chapters after that, we will put attribute and function-like macros into the spotlight.

Because this is an 'advanced' topic, I expect at least basic knowledge of the Rust language (structs, functions, tests, lifetimes…) and its surrounding ecosystem from the reader. I also assume you have Rust and Cargo installed on your computer. The book will be easier to digest if you have some professional experience as a programmer since we touch on other topics like DDD, TDD, functional programming, etc. But if those concepts do not ring a bell, there will usually be a note for each one, giving you the gist of it.

Writing a book is an exercise in humility. It reminds me of just how much I don't know, or never thought deeply about. This is especially true when your day-to-day work as an engineer is focused on getting things working quickly to produce value.

In any case, please, let me know if you spot mistakes, find an example that needs a bit of work, or have suggestions that might improve the current (or future!) chapters in Manning's liveBook's Discussion forum for my book. I appreciate your feedback.

Thanks,

—Sam Van Overmeire

brief contents

1 Going meta

You are your own forerunner, and the towers you have builded are but the foundation of your giant-self. And that self too shall be a foundation.

– Kahlil Gibran

This chapter covers

- What metaprogramming is
- Metaprogramming in Rust
- When to use macros
- What this book will teach you

Among the many powerful tools that Rust has to offer, macros are some of the most important and common. They serve as "a light in dark places, when all other lights go out", to be used when normal tools don't suffice. That of itself is enough to make macros a core topic of this book. They have another neat quality though: they can be a *"`pathway to many abilities`"*, as the emperor would put it. When you want to write a macro, you need knowledge of testing and debugging. You have to know how to set up a library because you cannot write a *procedural macro* without creating a library. Some knowledge about Rust internals, compilation, types, code organization, pattern matching, parsing… also comes in handy. Thus learning about macros allows me to talk about a variety of other programming topics! So we will be learning about Rust macros and using them to explore other subjects. *Procedural macros*, specifically, will take center stage because they are powerful and versatile… and you can already find a lot of useful content on *declarative macros*!

But we are getting ahead of ourselves. Let's take a step back and start from the beginning.

1.1 A day in the life

You are a Rust developer, starting a new project. You add a function that generates a person's full name based on a combination of his first and last name using `format!`. Another part of your code will turn your `struct` into JSON, so you annotate it with `#[derive(Deserialize)]`. And since you are a 10x programmer, you always write your tests first. So you add a function for testing, annotating it with the `#[test]` attribute. To make sure everything matches your expectations, you use `assert_eq!`. And when something goes wrong, you either turn to a debugger or… add some logging with `println!`.

Listing 1.1 The Program You Just Wrote

```
use serde::Deserialize;

#[derive(Deserialize)]   ❶
struct Request {
  given_name: String,
  last_name: String,
}

fn full_name(given: &str, last: &str) -> String {
  format!("{} {}", given, last)   ❷
}

fn main() {
  let r = Request {
     given_name: "Sam".to_string(),
     last_name: "Hall".to_string()
  };
  println!(
    "{}", full_name(&r.given_name, &r.last_name)
  );   ❷
}

#[cfg(test)]
mod tests {
  use super::*;

  #[test]
  fn test_deserialize() {
    let actual: Request =
        serde_json::from_str("{ \"given_name\": \"Test\",
    [CA] \"last_name\": \"McTest\" }")
            .expect("Deserialize to work");

    assert_eq!(actual.given_name, "Test".to_string());   ❷
    assert_eq!(actual.last_name, "McTest".to_string());   ❷
  }
}
```

❶ A procedural macro

❷ Declarative macros

And suddenly it dawns on you. Even when writing the simplest of Rust code, you just cannot stop using macros. You are surrounded by the fruits of Rust's *metaprogramming*.

1.2 What is metaprogramming

In brief, metaprogramming is when you write code that takes other code as input. If it had been invented in 2010, there is little doubt in my mind that it would have been called 'inception' programming. Using your own code as data allows you to manipulate existing code or to generate additional code, adding new capabilities to an application or making it more powerful. To enable metaprogramming, Rust has macros, which are a specific form of metaprogramming. Rust's macros run at compile time, 'expanding' into 'normal' code (structs, functions and the like). Only when this process is complete, is your code transformed into a - runnable - binary by `rustc`.

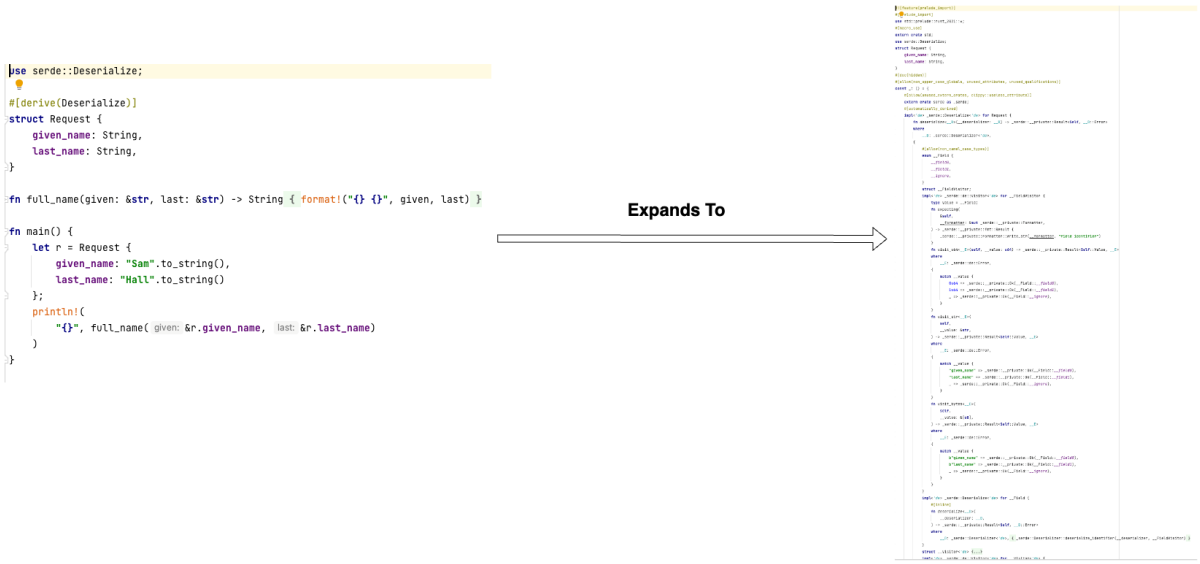

Figure 1.1 I contain multitudes! How our simple example hides many more lines of code, generated by macros

Rust is not the only language to offer metaprogramming capabilities. C++ and Clojure also have powerful macros. Java has reflection to manipulate classes, which famously allowed the Spring framework to develop some of its most impressive capabilities. Both Javascript and Python have `eval`, a function that takes in a string that will be evaluated as an instruction at runtime, as well as other - limited - metaprogramming options.

1.3 Downsides of metaprogramming in other languages

I have spent most of my career as a developer on teams where Java - with the Spring framework - was one of the principal company languages. (And yes, Javascript/Typescript was almost always in use as well.) So I will let you in on a little secret, which I hope you will not share with any of my potential future employers: I have come to really dislike the 'Spring approach' to writing applications. Now, don't get me wrong, Spring offers a lot of really useful libraries. But I am no longer a fan of the 'magic' involved in using the *framework*, which often boils down to adding one or more annotations somewhere in your application and hoping that this will cause good things to happen at runtime.

Powerful, yes. And dependency injection makes it easy to replace or mock a library, or to replace a database without changing any code. (Except that implementation details tend to leak to other parts of an application.) But it is also complex, even for a seasoned Spring developer. Let alone for someone new in the Java/Spring world. Plus, unexpected things will happen at runtime that running locally or testing did not reveal. At least with regard to the runtime part, though, a new wind is blowing. Spring and competitors like Quark and Micronaut have started using a lot more compile-time generation of code and dependency injection. This move is primarily driven by the need for faster startup times in serverless or containerized environments, but a bit less 'magic' and more safety may prove to be beneficial side effects. When it comes to Java reflection, I have to admit that I have occasionally used it in testing, to verify things that Java can't verify, or to inject dummy values into objects. But reflection is difficult to use, easy to abuse, and not very performant. So my personal Java experience with metaprogramming has not been great, even if the technique has sometimes proven useful.

Metaprogramming in those other popular languages is not much better. It has been said that "eval is evil" and there is even a chapter called "Never use eval()!" in the Mozilla documentation! Issues with minification and performance are cited, but the most famous issue has always been security. `eval` makes it incredibly easy for an attacker to add malicious code. Not great for a language that runs everywhere, from server to browser! There is also the issue that if you make a mistake, you will get back a variation of `SyntaxError` *at runtime*. And personally, I prefer to get my mistakes reported before shipping to production, thank you very much. But for a dynamic language like Javascript, that is a given. Python's `eval` function comes with a similar set of warnings. Macros in C++ have a mixed reputation. They are not seen as completely evil, but they have enough downsides that developers seem wary of overuse. They are harder to write and read than normal functions and might cause unintended side effects. In addition, they do not offer type safety, since they don't know anything about types. So they won't fail if you pass them a number when they were expecting strings or vice versa.

1.4 Metaprogramming in Rust

Given this brief and incomplete overview, I would guess that most programmers at some point in their career come into contact with a form of metaprogramming. Often to do things that would be hard to do with 'normal' coding tools. But since most of us are not LISP or Clojure developers, on average such experiences are bound to be limited, or even painful. So why on earth should we care about metaprogramming - using macros - in Rust? Because Rust is *different*. Which is something you have heard too many times before, but hear me out! The first difference with the above selection of popular languages is that, similar to Lisp and Clojure, it is hard to imagine Rust code without macros! They are used extensively in both the standard library and custom crates. At the time of writing, among the top 10 downloaded packages on crates.io, three are for creating procedural macros (`syn`, `quote`, and `proc-macro2`). One of the others is `serde`, where procedural macros ease your serialization work. Or search for the keyword 'derive' - which often signifies that the package has a derive macro - and you will get back over 7000 results, about 7% of all packages!

Why are so many people writing macros? Because in Rust they offer a very powerful form of metaprogramming *that is also relatively easy and safe to use*. Clojure offers power through its macros, yes. And even if (in my opinion) the language itself is difficult, the macros do not add much complexity on top of that. But the compile-time safety of a dynamic language is clearly a lot more limited. Meanwhile, all of Rust's macros are evaluated at compile time and have to withstand the same thorough checks that the language employs to verify other code. Meaning that what you generate is as safe as normal code, and you still get to enjoy the compiler telling you exactly why you are wrong! Especially for declarative macros, 'hygiene' is part of that safety, avoiding clashes with names used elsewhere in your code. That safety is missing from languages like Lisp and C and is the reason for some of the bugs and unexpected behavior that macros can cause in those languages.

Another advantage of doing everything at compile-time is that the performance impact on your final binary is - in most cases - negligible. You are just adding a bit of code. Nothing to lose sleep over. There is also obviously an impact on compile-times, but those are annoyingly long with or without macros. Still, how are macros different from the 'magic' that I just confessed to disliking? Isn't this more of the same? As I hope you will know by the end of this book, there are some differences. Importantly, reflection in Java is, to paraphrase Einstein, 'spooky action at runtime'. Macros in Rust are a) more localized and b) run at compile-time. That allows for easier inspection and better verification.

1.4.1 Macro galore

To make the localized argument more concrete, let me briefly introduce the protagonist of this book, the procedural macro. Procedural macros *take a piece of your code as a stream of tokens and return another stream of tokens* which will in turn be transformed into Rust code. This low-level manipulation stands in contrast with the approach of the better-known declarative macros. Those allow you to generate code using a higher ('declarative') level of abstraction. This makes declarative macros a safe and easy option to get started with - even if they lack the raw power of their procedural brothers.

> **NOTE** **Streams of tokens, expanding macros... As you may have guessed, we will talk about all this in more depth in the upcoming chapters.**

There are three kinds of procedural macros. First, *derive macros*. You use them by adding a `#[derive]` attribute to a struct, enum, or union. When that is done, the code of that struct/enum/union will be passed as an input to your macro. This input is not modified. Instead, new code is generated as an output. These macros are for extending the capabilities of types by adding functions or implementing traits. So whenever you see `#[derive]` decorating a struct, you know it is adding some kind of additional functionality *to that specific struct*. No functionality is added to some random part of your application. Neither is what is passed along modified in any way. Despite these limits (or maybe because of?), these are probably the most widely used procedural macros

Attribute macros, the second type, can be placed on structs, enums, unions as well as trait definitions and functions. They get their name from the fact that they define a new, custom attribute (one well-known example is `[tokio::main]`), whereas derive macros are required to use `\[derive]`. They are more powerful, and thus more dangerous because they transform the item they are decorating: the output they produce replaces the input. Whereas derive macros were only additive, with an attribute macro the definition of your type might change. But at least the annotation is telling you what struct it is transforming and is not changing other code and other files.

The third kind of procedural macro is called *'function-like'*. This one is invoked with the `!` operator and works with whatever input you pass in. Unlike the previous two macros, this one is not limited to annotating things like structs or functions. Instead, it can appear almost anywhere within your code. As we shall see, they can perform some powerful magic. But - you probably already know where I am going with this - the input of that magic is whatever you decided to pass along. Rust, once again, seems to have found a way to take a known programming concept and make it safe(r) to work with.

1.4.2 Appropriate use cases

"Ok, so since macros are so great and safe, I should use them everywhere for everything!". Wow, slow down there strawman! Obviously, you should start any application without turning to *custom* macros. Zero to Production in Rust build an entire deployable newsletter-application without ever writing a macro. (The author uses a lot of those that are provided by the language and its libraries though.) Structs, enums, and functions are just easier to understand and use, plain and simple. And while macros won't have a lot of impact on runtime performance, they still add something to compile-times and binary size, albeit negligibly in many cases.

In larger applications, using macros might be tempting because there is more boilerplate and macros excel at removing that. But for people unfamiliar with your project, the code will become harder to understand because they have to know what the macro is doing. This is often only possible by parsing the code since - compared to a function - the signature of a macro does not give you much insight into what is happening. In addition, your reader is bound to have more experience parsing ordinary Rust code, so even if he has to dive into a function definition, it will take him less time to get the gist of it.

Generic functions are a great tool for avoiding duplication, so they might provide you with an alternative. Similarly, generic implementation blocks (blanket implementations) are very powerful. Just look at this crazy piece of code. We implement our trait for everything that implements `Copy`. Numbers, characters, booleans… suddenly have a new function available. Maybe we should be afraid of using blanket implementations as well as macros.

Listing 1.2 The Powers and Dangers of Generics

```
trait Hello {
    fn hello(&self);
}

impl<T:Copy> Hello for T {
    fn hello(&self) {
        println!("Hello world");
    }
}

fn main() {
    2.hello();
    true.hello();
    'c'.hello();
}
```

Getting back on topic, what would be good use cases for macros? As we already mentioned, within Rust they are often used to avoid duplication and boilerplate. Take a look at the macros offered by the standard library, like `Debug`, `Clone`, `Default`, and others. They all do the grunt work for one well-defined, repetitive task. `Clone` does only one thing: it makes your object cloneable. As a bonus, a Rust developer reading your code will immediately grasp your intent when he sees the `#[derive(Clone)]` attribute. And he probably won't care about the actual

details of how this is done. This is perfect, as it avoids the additional mental strain involved in 'diving into' the macro code. This approach to avoiding duplication is far better than the automatic code generation offered by some languages (looking at you C#) because generation might make it easier to write code but *harder to read it*. And writing is often not the difficult part. Making code understandable for those who come after you, on the other hand, is.

> **NOTE** **I was reading about the `Decode` trait that `sqlx` offers and instinctively thought: "That trait probably has a derive macro, seems like a perfect use case". Lo and behold, there was indeed a derive available!**

So as a first use case for macros, look for repetitive tasks that are very easy to describe from a bird's eye view ("clone this, copy that, print it"). These are often tasks with a 'universal' appeal, useful in many applications, and easy to understand. E.g. making sure that something can be compared to others of the same kind (`PartialEq`) is a common task that most developers have been confronted with before.

Functions can help fight duplication as well, but they can't manipulate structs or add helpers to them. (Blanket implementations can, but they are limited to working through traits.) Outside the standard library, you can find a lot of other examples that help you avoid duplication and boilerplate. Serde allows for easy serialization/deserialization of structs. Tokio manages the boilerplate involved in creating an `async main` for you.

Another reason to turn to macros, closely related to the previous category, would be ease of use. You want to take away the uninteresting technical details of a task that a developer does not need to know about when he is writing his application. You could argue that Serde and Tokio belong to this category since they hide the details of serialization and asynchronous behavior. Only rarely will you have to look under the hood of these macros, most of the time they will 'just work'. How won't matter. Once again, a win for both the reader and the writer. Also worth mentioning are `Clasp` which hides the details - and boilerplate - of parsing command-line arguments and Rocket, which uses macros to hide REST application complexities.

In many cases making things easier is something you can also accomplish with functions. So the question is: what will be easier for developers to understand? If the clearest approach is to add an annotation to a relevant struct (or enum), you can only go for macros. If this is something that should *not* be managed per type but belongs together in a single file or function, that might be a reason to avoid macros and use functions. But maybe that would lead to a lot of boilerplate and less ease of use than a simple macro! Decisions, decisions… And I cannot make them for you, thank God.

One other use case is creating new capabilities that are not available in the Rust language. Again worth mentioning is how Tokio enables you to have asynchronous main functions. But there are lots of other examples in this category. The no-panic crate gives you a compile-time check that

your code does not panic. Static assertions also make guarantees about your code without ever running it, checking for instance whether a struct implements given traits. slqx lets you write SQL strings, checks whether they are valid at compile-time, and transforms the results into structs. Yew and Leptos allow you to write type-checked HTML within Rust. Lazy static gives you a static that is only instantiated when first used. Shuttle sets up cloud infrastructure for you based on annotations. Obviously, a lot of these offer validation at compile-time. After all, that is when your macro will run! But it is also the most interesting time to check and verify before you start doing more expensive, time-consuming things to verify your code. Like unit testing, integration testing, end-to-end testing, or even… testing in production. All of these have their place in modern application building. However, when a simple `cargo check` can point out errors before a single test has run, you are saving yourself a lot of time and effort. In addition, all the things that you can do at compile-time, like setting up those lazy statics, are additional performance wins for your users.

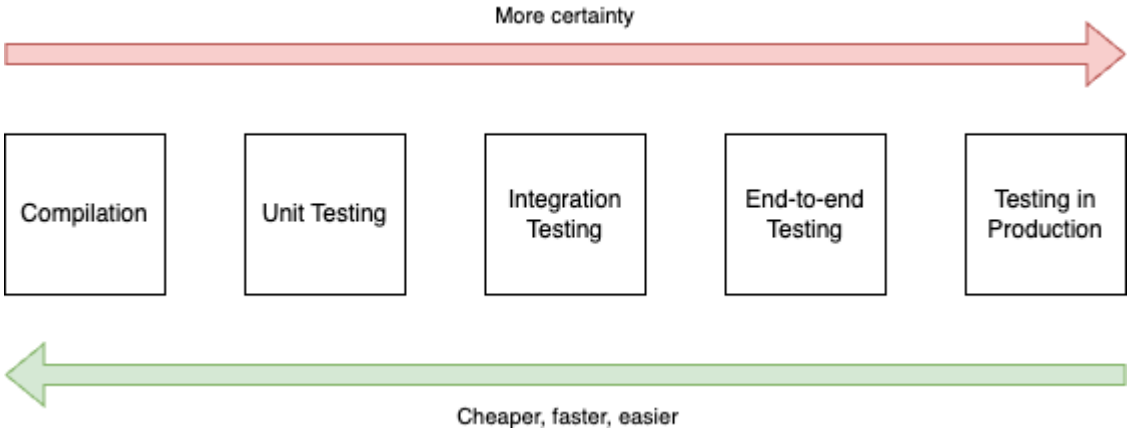

Figure 1.2 Spectrum of testing

Besides verification, macros from this category add *Domain Specific Language* capabilities that allow you to write code in an easier, more elegant way than would be possible with native Rust. Using macros for DSLs is also interesting for application developers that want to enable easier expression of ideas in a way that is closer to the language of their business. When done well, this type also belongs to the ease of use category.

NOTE **What is a Domain Specific Language (DSL)? The programming languages with which we programmers are most familiar are general-purpose languages, applicable to practically any domain. You can use Javascript to write code regardless of the sector you are working in. But DSLs are written with a specific domain in mind. That means the creator can focus on making business concepts easier to express. If you were writing a language for use by banks, you might make it very easy for developers (or even end users! a man can dream) to write code that transfers money between accounts. A DSL can also allow you to optimize. If you were writing one for working with DNA, you could assume you would only need four letters (A, C, G, T) to represent your data, which could allow for better compression (and since A always pairs with T and G with C, maybe you would only need two options!). DSLs come in two varieties: some are created entirely anew, and others are created by using a general-purpose language like Rust as a base. In this book, we are interested in the latter.**

1.4.3 Unfit for purpose - when not to use macros

When it comes to inappropriate use cases for macros, two categories come to mind. The first was mentioned already: things that you can easily accomplish with functions. Starting with functions, and moving to macros when things become too complicated or require too much boilerplate, is often a good idea. Don't try to over-engineer things. The other category where I have some doubts is business logic. Your business code is specific to your use case and application. So almost all publicly available macros are disqualified from the get-go. Now you might write a custom macro for use inside your company. But in a microservice world, sharing business code between services and teams is often a bad idea. Your idea of a 'user', 'aircraft', 'basket', or 'factory' within a microservice will differ from that in the next microservice. It's a road that leads to confusion and bugs, or customization of an already custom macro. There are exceptions to this category though. First, in larger codebases macros could help you avoid some rare business boilerplate. Second, we already noted how DSLs can improve your quality of life as an application engineer - especially in a complex domain. And macros are a great tool for writing DSLs.

One final - but minor - point to keep in mind before we move to the next section: IDE support for macros will always be less advanced than that for 'normal' programming. This is pretty much an unavoidable downside. With more powerful tools come more options. That makes it harder for your computer to guess what you can and cannot legally do. Imagine a programming language whose only valid statement is `2 + 2 = 4`. An IDE would be *incredibly* helpful in pointing out mistakes (*"you typed b - @? !, did you mean 2 + 2 = 4?"*) and giving code completions. Now imagine a language where everything is allowed. Does `struvt Example {}` have a typo? Maybe, maybe not. Who knows? This is related to why it is harder for an IDE to help you when

you work with dynamic languages. I.e. types are helpful for the machine too! A type system limits your options and that can limit the power of your language. But it can offer things like more safety, performance, and ease of use in return.

1.5 Approach of this book

The approach of this book can be summarized as 'example driven, step by step'. Most chapters will have one application as a central theme to explore a macro topic as well as other relevant themes from Rust. Starting with a simple 'hello world', we will add layers of knowledge, piece by piece: how to parse, how to test, how to handle errors. We will also point out common errors that you might run into and give you some debugging hints. Finally, chapters will briefly point out how popular crates (including those mentioned in this chapter) use the explained techniques or accomplish specific feats. This will give you insights into how you can apply what you have learned.

1.6 Exercises

- Ask a generative AI to tell you about metaprogramming in Rust. Based on what you learned, what does it get right? What mistakes does it make?
- Think of a recent application that you worked on. Could macros help avoid duplication or boilerplate? Make the application easier to use? Was there something that you could not easily do within the constraints of the language?

1.7 Summary

- Metaprogramming allows you to write code that generates more code
- Many languages offer some way to do metaprogramming
- But these tools are often difficult to use and not well-integrated into the language, which can lead to hard-to-understand or buggy code
- Rust’s macros are powerful but avoid many of these issues. They are 'expanded' into normal code that will be checked by the compiler
- Rust has declarative macros and three kinds of procedural macros (derive macros, attribute macros, and function-like macros)
- Metaprogramming should not be your first choice when solving problems, but it can help you avoid boilerplate and duplication, make your applications easier to use, or do things that are difficult to do with 'normal' Rust
- This book will explore macros, all the while using them to discuss other advanced subjects through an example-driven approach

2 Declarative macros

This chapter covers

- Writing declarative macros
- Avoiding boilerplate and duplication
- Accomplishing things that are hard to do with functions
- Automatically implementing newtype methods
- Implementing a simple DSL for exchanging money
- Making composition of functions easier
- Debugging macros by expanding
- The basics of macro hygiene
- Understanding the lazy static crate

We will start this book in *easy mode* with *declarative macros*. These macros consist of a syntax that will immediately remind you of pattern matching, with a combination of *matchers* and *transcribers*. The matchers contain what you want to match against; the transcriber has the code you will generate when you find that match. It's just that simple.

NOTE This chapter's focus is a broad overview of declarative macros and their usage. This stands in contrast with the rest of this book, where we will focus on specific topics and a limited number of examples. The reason is that declarative macros are not the main focus of this book and I expect the reader to know more about them than procedural macros. That means we can go through the subject of this chapter more quickly.

2.1 Creating vectors

But wait, this was an example-driven book! That means we should drag a first example into this.

2.1.1 Writing our own vec macro

`vec!` is a classic for explaining declarative macros. We will go through a *simplified* implementation that shows how the aforementioned matchers and transcribers work together to generate the correct kind of code output for any given situation. Here's the code.

Listing 2.1 my_vec, our first declarative macro

```
macro_rules! my_vec {   ❶
    () => [   ❷
        Vec::new()
    ];   ❸
    (make an empty vec) => (   ❹
        Vec::new()
    );   ❺
    {$x:expr} => {
        {
            let mut v = Vec::new();
            v.push($x);
            v
        }
    };   ❻
    [$($x:expr),+] => (
        {
            let mut v = Vec::new();
            $(
                v.push($x);
            )+
            v
        }
    )   ❻
}

fn main() {
    let empty: Vec<i32> = my_vec![];
    println!("{:?}", empty);   ❼
    let also_empty: Vec<i32> = my_vec!(make an empty vec);
    println!("{:?}", also_empty);   ❼
    let three_numbers = my_vec!(1, 2, 3);
    println!("{:?}", three_numbers);   ❽
}
```

❶ We declare a new macro called `my_vec`

❷ `()` is our first matcher. Since it is empty, it will match a macro call without any arguments

❸ Everything between the pair of square brackets is the first transcriber. This is what we will generate for an empty invocation of our macro. Note the semicolon at the end

❹ `(make an empty vec)` is a second matcher. It will match when our input literally matches 'make an empty vec'

❺ This is our second transcriber, this time between parentheses. We generate the same output as before, in the first transcriber

❻ The next two matcher-transcriber pairs. The first accepts one expression (`expr`) and will bind it to `x`. The second accepts multiple expressions separated by a comma. These will similarly be bound to `x`.

❼ These two print `[]`

❽ This one prints `[1, 2, 3]`

2.1.2 Syntax basics

You start your declaration of a declarative macro with `macro_rules!`, followed by the name you would like to use for the macro, similar to how you would create a function by writing `fn` followed by a function name. Inside the curly braces, you put the desired matchers and transcribers. A matcher and its transcriber are (similar to the syntax of pattern matching) separated by an arrow: `(matcher) (transcriber)`. In this case, we have four pairs of matchers and transcribers. Our first pair consists of an empty matcher, represented by some empty brackets, and a transcriber whose content is wrapped in square brackets. Squared brackets are not a requirement though: for both matcher and transcriber you have your choice of brackets: `()`, `{}`, and `[]` are all valid.

> **NOTE** **The alert reader will notice that every pair in the example looks a bit different. This is only intended as a demonstration of your options regarding brackets. Your code will look cleaner if you settle for one of these options. Which one should you pick? Going for the curly braces can have the downside of making your code a bit less clear if you have code blocks within your transcriber (see the second pair). And square brackets seem to be the less popular choice… So normal brackets are probably a good default.**

Another important syntactic element is that the pairs are separated by a semicolon. If you forget to do this, Rust will complain.

```
5 |     {$x:expr} => {
  |     ^ no rules expected this token in macro call
```

Which is its way of saying that there should not be any rules if you end a matcher-transcriber without a semicolon. So keep adding them as long as you have more matcher-transcriber pairs coming. When you get to your last pair, the semicolon is optional.

2.1.3 Declaring and exporting declarative macros

A limitation to consider is that declarative macros can only be used after they have been declared. If I had placed the macro below the main function, Rust would complain like this:

```
error: cannot find macro `my_vec` in this scope
 --> src/main.rs:5:25
  |
5 |     let three_numbers = my_vec!(1, 2, 3);
  |                         ^^^^^^
  |
  = help: have you added the `#[macro_use]` on the module/import?
```

Once you start exporting macros, this is no longer a problem since they are 'hoisted' to the top of your file by Rust. This is why the tip in this error message will solve the error as well, albeit - in this case - for the wrong reasons.

When you need to invoke your macro, you use its name followed by an exclamation mark and arguments between brackets. Similar to the macro itself, during invocation you can have any bracket you like, as long as it is normal, curly, or square. No doubt you have often seen `vec![]`, but `vec!()` and `vec!{}` are also valid, though curly brackets do not seem to be very popular for brief invocations. In this book, you will see me use curly braces for multiline `quote!` calls though.

2.1.4 The first matcher explained

Now that we have covered the basic syntax, here is our first matcher again.

```
() => [
    Vec::new()
];
```

Since our matcher is empty, it will match any empty invocation of our macro. So when we called `let empty: Vec<i32> = my_vec!();` in our main function, this is the matcher we ended up in, since a) Rust goes through the matcher from top to bottom and b) we did not pass anything in within the square brackets. We said that the content of the transcriber is located between the (in this case square) brackets, so that means `Vec::new()` is the code that Rust will generate when we have a match. So in this case, we are telling it that we want to call the `new` method of the vector struct. This piece of code will be added to our application in the location where the macro was called.

That brings us back to the first call in main. Rust sees `my_vec!()` and thinks *"an exclamation mark! This must be a macro invocation"*. And since there are no imports in our file, this is either a macro from the standard library or a custom one. It turns out to be a custom one because Rust finds it in the same file. With the macro found, Rust starts with the first matcher, which turns out to be the correct one. Now it can replace `my_vec!()` with the content of the transcriber, `Vec::new()`. So before starting to compile your code to a binary, `let empty: Vec<i32> = my_vec!();` has already changed to `let empty: Vec<i32> = Vec::new();`. A minor but important detail: since only `my_vec!()` is being replaced, the semicolon at the end of the statement remains where it is. Because of this, we did not need to add one to our transcriber.

2.1.5 Non-emtpy matchers

Let’s turn to the second matcher, which looks like this:

```
(make an empty vec) => (
    Vec::new()
);
```

In this case, the matcher contains *literal values*. This means that to match this particular 'arm' of the macro, you would need to put that exact literal value between brackets when calling the macro, which is what we do in the second example from our main function: `let also_empty: Vec<i32> = my_vec!(make me an empty vec);`. Our transcriber has not changed, so the output is still `Vec::new()` and the code becomes `let also_empty: Vec<i32> = Vec::new();`. In this case, the literals do not add anything interesting. But we will see some more interesting examples later on.

The next pair is more interesting.

```
{$x:expr} => {
    {
        let mut v = Vec::new();
        v.push($x);
        v
    }
};
```

This time we are telling Rust that we want it to *match any single Rust expression* (`expr`) and *bind it* to a value called `x`. The dollar sign preceding `x` is significant, since it signifies that this is a 'macro variable'. Without it, Rust thinks - like previously - that this is just another literal, in which case there would be exactly one match (i.e. `my_vec![x:expr]`). Besides expressions, which are a common target for matching, you can also match identifiers, literals, types…

> **NOTE** **The most powerful of all these target types is `tt`. `tt` stands for `TokenTree`, basically 'a piece of Rust code'. It's obviously a powerful option. But its comprehensiveness can also be a downside. For simpler types, Rust can catch mistakes (e.g. when you pass in a `literal` while the macro only matches an `ident`). Plus, your matchers become less fine-grained, since this one is screaming "GIVE ME ANYTHING YOU'VE GOT". It is advisable to start with a more concrete type and only move up to things like `tt` when that proves necessary.**

Within the transcriber we are creating a new vector, adding the input expression, and returning the entire vector, which now contains the expression as its only element. This is basic Rust code with only two things worth mentioning. The first is that we have to use the dollar sign within the transcriber as well. Remember, with `$` we have identified 'x' as a macro variable. So what we are telling Rust is to push this variable, which was bound to the input, into the vector. Without the dollar sign, Rust will tell you that it `cannot find value x in this scope`. Because there is no `x`, only `$x`.

The second thing to note is the extra pair of curly braces. Without those, Rust gives you back an error saying `expected expression, found let statement`. The reason becomes clear once you try to mentally substitute the macro call with its output. Take this example, which should

match our current rule: `let a_number = my_vec!(1);`. We know that `my_vec!(1)` will be replaced with the content of the transcriber. So since `let a_number =` will stay in place, we need something *that can be assigned to a let*. Say, an expression. Instead, we are giving back two statements and an expression! How is Rust supposed to give that to `let`? Once again, the error sounded cryptic, but it makes perfect sense. And the solution is simply to turn our output into a single expression. The curly braces do just that. Here is the code after our macro has run:

```
let a_number = {
    let mut v = Vec::new();
    v.push(1);
    v
}
```

Yeah, writing macros does require some thinking and (a lot of) tinkering. But since you have chosen Rust, you know that thinking is definitely on the menu.

We are now almost past the basics! Final matcher-transcriber pair:

```
[$($x:expr),+] => (
    {
        let mut v = Vec::new();
        $(
            v.push($x);
        )+
        v
    }
)
```

This is basically the same one as before, except with more dollar signs and some pluses. Within our matcher, we can see that `$x:expr` is now wrapped with `$( ),+`. That tells Rust to accept 'one or more expressions, separated by a comma'. As a programmer, it will not surprise you to hear that - in addition to a `+` - you can use a `*` for zero or more occurrences and `?` for zero or one. Like macros, regular expressions are everywhere. A slight gotcha is that the above will not match an input with a trailing comma. `my_vec![1,2,3]` will work, whereas `my_vec![1,2,3,]` will not. For that, you need an extra rule (see the exercises).

Inside the transcriber, the only thing that has changed is that a similar dollar-bracket-plus combo is surrounding our push statement. Except this time without the comma. Here too, this indicates repetition. 'For every expression from the matcher, repeat what is inside these brackets'. I.e. write a push statement for every expression that you found. That means `my_vec![1,2,3]` will generate *three* push statements.

NOTE **By now it might be obvious that the third matcher-transcriber pair is covered by the last pair. But that additional pair made it easier to explain things step-by-step.**

One final point before we end this section. What happens when you try to do illegal things inside

a macro? What if you try to mix integers and strings as input, something a `Vec` cannot accept? Your IDE might not realize that anything is amiss. After all, it's all valid expressions being passed in! But Rust is not fooled, because it generates 'normal' code from the 'rules' of your macro. And that code has to obey Rust's compilation rules. This means that you will get an error `expecting x, found y` (with the names depending on what you passed in first) if you try to mix types.

Now that you have seen the basics, we can move on to more interesting stuff.

2.2 Use cases

In this section, we will show common ways declarative macros increase the power of applications. In some cases, their utility is straightforward: they help you avoid writing boilerplate code, as we will see in our *newtypes* example. But other examples show you how we can do things with macros that are hard or impossible to do in any other way. Like creating *Domain Specific Languages*, fluent composition of functions, or adding additional functionality to functions.

Let's get started.

2.2.1 Varargs and default arguments

First, how about when we bump into the limits of functions? For example, unlike Java or C#, Rust functions do not allow variadic arguments. One reason might be that variadic arguments make the compiler's life harder. Or that it is not important enough a feature. But if you do need varargs, there are always macros. In fact, our vector macro performs this exact trick! Pass in any number of arguments and Rust will generate code to handle your needs.

If you are coming to Rust from one of the many, many languages that permit overloading or default arguments, macros have you covered as well. An example: I have a function for greeting and I would like it to default to "Hello", while also allowing more creative, custom salutations. I could create two functions with slightly different names to cover these cases. But it's a bit annoying that the names would differ when they offer the same functionality. Instead, we will write a `greeting` macro.

Listing 2.2 Greeting people, with defaults, in `greeting.rs`

```
pub fn greeting(name: &str, greeting: &str) -> String {
    format!("{}, {}!", greeting, name)
}

macro_rules! greeting {
    ($name:literal) => {
        greeting($name,"Hello")
    };
    ($name:literal,$greeting:literal) => {
        greeting($name,$greeting)
    }
}
```

For the first time in this chapter, our implementation is not located in the same file as our main function. Instead, it is placed in a separate file called `greeting.rs`. To use the macro outside the file with its definition, we have to put `#[macro_use]` above the module declaration that we add in main.

Listing 2.3 Example usage of our greeting macro in `main.rs`

```
use crate::greeting::greeting;  ❶

#[macro_use]
mod greeting;  ❷

fn main() {
    let greet = greeting!("Sam", "Heya");
    println!("{}", greet);  ❸
    let greet_with_default = greeting!("Sam");
    println!("{}", greet_with_default);  ❹
}
```

❶ Import the greeting *function* that we wrote.

❷ Import the module that contains our macro. With the annotation `#[macro_use]` we tell Rust that we want to import macros that are defined in that file

❸ Prints "Heya, Sam!"

❹ Prints "Hello, Sam!"

In a more complicated setup, with `mod.rs` importing and re-exporting modules, you will need to put the annotation both in the 'root' (your `main.rs` file) **and** any `mod.rs` files that do re-exporting. But don't worry, Rust will keep complaining with `have you added the #[macro_use] on the module/import?` until you fix all of them. It can be tedious at times, but this focus on keeping things private unless they are explicitly made public does force you to think about information hiding.

Note that we had to make our `greeting` function public (and import it into our `main.rs`). When you consider it, the reason is once again obvious: our declarative macro is *expanded in our main function*. In the previous section, we already learned that we can mentally take the content of our

transcriber and replace the invocation with that content. In this case, `greeting!("Sam", "Heya")` is replaced by the `greeting` function. And if `greeting` is not public, you are trying to invoke an unknown function. This behavior might not be what you desire (because you might want the macro to be the entry point to all your holiday greetings) but it is a logical consequence of the way macros and visibility work in Rust.

2.2.2 More than one way to expand code

We interrupt this broadcast to talk a bit more about expanding, a more official term for 'replacing with the content of the transcriber'. Because while replacing content in your mind is great, sometimes you want to see what is really going on. To help with that, Rust has a nice feature called trace macros (itself a declarative macro… turtles all the way down). They are unstable, which means you have to activate them as a feature and run your code with the nightly Rust build. You can do that with `rustup default nightly` which sets nightly as your default. Or - if you would like to stay on stable - you can instruct cargo to run a specific command with nightly using `cargo +nightly your-command`.

The following code shows how to activate and deactivate the trace macros feature.

Listing 2.4 Using the trace macros feature

```
#![feature(trace_macros)]   ❶

use crate::composing_example::composed_example;
use crate::starting_example::calculate_raise;
use crate::default_args::greeting;
use crate::simple_dsl::example_money_exchange;

#[macro_use]
mod default_args;

fn main() {
    trace_macros!(true);   ❷
    let _greet = greeting!("Sam", "Heya");
    let _greet_with_default = greeting!("Sam");
    trace_macros!(false);   ❸
}
```

❶ Add the unstable trace macros feature

❷ Activate the trace macros

❸ And deactivate the trace macros

This is our code from before with the `println!` statements removed and calls to `trace_macros!` added. With `true`, they are activated, with `false` they are disabled. In our case, deactivation is not strictly necessary since we have reached the end of our program in any case. Running this code will print something like this:

```
--> ch2-trace-macros/src/main.rs:9:18
  |
9 |     let _greet = greeting!("Sam", "Heya");
  |                  ^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: expanding `greeting! { "Sam", "Heya" }`
  = note: to `greeting("Sam", "Heya")`

--> ch2-trace-macros/src/main.rs:10:31
   |
10 |     let _greet_with_default = greeting!("Sam");
   |                               ^^^^^^^^^^^^^^^^
   |
   = note: expanding `greeting! { "Sam" }`
   = note: to `greeting("Sam", "Hello")`
```

The logs show our default at work! `greeting!("Sam");` is apparently transformed to `greeting("Sam", "Hello")`. How ingenious. The trace macros feature is now doing all the substitution work for us, which can save you a lot of mental effort.

Another occasionally useful tool is the `log_syntax!` macro (also unstable) which allows you to log at compile time. If you have never written macros before, you may not have considered why this might be important. As a minor demonstration, we can add a third option to our greeting macro. This third option uses `log_syntax` to tell the user what arguments were received, calls `println!` to inform him that the default greeting was returned, and calls the `greeting` function. All of this is wrapped in an additional pair of braces because we want a single expression to bind to `let`.

Listing 2.5 Using log syntax

```
macro_rules! greeting {
    ($name:literal) => {
        greeting($name,"Hello")
    };
    ($name:literal,$greeting:literal) => {
        greeting($name,$greeting)
    };
    (test $name:literal) => {{
        log_syntax!("The name passed to test is ", $name);   ❶
        println!("Returning default greeting");
        greeting($name,"Hello")
   }}
}
```

❶ We are using `log_syntax!` to log our input.

In our main file, nothing much has changed. We have added another feature and a macro invocation that will end up in our third matcher.

```
#![feature(trace_macros)]
#![feature(log_syntax)]   ❶

use crate::greeting::greeting;
#[macro_use]
mod greeting;

fn main() {
    trace_macros!(true);
    let _greet = greeting!("Sam", "Heya");
    let _greet_with_default = greeting!("Sam");
    let _greet_with_default_test = greeting!(test "Sam");   ❷
    trace_macros!(false);
}
```

❶ Log syntax is unstable, so we need to activate it with a feature

❷ We invoke the new branch of our greeting macro

Now if you run this with `cargo +nightly check` you will see the `log_syntax!` output ("The name passed to test is Sam") because it is executed at compile time. Only when we invoke `cargo +nightly run` will we see both the log syntax output and the `println!`. This difference is important for anyone debugging things that happen at compile time. Using the former you can debug declarative macros via print statements (objectively the best kind of debugging). Together, these tools allow you to trace… the route from macro to expanded Rust code. It is not the power of a real debugger, but certainly better than nothing. A pity that you have to use nightly Rust. Later, we will see some tooling that works with stable.

2.3 Newtypes

Another way declarative macros can help you is by avoiding boilerplate and duplication. To explore that theme, we will introduce *newtypes*. Newtypes are a concept from the world of functional programming. Essentially, they are a 'wrapper' around an existing value that forces your type system to help you avoid bugs. Say I have a function that calculates pay raises. The problem with that function is that it requires four parameters and two pairs have the same type. That means it is easy to make stupid mistakes.

Listing 2.6 Making mistakes

```
fn calculate_raise(first_name: String,
                   _last_name: String,
                   _age: i32,
                   current_pay: i32) -> i32 {
    if first_name == "Sam" {
        current_pay + 1000
    } else {
        current_pay
    }
}   ❶

fn main() {
    let _first_raise = calculate_raise(
        "Smith".to_string(),
        "Sam".to_string(),
        20,
        1000
    );   ❷

    let _second_raise = calculate_raise(
        "Sam".to_string(),
        "Smith".to_string(),
        1000,
        20
    );   ❸
}
```

❶ Only people with first name "Sam" will get a raise

❷ Whoops, I switched my first and last name. No raise for me!

❸ This time, at least the names are in the right order and I will get a raise. But despite being a thousand years old (!), my raise is only 20 dollars. Definitely a bug.

By creating unique wrappers (`FirstName`, `LastName`, `Age`, `CurrentPay`) for our parameters the type system can keep us from making such mistakes. In addition, our code becomes more readable because we are making everything a bit more explicit. And hiding the 'real' value of our parameters and giving them types with more meaning within our domain also makes newtypes ideal for public APIs. Not only making the API easier to understand but easier to evolve.

> **NOTE** **Rust also has type aliases that create an alternative name (alias) for a chosen type. If we wanted to do this for `FirstName`, we would write `type FirstName = String;`. This can make your code more readable to other developers, who can now see that you desire one specific type of String, a first name. Type aliases are often used when you have a more complex type that you want to make easier to read and use. Crates often have one custom error that is used everywhere. The `syn` crate, for example, has `syn::Error`. Because of this, it is also convenient to offer a type alias for `Result` with the default crate error already filled in. E.g. `type Result<T> = std::result::Result<T, syn::Error>`. Now code in the package can use `Result<T>` for its return type. But type aliases do not make your type system smarter: I can still pass in a `String` when my function requires a `FirstName` type alias.**

You can see an example for the `FirstName` newtype below. I have kept the internal value of the wrapper private while presenting the rest of my application with a `new` method. In that method I can do additional checks to make sure a valid value was passed in, giving back an error when this is not the case. Why would I do that? Because it will make using these types easier in other functions that can now rely on this validation to make additional assumptions. If there is only one way to create these wrappers, we *know that our newtype passed all its validations*. In the case of `FirstName` I now know that it is not empty. For `Age`, functions can assume validation will check that we have, say, a positive number under 150. But even without safeguards, newtypes have their value because they make the type system more powerful and force you to think about the values you pass in. If a function requires a `FirstName` and you are required to manually wrap your String into one, maybe that will keep you from accidentally passing in a String called `last_name`.

> **NOTE** **A built-in way to mitigate mistakes for our `age` vs `pay` issue would be to make the `age` parameter a `u8`. This would already guarantee a positive number below 256.**

Besides the constructor, we are also exposing a method `get_value` that will give back an immutable reference to my value to be used, safely, by other parts of my code. We could also add some conveniences like `AsRef` and `AsRefMut` trait implementations, but that is beyond the scope of this example.

Listing 2.7 The FirstName newtype

```
struct FirstName {
    value: String,
}

impl FirstName {
    pub fn new(name: &str) -> Result<FirstName, String> {
        if name.len() < 2 {
            Err("Name should be at least two characters".to_string())
        } else {
            Ok(FirstName {
                value: name.to_string(),
            })
        }
    }

    pub fn get_value(&self) -> &String {
        &self.value
    }
}

// code for the other three newtypes

fn calculate_raise(first_name: FirstName,
                   _last_name: LastName,
                   _age: Age,
                   current_pay: Pay) -> Pay {
    // ...
}   ❶
```

❶ Usage example: pass in the `newtypes` instead of the built-in types. If you accidentally switch arguments one and two, or three and four, the compiler will throw an error.

Everything is a trade-off in programming. Here, one downside to our approach is how bloated our code has become. Just to create a single newtype, we had to write 18 additional lines of code! We can turn to macros to reduce this overhead. Let's start with the `get_value` method for the `FirstName` struct.

Listing 2.8 A macro for the newtype get_value method

```
struct FirstName {
    value: String,
}

struct LastName {
    value: String,
}

macro_rules! generate_get_value {
    ($struct_type:ident) => {   ❶
        impl $struct_type {   ❷
            pub fn get_value(&self) -> &String {
                &self.value
            }
        }
    }
}

generate_get_value!(FirstName);   ❸
generate_get_value!(LastName);   ❸
```

❶ We want one input argument, an identifier

❷ And we use that argument, so we can implement a `get_value` method for the given struct

❸ Now our FirstName and LastName struct will get the new method implemented for them

By now, you can read this code with ease. We declare a new macro with `macro_rules!` and write a single matcher-transcriber pair. The only thing that we need to receive is the name of our struct. `ident` (identifier) seems like the most suitable kind of input. Everything except this name is hardcoded in the transcriber. Note that you can have multiple `impl` blocks for a struct. If this was not the case, our macro would prevent users from implementing any methods of their own!

So are we done? Unfortunately, no. When we try this for the next two struct, `Age` and `Pay`, we are greeted by a compile error: `mismatched types: expected reference &String found reference &i32`. Rust is right! We were keeping things simple assuming the return type would always be `String`. But that is not the case. One way to solve this issue is to make our macro accept an additional argument, a return type override. While `ident` would work, `ty` - which stands for 'Type' - is more suitable. `String` will be the default when we only get one identifier. That way our existing code will keep working. With this override in place, we can use the macro for `Age` and `Pay` as well.

Listing 2.9 Making it work for other types than Strings

```
struct Age {
    value: i32,
}

struct Pay {
    value: i32,
}

macro_rules! generate_get_value {
    ($struct_type:ident) => {
        impl $struct_type {
            pub fn get_value(&self) -> &String {
                &self.value
            }
        }
    };
    ($struct_type:ident,$return_type:ty) => {
        impl $struct_type {
            pub fn get_value(&self) -> &$return_type {
                &self.value
            }
        }
    }  (1)
}

generate_get_value!(FirstName);
generate_get_value!(LastName);
generate_get_value!(Age,i32);
generate_get_value!(Pay,i32);
```

(1) This pair is new. It takes an additional input, a type (`ty`) which is used to specify the correct return type in the signature of our `get_value` method.

And so we have exchanged some 18 lines of boilerplate code for… 20? Ok, but we can do better than that. Another nice feature of macros, one that we have avoided using for simplicity's sake, is that they can *call themselves*. And you may have noticed that there is a lot of overlap between our two code paths. So why not have one of our cases simply use the other? Since the 'String' pair is a special case of the other one (which can handle any type), it makes sense for our special case to use the generic internally. So we will call our macro and pass along the input, with `String` as a second argument. Rust sees the two identifiers and knows it has to move to the second matcher.

Listing 2.10 Final version of the get_value macro

```
macro_rules! generate_get_value {
    ($struct_type:ident) => {
        generate_get_value!($struct_type,String);   (1)
    };
    ($struct_type:ident,$return_type:ident) => {
        impl $struct_type {
            pub fn get_value(&self) -> &$return_type {
                &self.value
            }
        }  (2)
    }
}
```

❶ Inside our first transcriber, we are now calling our own macro with two identifiers

❷ This means we end up in this matcher which accepts two identifiers

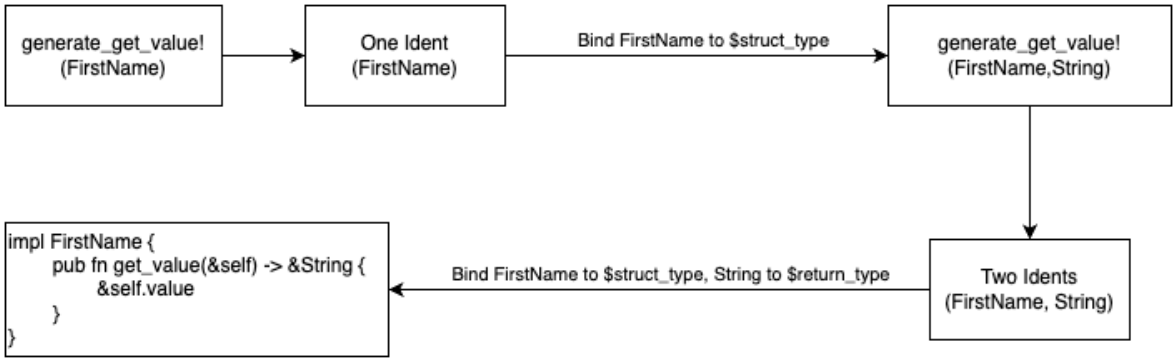

Figure 2.1 What happens when we call generate_get_value with only one identifier

We are now writing less boilerplate than before, all our logic for generating `get_value` methods is in one place, and the more newtypes we use, the more profit we gain from our macros. In a larger project, we could have additional macros for additional convenience methods. And perhaps another one that calls all the others. But that is left as an exercise for the reader.

> **NOTE** **The derive_more crate is worth a mention before you start writing too much yourself. It can automatically implement some basic traits for wrappers like these. It uses procedural macros though, not declarative ones.**

2.4 DSLs

Declarative macros are also good for creating *Domain Specific Languages*, or DSLs. As mentioned in the introduction, DSLs encapsulate knowledge about the domain - that the developers have learned from domain experts - into the application. This idea of capturing domain knowledge is related to that of a ubiquitous language, which comes from Domain Driven Design and argues that communication between experts and developers is easier when they use the same words and concepts. This leads to better code.

The goal of a DSL is to create a specialized language that is suitable to the domain and hides irrelevant complexities. Among these complexities could be things like additional validation and taking care of certain subtleties. Some think that a DSL could make a codebase understandable to non-programmers and one idea behind the cucumber.io [cucumber testing framework] was to write tests in a language that the domain experts could understand. Another idea was that experts would even be able to add their own tests using these tools! Within the Rust ecosystem, mini-DSLs abound. And they are often created by using macros. Two simple examples from the standard library are `println!` and `format!`, which offer a special syntax using curly braces to determine how to print specified variables.

As an example, we will write a DSL for handling transfers between accounts. First, we create an `Account` struct that contains the amount of `money` in our account. It also has methods for adding and removing money from the account. We only want positive amounts (hence the use of `u32`) but handling accounts going into the negative is beyond scope. `Account` also derives `Debug`. A good idea in general, though our motivation here is simply to print some outcomes.

Listing 2.11 The Account structure

```
use std::ops::{Add, Sub};

#[derive(Debug)]
struct Account {
    money: u32,
}

impl Account {
    fn add(&mut self, money: u32) {
        self.money = self.money.add(money)
    }

    fn subtract(&mut self, money: u32) {
        self.money = self.money.sub(money)
    }
}
```

Now check out the `exchange` macro below, which presents users with a mini-DSL. As you can see, this macro allows us to use natural language, understandable to outsiders, to describe actions. And when the macro does not understand a command, it will complain about this at compile time. Also, when transferring money between two accounts (third pair), we are hiding the complexity of a transaction from the DSL user.

Listing 2.12 This macro presents a mini-DSL for exchanging money

```
macro_rules! exchange {
    (Give $amount:literal to $name:ident) => {
        $name.add($amount)
    };
    (Take $amount:literal from $name:ident) => {
        $name.subtract($amount)
    };
    (Give $amount:literal from $giver:ident to $receiver:ident) => {
        $giver.subtract($amount);
        $receiver.add($amount)
    };   ❶
}

fn main() {
    let mut the_poor = Account {
        money: 0,
    };
    let mut the_rich = Account {
        money: 200,
    };

    exchange!(Give 20 to the_poor);   ❷
    exchange!(Take 10 from the_rich);   ❷
    exchange!(Give 30 from the_rich to the_poor);   ❷

    println!("Poor: {:?}, rich: {:?}", the_poor, the_rich);   ❸
}
```

❶ We have to end `$giver.subtract($amount)` with a semicolon because the transcriber has two statements instead of one, and the compiler would complain about `expecting a ;`.

❷ Use natural language to specify transactions.

❸ This prints "Poor: Account { money: 50 }, rich: Account { money: 160 }"

And there is no need to stop here. You could add currency types and automatically convert them. Or you could have special rules for over-draught. Good testing is needed to make sure that your macro scenarios do not 'collide' with each other though. (Where you think you will land in matcher X, but you end up in Y.) To give a simple example, the below macro is meant for giving money to the poor and berates people who don't give anything. We call our macro and… are complimented for giving nothing. That's because the first clause accepts any literal, *which includes zero*. And since the macro checks the matcher in order and always matches the first more general clause, the second one is never reached. The solution - in this case - is simple, just switch the order of the cases! And the underlying rule is "write macro rules/matchers from most-specific to least-specific"

More worrisome is that this is a valid implementation to the compiler, so static checks are insufficient to root out bugs like these.

Listing 2.13 A faulty wealth-transfer macro

```
macro_rules! give_money_to_the_poor {
    (Give $example:literal) => {
        println!("How generous");
    };
    (Give 0) => {
        println!("Cheapskate");
    };
}

fn main() {
    give_money_to_the_poor!(Give 0);   ❶
}
```

❶ Prints "How generous"! That is not what we expected.

2.5 Composing is easy

Besides avoiding boilerplate, macros also help us do things that are inelegant, hard, or even impossible to do in plain old Rust. (And you could see DSLs as a specific example of this more general category.) Take *composition*, which is a common feature of functional programming. Composition allows you to combine simple functions into bigger ones, creating larger functionality from smaller building blocks. This is a very interesting way to build applications when you want to keep your functions simple, understandable, and easy to test. And it is useful as a way to combine components in a paradigm that avoids object interactions for application building. To make things more concrete: say we have functions to increment a number, to change a number to a string, and to prefix a string. These three are shown below.

Listing 2.14 Three simple functions

```
fn add_one(n: i32) -> i32 {
    n + 1
}

fn stringify(n: i32) -> String {
    n.to_string()
}

fn prefix_with(prefix: &str) -> impl Fn(String) -> String + '_ {
    move |x| format!("{}{}", prefix, x)
}   ❶
```

❶ The `String + _` in the return type is needed because we are passing in a `&str` and that lifetime has to be made explicit.

The pseudocode below shows how we would like to compose these functions, similar to how it works in other languages. We pass our three functions to `compose` and get back a *new function that expects one input*. That input is the same type as the parameter of the first function we passed in. In our case, that function is `add_one`, which expects an `i32`. Internally, our `composed` will pass this argument to `add_one`, which will output an incremented number. That incremented

number is given to the next function, `stringify`, which turns the number into a string. Finally, this string is passed to `prefix_with`. Since this last function needs two arguments, both a prefix and an input string, we already passed in a prefix. In functional lingo, the function has been *partially applied*.

```
fn main() {
    let composed = compose(
        add_one,
        stringify,
        prefix_with("Result: ")
    );
    println!("{}", composed(5));   ❶
}
```

❶ This should print "Result: 6"

And you are not limited to a few functions! You can keep adding more, as long as they accept a single parameter that matches the output of the previous function and return a value that the next one accepts.

> **NOTE** **In some implementations, the order in which `compose` calls the functions might be reversed, i.e. from the rightmost argument to the leftmost. In that case, you first give it the last function it should call, and end with the first!**

But how do we write this `compose`? We will start simple with two input functions. No macros are required to make this work, just a lot of generics. `compose_two` (see the code in 2.14) requires two functions as parameters, both of which take a single parameter and return a result. The second one has to take the first one's output as its single input:

```
Fn(FIRST)  SECOND

Fn(SECOND)  THIRD
```

We then return a function that takes the first function's input and has an output equal to the second one's output:

```
Fn(FIRST)  THIRD
```

Compared to the signature, the implementation is simple: it's a closure that takes one argument (our generic 'FIRST'), gives it to our first function, and gives that output ('SECOND') to our second function, which will produce the final result ('THIRD'). Because of the closure, we use `impl` in our return type, since every closure is unique and won't match a normal generic function type.

Listing 2.15 A function to compose two functions

```
fn compose_two<FIRST, SECOND, THIRD, F, G>(f1: F, f2: G)
    -> impl Fn(FIRST) -> THIRD   ❶
    where F: Fn(FIRST) -> SECOND,   ❷
          G: Fn(SECOND) -> THIRD   ❸
{
    move |x| f2(f1(x))   ❹
}

fn main() {
    let two_composed_function = compose_two(
        compose_two(add_one, stringify),
        prefix_with("Result: ")
    );   ❺
}
```

❶ `compose_two` is a function that takes two generic parameters, called `f1` and `f2`, and returns a function (well, something that implements a function) that takes one generic argument and returns another generic argument.

❷ In this part of the where clause, we determine that `f1` is a function that takes a generic argument called `FIRST` and returns a generic result which we call `SECOND`. `FIRST` becomes the input of `compose_two`.

❸ Also in the where clause, we decide that `f2` takes `SECOND` and returns `THIRD`. `THIRD` is the output of our `compose_two`.

❹ Given an argument, pass it to the first function we received and pass that result to the second one we received.

❺ Repeatedly call `compose_two` to compose our three functions from before.

So that is the first step. How would we go about making this work for multiple functions? One approach that I have occasionally seen in other languages: just write an implementation for the most common number of arguments: `compose_three`, `compose_four`… up to `compose_ten`. That should cover 90% of all cases. And if you need more, you could also start nesting the existing implementations (a `compose_ten` within a `compose_ten`). It is a decent workaround. And it is hard to do better with plain Rust tooling. Say I decided to write a `compose` with a vector of functions as a parameter. I would need some way to tell the compiler that each function in this vector takes the previous output as input if I want my code to compile. That's hard to express in Rust.

But expressing this idea in a declarative macro is trivial. Let's go through it step by step. If our `compose` macro receives a single expression, a function, it returns that expression. A simple base case. If the macro is invoked with two or more arguments (note the `+` after `$tail` which is for when there is more than a single expression in our tail) we need recursive magic. What we do is call our `compose_two` and pass in the two required arguments. The first is the first function we receive, `head`. The second is the *result* of calling our `compose` macro again, this time on the remaining arguments. If this second compose invocation receives a single expression, we end up

in the first matcher, and we only have to compose two functions with a simple `compose_two` call. In the other case, we end up in our second branch - again - and return a function that is the result of `compose_two` - again. This is, in fact, what we suggested doing manually with nesting! Except now the macro takes care of everything in the background.

Listing 2.16 Compose macro

```
macro_rules! compose {
    ($last:expr) => { $last };
    ($head:expr,$($tail:expr),+) => {
        compose_two($head, compose!($($tail),+))
    }
}

fn main() {
    let composed = compose!(
        add_one,
        stringify,
        prefix_with("Result: ")
    );
    println!("{}", composed(5));   ❶
}
```

❶ Prints "Result: 6"

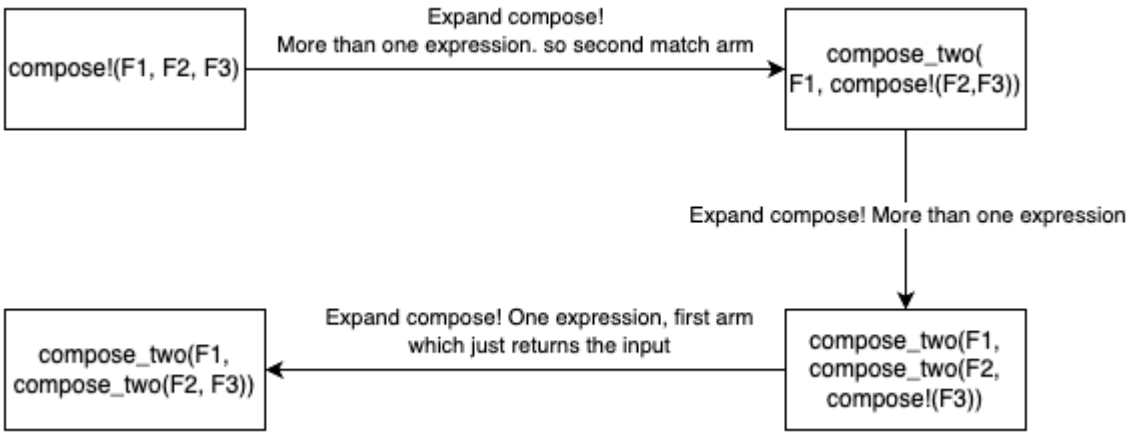

Figure 2.2 Compose macro with three functions, creatively called F1, F2 and F3

Commas are not the only way to separate the arguments that you pass to your macro. Similar to the brackets, you have some alternatives. In the case of `expr`, you can also go for a semicolon or an arrow. But if you're a Haskell fan, and you would like to compose using periods, that won't do. You will get an error that says `$head:expr is followed by '.', which is not allowed for expr fragments`. Other macro input types, like `tt`, have more options though.

Listing 2.17 Alternative with arrows separating the inputs to the macro

```
macro_rules! compose_alt {
    ($last:expr) => { $last };
    ($head:expr => $($tail:expr)=>+) => {
        compose_two($head, compose!($($tail),+))
    }  ❶
}

fn main() {
    let composed = compose_alt!(
        add_one => stringify => prefix_with("Result: ")
    );  ❷
    println!("{}", composed(5));
}
```

❶ Within the second matcher, we have replaced the commas with arrows.

❷ And now we use arrows in our invocation as well.

2.6 Currying on the other hand

Suppose you are now hooked on a functional style of Rust, but you find the 'one input' requirement of `compose` an annoyance. Luckily, you find out that there is something called currying, which takes a function with multiple arguments and turns it into multiple functions each taking one parameter. E.g. currying turns

`Fn(i32, i32) → i32`

into

`Fn(i32) → Fn(i32) → i32`.

That would make partial application of functions easier, which would in turn make composing easier. So you want to make this possible in Rust. And after your positive experience with composing, you think declarative macros are the way to go.

You would soon discover that currying with declarative macros is harder, though. One issue is the lack of 'visibility' of the function's signature: what does it require as parameters and what does it return? Going back to our previous approach, we could start with the easiest case, a `curry2` function for changing a function with two arguments into a curried version. Now all I have to do is recursively call this function. But how many times should I (well, the macro) call it? Without access to the signature, I'm in the dark, whereas with composing I knew how many calls had to happen because the functions *were given to my macro* as arguments.

So whereas composing was trivial, currying is hard. And while explicitly passing in details would help, that is exactly the kind of busywork we are trying to avoid. Because they are explicit about their number of arguments, closures are easier. And so there's a crate for currying them.

Take a look at the simplest rule in that crate.

```
macro_rules! curry {
    (|$first_arg:ident $(, $arg:ident )*| $function_body:expr) => {
       move |$first_arg| $(move |$arg|)* {
          $function_body
       }
    };
    // ...
}
```

This matcher expects one or more identifiers within pipes (`| |`), followed by an expression. Take the call `curry!(|a, b| a + b);` as an example. `a` and `b` are our two identifiers and `a + b` is our expression. All the transcriber has to do is add a `move` for each of these identifiers and pass them to the function. In our example, this becomes `move |a| move |b| a + b;`. Suddenly, we have two closures, each of which takes one argument. But again, this works because everything you need to know is passed in as an argument. Normal functions have this information in their signature, making good solutions harder (though probably not impossible). Instead, as this blog post shows, procedural macros offer the right tools for the job. And while its result is more complicated and has more lines of code than the average declarative macro from this chapter, the actual solution is still decently short: under a hundred lines.

2.7 Hygiene is something to consider as well

You should be aware that not everything you generate will be added 'as-is' to your code because declarative macros have a concept of hygiene for identifiers. Simply put, identifiers inside your macro will *always* be different from those in code outside the macro, even if their names overlap. Which means a number of things are impossible. For example, I cannot let `generate_x!()` output `let x = 0` *and then increment that named variable* in my ordinary code. If I try that, Rust will complain that it `cannot find value x in this scope`. That's because the `x` I initialized in my macro is not the `x` I am trying to increment in my application.

The way I am presenting this, hygiene sounds like a bad thing. But it is useful as a safeguard against contamination. A difference in intent is something to consider. If I am getting an identifier via my input and writing an implementation block, I *want to affect code outside of the macro*. Else what would be the point of that implementation? But identifiers created within my macro can serve other goals. Maybe I want to perform a calculation? Or push things into a vector. That stuff is independent of what I am doing outside the macro. And since developers often go for variable names that are easy to understand, there is a chance that a user of my macro will have the same variable name(s) in his own code. So when you see compiler errors around 'unresolved' and 'unknown' identifiers that are somehow related to macro code, remember this behavior. And if you want to have an effect on an identifier, just pass it into the macro as an argument.

2.8 From the real world: Lazy Static

Lazy static is a well-known crate that helps you create lazily initiated `static` values. This is useful for several reasons. For one, maybe the value inside your static requires a lot of computation. If it turns out the value is never needed, you never pay its initialization price. Another advantage is that the initialization happens at runtime, which has some additional options on top of those available at compile time. (Yes, this seems to contradict the preference for compile-time stuff that often bubbles up in this book. As always in software: it depends. Compile time work makes things safer and faster. But at runtime, you can do some things you could not do during compilation.)

At the very root of the crate is the lazy static macro. A `#[macro_export]` makes sure we can use it outside this file. The matchers can have `meta`, i.e. metadata, optionally (note the asterisk) passed in and the `static` is *literally* required as an input. A static also has an identifier, a type, an initialization expression as well as some optional additional info (thrown together in TokenTrees). The transcriber makes some modifications to the input and passes it along to another, internal macro.

Listing 2.18 Lazy static macro entrypoint, simplified slightly

```
#[macro_export]
macro_rules! lazy_static {
    ($(#[$attr:meta])* static ref $N:ident : $T:ty = $e:expr; $($t:tt)*) => {
        __lazy_static_internal!($(#[$attr])* () static ref $N : $T = $e; $($t)*);
    };
    // ...
}
```

Most of the implementation is hidden inside this internal macro, shown below. You could call the `@MAKE TY` and `@TAIL` a mini (nano?) DSL. It is used to make sure the other matcher-transcriber pairs within the macro are called. It is a pattern that also appears in The Little Book of Rust Macros. The first of these two additional arms is responsible for creating the type, which is just a struct with an empty internal field, decorated with the original metadata that was passed in (so those don't get lost!). The second arm creates a `Deref` and initialization. This is where the magic happens. If you need your lazily initialized static, you will dereference it when you start using it. Only at that point will the initialization expression that you provided run (you can see it being passed to the `__static_ref_initialize` function within `deref`). Underneath this all, `Once` from the spin library is used to make sure this initialization only happens one time.

Listing 2.19 Lazy static internal, again simplified

```
macro_rules! __lazy_static_internal {   ❶
    ($(#[$attr:meta])* ($($vis:tt)*) static ref
    [CA] $N:ident : $T:ty = $e:expr; $($t:tt)*) => {   ❷
        __lazy_static_internal!(@MAKE TY, $(#[$attr])*, ($($vis)*), $N);   ❸
        __lazy_static_internal!(@TAIL, $N : $T = $e);   ❹
    };
    (@TAIL, $N:ident : $T:ty = $e:expr) => {   ❹
        impl $crate::__Deref for $N {
            type Target = $T;
            fn deref(&self) -> &$T {
                fn __static_ref_initialize() -> $T { $e }

                fn __stability() -> &'static $T {
                    __lazy_static_create!(LAZY, $T);
                    LAZY.get(__static_ref_initialize)
                }
                __stability()
            }   ❺
        }
        impl $crate::LazyStatic for $N {
            fn initialize(lazy: &Self) {
                let _ = &**lazy;
            }
        }
    };
    (@MAKE TY, $(#[$attr:meta])*, ($($vis:tt)*), $N:ident) => {   ❸
        $(#[$attr])*
        $($vis)* struct $N {__private_field: ()}
        $($vis)* static $N: $N = $N {__private_field: ()};
    };
}

macro_rules! __lazy_static_create {   ❶
    ($NAME:ident, $T:ty) => {
        static $NAME: $crate::lazy::Lazy<$T> = $crate::lazy::Lazy::INIT;
    };
}
```

❶ These are declarative macro declarations, one for a macro called `lazy_static_internal` and one for `lazy_static_create`

❷ The first matcher expects optional attributes, optional visibility, a literal 'static ref' value, an identifier, type and expression. Finally, and again optionally, we capture everything else in a `TokenTree`

❸ We call our own macro and the `@MAKE TY` literal makes sure we end up in the last arm

❹ Similarly, here `@TAIL` makes sure we end up in the next arm

❺ This contains a lot of the magic of lazy static, using `Deref` to initialize our static

There, now you know like 80% (citation needed) of how the lazy static crate works!

2.9 Exercises

- In our first declarative macro example we use `expr` in some of our matches. But that was not our only option. Try to replace that with `literal`, `tt`, `ident`, or `ty`. Which ones work? Which don't? Do you understand why?
- Add a matcher to `my_vec` that allows for trailing commas. Try to avoid code duplication. If you need help, take a look at the `vec` macro from the standard library for inspiration.
- Another thing I like for newtypes is convenient `From` implementations. Write a macro that generates them for our four newtypes. Alternatively, you can go for `TryFrom` since that is a more suitable choice when input validation is required.
- Now that we have two macros, we could make our lives even easier by creating a third macro, `generate_newtypes_methods`, that calls our existing two macros behind the scene.
- Expand our Account example with dollar and euro currencies. You can use a hardcoded exchange rate of 2 dollars to 1 euro. All existing commands will require a currency type.
- In the upcoming procedural chapters, ask yourself: could I have done this with a declarative macro? Why not? What would be hard to do?

2.10 Summary

- Declarative macros are the first group of macros that Rust has to offer.
- They consist of one or more pairs of matchers and transcribers.
- The matcher has to match the content that was passed into the macro when it was invoked.
- If there is a match, the code inside the transcriber will be written to where the macro was invoked.
- Pieces of input can be captured in the matcher and used in the transcriber.
- Macros can call themselves, to avoid duplication and to allow for more complex scenarios.
- To use macros outside the file where they were defined, you will need to export them.
- Declarative macros have hygiene, which means local identifiers do not collide with external ones.
- There are several use cases for declarative macros: avoiding duplication and boilerplate is a major one. Another is doing things that are hard - or impossible - to do otherwise, like default arguments, varargs, or DSLs.
- If declarative macros fall short, you still have procedural macros waiting in the corridor to assist you with even more powerful weapons.

3 A 'hello world' procedural macro

In the beginning was the Word

– John 1:1

This chapter covers

- Setting up a procedural macro
- Getting the name of a struct by parsing a stream of tokens
- Generating hardcoded output
- Using variables in generated code
- Inspecting generated code with cargo expand
- Writing a macro without help from `syn` and `quote`
- Understanding how Rust's internal macros are different

We now come to the meat of our book, procedural macros. As we explained earlier, both procedural and declarative macros are forms of metaprogramming and allow you to manipulate and extend code. But they go at it differently. Declarative macros offer a DSL that allows you to generate code based on a combination of matchers and transcribers. Procedural macros, on the other hand, deal with lower-level information. They receive a stream of tokens containing *every detail* of the code you want to work with.

In my mind, the difference between declarative and procedural macros - and when you should use them - is a bit like SQL and general-purpose programming languages when it comes to querying databases. SQL is powerful, expressive, and user-friendly. It should be your first choice for querying. But at a certain level of complexity and for some kinds of tasks, it breaks down. It becomes complicated, difficult to read and extend. At that point, it can be worthwhile to replace SQL with a general-purpose language. Querying might require more effort and setup, but you will have more options and power. So the advice here is to start with declarative macros. They are simple, powerful, require only minimal setup, and have better IDE support to boot. If you

want to do things that declarative macros can't (for example: manipulating existing structs), you should turn to procedural macros.

And as we shall see, there are quite a few things that are easier or nicer to develop with procedural macros. Here we start simple, with a macro that adds a "hello world" printing method to a struct or enum. Adding new functionality to a struct is a good use case for a *derive* macro. On the other hand, if we wanted to *change* existing code, we would have to look elsewhere, since these macros are incapable of doing that. Derive macros are activated by adding the `[derive]` annotation to your code, putting the name of the macro between brackets. No doubt you have encountered these annotations before when you wanted to add `Debug` (`[derive(Debug)]`) or `Clone` (`#[derive(Clone)]`) functionality to your code.

3.1 Basic setup of a procedural macro project

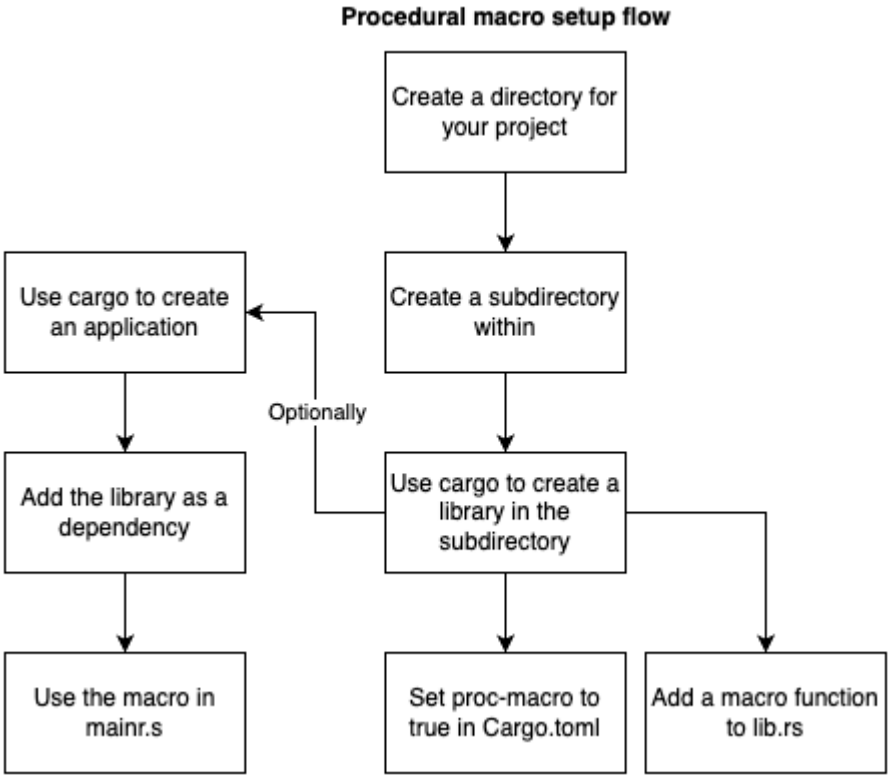

Figure 3.1 Basic setup of a procedural macro project

Creating a procedural macro takes some work. So bear with me while we go through the setup. First, we need a `hello world` directory with another directory called `hello-world-macro` inside the former. In `hello-world-macro` we will put our macro project. You are probably used to creating new Rust projects with `cargo init` However, in this case, we do not need an application but a **library**. Developers that want to use our macro will import our library as a dependency. So run `cargo init --lib` in `hello-world-macro`.

We have to modify the generated `Cargo.toml`. The most important change is adding a `lib` section, with the `proc-macro` property set to `true`. This tells Rust that this library will expose one - or more - procedural macros, and will make some tooling available to us. We also want to add `quote` and `syn` as dependencies. These are not strictly required, but they will make our lives much easier.

Listing 3.1 The Cargo.toml file from hello-world-macro, containing the `lib` section and two useful dependencies

```
[package]
name = "hello-world-macro"
version = "0.1.0"
edition = "2021"

[dependencies]
quote = "1.0.20"
syn = "1.0.98"

[lib]
proc-macro = true
```

Inside the automatically generated `lib.rs` file, we add a basic implementation - that does not really do anything yet. We will discuss the code in the next section. For now, let us continue our setup.

Listing 3.2 The initial lib.rs file from our `hello-world-macro` library

```
use quote::quote;
use proc_macro::{TokenStream};

#[proc_macro_derive(Hello)]
pub fn hello(_item: TokenStream) -> TokenStream {
    let add_hello_world = quote! {};
    add_hello_world.into()
}
```

The library is now ready. In the outer `hello-world` directory, we will set up a Rust example **application** with `cargo init`. This time, we add a dependency on the macro library we just created using a relative path. The name of the dependency is important and should match the *name of the package* you are importing (the path and directory name can be different though). Try renaming it to 'foo-bar', and do a `cargo run`. You will get something like `error: no matching package named foo-bar found`.

Listing 3.3 The Cargo.toml file from our outer Rust application

```
[package]
name = "hello-world"
version = "0.1.0"
edition = "2021"

[dependencies]
hello-world-macro = { path = "./hello-world-macro" }
```

Finally, modify the default `main.rs`. You should now be able to compile and run your application with `cargo run`. It currently prints nothing.

Listing 3.4 The initial main.rs file from the outer application

```
#[macro_use]
extern crate hello_world_macro;

#[derive(Hello)]
struct Example;

fn main() {}
```

NOTE If you don't like this setup or all the manual work, you could use cargo generate which has a template for generating a sensible macro setup. For learning purposes, setting things up manually is much more interesting though.

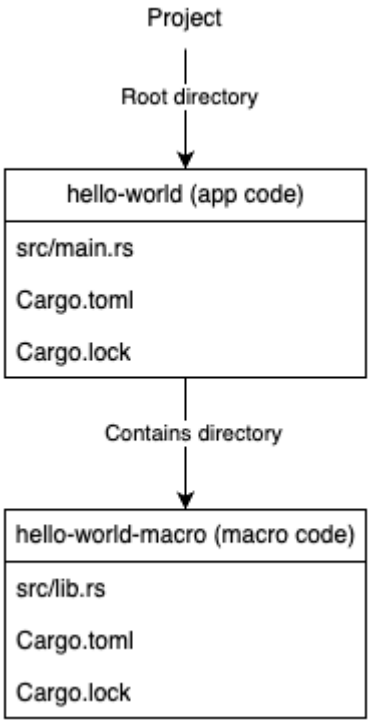

Figure 3.2 Our project structure

3.2 Analyzing the procedural macro setup

Now let us analyze the setup. The nested directory with `lib.rs` is the only piece of code we really need.

```
use quote::quote;
use proc_macro::{TokenStream};

#[proc_macro_derive(Hello)]   ❶
pub fn hello(_item: TokenStream) -> TokenStream {
    let add_hello_world = quote! {};   ❷
    add_hello_world.into()   ❸
}
```

❶ Declare this function to be a derive macro named 'Hello'

❷ Generate a new (currently empty) TokenStream using the `quote` macro

❸ Use the `Into` trait to change this `TokenStream` into the normal/standard library `TokenStream` with the same name

Below the import of dependencies, we have `#[proc_macro_derive(Hello)]`. This is an attribute, a piece of metadata that informs Rust that something should happen. In our case, it tells Rust that this function is an entry point to a derive macro. It requires a name, which we set to 'Hello'. This name will be used when invoking our macro.

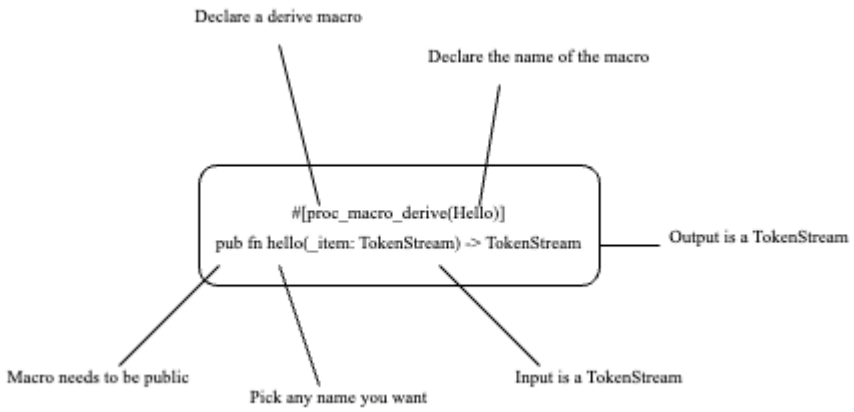

Figure 3.3 The signature of a derive macro

Moving on: the function signature. It has a single parameter of type `Tokenstream`, which we are not using yet. Therefore, it has been prefixed with an underscore. In earlier chapters, we already mentioned how procedural macros operated on a stream of tokens, a representation of the code that we want to modify, a struct for example. Everything there is to know about that struct will be contained inside this Tokenstream. Meanwhile, the `TokenStream` that we are returning will be used by Rust to generate additional code.

NOTE **Briefly about parsing and tokens**

Here is a high-level overview of what Rust does when you tell it to compile your application. (And I am not an expert on the details.) There are two steps involved: lexing and parsing. Lexing - also called tokenization - is the first step and is used to turn your code into a stream of tokens. Parsing turns this information into an Abstract Syntax Tree or AST, a tree-like representation of all relevant data in your program. This AST makes it easier for the Rust compiler (as well as compilers in other languages) to do its work, i.e. creating an executable that computers can understand. Meanwhile, 'spans' are used to link back to the original text - which is useful for things like error messages. Once the AST has been constructed, macros can be processed by the compiler.

While these are interesting factoids, you do not need to know the details of this process to write procedural macros. The most important takeaway is that Rust gives us code as parsed tokens. In return, we graciously give back information in the same format. Rust knows how to use these tokens. After all, all 'normal' code is also turned into tokens! Which means it is easy to turn your tokens into additional application code.

This brings us to the body of our function, where we are calling the `quote` macro from the dependency with the same name. Right now we are not passing any parameters into `quote`, which means we are generating an empty Tokenstream. Because `quote` is actually using the *proc_macro2* `TokenStream`, we still have to use the `Into` trait to transform it into a 'normal' *proc_macro* `TokenStream`: `add_hello_world.into()`. So the end result is a macro that… generates nothing. Zero runtime overhead would be the marketeer's pitch for this version of our library. But there is no obligation for a macro to generate code, so this is acceptable to Rust.

Now take a look at `main.rs`.

```
#[macro_use]
extern crate hello_world_macro;   ❶

#[derive(Hello)]   ❷
struct Example;   ❸

fn main() {}
```

❶ Make the macros inside the `hello_world_macro` directory available for use

❷ Now add the 'Hello' derive macro…

❸ … to our empty `Example` struct

The first two lines tell Rust that we will want to use macros from this dependency, similar to how we import declarative macros. (Note that this time we need to specify the dependency using underscores. In our `Cargo.toml` we used hyphens.) This is the older style for importing

procedural macros. Nowadays, you can also import them with a `use` one-liner, e.g. `use hello_world_macro::Hello;`. We will use both styles in this book. Next, the `#[derive(Hello)]` attribute tells Rust to run our derive macro named 'Hello' for the `Example` struct, passing that struct along as an input of type `TokenStream`. The (currently empty) output code will be added to our `main.rs` file.

Earlier we said that the outer application was not strictly necessary. Instead, it is a convenience, a 'consumer' that helps us verify that our code compiles. This is useful because Rust's compiler will not catch errors in code generated by your library. To make this more concrete, say you make a typo when adding the parameter for the derive macro function and write `TokenStrea`. If you run `cargo check`, Rust will point out that the `TokenStrea` does not exist. Failure prevented! But what if we write invalid Rust within the `quote` macro invocation?

```
#[proc_macro_derive(Hello)]
pub fn hello(item: TokenStream) -> TokenStream {
    let add_hello_world = quote! {
        fn this should not work () {}
    };

    add_hello_world.into()
}
```

Running `cargo check` inside our macro library will still work. But when we do a `check` in the outer application, we get an error:

```
error: expected one of `(` or `<`, found `should`
 --> src/main.rs:4:10
  |
4 | #[derive(Hello)]
  |          ^^^^^ expected one of `(` or `<`
  |
  = note: this error originates in the derive macro `Hello`
```

Apparently, Rust thinks that `fn this` means we want to create a function called `this`. A function and its name are followed by either parenthesis containing the parameters or generics (which are surrounded by <>). Instead, the word `should` appears… The lesson here is that `cargo check` inside your procedural macro library will only check for mistakes in your *library code*, not in the *code it generates*.

This makes sense because within the library we are not using the generated code. All Rust cares about, is that we declared `TokenStream` to be the return type of our function. And as long as we return *any* stream of tokens, even an invalid one, it is happy. Even if it did care, the compiler has no way of knowing in what context your generated code will be used, which makes checking hard. So instead we use a simple application with a basic usage example to generate code and force the compiler to stop being lazy and do some work.

3.3 Generating output

So how about actually producing some code? Start by adding the following to `lib.rs`:

Listing 3.5 Parsing the input and producing hardcoded output

```
// earlier imports
use syn::{parse_macro_input, DeriveInput};

#[proc_macro_derive(Hello)]
pub fn hello(_item: TokenStream) -> TokenStream {
    let add_hello_world = quote! {
        impl Example {
            fn hello_world(&self) {
                println!("Hello world")
            }
        }
    };  (1)

    add_hello_world.into()
}
```

(1) Return a hardcoded implementation block

We are still not doing anything with the incoming tokens. But we are using `quote` to generate new code. Right now that new code is a hardcoded implementation block for the `Example` struct we added to `main.rs`. This means we can now call this method and the below code should run. Do remember that the target for `cargo run` should be the application. Else you will get `a bin target must be available for cargo run` since a library cannot be executed.

Listing 3.6 Calling our generated function in our main file

```
// the macro import

#[derive(Hello)]
struct Example;

fn main() {
    let e = Example {};
    e.hello_world();
}  (1)
```

(1) Prints 'Hello world' when executed

The only downside is that our code only works for structs named 'Example'. Rename the struct and Rust will complain that it `cannot find type Example in this scope` as it can't match the implementation block to an existing type. The solution is to retrieve the name of the struct from the incoming tokens and use it for the `impl` that we are generating. In the terminology of macros, both declarative and procedural, this name is an *identifier*. And since our tokens represent the code that we decorated (a struct and its contents) we will need the topmost identifier. Once we have the name, we can combine it with `quote` to produce more interesting output.

Listing 3.7 Parsing the input and using it in our output

```
use quote::quote;
use proc_macro::{TokenStream};
use syn::{parse_macro_input, DeriveInput};

#[proc_macro_derive(Hello)]
pub fn hello(item: TokenStream) -> TokenStream {
    let ast = parse_macro_input!(item as DeriveInput);   ❶
    let name = ast.ident;   ❷

    let add_hello_world = quote! {
        impl #name {   ❸
            fn hello_world(&self) {
                println!("Hello world")
            }
        }   ❹
    };

    add_hello_world.into()
}
```

❶ Parse the incoming `TokenStream` into a more user-friendly Abstract Syntax Tree, creatively called `ast`

❷ Retrieve the top-level identifier. In our case, this will be the name of the annotated (Example) struct

❸ Use the special syntax that `quote` provides to add it to our output

❹ Everything else can remain hardcoded

We start by parsing the input tokens into an Abstract Syntax Tree by using `parse_macro_input`, provided by the `syn` crate. `syn` offers a lot of tools to help you parse Rust tokens, and this declarative macro is one tool in its arsenal. The '**as** DeriveInput' is a bit of custom syntactic sugar. Behind the scenes, the argument after 'as' is used to determine the type that will be generated. We have gone for `DeriveInput`. As the name suggests, this is any kind of input that you might get when writing a derive macro. So it basically contains either an enum or a struct.

Once we have the `DeriveInput`, we can get the name of the struct by retrieving the topmost ident(ifier), which we save in a variable called `name`. If we want to use `name` in our output, we need a special syntax to tell `quote` that this is not a literal value but should be replaced with the content of a variable with a matching name. You can do that by prefixing the name of your variable with a hashtag, i.e. `#name`. Which, in our case, will be replaced with the identifier 'Example'. As you can see, generating output with `quote` is delightfully easy.

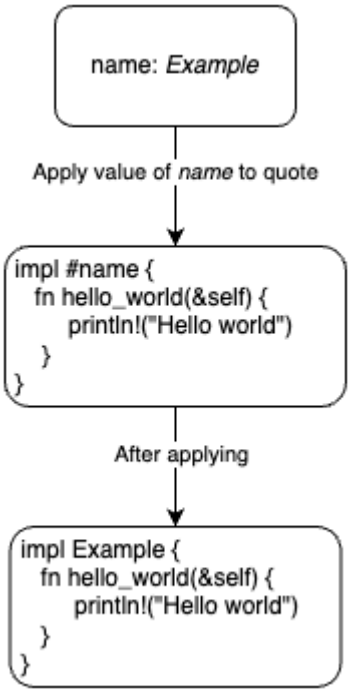

Figure 3.4 Using the name variable in quote for our `Example` struct

Run the code again, it should produce the same output as before, even if you change the name of your struct!

3.4 Experimenting with our code

Now let us run some experiments. Does our macro work with enums? Yes! Add the following to `main.rs`.

Listing 3.8 Invoking our macro for an enum

```
// previous code

#[derive(Hello)]
enum Pet {
    Cat,
}

fn main() {
    // previous code

    let p = Pet::Cat;
    p.hello_world();
}
```

How about functions? Unfortunately, no. Not only because `impl` cannot be used for functions, but also because derive is not allowed for functions. The error brings this point across quite clearly: `derive may only be applied to structs, enums and unions`.

> **NOTE** Though unions are a valid derive target, they do not feature in this book. Mostly because they are much less ubiquitous than structs and enums.

Would our generated code overwrite existing implementation blocks? Thankfully no. As we mentioned in an earlier chapter, Rust supports multiple `impl` blocks. So this works:

Listing 3.9 Multiple implementation blocks

```
// earlier code

impl Example {
    fn another_function(&self) {
        println!("Something else");
    }
}

fn main() {
    let e = Example {};
    e.hello_world();
    e.another_function();
    // other code
}
```

Other things might still go wrong though. For example, if you were to define an additional `hello_world` function, you would get an error `duplicate definitions for hello_world`. You'd better hope no users of your macro will ever think to add a function with that name! Name overlap is a real risk - and something we will talk about later in this book.

3.5 Cargo expand

In our previous chapter, we introduced a couple of ways to debug macros and to show what they expanded to. But we skipped over one very useful tool: cargo expand, which was unstable for some time but can now be used with stable Rust. You can install it with `cargo install cargo-expand`, after which you can run `cargo expand` in the root of our application directory, or the `src` folder. This will print your application code after macro expansion. Below you can see that all macros, including `println!`, were expanded.

> **NOTE** Except... `format_args!` was not expanded? That's a Rust issue and fixing it is a work in progress.

Listing 3.10 Our cargo expand output (with changed formatting)

```
// other generated code
impl Example {
    fn hello_world(&self) {
        {
            ::std::io::_print(
                format_args!("Hello world\n")
            );  ❶
        }
    }
}  ❷
```

❶ An expanded `println!`

❷ An expanded `#[derive(Hello)]`

`cargo expand` is a useful tool for a visual inspection of our code. And it runs even when our output is invalid, making it useful for debugging compilation issues. Say I had mistyped `self` in our `quote` macro invocation. I would get a compilation error and the below output, showing that something was very wrong with the function parameter. If the error still wasn't clear, I could pipe the output to a file (`cargo expand > somefile.rs`) and have my IDE help me track down the issue. Or I could temporarily replace my `main.rs` and get pointed to the incorrect lines by `cargo check`.

```
impl ExampleStruct {
    fn hello_world(sel: ()) {
        {
            ::std::io::_print(format_args!("Hello world\n"));
        }
    }
}
```

You can also use `expand` within the `hello-world-macro` library. But that will only show you how the macros you use - not the ones you create for others - are expanded, similar to how `check` will only point out errors in your code, and not in generated code. So most of the time, you want to use the `expand` command in applications that utilize your library and its macro.

3.6 The Same Macro - without syn and quote

The `quote` and `syn` libraries are very useful but not strictly necessary for writing macros. Below is the same application without them. To retrieve the name, we iterate over the incoming stream and take the second element with `nth(1)`. That item is a `TokenTree` containing the name of the struct or enum. The first element, contained in `nth(0)`, has the type, i.e. `struct` or `enum`, and is not relevant in this situation - so we skip it. With the `ident_name` function we take the tree and return the name identifier, or throw an error if we cannot find it.

NOTE **A `TokenTree` sits somewhere between a `TokenStream` and simple tokens. Basically, a `TokenStream` is a sequence of `TokenTrees`, which are themselves (recursively) composed of more trees and/or tokens. This is why we can iterate over our stream, pick an element and assure Rust that the type is `TokenTree`.**

To generate output, we use the `format` macro to inject our `name` variable in a string. For the transformation from string to `TokenStream`, we can use `parse`. We expect this to work, so we just use an `unwrap` on the `Result` that our parsing returns. This approach might seem very doable. We have removed two dependencies at the cost of a little bit of extra code. But even with such a basic example, we had to put in work to get the identifier and to output the new code. And with more complete examples, the complexity and additional burden of work would only grow.

Listing 3.11 Without syn and quote

```
use proc_macro::{TokenStream, TokenTree};

#[proc_macro_derive(Hello)]
pub fn hello_alt(item: TokenStream) -> TokenStream {
    fn ident_name(item: TokenTree) -> String {
        match item {
            TokenTree::Ident(i) => i.to_string(),
            _ => panic!("No ident")
        }
    }  ❶
    let name = ident_name(item.into_iter().nth(1).unwrap());  ❷

    format!("impl {} {{ fn hello_world(&self) \
    {{ println!(\"Hello world\") }} }} ", name)
        .parse().unwrap()  ❸
}
```

❶ If we have a `TokenTree` containing an identifier, we return it as a string

❷ As an input for the previous function, we provide the second element of the `TokenStream`, which should be the name

❸ We write our `impl` block as a string and use parse to turn it into the expected `TokenStream`. This will only fail if we made a mistake, in which case `unwrap` panicking is fine

Even so, compilation speed is a reason why you might want to opt out of using `syn`. While it is a very powerful library, it is also big and slow to compile. So if our example was a real macro, and we only needed the name of the struct/enum, our 'naive' example would compile a lot faster. Several libraries try to offer a lightweight alternative to `syn`, venial for example. Below is what our macro looks like with that library. It looks very similar to what we had before. We now use `parse_declaration`, which gives back an enum `Declaration`. With a `match` we retrieve the name from that enum.

Listing 3.12 Using a lightweight parser like venial

```
use quote::quote;
use proc_macro::{TokenStream};
use venial::{parse_declaration, Declaration, Struct, Enum};

#[proc_macro_derive(Hello)]
pub fn hello(item: TokenStream) -> TokenStream {
    let declaration = parse_declaration(item.into()).unwrap();

    let name = match declaration {
        Declaration::Struct(Struct { name, .. }) => name,
        Declaration::Enum(Enum { name, .. }) => name,
        _ => panic!("Only implemented for struct and enum")
    }; ❶

    let add_hello_world = quote! {
        impl #name {
            fn hello_world(&self) {
                println!("Hello world")
            }
        }
    };

    add_hello_world.into()
}
```

❶ Retrieve the name if we receive an enum or a struct, panic in all other cases.

Even in this simple example, build times measured with `cargo build --timings` drop from 3.1 seconds to 1.8 on my machine. Still, in this book, we will use `syn` because it is well-known, widely used, and very powerful. Plus, once you are familiar with how it handles the `TokenStream` parsing, switching to a lightweight alternative should not be too hard: many of the parsing concepts are always the same.

3.7 From the real world: Rocket's import documentation and Rust's internal derives

We will save further library explorations for subsequent chapters and limit ourselves to a few observations for now. First, the developers of Rocket are actually kind enough to teach you that macros can be imported in two ways (the ones we described in this chapter).

```
//! And to import all macros, attributes, and derives via `#[macro_use]`
//! in the crate root:
//!
//! ```rust
//! #[macro_use] extern crate rocket;
//! # #[get("/")] fn hello() { }
//! # fn main() { rocket::build().mount("/", routes![hello]); }
//! ```
//!
//! Or, alternatively, selectively import from the top-level scope:
//!
//! ```rust
//! # extern crate rocket;
//!
//! use rocket::{get, routes};
//! # #[get("/")] fn hello() { }
//! # fn main() { rocket::build().mount("/", routes![hello]); }
//! ```
```

Second, you may also be wondering how the standard library parses and outputs macros, since it cannot use external libraries like `syn` and `quote`. Instead, the standard library uses built-ins with similar concepts and names. For example `rustc_ast`, the 'Rust Abstract Syntax Tree' is used for parsing input. Outputting code is done with `rustc_expand`. And `rustc_span` contains utilities like `Ident` and `Span`. It is both familiar and alien when you are used to working with `syn` and `quote`.

Finally, since procedural macros have to be placed in a library & have to placed in the root of the crate (else you will get `functions tagged with #[proc_macro_derive] must currently reside in the root of the crate`), `lib.rs` is a great starting point for exploring other people's procedural macro code. You will see what macros they have and can dig in when needed.

3.8 Exercises

- Try changing the name of the macro inside `lib.rs` and running the application. What error do you get? What do you have to do to fix things?
- Add a function called `testing_testing` to the output of our macro. This is an *associated* function, one that takes no `&self` parameter. It should write "One two three" to the console.
- See if you can output a greeting followed by the name of the input (e.g. "Hello Example"). Fair warning: passing `#name` to print will not be enough, because that's an identifier, and you need a string! So either call `to_string` on the identifier and save the result in a variable, or use the `stringify` macro to change the `#name` into a string for you.

3.9 Summary

- Derive macros, the first kind of procedural macro, allow us to add functionality to structs and enums.
- To write procedural macros we need to create a library with a function that accepts a `TokenStream` input and outputs another `TokenStream`.
- That output is the code that will be generated and added to our application.
- To verify our macro, we cannot rely solely on `cargo check` within the library, because that will only check the code within the library itself, not the code we generate. Therefore, it is useful to have an application that uses and tests our macro. At the very least, this way we can verify that the generated code compiles.
- You can write procedural macros with standard Rust tooling, but `syn` is of great help parsing input, and `quote` has a macro to generate output.
- You can retrieve values from your input and pass them into your output.
- Cargo expand is a great tool that allows you to see the code generated by macros in your code.