The Little Book of Rust Macros

Note: This is a continuation of Daniel Keep's Book which has not been updated since the early summer of 2016, adapted to make use of mdBook.

View the rendered version here and the repository here.

A chinese version of this book can be found here.

This book is an attempt to distill the Rust community's collective knowledge of Rust macros, the `Macros by Example` ones as well as procedural macros(WIP). As such, both additions (in the form of pull requests) and requests (in the form of issues) are very much welcome. If something's unclear, opens up questions or is not understandable as written down, fear not to make an issue asking for clarification. The goal is for this book to become the best learning resource possible.

The original Little Book of Rust Macros has helped me immensely with understanding **Macros by Example** style macros while I was still learning the language. Unfortunately, the original book hasn't been updated since April of 2016, while the Rust language as well as its macro-system keeps evolving. Which is why I took up the task to update the book and keep it updated as well as I can while also adding newfound things to it. In hopes that it will help out all the fresh faces coming to Rust understanding its macro systems, a part of the language a lot of people tend to have trouble with.

This book expects you to have basic knowledge of Rust, it will not explain language features or constructs that are irrelevant to macros. No prior knowledge of macros is assumed. Having read and understood the first seven chapters of the Rust Book is a must, though having read the majority of the book is recommended.

Thanks

A big thank you to Daniel Keep for the original work as well as all the contributors that added to the original which can be found here.

License

This work inherits the licenses of the original, hence it is licensed under both the Creative Commons Attribution-ShareAlike 4.0 International License and the MIT license.

Syntax Extensions

Before talking about Rust's different macro systems it is worthwhile to discuss the general mechanism they are built on: *syntax extensions*.

To do that, we must first discuss how Rust source is processed by the compiler, and the general mechanisms on which user-defined macros and proc-macros are built upon.

Note: This book will use the term *syntax extension* from now on when talking about all of rust's different macro kinds in general to reduce potential confusion with the upcoming declarative macro 2.0 proposal which uses the `macro` keyword.

Source Analysis

Tokenization

The first stage of compilation for a Rust program is tokenization. This is where the source text is transformed into a sequence of tokens (*i.e.* indivisible lexical units; the programming language equivalent of "words"). Rust has various kinds of tokens, such as:

- Identifiers: `foo`, `Bambous`, `self`, `we_can_dance`, `LaCaravane`, …
- Literals: `42`, `72u32`, `0_____0`, `1.0e-40`, `"ferris was here"`, …
- Keywords: `_`, `fn`, `self`, `match`, `yield`, `macro`, …
- Symbols: `[`, `:`, `::`, `?`, `~`, `@` [1], …

…among others. There are some things to note about the above: first, `self` is both an identifier *and* a keyword. In almost all cases, `self` is a keyword, but it *is* possible for it to be *treated* as an identifier, which will come up later (along with much cursing). Secondly, the list of keywords includes some suspicious entries such as `yield` and `macro` that aren't *actually* in the language, but *are* parsed by the compiler—these are reserved for future use. Third, the list of symbols *also* includes entries that aren't used by the language. In the case of `<-`, it is vestigial: it was removed from the grammar, but not from the lexer. As a final point, note that `::` is a distinct token; it is not simply two adjacent `:` tokens. The same is true of all multi-character symbol tokens in Rust, as of Rust 1.2.[2]
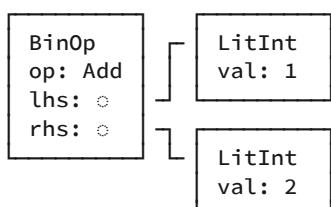
As a point of comparison, it is at *this* stage that some languages have their macro layer, though Rust does *not*. For example, C/C++ macros are *effectively* processed at this point. [3] This is why the following code works: [4]

```
#define SUB int
#define BEGIN {
#define END }

SUB main() BEGIN
    printf("Oh, the horror!\n");
END
```

Parsing

The next stage is parsing, where the stream of tokens is turned into an Abstract Syntax Tree (AST). This involves building up the syntactic structure of the program in memory. For example, the token sequence `1 + 2` is transformed into the equivalent of:

The AST contains the structure of the *entire* program, though it is based on purely *lexical* information. For example, although the compiler may know that a particular expression is referring

to a variable called a , at this stage, it has *no way* of knowing what a is, or even *where* it comes from.

It is *after* the AST has been constructed that macros are processed. However, before we can discuss that, we have to talk about token trees.
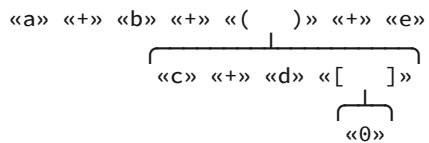
Token trees

Token trees are somewhere between tokens and the AST. Firstly, *almost* all tokens are also token trees; more specifically, they are *leaves*. There is one other kind of thing that can be a token tree leaf, but we will come back to that later.
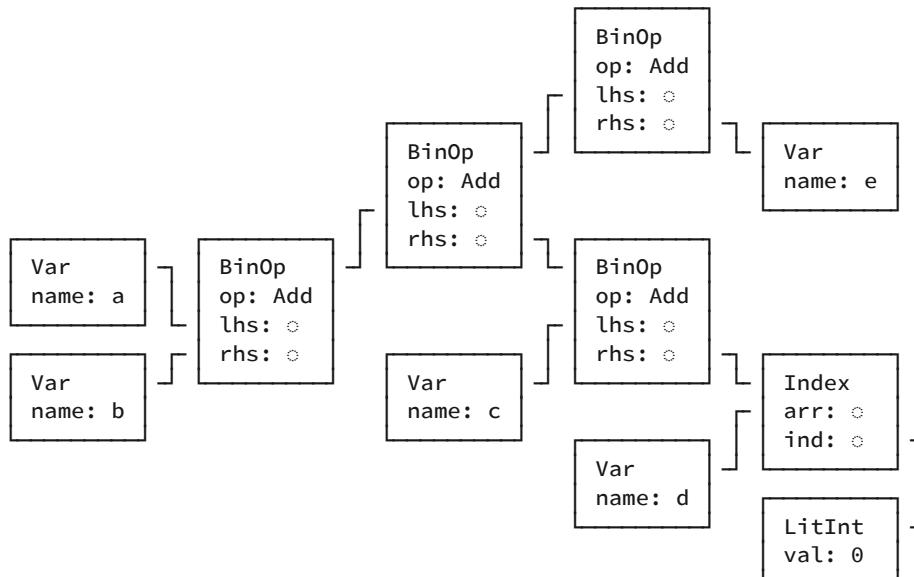
The only basic tokens that are *not* leaves are the "grouping" tokens: (...) , [...] , and {...} . These three are the *interior nodes* of token trees, and what gives them their structure. To give a concrete example, this sequence of tokens:

```
 a + b + (c + d[0]) + e
```

would be parsed into the following token trees:

Note that this has *no relationship* to the AST the expression would produce; instead of a single root node, there are *seven* token trees at the root level. For reference, the AST would be:

It is important to understand the distinction between the AST and token trees. When writing macros, you have to deal with *both* as distinct things.

One other aspect of this to note: it is *impossible* to have an unpaired parenthesis, bracket or brace; nor is it possible to have incorrectly nested groups in a token tree.

1. `@` has a purpose, though most people seem to forget about it completely: it is used in patterns to bind a non-terminal part of the pattern to a name. ↵

2. Technically rust currently(1.46) has two lexers, `rustc_lexer` which only emits single character symbols as tokens and the lexer in `rustc_parse` which sees multi-character symbols as distinct tokens. ↵

3. In fact, the C preprocessor uses a different lexical structure to C itself, but the distinction is *broadly* irrelevant. ↵

4. *Whether* it should work is an entirely *different* question. ↵

Macros in the AST

As previously mentioned, macro processing in Rust happens *after* the construction of the AST. As such, the syntax used to invoke a macro *must* be a proper part of the language's syntax. In fact, there are several "syntax extension" forms which are part of Rust's syntax. Specifically, the following 4 forms (by way of examples):

1. `# [ $arg ]` ; *e.g.* `#[derive(Clone)]` , `#[no_mangle]` , ...
2. `# ! [ $arg ]` ; *e.g.* `#![allow(dead_code)]` , `#![crate_name="blang"]` , ...
3. `$name ! $arg` ; *e.g.* `println!("Hi!")` , `concat!("a", "b")` , ...
4. `$name ! $arg0 $arg1` ; *e.g.* `macro_rules! dummy { () => {}; }` .

The first two are attributes which annotate items, expressions and statements. They can be classified into different kinds, built-in attributes, proc-macro attributes and derive attributes. proc-macro attributes and derive attributes can be implemented with the second macro system that Rust offers, procedural macros. built-in attributes on the other hand are attributes implemented by the compiler.

The third form `$name ! $arg` are function-like macros. It is the form available for use with `macro_rules!` , `macro` and also procedural macros. Note that this form is not *limited* to `macro_rules!` macros: it is a generic syntax extension form. For example, whilst `format!` is a `macro_rules!` macro, `format_args!` (which is used to *implement* `format!`) is *not* as it is a compiler builtin.

The fourth form is essentially a variation which is *not* available to macros. In fact, the only case where this form is used *at all* is with the `macro_rules!` construct itself.

So, starting with the third form, how does the Rust parser know what the `$arg` in (`$name ! $arg`) looks like for every possible syntax extension? The answer is that it doesn't *have to*. Instead, the argument of a syntax extension invocation is a *single* token tree. More specifically, it is a single, *non-leaf* token tree; `(...)` , `[...]` , or `{...}` . With that knowledge, it should become apparent how the parser can understand all of the following invocation forms:

```
bitflags! {
    struct Color: u8 {
        const RED    = 0b0001,
        const GREEN  = 0b0010,
        const BLUE   = 0b0100,
        const BRIGHT = 0b1000,
    }
}

lazy_static! {
    static ref FIB_100: u32 = {
        fn fib(a: u32) -> u32 {
            match a {
                0 => 0,
                1 => 1,
                a => fib(a-1) + fib(a-2)
            }
        }

        fib(100)
    };
}

fn main() {
    use Color::*;
    let colors = vec![RED, GREEN, BLUE];
    println!("Hello, World!");
}
```

Although the above invocations may *look* like they contain various kinds of Rust code, the parser simply sees a collection of meaningless token trees. To make this clearer, we can replace all these syntactic "black boxes" with ⬚, leaving us with:

```
bitflags! ⬚

lazy_static! ⬚

fn main() {
    let colors = vec! ⬚;
    println! ⬚;
}
```

Just to reiterate: the parser does not assume *anything* about ⬚; it remembers the tokens it contains, but doesn't try to *understand* them. This means ⬚ can be anything, even invalid Rust! As to why this is a good thing, we will come back to that at a later point.

So, does this also apply to `$arg` in form 1 and 2, and to the two args in form 4? Kind of. The `$arg` for form 1 and 2 is a bit different in that it is not directly a token tree, but a *simple path* that is either followed by an `=` token and a literal expression, or a token tree. We will explore this more in-depth in the appropriate proc-macro chapter. The important part here is that this form as well, makes use of token trees to describe the input. The 4th form in general is more special and accepts a very specific grammar that also makes use of token trees though. The specifics of this form do not matter at this point so we will skip them until they become relevant.

The important takeaways from this are:

- There are multiple kinds of syntax extensions in Rust.
- Just seeing something of the form `$name! $arg`, doesn't tell you what kind of syntax extension it might be. It could be a `macro_rules!` macro, a `proc-macro` or maybe even a builtin.

- The input to every `!` macro invocation, that is form 3, is a single non-leaf token tree.
- Syntax extensions are parsed as *part* of the abstract syntax tree.

The last point is the most important, as it has *significant* implications. Because syntax extensions are parsed into the AST, they can **only** appear in positions where they are explicitly supported. Specifically syntax extensions can appear in place of the following:

- Patterns
- Statements
- Expressions
- Items(this includes `impl` items)
- Types

Some things *not* on this list:

- Identifiers
- Match arms
- Struct fields

There is absolutely, definitely *no way* to use syntax extensions in any position *not* on the first list.

Expansion

Expansion is a relatively simple affair. At some point *after* the construction of the AST, but before the compiler begins constructing its semantic understanding of the program, it will expand all syntax extensions.

This involves traversing the AST, locating syntax extension invocations and replacing them with their expansion.

Once the compiler has run a syntax extension, it expects the result to be parsable as one of a limited set of syntax elements, based on context. For example, if you invoke a syntax extension at module scope, the compiler will parse the result into an AST node that represents an item. If you invoke a syntax extension in expression position, the compiler will parse the result into an expression AST node.

In fact, it can turn a syntax extension result into any of the following:

- an expression,
- a pattern,
- a type,
- zero or more items, or
- zero or more statements.
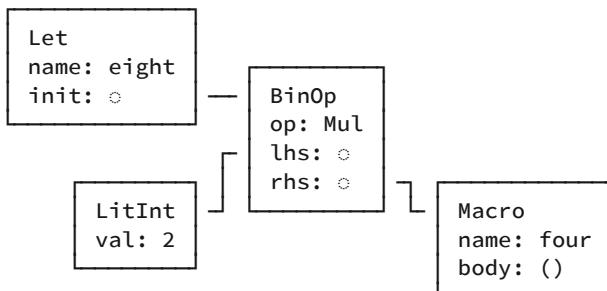
In other words, *where* you can invoke a syntax extension determines what its result will be interpreted as.

The compiler will take this AST node and completely replace the syntax extension's invocation node with the output node. *This is a structural operation*, not a textual one!
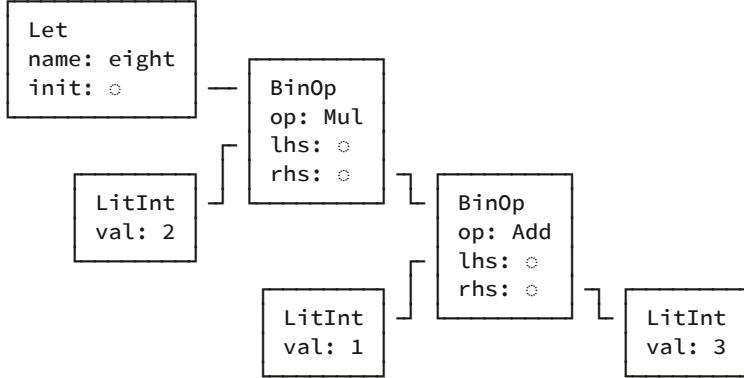
For example, consider the following:

```
let eight = 2 * four!();
```

We can visualize this partial AST as follows:

From context, `four!()` *must* expand to an expression (the initializer can *only* be an expression). Thus, whatever the actual expansion is, it will be interpreted as a complete expression. In this case, we will assume `four!` is defined such that it expands to the expression `1 + 3`. As a result, expanding this invocation will result in the AST changing to:

```
┌──────────────┐
│ Let          │
│ name: eight  │    ┌──────────────┐
│ init: ○      │────│ BinOp        │
└──────────────┘    │ op: Mul      │
                    │ lhs: ○       │
                    │ rhs: ○       │    ┌──────────────┐
    ┌──────────────┐└──────────────┘    │ BinOp        │
    │ LitInt       │                    │ op: Add      │
    │ val: 2       │                    │ lhs: ○       │
    └──────────────┘                    │ rhs: ○       │    ┌──────────────┐
                    ┌──────────────┐    └──────────────┘    │ LitInt       │
                    │ LitInt       │                        │ val: 3       │
                    │ val: 1       │                        └──────────────┘
                    └──────────────┘
```

This can be written out like so:

```
let eight = 2 * (1 + 3);
```

Note that we added parentheses *despite* them not being in the expansion. Remember that the compiler always treats the expansion of a syntax extension as a complete AST node, **not** as a mere sequence of tokens. To put it another way, even if you don't explicitly wrap a complex expression in parentheses, there is no way for the compiler to "misinterpret" the result, or change the order of evaluation.

It is important to understand that syntax extension expansions are treated as AST nodes, as this design has two further implications:

- In addition to there being a limited number of invocation *positions*, syntax extension can *only* expand to the kind of AST node the parser *expects* at that position.
- As a consequence of the above, syntax extension *absolutely cannot* expand to incomplete or syntactically invalid constructs.

There is one further thing to note about expansion: what happens when a syntax extension expands to something that contains *another* syntax extension invocation. For example, consider an alternative definition of `four!`; what happens if it expands to `1 + three!()`?

```
let x = four!();
```

Expands to:

```
let x = 1 + three!();
```

This is resolved by the compiler checking the result of expansions for additional syntax extension invocations, and expanding them. Thus, a second expansion step turns the above into:

```
let x = 1 + 3;
```

The takeaway here is that expansion happens in "passes"; as many as is needed to completely expand all invocations.

Well, not *quite*. In fact, the compiler imposes an upper limit on the number of such recursive passes it is willing to run before giving up. This is known as the syntax extension recursion limit and defaults to 128. If the 128th expansion contains a syntax extension invocation, the compiler will abort with an error indicating that the recursion limit was exceeded.

This limit can be raised using the `#![recursion_limit="…"]` attribute, though it *must* be done crate-wide. Generally, it is recommended to try and keep syntax extension below this limit wherever possible as it may impact compilation times.

Hygiene

Hygiene is an important concept for macros. It describes the ability for a macro to work in its own syntax context, not affecting nor being affected by its surroundings. In other words this means that a syntax extension should be invocable anywhere without interfering with its surrounding context.

In a perfect world all syntax extensions in Rust would be fully hygienic, unfortunately this isn't the case, so care should be taken to avoid writing syntax extensions that aren't fully hygienic. We will go into general hygiene concepts here which will be touched upon in the corresponding hygiene chapters for the different syntax extensions Rust has to offer.

Hygiene mainly affects identifiers and paths emitted by syntax extensions. In short, if an identifier created by a syntax extension cannot be accessed by the environment where the syntax extension has been invoked it is hygienic in regards to that identifier. Likewise, if an identifier used in a syntax extension cannot reference something defined outside of a syntax extension it is considered hygienic.

Note: The terms `create` and `use` refer to the position the identifier is in. That is the `Foo` in `struct Foo {}` or the `foo` in `let foo = …;` are created in the sense that they introduce something new under the name, but the `Foo` in `fn foo(_: Foo) {}` or the `foo` in `foo + 3` are usages in the sense that they are referring to something existing.

This is best shown by example.

Let's assume we have some syntax extension `make_local` that expands to `let local = 0;`, that is it *creates* the identifier `local`. Then given the following snippet:

```
make_local!();
assert_eq!(local, 0);
```

If the `local` in `assert_eq!(local, 0);` resolves to the local defined by the syntax extension, the syntax extension is not hygienic (at least in regards to local names/bindings).

Now let's assume we have some syntax extension `use_local` that expands to `local = 42;`, that is it makes *use* of the identifier `local`. Then given the following snippet:

```
let mut local = 0;
use_local!();
```

If the `local` inside of the syntax extension for the given invocation resolves to the local defined before its invocation, the syntax extension is not hygienic either.

This is a rather short introduction to the general concept of hygiene. It will be explained in more depth in the corresponding `macro_rules! hygiene` and proc-macro `hygiene` chapters, with their specific peculiarities.

Debugging

`rustc` provides a number of tools to debug general syntax extensions, as well as some more specific ones tailored towards declarative and procedural macros respectively.

Sometimes, it is what the extension *expands to* that proves problematic as you do not usually see the expanded code. Fortunately `rustc` offers the ability to look at the expanded code via the unstable `-Zunpretty=expanded` argument. Given the following code:

```
// Shorthand for initializing a `String`.
macro_rules! S {
    ($e:expr) => {String::from($e)};
}

fn main() {
    let world = S!("World");
    println!("Hello, {}!", world);
}
```

compiled with the following command:

```
rustc +nightly -Zunpretty=expanded hello.rs
```

produces the following output (modified for formatting):

```
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2018::*;
#[macro_use]
extern crate std;
// Shorthand for initializing a `String`.
macro_rules! S { ($e : expr) => { String :: from($e) } ; }

fn main() {
    let world = String::from("World");
    {
        ::std::io::_print(
            ::core::fmt::Arguments::new_v1(
                &["Hello, ", "!\n"],
                &match (&world,) {
                    (arg0,) => [
                        ::core::fmt::ArgumentV1::new(arg0, ::core::fmt::Display::fmt)
                    ],
                }
            )
        );
    };
}
```

But not just `rustc` exposes means to aid in debugging syntax extensions. For the aforementioned `-Zunpretty=expanded` option, there exists a nice `cargo` plugin called `cargo-expand` made by `dtolnay` which is basically just a wrapper around it.

You can also use the playground, clicking on its `TOOLS` button in the top right gives you the option to expand syntax extensions as well!

Declarative Macros

This chapter will introduce Rust's declarative macro system: `macro_rules!` .

There are two different introductions in this chapter, a methodical and a practical.

The former will attempt to give you a complete and thorough explanation of *how* the system works, while the latter one will cover more practical examples. As such, the methodical introduction is intended for people who just want the system as a whole explained, while the practical introduction guides one through the implementation of a single macro.

Following up the two introductions it offers some generally very useful patterns and building blocks for creating feature-rich macros.

Other resources about declarative macros include the Macros chapter of the Rust Book which is a more approachable, high-level explanation as well as the reference chapter which goes more into the precise details of things.

Note: This book will usually use the term *mbe*(**M**acro-**B**y-**E**xample), *mbe macro* or `macro_rules!` macro when talking about `macro_rules!` macros.

Macros, A Methodical Introduction

This chapter will introduce Rust's declarative [Macro-By-Example](#) system by explaining the system as a whole. It will do so by first going into the construct's syntax and its key parts and then following it up with more general information that one should at least be aware of.

`macro_rules!`

With all that in mind, we can introduce `macro_rules!` itself. As noted previously, `macro_rules!` is *itself* a syntax extension, meaning it is *technically* not part of the Rust syntax. It uses the following forms:

```
macro_rules! $name {
    $rule0 ;
    $rule1 ;
    // …
    $ruleN ;
}
```

There must be *at least* one rule, and you can omit the semicolon after the last rule. You can use brackets(`[]`), parentheses(`()`) or braces(`{}`).

Each *"rule"* looks like the following:

```
($matcher) => {$expansion}
```

Like before, the types of parentheses used can be any kind, but parentheses around the matcher and braces around the expansion are somewhat conventional. The expansion part of a rule is also called its *transcriber*.

Note that the choice of the parentheses does not matter in regards to how the mbe macro may be invoked. In fact, function-like macros can be invoked with any kind of parentheses as well, but invocations with `{ .. }` and `( ... );` , notice the trailing semicolon, are special in that their expansion will *always* be parsed as an *item*.

If you are wondering, the `macro_rules!` invocation expands to... *nothing*. At least, nothing that appears in the AST; rather, it manipulates compiler-internal structures to register the mbe macro. As such, you can *technically* use `macro_rules!` in any position where an empty expansion is valid.

Matching

When a `macro_rules!` macro is invoked, the `macro_rules!` interpreter goes through the rules one by one, in declaration order. For each rule, it tries to match the contents of the input token tree against that rule's `matcher` . A matcher must match the *entirety* of the input to be considered a match.

If the input matches the matcher, the invocation is replaced by the `expansion` ; otherwise, the next rule is tried. If all rules fail to match, the expansion fails with an error.

The simplest example is of an empty matcher:

```
macro_rules! four {
    () => { 1 + 3 };
}
```

This matches if and only if the input is also empty (*i.e.* `four!()`, `four![]` or `four!{}`).

Note that the specific grouping tokens you use when you invoke the function-like macro *are not* matched, they are in fact not passed to the invocation at all. That is, you can invoke the above macro as `four![]` and it will still match. Only the *contents* of the input token tree are considered.

Matchers can also contain literal token trees, which must be matched exactly. This is done by simply writing the token trees normally. For example, to match the sequence `4 fn ['spang "whammo"] @_@`, you would write:

```
macro_rules! gibberish {
    (4 fn ['spang "whammo"] @_@) => {...};
}
```

You can use any token tree that you can write.

Metavariables

Matchers can also contain captures. These allow input to be matched based on some general grammar category, with the result captured to a metavariable which can then be substituted into the output.

Captures are written as a dollar (`$`) followed by an identifier, a colon (`:`), and finally the kind of capture which is also called the fragment-specifier, which must be one of the following:

- `block` : a block (i.e. a block of statements and/or an expression, surrounded by braces)
- `expr` : an expression
- `ident` : an identifier (this includes keywords)
- `item` : an item, like a function, struct, module, impl, etc.
- `lifetime` : a lifetime (e.g. `'foo`, `'static`, ...)
- `literal` : a literal (e.g. `"Hello World!"`, `3.14`, `'🦀'`, ...)
- `meta` : a meta item; the things that go inside the `#[...]` and `#![...]` attributes
- `pat` : a pattern
- `path` : a path (e.g. `foo`, `::std::mem::replace`, `transmute::<_, int>`, ...)
- `stmt` : a statement
- `tt` : a single token tree
- `ty` : a type
- `vis` : a possible empty visibility qualifier (e.g. `pub`, `pub(in crate)`, ...)

For more in-depth description of the fragment specifiers, check out the Fragment Specifiers chapter.

For example, here is a `macro_rules!` macro which captures its input as an expression under the metavariable `$e` :

```
macro_rules! one_expression {
    ($e:expr) => {...};
}
```

These metavariables leverage the Rust compiler's parser, ensuring that they are always "correct". An `expr` metavariable will *always* capture a complete, valid expression for the version of Rust being compiled.

You can mix literal token trees and metavariables, within limits (explained in Metavariables and Expansion Redux).

To refer to a metavariable you simply write `$name`, as the type of the variable is already specified in the matcher. For example:

```
macro_rules! times_five {
    ($e:expr) => { 5 * $e };
}
```

Much like macro expansion, metavariables are substituted as complete AST nodes. This means that no matter what sequence of tokens is captured by `$e`, it will be interpreted as a single, complete expression.

You can also have multiple metavariables in a single matcher:

```
macro_rules! multiply_add {
    ($a:expr, $b:expr, $c:expr) => { $a * ($b + $c) };
}
```

And use them as often as you like in the expansion:

```
macro_rules! discard {
    ($e:expr) => {};
}
macro_rules! repeat {
    ($e:expr) => { $e; $e; $e; };
}
```

There is also a special metavariable called `$crate` which can be used to refer to the current crate.

Repetitions

Matchers can contain repetitions. These allow a sequence of tokens to be matched. These have the general form `$ ( ... ) sep rep`.

- `$` is a literal dollar token.

- `( ... )` is the paren-grouped matcher being repeated.

- `sep` is an *optional* separator token. It may not be a delimiter or one of the repetition operators. Common examples are `,` and `;`.

- `rep` is the *required* repeat operator. Currently, this can be:

  - `?` : indicating at most one repetition
  - `*` : indicating zero or more repetitions
  - `+` : indicating one or more repetitions

Since `?` represents at most one occurrence, it cannot be used with a separator.

Repetitions can contain any other valid matcher, including literal token trees, metavariables, and other repetitions allowing arbitrary nesting.

Repetitions use the same syntax in the expansion and repeated metavariables can only be accessed inside of repetitions in the expansion.

For example, below is a mbe macro which formats each element as a string. It matches zero or more comma-separated expressions and expands to an expression that constructs a vector.

```rust
macro_rules! vec_strs {
    (
        // Start a repetition:
        $(
            // Each repeat must contain an expression...
            $element:expr
        )
        // ...separated by commas...
        ,
        // ...zero or more times.
        *
    ) => {
        // Enclose the expansion in a block so that we can use
        // multiple statements.
        {
            let mut v = Vec::new();

            // Start a repetition:
            $(
                // Each repeat will contain the following statement, with
                // $element replaced with the corresponding expression.
                v.push(format!("{}", $element));
            )*

            v
        }
    };
}

fn main() {
    let s = vec_strs![1, "a", true, 3.14159f32];
    assert_eq!(s, &["1", "a", "true", "3.14159"]);
}
```

You can repeat multiple metavariables in a single repetition as long as all metavariables repeat equally often. So this invocation of the following macro works:

```rust
macro_rules! repeat_two {
    ($($i:ident)*, $($i2:ident)*) => {
        $( let $i: (); let $i2: (); )*
    }
}

repeat_two!( a b c d e f, u v w x y z );
```

But this does not:

```rust
repeat_two!( a b c d e f, x y z );
```

failing with the following error

```
error: meta-variable `i` repeats 6 times, but `i2` repeats 3 times
 --> src/main.rs:6:10
   |
6  |            $( let $i: (); let $i2: (); )*
   |                ^^^^^^^^^^^^^^^^^^^^^^^^^
```

Metavariable Expressions

RFC: rfcs#1584
Tracking Issue: rust#83527
Feature: `#![feature(macro_metavar_expr)]`

Transcriber can contain what is called metavariable expressions. Metavariable expressions provide transcribers with information about metavariables that are otherwise not easily obtainable. With the exception of the `$$` expression, these have the general form `$ { op(...) }`. Currently all metavariable expressions but `$$` deal with repetitions.

The following expressions are available with `ident` being the name of a bound metavariable and `depth` being an integer literal:

- `${count(ident)}` : The number of times `$ident` repeats in the inner-most repetition in total. This is equivalent to `${count(ident, 0)}` .
- `${count(ident, depth)}` : The number of times `$ident` repeats in the repetition at `depth` .
- `${index()}` : The current repetition index of the inner-most repetition. This is equivalent to `${index(0)}` .
- `${index(depth)}` : The current index of the repetition at `depth` , counting outwards.
- `${len()}` : The number of times the inner-most repetition will repeat for. This is equivalent to `${len(0)}` .
- `${len(depth)}` : The number of times the repetition at `depth` will repeat for, counting outwards.
- `${ignore(ident)}` : Binds `$ident` for repetition, while expanding to nothing.
- `$$` : Expands to a single `$` , effectively escaping the `$` token so it won't be transcribed.

For the complete grammar definition you may want to consult the Macros By Example chapter of the Rust reference.

Macros, A Practical Introduction

This chapter will introduce Rust's declarative Macro-By-Example system using a relatively simple, practical example. It does *not* attempt to explain all of the intricacies of the system; its goal is to get you comfortable with how and why macros are written.

There is also the Macros chapter of the Rust Book which is another high-level explanation, and the methodical introduction chapter of this book, which explains the macro system in detail.

A Little Context

Note: don't panic! What follows is the only math that will be talked about. You can quite safely skip this section if you just want to get to the meat of the article.

If you aren't familiar, a recurrence relation is a sequence where each value is defined in terms of one or more *previous* values, with one or more initial values to get the whole thing started. For example, the Fibonacci sequence can be defined by the relation:

$$F_n = 0, 1, \ldots, F_{n-2} + F_{n-1}$$

Thus, the first two numbers in the sequence are 0 and 1, with the third being $F_0 + F_1 = 0 + 1 = 1$, the fourth $F_1 + F_2 = 1 + 1 = 2$, and so on forever.

Now, *because* such a sequence can go on forever, that makes defining a `fibonacci` function a little tricky, since you obviously don't want to try returning a complete vector. What you *want* is to return something which will lazily compute elements of the sequence as needed.

In Rust, that means producing an `Iterator`. This is not especially *hard*, but there is a fair amount of boilerplate involved: you need to define a custom type, work out what state needs to be stored in it, then implement the `Iterator` trait for it.

However, recurrence relations are simple enough that almost all of these details can be abstracted out with a little `macro_rules!` macro-based code generation.

So, with all that having been said, let's get started.

Construction

Usually, when working on a new `macro_rules!` macro, the first thing I do is decide what the invocation should look like. In this specific case, my first attempt looked like this:

```
let fib = recurrence![a[n] = 0, 1, ..., a[n-2] + a[n-1]];

for e in fib.take(10) { println!("{}", e) }
```

From that, we can take a stab at how the `macro_rules!` macro should be defined, even if we aren't sure of the actual expansion. This is useful because if you can't figure out how to parse the input syntax, then *maybe* you need to change it.

```
macro_rules! recurrence {
    ( a[n] = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
}
```

Assuming you aren't familiar with the syntax, allow me to elucidate. This is defining a syntax extension, using the `macro_rules!` system, called `recurrence!`. This `macro_rules!` macro has a single parsing rule. That rule says the input to the invocation must match:

- the literal token sequence `a [ n ] =`,
- a repeating (the `$( ... )`) sequence, using `,` as a separator, and one or more (`+`) repeats of:
    - a valid *expression* captured into the metavariable `inits` (`$inits:expr`)
- the literal token sequence `, ... ,`,
- a valid *expression* captured into the metavariable `recur` (`$recur:expr`).

Finally, the rule says that *if* the input matches this rule, then the invocation should be replaced by the token sequence `/* ... */`.

It's worth noting that `inits`, as implied by the name, actually contains *all* the expressions that match in this position, not just the first or last. What's more, it captures them *as a sequence* as opposed to, say, irreversibly pasting them all together. Also note that you can do "zero or more" with a repetition by using `*` instead of `+` and even optional, "zero or one" with `?`.

As an exercise, let's take the proposed input and feed it through the rule, to see how it is processed. The "Position" column will show which part of the syntax pattern needs to be matched against next, denoted by a "⌂". Note that in some cases, there might be more than one possible "next" element to match against. "Input" will contain all of the tokens that have *not* been consumed yet. `inits` and `recur` will contain the contents of those bindings.

Position	Input	inits	recur
a[n] = $($inits:expr),+ , ... , $recur:expr	a[n] = 0, 1, ..., a[n-2] + a[n-1]		
a[n] = $($inits:expr),+ , ... , $recur:expr	[n] = 0, 1, ..., a[n-2] + a[n-1]		
a[n] = $($inits:expr),+ , ... , $recur:expr	n] = 0, 1, ..., a[n-2] + a[n-1]		
a[n] = $($inits:expr),+ , ... , $recur:expr	] = 0, 1, ..., a[n-2] + a[n-1]		
a[n] = $($inits:expr),+ , ... , $recur:expr	= 0, 1, ..., a[n-2] + a[n-1]		
a[n] = $($inits:expr),+ , ... , $recur:expr	0, 1, ..., a[n-2] + a[n-1]		
a[n] = $($inits:expr),+ , ... , $recur:expr	0, 1, ..., a[n-2] + a[n-1]		
a[n] = $($inits:expr),+ , ... , $recur:expr	, 1, ..., a[n-2] + a[n-1]	0	

Note: there are two ⌂ here, because the next input token might match *either* the comma separator *between* elements in the repetition, *or* the comma *after* the repetition. The macro system will keep track of both possibilities, until it is able to decide which one to follow.

a[n] = $($inits:expr),+ , ... , $recur:expr	1, ..., a[n-2] + a[n-1]	0	
a[n] = $($inits:expr),+ , ... , $recur:expr	, ..., a[n-2] + a[n-1]	0, 1	

Note: the third, crossed-out marker indicates that the macro system has, as a consequence of the last token consumed, eliminated one of the previous possible branches.

a[n] = $($inits:expr),+ , ... , $recur:expr	..., a[n-2] + a[n-1]	0, 1	
a[n] = $($inits:expr),+ , ... , $recur:expr	, a[n-2] + a[n-1]	0, 1	
a[n] = $($inits:expr),+ , ... , $recur:expr	a[n-2] + a[n-1]	0, 1	
a[n] = $($inits:expr),+ , ... , $recur:expr		0, 1	a[n-2] + a[n-1]

Note: this particular step should make it clear that a binding like $recur:expr will consume an *entire expression*, using the compiler's knowledge of what constitutes a valid expression. As will be noted later, you can do this for other language constructs, too.

The key take-away from this is that the macro system will *try* to incrementally match the tokens provided as input to the macro against the provided rules. We'll come back to the "try" part.

Now, let's begin writing the final, fully expanded form. For this expansion, I was looking for something like:

```
let fib = {
    struct Recurrence {
        mem: [u64; 2],
        pos: usize,
    }
```

This will be the actual iterator type. `mem` will be the memo buffer to hold the last few values so the recurrence can be computed. `pos` is to keep track of the value of `n`.

Aside: I've chosen `u64` as a "sufficiently large" type for the elements of this sequence. Don't worry about how this will work out for *other* sequences; we'll come to it.

```
    impl Iterator for Recurrence {
        type Item = u64;

        fn next(&mut self) -> Option<Self::Item> {
            if self.pos < 2 {
                let next_val = self.mem[self.pos];
                self.pos += 1;
                Some(next_val)
```

We need a branch to yield the initial values of the sequence; nothing tricky.

```
            } else {
                let a = /* something */;
                let n = self.pos;
                let next_val = a[n-2] + a[n-1];

                self.mem.TODO_shuffle_down_and_append(next_val);

                self.pos += 1;
                Some(next_val)
            }
        }
    }
```

This is a bit harder; we'll come back and look at *how* exactly to define `a`. Also, `TODO_shuffle_down_and_append` is another placeholder; I want something that places `next_val` on the end of the array, shuffling the rest down by one space, dropping the 0th element.

```
    Recurrence { mem: [0, 1], pos: 0 }
};

for e in fib.take(10) { println!("{}", e) }
```

Lastly, return an instance of our new structure, which can then be iterated over. To summarize, the complete expansion is:

```
let fib = {
    struct Recurrence {
        mem: [u64; 2],
        pos: usize,
    }

    impl Iterator for Recurrence {
        type Item = u64;

        fn next(&mut self) -> Option<u64> {
            if self.pos < 2 {
                let next_val = self.mem[self.pos];
                self.pos += 1;
                Some(next_val)
            } else {
                let a = /* something */;
                let n = self.pos;
                let next_val = (a[n-2] + a[n-1]);

                self.mem.TODO_shuffle_down_and_append(next_val.clone());

                self.pos += 1;
                Some(next_val)
            }
        }
    }

    Recurrence { mem: [0, 1], pos: 0 }
};

for e in fib.take(10) { println!("{}", e) }
```

Aside: Yes, this *does* mean we're defining a different `Recurrence` struct and its implementation for each invocation. Most of this will optimise away in the final binary.

It's also useful to check your expansion as you're writing it. If you see anything in the expansion that needs to vary with the invocation, but *isn't* in the actual accepted syntax of our macro, you should work out where to introduce it. In this case, we've added `u64`, but that's not necessarily what the user wants, nor is it in the macro syntax. So let's fix that.

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
}

/*
let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-2] + a[n-1]];

for e in fib.take(10) { println!("{}", e) }
*/
```

Here, I've added a new metavariable: `sty` which should be a type.

Aside: if you're wondering, the bit after the colon in a metavariable can be one of several kinds of syntax matchers. The most common ones are `item`, `expr`, and `ty`. A complete explanation can be found in Macros, A Methodical Introduction; `macro_rules!` (Matchers).

There's one other thing to be aware of: in the interests of future-proofing the language, the

compiler restricts what tokens you're allowed to put *after* a matcher, depending on what kind it is. Typically, this comes up when trying to match expressions or statements; those can *only* be followed by one of `=>` , `,` , and `;` .

A complete list can be found in Macros, A Methodical Introduction; Minutiae; Metavariables and Expansion Redux.

Indexing and Shuffling

I will skim a bit over this part, since it's effectively tangential to the macro-related stuff. We want to make it so that the user can access previous values in the sequence by indexing `a` ; we want it to act as a sliding window keeping the last few (in this case, 2) elements of the sequence.

We can do this pretty easily with a wrapper type:

```rust
struct IndexOffset<'a> {
    slice: &'a [u64; 2],
    offset: usize,
}

impl<'a> Index<usize> for IndexOffset<'a> {
    type Output = u64;

    fn index<'b>(&'b self, index: usize) -> &'b u64 {
        use std::num::Wrapping;

        let index = Wrapping(index);
        let offset = Wrapping(self.offset);
        let window = Wrapping(2);

        let real_index = index - offset + window;
        &self.slice[real_index.0]
    }
}
```

Aside: since lifetimes come up *a lot* with people new to Rust, a quick explanation: `'a` and `'b` are lifetime parameters that are used to track where a reference (*i.e.* a borrowed pointer to some data) is valid. In this case, `IndexOffset` borrows a reference to our iterator's data, so it needs to keep track of how long it's allowed to hold that reference for, using `'a` .

`'b` is used because the `Index::index` function (which is how subscript syntax is actually implemented) is *also* parameterized on a lifetime, on account of returning a borrowed reference. `'a` and `'b` are not necessarily the same thing in all cases. The borrow checker will make sure that even though we don't explicitly relate `'a` and `'b` to one another, we don't accidentally violate memory safety.

This changes the definition of `a` to:

```rust
let a = IndexOffset { slice: &self.mem, offset: n };
```

The only remaining question is what to do about `TODO_shuffle_down_and_append` . I wasn't able to find a method in the standard library with exactly the semantics I wanted, but it isn't hard to do by

hand.

```
{
    use std::mem::swap;

    let mut swap_tmp = next_val;
    for i in (0..2).rev() {
        swap(&mut swap_tmp, &mut self.mem[i]);
    }
}
```

This swaps the new value into the end of the array, swapping the other elements down one space.

Aside: doing it this way means that this code will work for non-copyable types, as well.

The working code thus far now looks like this:

```rust
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => { /* ... */ };
}

fn main() {
    /*
    let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-2] + a[n-1]];

    for e in fib.take(10) { println!("{}", e) }
    */
    let fib = {
        use std::ops::Index;

        struct Recurrence {
            mem: [u64; 2],
            pos: usize,
        }

        struct IndexOffset<'a> {
            slice: &'a [u64; 2],
            offset: usize,
        }

        impl<'a> Index<usize> for IndexOffset<'a> {
            type Output = u64;

            #[inline(always)]
            fn index<'b>(&'b self, index: usize) -> &'b u64 {
                use std::num::Wrapping;

                let index = Wrapping(index);
                let offset = Wrapping(self.offset);
                let window = Wrapping(2);

                let real_index = index - offset + window;
                &self.slice[real_index.0]
            }
        }

        impl Iterator for Recurrence {
            type Item = u64;

            #[inline]
            fn next(&mut self) -> Option<u64> {
                if self.pos < 2 {
                    let next_val = self.mem[self.pos];
                    self.pos += 1;
                    Some(next_val)
                } else {
                    let next_val = {
                        let n = self.pos;
                        let a = IndexOffset { slice: &self.mem, offset: n };
                        a[n-2] + a[n-1]
                    };

                    {
                        use std::mem::swap;

                        let mut swap_tmp = next_val;
                        for i in [1,0] {
                            swap(&mut swap_tmp, &mut self.mem[i]);
                        }
                    }

                    self.pos += 1;
                    Some(next_val)
```

```
                }
            }
        }

        Recurrence { mem: [0, 1], pos: 0 }
    };

    for e in fib.take(10) { println!("{}", e) }
}
```

Note that I've changed the order of the declarations of `n` and `a`, as well as wrapped them(along with the recurrence expression) in a block. The reason for the first should be obvious(`n` needs to be defined first so I can use it for `a`). The reason for the second is that the borrowed reference `&self.mem` will prevent the swaps later on from happening (you cannot mutate something that is aliased elsewhere). The block ensures that the `&self.mem` borrow expires before then.

Incidentally, the only reason the code that does the `mem` swaps is in a block is to narrow the scope in which `std::mem::swap` is available, for the sake of being tidy.

If we take this code and run it, we get:

```
0
1
1
2
3
5
8
13
21
34
```

Success! Now, let's copy & paste this into the macro expansion, and replace the expanded code with an invocation. This gives us:

```rust
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ , ... , $recur:expr ) => {
        {
            /*
                What follows here is *literally* the code from before,
                cut and pasted into a new position. No other changes
                have been made.
            */

            use std::ops::Index;

            struct Recurrence {
                mem: [u64; 2],
                pos: usize,
            }

            struct IndexOffset<'a> {
                slice: &'a [u64; 2],
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = u64;

                fn index<'b>(&'b self, index: usize) -> &'b u64 {
                    use std::num::Wrapping;

                    let index = Wrapping(index);
                    let offset = Wrapping(self.offset);
                    let window = Wrapping(2);

                    let real_index = index - offset + window;
                    &self.slice[real_index.0]
                }
            }

            impl Iterator for Recurrence {
                type Item = u64;

                fn next(&mut self) -> Option<u64> {
                    if self.pos < 2 {
                        let next_val = self.mem[self.pos];
                        self.pos += 1;
                        Some(next_val)
                    } else {
                        let next_val = {
                            let n = self.pos;
                            let a = IndexOffset { slice: &self.mem, offset: n };
                            (a[n-2] + a[n-1])
                        };

                        {
                            use std::mem::swap;

                            let mut swap_tmp = next_val;
                            for i in (0..2).rev() {
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
                        }

                        self.pos += 1;
                        Some(next_val)
                    }
                }
            }
```

```
                    Recurrence { mem: [0, 1], pos: 0 }
            }
        };
    }

    fn main() {
        let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-2] + a[n-1]];

        for e in fib.take(10) { println!("{}", e) }
    }
```

Obviously, we aren't *using* the metavariables yet, but we can change that fairly easily. However, if we try to compile this, `rustc` aborts, telling us:

```
error: local ambiguity: multiple parsing options: built-in NTs expr ('inits') or 1
other option.
  --> src/main.rs:75:45
   |
75 |     let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-2] + a[n-1]];
   |
```

Here, we've run into a limitation of the `macro_rules` system. The problem is that second comma. When it sees it during expansion, `macro_rules` can't decide if it's supposed to parse *another* expression for `inits`, or `...`. Sadly, it isn't quite clever enough to realise that `...` isn't a valid expression, so it gives up. Theoretically, this *should* work as desired, but currently doesn't.

Aside: I *did* fib a little about how our rule would be interpreted by the macro system. In general, it *should* work as described, but doesn't in this case. The `macro_rules` machinery, as it stands, has its foibles, and its worthwhile remembering that on occasion, you'll need to contort a little to get it to work.

In this *particular* case, there are two issues. First, the macro system doesn't know what does and does not constitute the various grammar elements (*e.g.* an expression); that's the parser's job. As such, it doesn't know that `...` isn't an expression. Secondly, it has no way of trying to capture a compound grammar element (like an expression) without 100% committing to that capture.

In other words, it can ask the parser to try and parse some input as an expression, but the parser will respond to any problems by aborting. The only way the macro system can currently deal with this is to just try to forbid situations where this could be a problem.

On the bright side, this is a state of affairs that exactly *no one* is enthusiastic about. The `macro` keyword has already been reserved for a more rigorously-defined future macro system. Until then, needs must.

Thankfully, the fix is relatively simple: we remove the comma from the syntax. To keep things balanced, we'll remove *both* commas around `...`:

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
//                                       ^~~ changed
        /* ... */
    };
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-2] + a[n-1]];
//                                        ^~~ changed

    for e in fib.take(10) { println!("{}", e) }
}
```

Success! ... or so we thought. Turns out this is being rejected by the compiler nowadays, while it was fine back when this was written. The reason for this is that the compiler now recognizes the `...` as a token, and as we know we may only use `=>`, `,` or `;` after an expression fragment. So unfortunately we are now out of luck as our dreamed up syntax will not work out this way, so let us just choose one that looks the most befitting that we are allowed to use instead, I'd say replacing `,` with `;` works.

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ ; ... ; $recur:expr ) => {
//                                     ^~~~~~^ changed
        /* ... */
    };
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1; ...; a[n-2] + a[n-1]];
//                                        ^~~~~^ changed

    for e in fib.take(10) { println!("{}", e) }
}
```

Success! But for real this time.

Substitution

Substituting something you've captured in a macro is quite simple; you can insert the contents of a metavariable `$sty:ty` by using `$sty`. So, let's go through and fix the `u64` s:

```
macro_rules! recurrence {
    ( a[n]: $sty:ty = $($inits:expr),+ ; ... ; $recur:expr ) => {
        {
            use std::ops::Index;

            struct Recurrence {
                mem: [$sty; 2],
//                      ^~~~ changed
                pos: usize,
            }

            struct IndexOffset<'a> {
                slice: &'a [$sty; 2],
//                          ^~~~ changed
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = $sty;
//                            ^~~~ changed

                #[inline(always)]
                fn index<'b>(&'b self, index: usize) -> &'b $sty {
//                                                          ^~~~ changed
                    use std::num::Wrapping;

                    let index = Wrapping(index);
                    let offset = Wrapping(self.offset);
                    let window = Wrapping(2);

                    let real_index = index - offset + window;
                    &self.slice[real_index.0]
                }
            }

            impl Iterator for Recurrence {
                type Item = $sty;
//                          ^~~~ changed

                #[inline]
                fn next(&mut self) -> Option<$sty> {
//                                           ^~~~ changed
                    /* ... */
                }
            }

            Recurrence { mem: [0, 1], pos: 0 }
        }
    };
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1; ...; a[n-2] + a[n-1]];

    for e in fib.take(10) { println!("{}", e) }
}
```

Let's tackle a harder one: how to turn `inits` into both the array literal `[0, 1]` *and* the array type, `[$sty; 2]`. The first one we can do like so:

```
            Recurrence { mem: [$($inits),+], pos: 0 }
//                            ^~~~~~~~~~~ changed
```

This effectively does the opposite of the capture: repeat `inits` one or more times, separating each with a comma. This expands to the expected sequence of tokens: `0, 1`.

Somehow turning `inits` into a literal `2` is a little trickier. It turns out that there's no direct way to do this, but we *can* do it by using a second `macro_rules!` macro. Let's take this one step at a time.

```
macro_rules! count_exprs {
    /* ??? */
}
```

The obvious case is: given zero expressions, you would expect `count_exprs` to expand to a literal `0`.

```
macro_rules! count_exprs {
    () => (0);
//   ^~~~~~~~~~ added
}
```

Aside: You may have noticed I used parentheses here instead of curly braces for the expansion. `macro_rules` really doesn't care *what* you use, so long as it's one of the "matcher" pairs: `( )`, `{ }` or `[ ]`. In fact, you can switch out the matchers on the macro itself(*i.e.* the matchers right after the macro name), the matchers around the syntax rule, and the matchers around the corresponding expansion.

You can also switch out the matchers used when you *invoke* a macro, but in a more limited fashion: a macro invoked as `{ ... }` or `( ... );` will *always* be parsed as an *item* (*i.e.* like a `struct` or `fn` declaration). This is important when using macros in a function body; it helps disambiguate between "parse like an expression" and "parse like a statement".

What if you have *one* expression? That should be a literal `1`.

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
//   ^~~~~~~~~~~~~~~~~ added
}
```

Two?

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (2);
//   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~ added
}
```

We can "simplify" this a little by re-expressing the case of two expressions recursively.

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (1 + count_exprs!($e1));
//                           ^~~~~~~~~~~~~~~~~~~~~ changed
}
```

This is fine since Rust can fold `1 + 1` into a constant value. What if we have three expressions?

```
macro_rules! count_exprs {
    () => (0);
    ($e:expr) => (1);
    ($e0:expr, $e1:expr) => (1 + count_exprs!($e1));
    ($e0:expr, $e1:expr, $e2:expr) => (1 + count_exprs!($e1, $e2));
//  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ added
}
```

Aside: You might be wondering if we could reverse the order of these rules. In this particular case, *yes*, but the macro system can sometimes be picky about what it is and is not willing to recover from. If you ever find yourself with a multi-rule macro that you *swear* should work, but gives you errors about unexpected tokens, try changing the order of the rules.

Hopefully, you can see the pattern here. We can always reduce the list of expressions by matching one expression, followed by zero or more expressions, expanding that into 1 + a count.

```
macro_rules! count_exprs {
    () => (0);
    ($head:expr) => (1);
    ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
//  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ changed
}
```

JFTE: this is not the *only*, or even the *best* way of counting things. You may wish to peruse the Counting section later for a more efficient way.

With this, we can now modify `recurrence` to determine the necessary size of `mem` .

```rust
            // added:
            macro_rules! count_exprs {
                () => (0);
                ($head:expr) => (1);
                ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
            }

            macro_rules! recurrence {
                ( a[n]: $sty:ty = $($inits:expr),+ ; ... ; $recur:expr ) => {
                    {
                        use std::ops::Index;

                        const MEM_SIZE: usize = count_exprs!($($inits),+);
            //          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ added

                        struct Recurrence {
                            mem: [$sty; MEM_SIZE],
            //                       ^~~~~~~~ changed
                            pos: usize,
                        }

                        struct IndexOffset<'a> {
                            slice: &'a [$sty; MEM_SIZE],
            //                              ^~~~~~~~ changed
                            offset: usize,
                        }

                        impl<'a> Index<usize> for IndexOffset<'a> {
                            type Output = $sty;

                            #[inline(always)]
                            fn index<'b>(&'b self, index: usize) -> &'b $sty {
                                use std::num::Wrapping;

                                let index = Wrapping(index);
                                let offset = Wrapping(self.offset);
                                let window = Wrapping(MEM_SIZE);
            //                                        ^~~~~~~~ changed

                                let real_index = index - offset + window;
                                &self.slice[real_index.0]
                            }
                        }

                        impl Iterator for Recurrence {
                            type Item = $sty;

                            #[inline]
                            fn next(&mut self) -> Option<$sty> {
                                if self.pos < MEM_SIZE {
            //                                 ^~~~~~~~ changed
                                    let next_val = self.mem[self.pos];
                                    self.pos += 1;
                                    Some(next_val)
                                } else {
                                    let next_val = {
                                        let n = self.pos;
                                        let a = IndexOffset { slice: &self.mem, offset: n };
                                        (a[n-2] + a[n-1])
                                    };

                                    {
                                        use std::mem::swap;

                                        let mut swap_tmp = next_val;
                                        for i in (0..MEM_SIZE).rev() {
```

```
//                                        ^~~~~~~~ changed
                        swap(&mut swap_tmp, &mut self.mem[i]);
                    }
                }

                self.pos += 1;
                Some(next_val)
            }
        }
    }

    Recurrence { mem: [$($inits),+], pos: 0 }
    }
};
}
/* ... */
```

With that done, we can now substitute the last thing: the `recur` expression.

```
/* ... */
                #[inline]
                fn next(&mut self) -> Option<u64> {
                    if self.pos < MEM_SIZE {
                        let next_val = self.mem[self.pos];
                        self.pos += 1;
                        Some(next_val)
                    } else {
                        let next_val = {
                            let n = self.pos;
                            let a = IndexOffset { slice: &self.mem, offset: n };
                            $recur
//                          ^~~~~~ changed
                        };
                        {
                            use std::mem::swap;
                            let mut swap_tmp = next_val;
                            for i in (0..MEM_SIZE).rev() {
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
                        }
                        self.pos += 1;
                        Some(next_val)
                    }
                }
/* ... */
```

And, when we compile our finished `macro_rules!` macro...

```
error[E0425]: cannot find value `a` in this scope
  --> src/main.rs:68:50
   |
68 |     let fib = recurrence![a[n]: u64 = 1, 1; ...; a[n-2] + a[n-1]];
   |                                                  ^ not found in this scope

error[E0425]: cannot find value `n` in this scope
  --> src/main.rs:68:52
   |
68 |     let fib = recurrence![a[n]: u64 = 1, 1; ...; a[n-2] + a[n-1]];
   |                                                    ^ not found in this scope

error[E0425]: cannot find value `a` in this scope
  --> src/main.rs:68:59
   |
68 |     let fib = recurrence![a[n]: u64 = 1, 1; ...; a[n-2] + a[n-1]];
   |                                                           ^ not found in this
scope

error[E0425]: cannot find value `n` in this scope
  --> src/main.rs:68:61
   |
68 |     let fib = recurrence![a[n]: u64 = 1, 1; ...; a[n-2] + a[n-1]];
   |                                                             ^ not found in this
scope
```

… wait, what? That can't be right… let's check what the macro is expanding to.

```
$ rustc +nightly -Zunpretty=expanded recurrence.rs
```

The `-Zunpretty=expanded` argument tells `rustc` to perform macro expansion, then turn the
resulting AST back into source code. The output (after cleaning up some formatting) is shown below;
in particular, note the place in the code where `$recur` was substituted:

```rust
#![feature(no_std)]
#![no_std]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;
fn main() {
    let fib = {
        use std::ops::Index;
        const MEM_SIZE: usize = 1 + 1;
        struct Recurrence {
            mem: [u64; MEM_SIZE],
            pos: usize,
        }
        struct IndexOffset<'a> {
            slice: &'a [u64; MEM_SIZE],
            offset: usize,
        }
        impl <'a> Index<usize> for IndexOffset<'a> {
            type Output = u64;
            #[inline(always)]
            fn index<'b>(&'b self, index: usize) -> &'b u64 {
                use std::num::Wrapping;
                let index = Wrapping(index);
                let offset = Wrapping(self.offset);
                let window = Wrapping(MEM_SIZE);
                let real_index = index - offset + window;
                &self.slice[real_index.0]
            }
        }
        impl Iterator for Recurrence {
            type Item = u64;
            #[inline]
            fn next(&mut self) -> Option<u64> {
                if self.pos < MEM_SIZE {
                    let next_val = self.mem[self.pos];
                    self.pos += 1;
                    Some(next_val)
                } else {
                    let next_val = {
                        let n = self.pos;
                        let a = IndexOffset{slice: &self.mem, offset: n,};
                        a[n - 1] + a[n - 2]
                    };
                    {
                        use std::mem::swap;
                        let mut swap_tmp = next_val;
                        {
                            let result =
                                match
::std::iter::IntoIterator::into_iter((0..MEM_SIZE).rev()) {
                                    mut iter => loop {
                                        match ::std::iter::Iterator::next(&mut iter) {
                                            ::std::option::Option::Some(i) => {
                                                swap(&mut swap_tmp, &mut self.mem[i]);
                                            }
                                            ::std::option::Option::None => break,
                                        }
                                    },
                                };
                            result
                        }
                    }
                    self.pos += 1;
                    Some(next_val)
                }
```

```
            }
        }
        Recurrence{mem: [0, 1], pos: 0,}
    };
    {
        let result =
            match ::std::iter::IntoIterator::into_iter(fib.take(10)) {
                mut iter => loop {
                    match ::std::iter::Iterator::next(&mut iter) {
                        ::std::option::Option::Some(e) => {
                            ::std::io::_print(::std::fmt::Arguments::new_v1(
                                {
                                    static __STATIC_FMTSTR: &'static [&'static str] =
&["", "\n"];

                                    __STATIC_FMTSTR
                                },
                                &match (&e,) {
                                    (__arg0,) => [::std::fmt::ArgumentV1::new(__arg0,
 ::std::fmt::Display::fmt)],
                                }
                            ))
                        }
                        ::std::option::Option::None => break,
                    }
                },
            };
        result
    }
}
```

But that looks fine! If we add a few missing `#![feature(...)]` attributes and feed it to a nightly build of `rustc`, it even compiles! … *what?!*

Aside: You can't compile the above with a non-nightly build of `rustc`. This is because the expansion of the `println!` macro depends on internal compiler details which are *not* publicly stabilized.

Being Hygienic

The issue here is that identifiers in Rust syntax extensions are *hygienic*. That is, identifiers from two different contexts *cannot* collide. To show the difference, let's take a simpler example.

```
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42;
            $e
        }
    }
}

let four = using_a!(a / 10);
```

This macro simply takes an expression, then wraps it in a block with a variable `a` defined. We then use this as a round-about way of computing `4`. There are actually *two* syntax contexts involved in this example, but they're invisible. So, to help with this, let's give each context a different colour. Let's start with the unexpanded code, where there is only a single context:

```
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42;
            $e
        }
    }
}
```

```
let four = using_a!(a / 10);
```

Now, let's expand the invocation.

```
let four = {
    let a = 42;
    a / 10
};
```

As you can see, the `a` that's defined by the macro invocation is in a different context to the `a` we provided in our invocation. As such, the compiler treats them as completely different identifiers, *even though they have the same lexical appearance*.

This is something to be *really* careful of when working on `macro_rules!` macros, syntax extensions in general even: they can produce ASTs which will not compile, but which *will* compile if written out by hand, or dumped using `-Zunpretty=expanded`.

The solution to this is to capture the identifier *with the appropriate syntax context*. To do that, we need to again adjust our macro syntax. To continue with our simpler example:

```
macro_rules! using_a {
    ($a:ident, $e:expr) => {
        {
            let $a = 42;
            $e
        }
    }
}
```

```
let four = using_a!(a, a / 10);
```

This now expands to:

```
let four = {
    let a = 42;
    a / 10
};
```

Now, the contexts match, and the code will compile. We can make this adjustment to our `recurrence!` macro by explicitly capturing `a` and `n`. After making the necessary changes, we have:

```rust
macro_rules! count_exprs {
    () => (0);
    ($head:expr) => (1);
    ($head:expr, $($tail:expr),*) => (1 + count_exprs!($($tail),*));
}

macro_rules! recurrence {
    ( $seq:ident [ $ind:ident ]: $sty:ty = $($inits:expr),+ ; ... ; $recur:expr ) => {
//        ^~~~~~~~~~    ^~~~~~~~~~ changed
        {
            use std::ops::Index;

            const MEM_SIZE: usize = count_exprs!($($inits),+);

            struct Recurrence {
                mem: [$sty; MEM_SIZE],
                pos: usize,
            }

            struct IndexOffset<'a> {
                slice: &'a [$sty; MEM_SIZE],
                offset: usize,
            }

            impl<'a> Index<usize> for IndexOffset<'a> {
                type Output = $sty;

                #[inline(always)]
                fn index<'b>(&'b self, index: usize) -> &'b $sty {
                    use std::num::Wrapping;

                    let index = Wrapping(index);
                    let offset = Wrapping(self.offset);
                    let window = Wrapping(MEM_SIZE);

                    let real_index = index - offset + window;
                    &self.slice[real_index.0]
                }
            }

            impl Iterator for Recurrence {
                type Item = $sty;

                #[inline]
                fn next(&mut self) -> Option<$sty> {
                    if self.pos < MEM_SIZE {
                        let next_val = self.mem[self.pos];
                        self.pos += 1;
                        Some(next_val)
                    } else {
                        let next_val = {
                            let $ind = self.pos;
//                              ^~~~ changed
                            let $seq = IndexOffset { slice: &self.mem, offset: $ind };
//                              ^~~~ changed                                    ^~~~
// changed

                            $recur
                        };

                        {
                            use std::mem::swap;

                            let mut swap_tmp = next_val;
                            for i in (0..MEM_SIZE).rev() {
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
```

```
                    }

                    self.pos += 1;
                    Some(next_val)
                }
            }
        }

        Recurrence { mem: [$($inits),+], pos: 0 }
    }
};
}

fn main() {
    let fib = recurrence![a[n]: u64 = 0, 1; ...; a[n-2] + a[n-1]];

    for e in fib.take(10) { println!("{}", e) }
}
```

And it compiles! Now, let's try with a different sequence.

```
for e in recurrence!(f[i]: f64 = 1.0; ...; f[i-1] * i as f64).take(10) {
    println!("{}", e)
}
```

Which gives us:

```
1
1
2
6
24
120
720
5040
40320
362880
```

Success!

Minutiae

This section goes through some of the finer details of the `macro_rules!` system. At a minimum, you should try to be at least *aware* of these details and issues.

Fragment Specifiers

As mentioned in the `methodical introduction` chapter, Rust, as of 1.60, has 14 fragment specifiers. This section will go a bit more into detail for some of them and shows a few example inputs of what each matcher matches.

Note: Capturing with anything but the `ident`, `lifetime` and `tt` fragments will render the captured AST opaque, making it impossible to further match it with other fragment specifiers in future macro invocations.

- `block`
- `expr`
- `ident`
- `item`
- `lifetime`
- `literal`
- `meta`
- `pat`
- `pat_param`
- `path`
- `stmt`
- `tt`
- `ty`
- `vis`

`block`

The `block` fragment solely matches a block expression, which consists of an opening `{` brace, followed by any number of statements and finally followed by a closing `}` brace.

```
macro_rules! blocks {
    ($($block:block)*) => ();
}

blocks! {
    {}
    {
        let zig;
    }
    { 2 }
}
```

`expr`

The `expr` fragment matches any kind of expression (Rust has a lot of them, given it *is* an expression-oriented language).

```
macro_rules! expressions {
    ($($expr:expr)*) => ();
}

expressions! {
    "literal"
    funcall()
    future.await
    break 'foo bar
}
```

ident

The `ident` fragment matches an identifier or *keyword*.

```
macro_rules! idents {
    ($($ident:ident)*) => ();
}

idents! {
    // _ <- This is not an ident, it is a pattern
    foo
    async
    O_____O
    _____O_____
}
```

item

The `item` fragment simply matches any of Rust's item *definitions*, not identifiers that refer to items.
This includes visibility modifiers.

```
macro_rules! items {
    ($($item:item)*) => ();
}

items! {
    struct Foo;
    enum Bar {
        Baz
    }
    impl Foo {}
    pub use crate::foo;
    /*...*/
}
```

lifetime

The `lifetime` fragment matches a lifetime or label. It's quite similar to `ident` but with a prepended
`'`.

```
macro_rules! lifetimes {
    ($($lifetime:lifetime)*) => ();
}

lifetimes! {
    'static
    'shiv
    '_
}
```

literal

The `literal` fragment matches any literal expression.

```
macro_rules! literals {
    ($($literal:literal)*) => ();
}

literals! {
    -1
    "hello world"
    2.3
    b'b'
    true
}
```

meta

The `meta` fragment matches the contents of an attribute. That is, it will match a simple path, one without generic arguments followed by a delimited token tree or an `=` followed by a literal expression.

Note: You will usually see this fragment being used in a matcher like `#[$meta:meta]` or `#![$meta:meta]` to actually capture an attribute.

```
macro_rules! metas {
    ($($meta:meta)*) => ();
}

metas! {
    ASimplePath
    super::man
    path = "home"
    foo(bar)
}
```

Doc-Comment Fact: Doc-Comments like `/// ...` and `//! ...` are actually syntax sugar for attributes! They desugar to `#[doc="..."]` and `#![doc="..."]` respectively, meaning you can match on them like with attributes!

---

pat

The `pat` fragment matches any kind of [pattern], including or-patterns starting with the 2021 edition.

```
macro_rules! patterns {
    ($($pat:pat)*) => ();
}

patterns! {
    "literal"
    _
    0..5
    ref mut PatternsAreNice
    0 | 1 | 2 | 3
}
```

pat_param

In the 2021 edition, the behavior for the `pat` fragment type has been changed to allow or-patterns to be parsed. This changes the follow list of the fragment, preventing such fragment from being followed by a `|` token. To avoid this problem or to get the old fragment behavior back one can use the `pat_param` fragment which allows `|` to follow it, as it disallows top level or-patterns.

```
macro_rules! patterns {
    ($( $( $pat:pat_param )|+ )*) => ();
}

patterns! {
    "literal"
    _
    0..5
    ref mut PatternsAreNice
    0 | 1 | 2 | 3
}
```

path

The `path` fragment matches a so called [TypePath] style path. This includes the function style trait forms, `Fn() -> ()`.

```
macro_rules! paths {
    ($($path:path)*) => ();
}

paths! {
    ASimplePath
    ::A::B::C::D
    G::<eneri>::C
    FnMut(u32) -> ()
}
```

stmt

The `statement` fragment solely matches a statement without its trailing semicolon, unless it is an item statement that requires one (such as a Unit-Struct).

Let's use a simple example to show exactly what is meant with this. We use a macro that merely emits what it captures:

```
macro_rules! statements {
    ($($stmt:stmt)*) => ($($stmt)*);
}

fn main() {
    statements! {
        struct Foo;
        fn foo() {}
        let zig = 3
        let zig = 3;
        3
        3;
        if true {} else {}
        {}
    }
}
```

Expanding this, via the playground for example[1], gives us roughly the following:

```
/* snip */

fn main() {
    struct Foo;
    fn foo() { }
    let zig = 3;
    let zig = 3;
    ;
    3;
    3;
    ;
    if true { } else { }
    { }
}
```

From this we can tell a few things.

The first you should be able to see immediately is that while the `stmt` fragment doesn't capture trailing semicolons, it still emits them when required, even if the statement is already followed by one. The simple reason for that is that semicolons on their own are already valid statements which the fragment captures eagerly. So our macro isn't capturing 8 times, but 10! This can be important when doing multiples repetitions and expanding these in one repetition expansion, as the repetition numbers have to match in those cases.

Another thing you should be able to notice here is that the trailing semicolon of the `struct Foo;` item statement is being matched, otherwise we would've seen an extra one like in the other cases. This makes sense as we already said, that for item statements that require one, the trailing semicolon will be matched with.

A last observation is that expressions get emitted back with a trailing semicolon, unless the expression solely consists of only a block expression or control flow expression.

The fine details of what was just mentioned here can be looked up in the reference.

Fortunately, these fine details here are usually not of importance whatsoever, with the small exception that was mentioned earlier in regards to repetitions which by itself shouldn't be a common problem to run into.

`tt`

The `tt` fragment matches a TokenTree. If you need a refresher on what exactly a TokenTree was you may want to revisit the TokenTree chapter of this book. The `tt` fragment is one of the most powerful fragments, as it can match nearly anything while still allowing you to inspect the contents of it at a later state in the macro.

This allows one to make use of very powerful patterns like the tt-muncher or the push-down-accumulator.

`ty`

The `ty` fragment matches any kind of type expression.

```
macro_rules! types {
    ($($type:ty)*) => ();
}

types! {
    foo::bar
    bool
    [u8]
    impl IntoIterator<Item = u32>
}
```

`vis`

The `vis` fragment matches a *possibly empty* Visibility qualifier.

```
macro_rules! visibilities {
    //         V~~Note this comma, since we cannot repeat a `vis` fragment on its own
    ($($vis:vis,)*) => ();
}

visibilities! {
    , // no vis is fine, due to the implicit `?`
    pub,
    pub(crate),
    pub(in super),
    pub(in some_path),
}
```

While able to match empty sequences of tokens, the fragment specifier still acts quite different from optional repetitions which is described in the following:

If it is being matched against no left over tokens the entire macro matching fails.

```
macro_rules! non_optional_vis {
    ($vis:vis) => ();
}
non_optional_vis!();
// ^^^^^^^^^^^^^^^ error: missing tokens in macro arguments
```

`$vis:vis $ident:ident` matches fine.

```
macro_rules! vis_ident {
    ($vis:vis $ident:ident) => ();
}
vis_ident!(pub foo); // this works fine
```

In contrast, `$(pub)? $ident:ident` is ambiguous, as `pub` denotes a valid identifier.

```
macro_rules! pub_ident {
    ($(pub)? $ident:ident) => ();
}
pub_ident!(pub foo);
        // ^^^ error: local ambiguity when calling macro `pub_ident`: multiple parsing
options: built-in NTs ident ('ident') or 1 other option.
```

Being a fragment that matches the empty token sequence also gives it a very interesting quirk in combination with `tt` fragments and recursive expansions.

When matching the empty token sequence, the metavariable will still count as a capture and since it is not a `tt`, `ident` or `lifetime` fragment it will become opaque to further expansions. This means if this capture is passed onto another macro invocation that captures it as a `tt` you effectively end up with token tree that contains nothing!

```
macro_rules! it_is_opaque {
    (()) => { "()" };
    (($tt:tt)) => { concat!("$tt is ", stringify!($tt)) };
    ($vis:vis ,) => { it_is_opaque!( ($vis) ) }
}
fn main() {
    // this prints "$tt is ", as the recursive calls hits the second branch with
    // an empty tt, opposed to matching with the first branch!
    println!("{}", it_is_opaque!(,));
}
```

---

1. See the [debugging chapter](#) for tips on how to do this. ↩

Metavariables and Expansion Redux

Once the parser begins consuming tokens for a metavariable, *it cannot stop or backtrack*. This means that the second rule of the following macro *cannot ever match*, no matter what input is provided:

```
macro_rules! dead_rule {
    ($e:expr) => { ... };
    ($i:ident +) => { ... };
}
```

Consider what happens if this macro is invoked as `dead_rule!(x+)`. The interpreter will start at the first rule, and attempt to parse the input as an expression. The first token `x` is valid as an expression. The second token is *also* valid in an expression, forming a binary addition node.

At this point, given that there is no right-hand side of the addition, you might expect the parser to give up and try the next rule. Instead, the parser will panic and abort the entire compilation, citing a syntax error.

As such, it is important in general that you write macro rules from most-specific to least-specific.

To defend against future syntax changes altering the interpretation of macro input, `macro_rules!` restricts what can follow various metavariables. The complete list, showing what may follow what fragment specifier, as of Rust 1.46 is as follows:

- `stmt` and `expr`: `=>`, `,`, or `;`
- `pat`: `=>`, `,`, `=`, `if`, `in` [1]
- `pat_param`: `=>`, `,`, `=`, `|`, `if`, `in`
- `path` and `ty`: `=>`, `,`, `=`, `|`, `;`, `:`, `>`, `>>`, `[`, `{`, `as`, `where`, or a macro variable of the `block` fragment specifier.
- `vis`: `,`, an identifier other than a non-raw `priv`, any token that can begin a type or a metavariable with an `ident`, `ty`, or `path` fragment specifier.
- All other fragment specifiers have no restrictions.

Repetitions also adhere to these restrictions, meaning if a repetition can repeat multiple times(`*` or `+`), then the contents must be able to follow themselves. If a repetition can repeat zero times (`?` or `*`) then what comes after the repetition must be able to follow what comes before.

The parser also does not perform any kind of lookahead. That means if the compiler cannot unambiguously determine how to parse the macro invocation one token at a time, it will abort with an ambiguity error. A simple example that triggers this:

```
macro_rules! ambiguity {
    ($($i:ident)* $i2:ident) => { };
}

// error:
//    local ambiguity: multiple parsing options: built-in NTs ident ('i') or ident
('i2').
ambiguity!(an_identifier);
```

The parser does not look ahead past the identifier to see if the following token is a `)`, which would allow it to parse properly.

One aspect of substitution that often surprises people is that substitution is *not* token-based, despite

very much *looking* like it.

Consider the following:

```
macro_rules! capture_then_match_tokens {
    ($e:expr) => {match_tokens!($e)};
}

macro_rules! match_tokens {
    ($a:tt + $b:tt) => {"got an addition"};
    (($i:ident)) => {"got an identifier"};
    ($($other:tt)*) => {"got something else"};
}

fn main() {
    println!("{}\n{}\n{}\n",
        match_tokens!((caravan)),
        match_tokens!(3 + 6),
        match_tokens!(5));
    println!("{}\n{}\n{}",
        capture_then_match_tokens!((caravan)),
        capture_then_match_tokens!(3 + 6),
        capture_then_match_tokens!(5));
}
```

The output is:

```
got an identifier
got an addition
got something else

got something else
got something else
got something else
```

By parsing the input into an AST node, the substituted result becomes *un-destructible*; *i.e.* you cannot examine the contents or match against it ever again.

Here is *another* example which can be particularly confusing:

```
macro_rules! capture_then_what_is {
    (#[$m:meta]) => {what_is!(#[$m])};
}

macro_rules! what_is {
    (#[no_mangle]) => {"no_mangle attribute"};
    (#[inline]) => {"inline attribute"};
    ($($tts:tt)*) => {concat!("something else (", stringify!($($tts)*), ")")};
}

fn main() {
    println!(
        "{}\n{}\n{}\n{}",
        what_is!(#[no_mangle]),
        what_is!(#[inline]),
        capture_then_what_is!(#[no_mangle]),
        capture_then_what_is!(#[inline]),
    );
}
```

The output is:

```
no_mangle attribute
inline attribute
something else (#[no_mangle])
something else (#[inline])
```

The only way to avoid this is to capture using the `tt`, `ident` or `lifetime` kinds. Once you capture with anything else, the only thing you can do with the result from then on is substitute it directly into the output.

---

1. **Edition Differences**: Before the 2021 edition, `pat` may also be followed by `|`. ↩

Metavariable Expressions

RFC: rfcs#1584
Tracking Issue: rust#83527
Feature: `#![feature(macro_metavar_expr)]`

Note: The example code snippets are very bare bones, trying to show off how they work. If you think you got small snippets with proper isolated usage of these expression please submit them!

As mentioned in the `methodical introduction`, Rust has special expressions that can be used by macro transcribers to obtain information about metavariables that are otherwise difficult or even impossible to get. This chapter will introduce them more in-depth together with usage examples.

- Metavariable Expressions
  - Dollar Dollar (`$$`)
  - `count($ident, depth)`
  - `index(depth)`
  - `len(depth)`
  - `ignore($ident)`

Dollar Dollar ($$)

The `$$` expression expands to a single `$`, making it effectively an escaped `$`. This enables the ability in writing macros emitting new macros as the former macro won't transcribe metavariables, repetitions and metavariable expressions that have an escaped `$`.

We can see the problem without using `$$` in the following snippet:

```
macro_rules! foo {
    () => {
        macro_rules! bar {
            ( $( $any:tt )* ) => { $( $any )* };
            // ^^^^^^^^^^^^ error: attempted to repeat an expression containing no
syntax variables matched as repeating at this depth
        }
    };
}

foo!();
```

The problem is obvious, the transcriber of foo sees a repetition and tries to repeat it when transcribing, but there is no `$any` metavariable in its scope causing it to fail. With `$$` we can get around this as the transcriber of `foo` will no longer try to do the repetition.[1]

```
#![feature(macro_metavar_expr)]

macro_rules! foo {
    () => {
        macro_rules! bar {
            ( $$( $$any:tt )* ) => { $$( $$any )* };
        }
    };
}

foo!();
bar!();
```

count($ident, depth)

The `count` metavariable expression expands to the repetition count of the metavariable `$ident` up to the given repetition depth.

- The `$ident` argument must be a declared metavariable in the scope of the rule.
- The `depth` argument must be an integer literal of value less or equal to the maximum repetition depth that the `$ident` metavariable appears in.
- The expression expands to an unsuffixed integer literal token.

The `count($ident)` expression defaults `depth` to the maximum valid depth, making it count the total repetitions for the given metavariable.

```
#![feature(macro_metavar_expr)]

macro_rules! foo {
    ( $( $outer:ident ( $( $inner:ident ),* ) ; )* ) => {
        println!("count(outer, 0): $outer repeats {} times", ${count($outer)});
        println!("count(inner, 0): The $inner repetition repeats {} times in the outer
repetition", ${count($inner, 0)});
        println!("count(inner, 1): $inner repeats {} times in the inner repetitions",
${count($inner, 1)});
    };
}

fn main() {
    foo! {
        outer () ;
        outer ( inner , inner ) ;
        outer () ;
        outer ( inner ) ;
    };
}
```

index(depth)

The `index(depth)` metavariable expression expands to the current iteration index of the repetition at the given depth.

- The `depth` argument targets the repetition at `depth` counting outwards from the inner-most repetition where the expression is invoked.

- The expression expands to an unsuffixed integer literal token.

The `index()` expression defaults `depth` to `0`, making it a shorthand for `index(0)`.

```
#![feature(macro_metavar_expr)]

macro_rules! attach_iteration_counts {
    ( $( ( $( $inner:ident ),* ) ; )* ) => {
        ( $(
            $((
                stringify!($inner),
                ${index(1)}, // this targets the outer repetition
                ${index()}  // and this, being an alias for `index(0)` targets the
inner repetition
            ),)*
        )* )
    };
}

fn main() {
    let v = attach_iteration_counts! {
        ( hello ) ;
        ( indices , of ) ;
        () ;
        ( these, repetitions ) ;
    };
    println!("{v:?}");
}
```

`len(depth)`

The `len(depth)` metavariable expression expands to the iteration count of the repetition at the given depth.

- The `depth` argument targets the repetition at `depth` counting outwards from the inner-most repetition where the expression is invoked.
- The expression expands to an unsuffixed integer literal token.

The `len()` expression defaults `depth` to `0`, making it a shorthand for `len(0)`.

```
#![feature(macro_metavar_expr)]

macro_rules! lets_count {
    ( $( $outer:ident ( $( $inner:ident ),* ) ; )* ) => {
        $(
            $(
                println!(
                    "'{}' in inner iteration {}/{} with '{}' in outer iteration {}/{}
",
                    stringify!($inner), ${index()}, ${len()},
                    stringify!($outer), ${index(1)}, ${len(1)},
                );
            )*
        )*
    };
}

fn main() {
    lets_count!(
        many (small , things) ;
        none () ;
        exactly ( one ) ;
    );
}
```

ignore($ident)

The `ignore($ident)` metavariable expression expands to nothing, making it possible to expand
something as often as a metavariable repeats without expanding the metavariable.

- The `$ident` argument must be a declared metavariable in the scope of the rule.

```
#![feature(macro_metavar_expr)]

macro_rules! repetition_tuples {
    ( $( ( $( $inner:ident ),* ) ; )* ) => {
        ($(
            $(
                (
                    ${index()},
                    ${index(1)}
                    ${ignore($inner)} // without this metavariable expression,
compilation would fail
                ),
            )*
        )*)
    };
}

fn main() {
    let tuple = repetition_tuples!(
        ( one, two ) ;
        () ;
        ( one ) ;
        ( one, two, three ) ;
    );
    println!("{tuple:?}");
}
```

1. Before `$$` was implemented, users had to resort to a tricky and not so well-known hack to declare nested macros with repetitions [via using](#) `$tt` [like this](#). ↵

Hygiene

`macro_rules!` macros in Rust are *partially* hygienic, also called mixed hygiene. Specifically, they are hygienic when it comes to *local variables*, *labels* and `$crate`, but nothing else.

Hygiene works by attaching an invisible "syntax context" value to all identifiers. When two identifiers are compared, *both* the identifiers' textual names *and* syntax contexts must be identical for the two to be considered equal.

To illustrate this, consider the following code:

```rust
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42;
            $e
        }
    }
}

let four = using_a!(a / 10);
```

We will use the background colour to denote the syntax context. Now, let's expand the macro invocation:

```rust
let four = {
    let a = 42;
    a / 10
};
```

First, recall that `macro_rules!` invocations effectively *disappear* during expansion.

Second, if you attempt to compile this code, the compiler will respond with something along the following lines:

```
error[E0425]: cannot find value `a` in this scope
  --> src/main.rs:13:21
   |
13 | let four = using_a!(a / 10);
   |                     ^ not found in this scope
```

Note that the background colour (*i.e.* syntax context) for the expanded macro *changes* as part of expansion. Each `macro_rules!` macro expansion is given a new, unique syntax context for its contents. As a result, there are *two different* `a`s in the expanded code: one in the first syntax context, the second in the other. In other words, `a` is not the same identifier as `a`, however similar they may appear.

That said, tokens that were substituted *into* the expanded output *retain* their original syntax context (by virtue of having been provided to the macro as opposed to being part of the macro itself). Thus, the solution is to modify the macro as follows:

```
macro_rules! using_a {
    ($a:ident, $e:expr) => {
        {
            let $a = 42;
            $e
        }
    }
}
```

```
let four = using_a!(a, a / 10);
```

Which, upon expansion becomes:

```
let four = {
    let a = 42;
    a / 10
};
```

The compiler will accept this code because there is only one `a` being used.

$crate

Hygiene is also the reason that we need the `$crate` metavariable when our macro needs access to other items in the defining crate. What this special metavariable does is that it expands to an absolute path to the defining crate.

```
//// Definitions in the `helper_macro` crate.
#[macro_export]
macro_rules! helped {
    // () => { helper!() } // This might lead to an error due to 'helper' not being in
scope.
    () => { $crate::helper!() }
}

#[macro_export]
macro_rules! helper {
    () => { () }
}

//// Usage in another crate.
// Note that `helper_macro::helper` is not imported!
use helper_macro::helped;

fn unit() {
    // but it still works due to `$crate` properly expanding to the crate path
`helper_macro`
    helped!();
}
```

Note that, because `$crate` refers to the current crate, it must be used with a fully qualified module path when referring to non-macro items:

```
pub mod inner {
    #[macro_export]
    macro_rules! call_foo {
        () => { $crate::inner::foo() };
    }

    pub fn foo() {}
}
```

Non-Identifier Identifiers

There are two tokens which you are likely to run into eventually that *look* like identifiers, but aren't. Except when they are.

First is `self` . This is *very definitely* a keyword. However, it also happens to fit the definition of an identifier. In regular Rust code, there's no way for `self` to be interpreted as an identifier, but it *can* happen with `macro_rules!` macros:

```
macro_rules! what_is {
    (self) => {"the keyword `self`"};
    ($i:ident) => {concat!("the identifier `", stringify!($i), "`")};
}

macro_rules! call_with_ident {
    ($c:ident($i:ident)) => {$c!($i)};
}

fn main() {
    println!("{}", what_is!(self));
    println!("{}", call_with_ident!(what_is(self)));
}
```

The above outputs:

```
the keyword `self`
the keyword `self`
```

But that makes no sense; `call_with_ident!` required an identifier, matched one, and substituted it! So `self` is both a keyword and not a keyword at the same time. You might wonder how this is in any way important. Take this example:

```
macro_rules! make_mutable {
    ($i:ident) => {let mut $i = $i;};
}

struct Dummy(i32);

impl Dummy {
    fn double(self) -> Dummy {
        make_mutable!(self);
        self.0 *= 2;
        self
    }
}
```

This fails to compile with:

```
error: `mut` must be followed by a named binding
 --> src/main.rs:2:24
  |
2 |     ($i:ident) => {let mut $i = $i;};
  |                        ^^^^^^ help: remove the `mut` prefix: `self`
...
9 |         make_mutable!(self);
  |         ------------------- in this macro invocation
  |
  = note: `mut` may be followed by `variable` and `variable @ pattern`
```

So the macro will happily match `self` as an identifier, allowing you to use it in cases where you can't actually use it. But, fine; it somehow remembers that `self` is a keyword even when it's an identifier, so you *should* be able to do this, right?

```rust
macro_rules! make_self_mutable {
    ($i:ident) => {let mut $i = self;};
}

struct Dummy(i32);

impl Dummy {
    fn double(self) -> Dummy {
        make_self_mutable!(mut_self);
        mut_self.0 *= 2;
        mut_self
    }
}
```

This fails with:

```
error[E0424]: expected value, found module `self`
  --> src/main.rs:2:33
   |
2  |        ($i:ident) => {let mut $i = self;};
   |                                    ^^^^ `self` value is a keyword only available
in methods with a `self` parameter
...
8  | /      fn double(self) -> Dummy {
9  | |          make_self_mutable!(mut_self);
   | |          --------------------------- in this macro invocation
10 | |          mut_self.0 *= 2;
11 | |          mut_self
12 | |      }
   | |_____- this function has a `self` parameter, but a macro invocation can only
access identifiers it receives from parameters
   |
```

Now the compiler thinks we refer to our module with `self`, but that doesn't make sense. We already have a `self` right there, in the function signature which is definitely not a module. It's almost like it's complaining that the `self` it's trying to use isn't the *same* `self` ... as though the `self` keyword has hygiene, like an... identifier.

```rust
macro_rules! double_method {
    ($body:expr) => {
        fn double(mut self) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {{
        self.0 *= 2;
        self
    }}
}
```

Same error. What about...

```
macro_rules! double_method {
    ($self_:ident, $body:expr) => {
        fn double(mut $self_) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {self, {
        self.0 *= 2;
        self
    }}
}
```

At last, *this works*. So `self` is both a keyword *and* an identifier when it feels like it. Surely this works for other, similar constructs, right?

```
macro_rules! double_method {
    ($self_:ident, $body:expr) => {
        fn double($self_) -> Dummy {
            $body
        }
    };
}

struct Dummy(i32);

impl Dummy {
    double_method! {_, 0}
}
```

```
error: no rules expected the token `_`
  --> src/main.rs:12:21
   |
1  | macro_rules! double_method {
   | -------------------------- when calling this macro
...
12 |     double_method! {_, 0}
   |                     ^ no rules expected this token in macro call
```

No, of course not. `_` is a keyword that is valid in patterns and expressions, but somehow *isn't* an identifier like the keyword `self` is, despite matching the definition of an identifier just the same.

You might think you can get around this by using `$self_:pat` instead; that way, `_` will match! Except, no, because `self` isn't a pattern. Joy.

The only work around for this (in cases where you want to accept some combination of these tokens) is to use a `tt` matcher instead.

Debugging

Note: This is a list of debugging tools specifically tailored towards declarative macros, additional means of debugging these can be found in the debugging chapter of syntax extensions.

One of the most useful is `trace_macros!`, which is a directive to the compiler instructing it to dump every `macro_rules!` macro invocation prior to expansion. For example, given the following:

```
#![feature(trace_macros)]

macro_rules! each_tt {
    () => {};
    ($_tt:tt $($rest:tt)*) => {each_tt!($($rest)*);};
}

each_tt!(foo bar baz quux);
trace_macros!(true);
each_tt!(spim wak plee whum);
trace_macros!(false);
each_tt!(trom qlip winp xod);
```

The output is:

```
note: trace_macro
  --> src/main.rs:11:1
   |
11 | each_tt!(spim wak plee whum);
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: expanding `each_tt! { spim wak plee whum }`
   = note: to `each_tt ! (wak plee whum) ;`
   = note: expanding `each_tt! { wak plee whum }`
   = note: to `each_tt ! (plee whum) ;`
   = note: expanding `each_tt! { plee whum }`
   = note: to `each_tt ! (whum) ;`
   = note: expanding `each_tt! { whum }`
   = note: to `each_tt ! () ;`
   = note: expanding `each_tt! {  }`
   = note: to ``
```

This is *particularly* invaluable when debugging deeply recursive `macro_rules!` macros. You can also enable this from the command-line by adding `-Z trace-macros` to the compiler command line.

Secondly, there is `log_syntax!` which causes the compiler to output all tokens passed to it. For example, this makes the compiler sing a song:

```
#![feature(log_syntax)]

macro_rules! sing {
    () => {};
    ($tt:tt $($rest:tt)*) => {log_syntax!($tt); sing!($($rest)*);};
}

sing! {
    ^ < @ < . @ *
    '\x08' '{' '"' _ # ' '
    - @ '$' && / _ %
    ! ( '\t' @ | = >
    ; '\x08' '\'' + '$' ? '\x7f'
    , # '"' ~ | ) '\x07'
}
```

This can be used to do slightly more targeted debugging than `trace_macros!`.

Another amazing tool is `lukaslueg`'s `macro_railroad`, a tool that allows you visualize and generate syntax diagrams for Rust's `macro_rules!` macros. It visualizes the accepted macro's grammar as an automata.

Scoping

The way in which mbe macros are scoped can be somewhat unintuitive. They use two forms of scopes: textual scope, and path-based scope.

When such a macro is invoked by an unqualified identifier(an identifier that isn't part of a multi-segment path), it is first looked up in textual scoping and then in path-based scoping should the first lookup not yield any results. If it is invoked by a qualified identifier it will skip the textual scoping lookup and instead only do a look up in the path-based scoping.

Textual Scope

Firstly, unlike everything else in the language, function-like macros will remain visible in sub-modules.

```
macro_rules! X { () => {}; }
mod a {
    X!(); // defined
}
mod b {
    X!(); // defined
}
mod c {
    X!(); // defined
}
```

Note: In these examples, remember that all of them have the *same behavior* when the module contents are in separate files.

Secondly, *also* unlike everything else in the language, `macro_rules!` macros are only accessible *after* their definition. Also note that this example demonstrates how `macro_rules!` macros do not "leak" out of their defining scope:

```
mod a {
    // X!(); // undefined
}
mod b {
    // X!(); // undefined
    macro_rules! X { () => {}; }
    X!(); // defined
}
mod c {
    // X!(); // undefined
}
```

To be clear, this lexical order dependency applies even if you move the macro to an outer scope:

```
mod a {
    // X!(); // undefined
}
macro_rules! X { () => {}; }
mod b {
    X!(); // defined
}
mod c {
    X!(); // defined
}
```

However, this dependency *does not* apply to macros themselves:

```
mod a {
    // X!(); // undefined
}
macro_rules! X { () => { Y!(); }; }
mod b {
    // X!(); // defined, but Y! is undefined
}
macro_rules! Y { () => {}; }
mod c {
    X!(); // defined, and so is Y!
}
```

Defining `macro_rules!` macros multiple times is allowed and the most recent declaration will simply shadow previous ones unless it has gone out of scope.

```
macro_rules! X { (1) => {}; }
X!(1);
macro_rules! X { (2) => {}; }
// X!(1); // Error: no rule matches `1`
X!(2);

mod a {
    macro_rules! X { (3) => {}; }
    // X!(2); // Error: no rule matches `2`
    X!(3);
}
// X!(3); // Error: no rule matches `3`
X!(2);
```

`macro_rules!` macros can be exported from a module using the `#[macro_use]` attribute. Using this on a module is similar to saying that you do not want to have the module's macro's scope end with the module.

```
mod a {
    // X!(); // undefined
}
#[macro_use]
mod b {
    macro_rules! X { () => {}; }
    X!(); // defined
}
mod c {
    X!(); // defined
}
```

Note that this can interact in somewhat bizarre ways due to the fact that identifiers in a

`macro_rules!` macro (including other macros) are only resolved upon expansion:

```rust
mod a {
    // X!(); // undefined
}
#[macro_use]
mod b {
    macro_rules! X { () => { Y!(); }; }
    // X!(); // defined, but Y! is undefined
}
macro_rules! Y { () => {}; }
mod c {
    X!(); // defined, and so is Y!
}
```

Another complication is that `#[macro_use]` applied to an `extern crate` *does not* behave this way: such declarations are effectively *hoisted* to the top of the module. Thus, assuming `X!` is defined in an external crate called `macs`, the following holds:

```rust
mod a {
    // X!(); // defined, but Y! is undefined
}
macro_rules! Y { () => {}; }
mod b {
    X!(); // defined, and so is Y!
}
#[macro_use] extern crate macs;
mod c {
    X!(); // defined, and so is Y!
}
```

Finally, note that these scoping behaviors apply to *functions* as well, with the exception of `#[macro_use]` (which isn't applicable):

```rust
macro_rules! X {
    () => { Y!() };
}

fn a() {
    macro_rules! Y { () => {"Hi!"} }
    assert_eq!(X!(), "Hi!");
    {
        assert_eq!(X!(), "Hi!");
        macro_rules! Y { () => {"Bye!"} }
        assert_eq!(X!(), "Bye!");
    }
    assert_eq!(X!(), "Hi!");
}

fn b() {
    macro_rules! Y { () => {"One more"} }
    assert_eq!(X!(), "One more");
}
```

These scoping rules are why a common piece of advice is to place all `macro_rules!` macros which should be accessible "crate wide" at the very top of your root module, before any other modules. This ensures they are available *consistently*. This also applies to `mod` definitions for files, as in:

```
#[macro_use]
mod some_mod_that_defines_macros;
mod some_mod_that_uses_those_macros;
```

The order here is important, swap the declaration order and it won't compile.

Path-Based Scope

By default, a `macro_rules!` macro has no path-based scope. However, if it has the `#[macro_export]` attribute, then it is declared in the crate root scope and can be referred to similar to how you refer to any other item. The Import and Export chapter goes more in-depth into said attribute.

Import and Export

Importing `macro_rules!` macros differs between the two Rust Editions, 2015 and 2018. It is recommended to read both parts nevertheless, as the 2018 Edition can still use the constructs that are explained in the 2015 Edition.

Edition 2015

In Edition 2015 you have to use the `#[macro_use]` attribute that has already been introduced in the scoping chapter. This can be applied to *either* modules or external crates. For example:

```
#[macro_use]
mod macros {
    macro_rules! X { () => { Y!(); } }
    macro_rules! Y { () => {} }
}

X!();
```

`macro_rules!` macros can be exported from the current crate using `#[macro_export]`. Note that this *ignores* all visibility.

Given the following definition for a library package `macs`:

```
mod macros {
    #[macro_export] macro_rules! X { () => { Y!(); } }
    #[macro_export] macro_rules! Y { () => {} }
}

// X! and Y! are *not* defined here, but *are* exported,
// despite `macros` being private.
```

The following code will work as expected:

```
X!(); // X is defined
#[macro_use] extern crate macs;
X!();
```

This works, as said in the scoping chapter, because `#[macro_use]` works slightly different on extern crates, as it basically *hoists* the exported macros out of the crate to the top of the module.

Note: you can *only* `#[macro_use]` an external crate from the root module.

Finally, when importing `macro_rules!` macros from an external crate, you can control *which* macros you import. You can use this to limit namespace pollution, or to override specific macros, like so:

```
// Import *only* the `X!` macro.
#[macro_use(X)] extern crate macs;

// X!(); // X is defined, but Y! is undefined

macro_rules! Y { () => {} }

X!(); // X is defined, and so is Y!

fn main() {}
```

When exporting `macro_rules!` macros, it is often useful to refer to non-macro symbols in the defining crate. Because crates can be renamed, there is a special substitution variable available: `$crate`. This will *always* expand to an absolute path prefix to the containing crate (*e.g.* `:: macs`).

Note that unless your compiler version is >= 1.30, this does *not* work for `macro_rules!` macros, because `macro_rules!` macros do not interact with regular name resolution in any way. Otherwise, you cannot use something like `$crate::Y!` to refer to a particular macro within your crate. The implication, combined with selective imports via `#[macro_use]` is that there is currently *no way* to guarantee any given macro will be available when imported by another crate.

It is recommended that you *always* use absolute paths to non-macro names, to avoid conflicts, *including* names in the standard library.

Edition 2018

The 2018 Edition made our lives a lot easier when it comes to `macro_rules!` macros. Why you ask? Quite simply because it managed to make them feel more like proper items than some special thing in the language. What this means is that we can properly import and use them in a namespaced fashion!

So instead of using `#[macro_use]` to import every exported macro from a crate into the global namespace we can now do the following:

```
use some_crate::some_macro;

fn main() {
    some_macro!("hello");
    // as well as
    some_crate::some_other_macro!("macro world");
}
```

Unfortunately, this only applies for external crates, if you use `macro_rules!` macros that you have defined in your own crate you are still required to go with `#[macro_use]` on the defining modules. So scoping applies there the same way as before as well.

The `$crate` prefix works in this version for everything, macros and items alike since this Edition came out with Rust 1.31.

Turing Completeness

Rust's Declarative Macros are Turing-Complete in their expansion. Here we will prove this fact.

Tag Systems

A Tag System is a model of computation that we will use in our proof.

Definition

Note: the definition below is taken verbatim from Wikipedia.

Briefly, a Tag System is a triplet (m, A, P) where:

- m is a positive integer, called the deletion number.
- A is a finite alphabet of symbols, one of which can be a special *halting symbol* H. All finite (possibly empty) strings on A are called **words**.
- P is a set of production rules, assigning a word $P(x)$ (called a production) to each symbol x in A. The production (say $P(H)$) assigned to the halting symbol is seen below to play no role in computations, but for convenience is taken to be $P(H) = H$.

A **halting word** is a word that either:

- Begins with the halting symbol H,
- or whose length is less than m.

A **transformation** t (called the tag operation) is defined on the set of non-halting words, such that if x denotes the leftmost symbol of a word S, then $t(S)$ is the result of deleting the leftmost m symbols of S and appending the word $P(x)$ on the right. Thus, the system processes the m-symbol head into a tail of variable length, but the generated tail depends solely on the first symbol of the head.

A **computation** by a tag system is a finite sequence of words produced by iterating the transformation t, starting with an initially given word and halting when a halting word is produced.

Computational Power

When its parameter m is greater than 1, a Tag System is capable of representing an arbitrary Turing Machine. We will use this fact in our proof.

Proof

The proof consists of the following steps:

1. We will take an **arbitrary** Tag System T of $m > 1$.

2. We will construct a `macro_rules!` definition `my_tag_system` that simulates this Tag System, such that:
    1. Every halting string in the Tag System halts in our macro.
    2. Every production rule in the Tag System is reproduced exactly in our macro.

This will prove that we can fully simulate any Tag System of $m > 1$ with a `macro_rules!` definition's expansion. And, since simulating a Tag System inherits its halting properties, our macro's expansion will halt **if and only if the Tag System halts**.

An Arbitrary Tag System

Let's specify an arbitrary Tag System $T = (m, A, P)$:

- $m \in \mathbb{N}, m > 1$
- A is our set of symbols.
    - $A = \{a, b, c, \ldots\}$
    - A is finite
    - The empty string is considered a string on A.
    - A may contain a special character for halting: H.
- P is a set of production rules $\{p_a, p_b, p_c, \ldots\}$, which can be equivalently seen as a function from A to the strings on A:

$$P : A \to A^\star$$

Constructing a Macro Definition for our Tag System

We will now construct a `macro_rules!` definition that fully simulates `T`.

Declaration

We begin with the declaration. Nothing special here.

```
macro_rules! my_tag_system {
```

Halting Input

First we define what inputs will halt. There are two cases.

A string beginning with the special character H will halt.

```
(H $($tail:tt)*) => {};
```

A string of length $[0 \ldots (m - 1)]$ will halt.

```
() => {};

($ident_1:ident) => {};

($ident_1:ident $ident_2:ident) => {};

/* ... */

($ident_1:ident $ident_2:ident /* ... */ $ident_m_minus_2:ident) => {};

($ident_1:ident $ident_2:ident /* ... */ $ident_m_minus_2:ident
$ident_m_minus_1:ident) => {};
```

Non-Halting Input

Note: given two strings a, b, the concatenation of both is written as $a|b$

For each symbol in A, we have a production rule. To recap:

A string $S = x|head|tail$ with $\mathtt{len}(head) = m - 1$ will be transformed into $t(S)$ in the following way:

$$t(S)$$

$$= t(x|head|tail)$$

$$= tail|P(x)$$

Thus, we write each of the rules in `P` like this:

```
// P(x)
(x $d_1:tt $d_2:tt /* ... */ $d_m_minus_1:tt $($tail:tt)*) => {
    my_tag_system!($($tail)* p_x)
};
```

Where we replace `p_x` with the expansion of $P(x)$.

All Together

Putting it all together, it looks like this:

```
macro_rules! my_tag_system {

    // Halting Input
    (H $($tail:tt)*) => {};

    () => {};

    ($ident_1:ident) => {};

    ($ident_1:ident $ident_2:ident) => {};

    /* ... */

    ($ident_1:ident $ident_2:ident /* ... */ $ident_m_minus_2:ident) => {};

    ($ident_1:ident $ident_2:ident /* ... */ $ident_m_minus_2:ident
 $ident_m_minus_1:ident) => {};

    // Non-Halting Input

    // P(a)
    (a $d_1:tt $d_2:tt /* ... */ $d_m_minus_1:tt $($tail:tt)*) => {
        my_tag_system!($($tail)* p_a)
    };

    // P(b)
    (b $d_1:tt $d_2:tt /* ... */ $d_m_minus_1:tt $($tail:tt)*) => {
        my_tag_system!($($tail)* p_b)
    };

    // ...
}
```

Analysis of our Macro Definition

When does our macro's expansion halt?

Well, as we can see from our definition:

- We can represent all strings on A.
- Our macro's definition halts with all strings for which T halts.
- Our macro's definition reproduces all production rules present in T. No more, and no less.

Therefore, the macro expansion process will quite literally simulate the Tag System we've encoded in our definition.

Our macro's expansion will halt **if and only if T halts**.

Caveat: Recursion Limit

The Rust Compiler has, very reasonably, a `recursion_limit` parameter for macro expansion. This allows the compiler to bail itself out in case a macro expansion is taking an unreasonably long time.

Such a parameter does technically mean that our macros will always halt. But for the purposes of formal analysis, that's just an implementation detail.

Note: it's the same as noting that a computer's RAM is finite, and thus doesn't represent a real

Turing Machine.

Besides, this limit can be bypassed by setting the `recursion_limit` attribute to a very large number, for example: `#![recursion_limit = "999999999999999999"]`

Relation to Turing-Completeness

- A Tag System with $m > 1$ is Turing-Complete.
  - This means that we can take an arbitrary Turing Machine M and convert it into a Tag System T with $m > 1$ that halts if and only if M halts.
- We have just proven that we can take an arbitrary Tag System T and convert it into a `macro_rules!` definition `my_tag_system` that halts in its expansion if and only if T halts.

Therefore, we are able to:

1. Take an arbitrary Turing Machine M,
2. Convert it into a Tag System T that halts if and only if M halts,
3. And then convert it into a `macro_rules!` definition `my_tag_system` that halts in its expansion if and only if T halts.

And since the "if and only if" relation is transitive, we've just proven that `macro_rules!` expansion is Turing-Complete.

Consequences of Turing-Completeness

The fact that Rust's Declarative Macros are Turing Complete imposes very important consequences on tooling:

- Will a macro always finish expanding? Undecidable.
- By Rice's Theorem, most properties of a Turing-Complete system are Undecidable.
  - Are two macros equivalent? Who knows! It's actually very likely that this is impossible to tell in general.
  - Is a macro's definition arm redundant? Might be impossible to tell in general.

All of this is to say: as far as the author of this fragment (Félix Fischer) is concerned, it's **To Be Determined** whether or not any of these questions and many more like them are even answerable.

Patterns

Parsing and expansion patterns.

Callbacks

Due to the order that macros are expanded in, it is (as of Rust 1.2) impossible to pass information to a macro from the expansion of *another* macro:

```
macro_rules! recognize_tree {
    (larch) => { println!("#1, the Larch.") };
    (redwood) => { println!("#2, the Mighty Redwood.") };
    (fir) => { println!("#3, the Fir.") };
    (chestnut) => { println!("#4, the Horse Chestnut.") };
    (pine) => { println!("#5, the Scots Pine.") };
    ($($other:tt)*) => { println!("I don't know; some kind of birch maybe?") };
}

macro_rules! expand_to_larch {
    () => { larch };
}

fn main() {
    recognize_tree!(expand_to_larch!());
    // first expands to:  recognize_tree! { expand_to_larch ! (  ) }
    // and then:          println! { "I don't know; some kind of birch maybe?" }
}
```

This can make modularizing macros very difficult.

An alternative is to use recursion and pass a callback:

```
// ...

macro_rules! call_with_larch {
    ($callback:ident) => { $callback!(larch) };
}

fn main() {
    call_with_larch!(recognize_tree);
    // first expands to:  call_with_larch! { recognize_tree }
    // then:              recognize_tree! { larch }
    // and finally:       println! { "#1, the Larch." }
}
```

Using a `tt` repetition, one can also forward arbitrary arguments to a callback.

```
macro_rules! callback {
    ($callback:ident( $($args:tt)* )) => {
        $callback!( $($args)* )
    };
}

fn main() {
    callback!(callback(println("Yes, this *was* unnecessary.")));
}
```

You can, of course, insert additional tokens in the arguments as needed.

Incremental TT Munchers

```
macro_rules! mixed_rules {
    () => {};
    (trace $name:ident; $($tail:tt)*) => {
        {
            println!(concat!(stringify!($name), " = {:?}"), $name);
            mixed_rules!($($tail)*);
        }
    };
    (trace $name:ident = $init:expr; $($tail:tt)*) => {
        {
            let $name = $init;
            println!(concat!(stringify!($name), " = {:?}"), $name);
            mixed_rules!($($tail)*);
        }
    };
}
```

This pattern is perhaps the *most powerful* macro parsing technique available, allowing one to parse grammars of significant complexity. However, it can increase compile times if used excessively, so should be used with care.

A *TT muncher* is a recursive `macro_rules!` macro that works by incrementally processing its input one step at a time. At each step, it matches and removes (munches) some sequence of tokens from the start of its input, generates some intermediate output, then recurses on the input tail.

The reason for "TT" in the name specifically is that the unprocessed part of the input is *always* captured as `$($tail:tt)*`. This is done as a `tt` repetition is the only way to *losslessly* capture part of a macro's input.

The only hard restrictions on TT munchers are those imposed on the `macro_rules!` macro system as a whole:

- You can only match against literals and grammar constructs which can be captured by `macro_rules!`.
- You cannot match unbalanced groups.

It is important, however, to keep the macro recursion limit in mind. `macro_rules!` does not have *any* form of tail recursion elimination or optimization. It is recommended that, when writing a TT muncher, you make reasonable efforts to keep recursion as limited as possible. This can be done by adding additional rules to account for variation in the input (as opposed to recursion into an intermediate layer), or by making compromises on the input syntax to make using standard repetitions more tractable.

Performance

TT munchers are inherently quadratic. Consider a TT muncher rule that consumes one token tree and then recursively calls itself on the remaining input. If it is passed 100 token trees:

- The initial invocation will match against all 100 token trees.
- The first recursive invocation will match against 99 token trees.
- The next recursive invocation will match against 98 token trees.

And so on, down to 1. This is a classic quadratic pattern, and long inputs can cause macro expansion to blow out compile times.

Try to avoid using TT munchers too much, especially with long inputs. The default value of the `recursion_limit` attribute is a good sanity check; if you have to exceed it, you might be heading for trouble.

If you have the choice between writing a TT muncher that can be called once to handle multiple things, or a simpler macro that can be called multiple times to handle a single thing, prefer the latter. For example, you could change a macro that is called like this:

```
f! {
    fn f_u8(x: u32) -> u8;
    fn f_u16(x: u32) -> u16;
    fn f_u32(x: u32) -> u32;
    fn f_u64(x: u64) -> u64;
    fn f_u128(x: u128) -> u128;
}
```

To one that is called like this:

```
f! { fn f_u8(x: u32) -> u8; }
f! { fn f_u16(x: u32) -> u16; }
f! { fn f_u32(x: u32) -> u32; }
f! { fn f_u64(x: u64) -> u64; }
f! { fn f_u128(x: u128) -> u128; }
```

The longer the input, the more likely this will improve compile times.

Also, if a TT muncher macro has many rules, put the most frequently matched rules as early as possible. This avoids unnecessary matching failures. (In fact, this is good advice for any kind of declarative macro, not just TT munchers.)

Finally, if you can write a macro using normal repetition via `*` or `+`, that should be preferred to a TT muncher. This is most likely if each invocation of the TT muncher would only process one token at a time. In more complicated cases, there is an advanced technique used within the `quote` crate that can avoid the quadratic behaviour, at the cost of some conceptual complexity. See this comment for details.

Internal Rules

```
#[macro_export]
macro_rules! foo {
    (@as_expr $e:expr) => {$e};

    ($($tts:tt)*) => {
        foo!(@as_expr $($tts)*)
    };
}
```

Internal rules can be used to unify multiple `macro_rules!` macros into one, or to make it easier to read and write TT Munchers by explicitly naming what rule you wish to call in a macro.

So why is it useful to unify multiple macros-by-example into one? The main reasoning for this is how they are handled in the 2015 Edition of Rust due to `macro_rules!` macros not being namespaced in said edition. This gives one the troubles of having to re-export all the internal `macro_rules!` macros as well as polluting the global macro namespace or even worse, macro name collisions with other crates. In short, it's quite a hassle. This fortunately isn't really a problem anymore nowadays with a rustc version >= 1.30, for more information consult the Import and Export chapter.

Nevertheless, let's talk about how we can unify multiple `macro_rules!` macros into one with this technique and what exactly this technique even is.

We have two `macro_rules!` macros, the common `as_expr!` macro and a `foo` macro that makes use of the first one:

```
#[macro_export]
macro_rules! as_expr { ($e:expr) => {$e} }

#[macro_export]
macro_rules! foo {
    ($($tts:tt)*) => {
        as_expr!($($tts)*)
    };
}
```

This is definitely not the nicest solution we could have for this macro, as it pollutes the global macro namespace as mentioned earlier. In this specific case `as_expr` is also a very simple macro that we only used once, so let's "embed" this macro in our `foo` macro with internal rules! To do so, we simply prepend a new matcher for our macro, which consists of the matcher used in the `as_expr` macro, but with a small addition. We prepend a tokentree that makes it match only when specifically asked to. In this case we can for example use `@as_expr`, so our matcher becomes `(@as_expr $e:expr) => {$e};`. With this we get the macro that was defined at the very top of this page:

```
#[macro_export]
macro_rules! foo {
    (@as_expr $e:expr) => {$e};

    ($($tts:tt)*) => {
        foo!(@as_expr $($tts)*)
    };
}
```

You see how we embedded the `as_expr` macro in the `foo` one? All that changed is that instead of

invoking the `as_expr` macro, we now invoke `foo` recursively but with a special token tree prepended to the arguments, `foo!(@as_expr $($tts)*)` . If you look closely you might even see that this pattern can be combined quite nicely with TT Munchers!

The reason for using `@` was that, as of Rust 1.2, the `@` token is *not* used in prefix position; as such, it cannot conflict with anything. This reasoning became obsolete later on when in Rust 1.7 macro matchers got future proofed by emitting a warning to prevent certain tokens from being allowed to follow certain fragments[1], which in Rust 1.12 became a hard-error. Other symbols or unique prefixes may be used as desired, but use of `@` has started to become widespread, so using it may aid readers in understanding your macro.

Note: in the early days of Rust the `@` token was previously used in prefix position to denote a garbage-collected pointer, back when the language used sigils to denote pointer types. Its only *current* purpose is for binding names to patterns. For this, however, it is used as an *infix* operator, and thus does not conflict with its use here.

Additionally, internal rules will often come *before* any "bare" rules, to avoid issues with `macro_rules!` incorrectly attempting to parse an internal invocation as something it cannot possibly be, such as an expression.

Performance

One downside of internal rules is that they can hurt compile times. Only one macro rule can match any (valid) macro invocation, but the compiler must try to match all rules in order. If a macro has many rules, there can be many such failures, and the use of internal rules will increase the number of such failures.

Also, the `@as_expr` -style identifier makes rules longer, slightly increasing the amount of work the compiler must do when matching.

Therefore, for best performance, avoiding internal rules is best. Avoiding them often makes complex macros easier to read, too.

---

1. ambiguity-restrictions ↩

Push-down Accumulation

The following macro uses *push-down accumulation*.

```
macro_rules! init_array {
    [$e:expr; $n:tt] => {
        {
            let e = $e;
            accum!([$n, e.clone()] -> [])
        }
    };
}
macro_rules! accum {
    ([3, $e:expr] -> [$($body:tt)*]) => { accum!([2, $e] -> [$($body)* $e,]) };
    ([2, $e:expr] -> [$($body:tt)*]) => { accum!([1, $e] -> [$($body)* $e,]) };
    ([1, $e:expr] -> [$($body:tt)*]) => { accum!([0, $e] -> [$($body)* $e,]) };
    ([0, $_:expr] -> [$($body:tt)*]) => { [$($body)*] };
}

let strings: [String; 3] = init_array![String::from("hi!"); 3];
```

All syntax extensions in Rust **must** result in a complete, supported syntax element (such as an expression, item, *etc.*). This means that it is impossible to have a syntax extension expand to a partial construct.

One might hope that the above example could be more directly expressed like so:

```
macro_rules! init_array {
    [$e:expr; $n:tt] => {
        {
            let e = $e;
            [accum!($n, e.clone())]
        }
    };
}
macro_rules! accum {
    (3, $e:expr) => { $e, accum!(2, $e) };
    (2, $e:expr) => { $e, accum!(1, $e) };
    (1, $e:expr) => { $e };
}
```

The expectation is that the expansion of the array literal would proceed as follows:

```
[accum!(3, e.clone())]
[e.clone(), accum!(2, e.clone())]
[e.clone(), e.clone(), accum!(1, e.clone())]
[e.clone(), e.clone(), e.clone()]
```

However, this would require each intermediate step to expand to an incomplete expression. Even though the intermediate results will never be used *outside* of a macro context, it is still forbidden.

Push-down, however, allows us to incrementally build up a sequence of tokens without needing to actually have a complete construct at any point prior to completion. In the example given at the top, the sequence of invocations proceeds as follows:

```
init_array!(String::from("hi!"); 3)
accum!([3, e.clone()] -> [])
accum!([2, e.clone()] -> [e.clone(),])
accum!([1, e.clone()] -> [e.clone(), e.clone(),])
accum!([0, e.clone()] -> [e.clone(), e.clone(), e.clone(),])
[e.clone(), e.clone(), e.clone(),]
```

As you can see, each layer adds to the accumulated output until the terminating rule finally emits it as a complete construct.

The only critical part of the above formulation is the use of `$($body:tt)*` to preserve the output without triggering parsing. The use of `($input) -> ($output)` is simply a convention adopted to help clarify the behavior of such macros.

Push-down accumulation is frequently used as part of incremental TT munchers, as it allows arbitrarily complex intermediate results to be constructed. Internal Rules were of use here as well, as they simplify creating such macros.

Performance

Push-down accumulation is inherently quadratic. Consider a push-down accumulation rule that builds up an accumulator of 100 token trees, one token tree per invocation.

- The initial invocation will match against the empty accumulator.
- The first recursive invocation will match against the accumulator of 1 token tree.
- The next recursive invocation will match against the accumulator of 2 token trees.

And so on, up to 100. This is a classic quadratic pattern, and long inputs can cause macro expansion to blow out compile times. Furthermore, TT munchers are also inherently quadratic over their input, so a macro that uses both TT munching *and* push-down accumulation will be doubly quadratic!

All the performance advice about TT munchers holds for push-down accumulation. In general, avoid using them too much, and keep them as simple as possible.

Finally, make sure you put the accumulator at the *end* of rules, rather than the beginning. That way, if a rule fails, the compiler won't have had to match the (potentially long) accumulator before hitting the part of the rule that fails to match. This can make a large difference to compile times.

Repetition Replacement

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}
```

This pattern is where a matched repetition sequence is simply discarded, with the variable being used to instead drive some repeated pattern that is related to the input only in terms of length.

For example, consider constructing a default instance of a tuple with more than 12 elements (the limit as of Rust 1.2).

```
macro_rules! tuple_default {
    ($($tup_tys:ty),*) => {
        (
            $(
                replace_expr!(
                    ($tup_tys)
                    Default::default()
                ),
            )*
        )
    };
}
```

JFTE: we *could* have simply used `$tup_tys::default()`.

Here, we are not actually *using* the matched types. Instead, we throw them away and replace them with a single, repeated expression. To put it another way, we don't care *what* the types are, only *how many* there are.

TT Bundling

```rust
macro_rules! call_a_or_b_on_tail {
    ((a: $a:ident, b: $b:ident), call a: $($tail:tt)*) => {
        $a(stringify!($($tail)*))
    };

    ((a: $a:ident, b: $b:ident), call b: $($tail:tt)*) => {
        $b(stringify!($($tail)*))
    };

    ($ab:tt, $_skip:tt $($tail:tt)*) => {
        call_a_or_b_on_tail!($ab, $($tail)*)
    };
}

fn compute_len(s: &str) -> Option<usize> {
    Some(s.len())
}

fn show_tail(s: &str) -> Option<usize> {
    println!("tail: {:?}", s);
    None
}

fn main() {
    assert_eq!(
        call_a_or_b_on_tail!(
            (a: compute_len, b: show_tail),
            the recursive part that skips over all these
            tokens does not much care whether we will call a
            or call b: only the terminal rules care.
        ),
        None
    );
    assert_eq!(
        call_a_or_b_on_tail!(
            (a: compute_len, b: show_tail),
            and now, to justify the existence of two paths
            we will also call a: its input should somehow
            be self-referential, so let us make it return
            some ninety-one!
        ),
        Some(87)
    );
}
```

In particularly complex recursive macros, a large number of arguments may be needed in order to carry identifiers and expressions to successive layers. However, depending on the implementation there may be many intermediate layers which need to forward these arguments, but do not need to *use* them.

As such, it can be very useful to bundle all such arguments together into a single TT by placing them in a group. This allows layers which do not need to use the arguments to simply capture and substitute a single `tt`, rather than having to exactly capture and substitute the entire argument group.

The example above bundles the `$a` and `$b` expressions into a group which can then be forwarded as a single `tt` by the recursive rule. This group is then destructured by the terminal rules to access the expressions.

Building Blocks

Reusable snippets of `macro_rules!` macro code.

AST Coercion

The Rust parser is not very robust in the face of `tt` substitutions. Problems can arise when the parser is expecting a particular grammar construct and *instead* finds a lump of substituted `tt` tokens. Rather than attempt to parse them, it will often just *give up*. In these cases, it is necessary to employ an AST coercion.

```
macro_rules! as_expr { ($e:expr) => {$e} }
macro_rules! as_item { ($i:item) => {$i} }
macro_rules! as_pat  { ($p:pat)  => {$p} }
macro_rules! as_stmt { ($s:stmt) => {$s} }
macro_rules! as_ty   { ($t:ty)   => {$t} }

as_item!{struct Dummy;}

fn main() {
    as_stmt!(let as_pat!(_): as_ty!(_) = as_expr!(42));
}
```

These coercions are often used with push-down accumulation macros in order to get the parser to treat the final `tt` sequence as a particular kind of grammar construct.

Note that this specific set of macros is determined by what macros are allowed to expand to, *not* what they are able to capture.

Counting

What follows are several techniques for counting in `macro_rules!` macros:

Note: If you are just interested in the most efficient way [look here](#)

Repetition with replacement

Counting things in a macro is a surprisingly tricky task. The simplest way is to use replacement with a repetition match.

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}

macro_rules! count_tts {
    ($($tts:tt)*) => {0usize $(+ replace_expr!($tts 1usize))*};
}
```

This is a fine approach for smallish numbers, but will likely *crash the compiler* with inputs of around 500 or so tokens. Consider that the output will look something like this:

```
0usize + 1usize + /* ~500 `+ 1usize`s */ + 1usize
```

The compiler must parse this into an AST, which will produce what is effectively a perfectly unbalanced binary tree 500+ levels deep.

Recursion

An older approach is to use recursion.

```
macro_rules! count_tts {
    () => {0usize};
    ($_head:tt $($tail:tt)*) => {1usize + count_tts!($($tail)*)};
}
```

Note: As of `rustc` 1.2, the compiler has *grievous* performance problems when large numbers of integer literals of unknown type must undergo inference. We are using explicitly `usize`-typed literals here to avoid that.

If this is not suitable (such as when the type must be substitutable), you can help matters by using `as` (*e.g.* `0 as $ty`, `1 as $ty`, *etc.*).

This *works*, but will trivially exceed the recursion limit. Unlike the repetition approach, you can extend the input size by matching multiple tokens at once.

```
macro_rules! count_tts {
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $_f:tt $_g:tt $_h:tt $_i:tt $_j:tt
     $_k:tt $_l:tt $_m:tt $_n:tt $_o:tt
     $_p:tt $_q:tt $_r:tt $_s:tt $_t:tt
     $($tail:tt)*)
        => {20usize + count_tts!($($tail)*)};
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $_f:tt $_g:tt $_h:tt $_i:tt $_j:tt
     $($tail:tt)*)
        => {10usize + count_tts!($($tail)*)};
    ($_a:tt $_b:tt $_c:tt $_d:tt $_e:tt
     $($tail:tt)*)
        => {5usize + count_tts!($($tail)*)};
    ($_a:tt
     $($tail:tt)*)
        => {1usize + count_tts!($($tail)*)};
    () => {0usize};
}

fn main() {
    assert_eq!(700, count_tts!(
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        // Repetition breaks somewhere after this
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,

        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
        ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,, ,,,,,,,,,,
    ));
}
```

This particular formulation will work up to ~1,200 tokens.

Slice length

A third approach is to help the compiler construct a shallow AST that won't lead to a stack overflow.
This can be done by constructing an array literal and calling the `len` method.

```
macro_rules! replace_expr {
    ($_t:tt $sub:expr) => {$sub};
}

macro_rules! count_tts {
    ($($tts:tt)*) => {<[()]>::len(&[$(replace_expr!($tts ())),*])};
}
```

This has been tested to work up to 10,000 tokens, and can probably go much higher.

Array length

Another modification of the previous approach is to use const generics stabilized in Rust 1.51. It's only slightly slower than slice length method on 20,000 tokens and works in const contexts.

```rust
const fn count_helper<const N: usize>(_: [(); N]) -> usize { N }

macro_rules! replace_expr {
    ($_t:tt $sub:expr) => { $sub }
}

macro_rules! count_tts {
    ($($smth:tt)*) => {
        count_helper([$(replace_expr!($smth ())),*])
    }
}
```

Enum counting

This approach can be used where you need to count a set of mutually distinct identifiers.

```rust
macro_rules! count_idents {
    () => {0};
    ($last_ident:ident, $($idents:ident),* $(,)?) => {
        {
            #[allow(dead_code, non_camel_case_types)]
            enum Idents { $($idents,)* $last_ident }
            const COUNT: u32 = Idents::$last_ident as u32 + 1;
            COUNT
        }
    };
}
```

This method does have two drawbacks. As implied above, it can *only* count valid identifiers (which are also not keywords), and it does not allow those identifiers to repeat.

Bit twiddling

Another recursive approach using bit operations:

```rust
macro_rules! count_tts {
    () => { 0 };
    ($odd:tt $($a:tt $b:tt)*) => { (count_tts!($($a)*) << 1) | 1 };
    ($($a:tt $even:tt)*) => { count_tts!($($a)*) << 1 };
}
```

This approach is pretty smart as it effectively halves its input whenever its even and then multiplying the counter by 2 (or in this case shifting 1 bit to the left which is equivalent). If the input is uneven it simply takes one token tree from the input `or` s the token tree to the previous counter which is

equivalent to adding 1 as the lowest bit has to be a 0 at this point due to the previous shifting. Rinse and repeat until we hit the base rule `() => 0`.

The benefit of this is that the constructed AST expression that makes up the counter value will grow with a complexity of `O(log(n))` instead of `O(n)` like the other approaches. Be aware that you can still hit the recursion limit with this if you try hard enough. Credits for this method go to Reddit user `YatoRust`.

Let's go through the procedure by hand once:

```
count_tts!(0 0 0 0 0 0 0 0 0 0);
```

This invocation will match the third rule due to the fact that we have an even number of token trees(10). The matcher names the odd token trees in the sequence `$a` and the even ones `$even` but the expansion only makes use of `$a`, which means it effectively discards all the even elements cutting the input in half. So the invocation now becomes:

```
count_tts!(0 0 0 0 0) << 1;
```

This invocation will now match the second rule as its input is an uneven amount of token trees. In this case the first token tree is discarded to make the input even again, then we also do the halving step in this invocation again since we know the input would be even now anyways. Therefore we can count 1 for the uneven discard and multiply by 2 again since we also halved.

```
((count_tts!(0 0) << 1) | 1) << 1;
```

```
((count_tts!(0) << 1 << 1) | 1) << 1;
```

```
((((count_tts!() << 1) | 1) << 1 << 1) | 1) << 1;
```

```
((((0 << 1) | 1) << 1 << 1) | 1) << 1;
```

Now to check if we expanded correctly manually we can use a one of the tools we introduced for `debugging`. When expanding the macro there we should get:

```
((((0 << 1) | 1) << 1 << 1) | 1) << 1;
```

That's the same so we didn't make any mistakes, great!

Abacus Counters

Provisional: needs a more compelling example. Matching nested groups that are *not* denoted by Rust groups is sufficiently unusual that it may not merit inclusion.

Note: this section assumes understanding of push-down accumulation and incremental TT munchers.

```rust
macro_rules! abacus {
    ((- $($moves:tt)*) -> (+ $($count:tt)*)) => {
        abacus!(($($moves)*) -> ($($count)*))
    };
    ((- $($moves:tt)*) -> ($($count:tt)*)) => {
        abacus!(($($moves)*) -> (- $($count)*))
    };
    ((+ $($moves:tt)*) -> (- $($count:tt)*)) => {
        abacus!(($($moves)*) -> ($($count)*))
    };
    ((+ $($moves:tt)*) -> ($($count:tt)*)) => {
        abacus!(($($moves)*) -> (+ $($count)*))
    };

    // Check if the final result is zero.
    (() -> ()) => { true };
    (() -> ($($count:tt)+)) => { false };
}

fn main() {
    let equals_zero = abacus!((++-+-+++--++---++----+) -> ());
    assert_eq!(equals_zero, true);
}
```

This technique can be used in cases where you need to keep track of a varying counter that starts at or near zero, and must support the following operations:

- Increment by one.
- Decrement by one.
- Compare to zero (or any other fixed, finite value).

A value of *n* is represented by *n* instances of a specific token stored in a group. Modifications are done using recursion and push-down accumulation. Assuming the token used is `x`, the operations above are implemented as follows:

- Increment by one: match `($($count:tt)*)`, substitute `(x $($count)*)`.
- Decrement by one: match `(x $($count:tt)*)`, substitute `($($count)*)`.
- Compare to zero: match `()`.
- Compare to one: match `(x)`.
- Compare to two: match `(x x)`.
- *(and so on...)*

In this way, operations on the counter are like flicking tokens back and forth like an abacus.[1]

In cases where you want to represent negative values, *-n* can be represented as *n* instances of a

different token. In the example given above, *+n* is stored as *n* `+` tokens, and *-m* is stored as *m* `-` tokens.

In this case, the operations become slightly more complicated; increment and decrement effectively reverse their usual meanings when the counter is negative. To which given `+` and `-` for the positive and negative tokens respectively, the operations change to:

- Increment by one:
  - match `()` , substitute `(+)` .
  - match `(- $($count:tt)*)` , substitute `($($count)*)` .
  - match `($($count:tt)+)` , substitute `(+ $($count)+)` .
- Decrement by one:
  - match `()` , substitute `(-)` .
  - match `(+ $($count:tt)*)` , substitute `($($count)*)` .
  - match `($($count:tt)+)` , substitute `(- $($count)+)` .
- Compare to 0: match `()` .
- Compare to +1: match `(+)` .
- Compare to -1: match `(-)` .
- Compare to +2: match `(++)` .
- Compare to -2: match `(--)` .
- *(and so on…)*

Note that the example at the top combines some of the rules together (for example, it combines increment on `()` and `($($count:tt)+)` into an increment on `($($count:tt)*)`).

If you want to extract the actual *value* of the counter, this can be done using a regular counter macro. For the example above, the terminal rules can be replaced with the following:

```
macro_rules! abacus {
    // ...

    // This extracts the counter as an integer expression.
    (() -> ()) => {0};
    (() -> (- $($count:tt)*)) => {
        - ( count_tts!($( $count_tts:tt )*) )
    };
    (() -> (+ $($count:tt)*)) => {
        count_tts!($( $count_tts:tt )*)
    };
}

// One of the many token tree counting macros in the counting chapter
macro_rules! count_tts {
    // ...
}
```

JFTE: strictly speaking, the above formulation of `abacus!` is needlessly complex. It can be implemented much more efficiently using repetition, provided you *do not* need to match against the counter's value in a macro:

```
macro_rules! abacus {
    (-) => {-1};
    (+) => {1};
    ($( $moves:tt )*) => {
        0 $(+ abacus!($moves))*
    }
}
```

---

1. This desperately thin reasoning conceals the *real* reason for this name: to avoid having *yet another* thing with "token" in the name. Talk to your writer about avoiding semantic satiation today!
   In fairness, it could *also* have been called "unary counting". ↵

Parsing Rust

Parsing some of Rust's items can be useful in certain situations. This section will show a few macros that can parse some of Rust's more complex items like structs and functions to a certain extent. The goal of these macros is not to be able to parse the entire grammar of the items but to parse parts that are in general quite useful without being too complex to parse. This means we ignore things like generics and such.

The main points of interest of these macros are their `matchers` . The transcribers are only there for example purposes and are usually not that impressive.

Function

```
macro_rules! function_item_matcher {
    (

        $( #[$meta:meta] )*
//      ^~~~attributes~~~~^
        $vis:vis fn $name:ident ( $( $arg_name:ident : $arg_ty:ty ),* $(,)? )
//                              ^~~~~~~~~~~~~~~argument list~~~~~~~~~~~~~~~^
        $( -> $ret_ty:ty )?
//      ^~~~return type~~~^
        { $($tt:tt)* }
//      ^~~~~body~~~~^
    ) => {
        $( #[$meta] )*
        $vis fn $name ( $( $arg_name : $arg_ty ),* ) $( -> $ret_ty )? { $($tt)* }
    }
}
```

A simple function matcher that ignores qualifiers like `unsafe`, `async`, ... as well as generics and where clauses. If parsing those is required it is likely that you are better off using a proc-macro instead.

This lets you for example, inspect the function signature, generate some extra things from it and then re-emit the entire function again. Kind of like a `Derive` proc-macro but weaker and for functions.

Ideally we would like to use a pattern fragment specifier instead of an ident for the arguments but this is currently not allowed. Fortunately people don't use non-identifier patterns in function signatures that often so this is okay(a shame, really).

Method

The macro for parsing basic functions is nice and all, but sometimes we would like to also parse methods, functions that refer to their object via some form of `self` usage. This makes things a bit trickier:

```rust
macro_rules! method_item_matcher {
    // self
    (
        $( #[$meta:meta] )*
    // ^~~~attributes~~~~^
        $vis:vis fn $name:ident ( $self:ident $(, $arg_name:ident : $arg_ty:ty )*
$(,)? )
    //                           ^~~~~~~~~~~~~~~~~~~~~argument
list~~~~~~~~~~~~~~~~~~~~~~^
        $( -> $ret_ty:ty )?
    //      ^~~~return type~~~^
        { $($tt:tt)* }
    //      ^~~~~body~~~~^
    ) => {
        $( #[$meta] )*
        $vis fn $name ( $self $(, $arg_name : $arg_ty )* ) $( -> $ret_ty )? { $($tt)*
}
    };

    // mut self
    (
        $( #[$meta:meta] )*
        $vis:vis fn $name:ident ( mut $self:ident $(, $arg_name:ident : $arg_ty:ty )*
$(,)? )
        $( -> $ret_ty:ty )?
        { $($tt:tt)* }
    ) => {
        $( #[$meta] )*
        $vis fn $name ( mut $self $(, $arg_name : $arg_ty )* ) $( -> $ret_ty )? {
$($tt)* }
    };

    // &self
    (
        $( #[$meta:meta] )*
        $vis:vis fn $name:ident ( & $self:ident $(, $arg_name:ident : $arg_ty:ty )*
$(,)? )
        $( -> $ret_ty:ty )?
        { $($tt:tt)* }
    ) => {
        $( #[$meta] )*
        $vis fn $name ( & $self $(, $arg_name : $arg_ty )* ) $( -> $ret_ty )? {
$($tt)* }
    };

    // &mut self
    (
        $( #[$meta:meta] )*
        $vis:vis fn $name:ident ( &mut $self:ident $(, $arg_name:ident : $arg_ty:ty )*
$(,)? )
        $( -> $ret_ty:ty )?
        { $($tt:tt)* }
    ) => {
        $( #[$meta] )*
        $vis fn $name ( &mut $self $(, $arg_name : $arg_ty )* ) $( -> $ret_ty )? {
$($tt)* }
    }
}
```

The four rules are identical except for the `self` receiver on both sides of the rule, which is `self`,
`mut self`, `&self`, and `&mut self`. You might not need all four rules.

`$self:ident` must be used in the matcher instead of a bare `self`. Without that, uses of `self` in

the body will cause compile errors, because a macro invocation can only access identifiers it receives from parameters.

Struct

```rust
macro_rules! struct_item_matcher {
    // Unit-Struct
    (
        $( #[$meta:meta] )*
//      ^~~~attributes~~~~^
        $vis:vis struct $name:ident;
    ) => {
        $( #[$meta] )*
        $vis struct $name;
    };

    // Tuple-Struct
    (
        $( #[$meta:meta] )*
//      ^~~~attributes~~~~^
        $vis:vis struct $name:ident (
            $(
                $( #[$field_meta:meta] )*
//              ^~~~field attributes~~~~^
                $field_vis:vis $field_ty:ty
//              ^~~~~~a single field~~~~~~^
            ),*
        $(,)? );
    ) => {
        $( #[$meta] )*
        $vis struct $name (
            $(
                $( #[$field_meta] )*
                $field_vis $field_ty
            ),*
        );
    };

    // Named-Struct
    (
        $( #[$meta:meta] )*
//      ^~~~attributes~~~~^
        $vis:vis struct $name:ident {
            $(
                $( #[$field_meta:meta] )*
//              ^~~~field attributes~~~!^
                $field_vis:vis $field_name:ident : $field_ty:ty
//              ^~~~~~~~~~~~~~~~~a single field~~~~~~~~~~~~~~~~~^
            ),*
        $(,)? }
    ) => {
        $( #[$meta] )*
        $vis struct $name {
            $(
                $( #[$field_meta] )*
                $field_vis $field_name : $field_ty
            ),*
        }
    }
}
```

Enum

Parsing enums is a bit more complex than structs so we will finally make use of some of the patterns we have discussed, Incremental TT Muncher and Internal Rules. Instead of just building the parsed enum again we will merely visit all the tokens of the enum, as rebuilding the enum would require us to collect all the parsed tokens temporarily again via a Push Down Accumulator.

```
macro_rules! enum_item_matcher {
    // tuple variant
    (@variant $variant:ident (
        $(
            $( #[$field_meta:meta] )*
    //      ^~~~field attributes~~~~^
            $field_vis:vis $field_ty:ty
    //      ^~~~~~a single field~~~~~~^
        ),* $(,)?
    //V~~rest of input~~V
    ) $(, $($tt:tt)* )? ) => {

        // process rest of the enum
        $( enum_item_matcher!(@variant $( $tt )*) )?
    };
    // named variant
    (@variant $variant:ident {
        $(
            $( #[$field_meta:meta] )*
    //      ^~~~field attributes~~~!^
            $field_vis:vis $field_name:ident : $field_ty:ty
    //      ^~~~~~~~~~~~~~~~~~a single field~~~~~~~~~~~~~~~~^
        ),* $(,)?
    //V~~rest of input~~V
    } $(, $($tt:tt)* )? ) => {
        // process rest of the enum
        $( enum_item_matcher!(@variant $( $tt )*) )?
    };
    // unit variant
    (@variant $variant:ident $(, $($tt:tt)* )? ) => {
        // process rest of the enum
        $( enum_item_matcher!(@variant $( $tt )*) )?
    };
    // trailing comma
    (@variant ,) => {};
    // base case
    (@variant) => {};
    // entry point
    (
        $( #[$meta:meta] )*
        $vis:vis enum $name:ident {
            $($tt:tt)*
        }
    ) => {
        enum_item_matcher!(@variant $($tt)*)
    };
}
```

Macros 2.0

RFC: rfcs#1584
Tracking Issue: rust#39412
Feature: `#![feature(decl_macro)]`

While not yet stable(or rather far from being finished), there is proposal for a new declarative macro system that is supposed to replace `macro_rules!` dubbed declarative macros 2.0, `macro`, `decl_macro` or confusingly also `macros-by-example`.

This chapter is only meant to quickly glance over the current state, showing how to use this macro system and where it differs. Nothing described here is final or complete, and may be subject to change.

Syntax

We'll do a comparison between the `macro` and `macro_rules` syntax for two macros we have implemented in previous chapters:

```
#![feature(decl_macro)]

macro_rules! replace_expr_ {
    ($_t:tt $sub:expr) => { $sub }
}
macro replace_expr($_t:tt $sub:expr) {
    $sub
}

macro_rules! count_tts_ {
    () => { 0 };
    ($odd:tt $($a:tt $b:tt)*) => { (count_tts!($($a)*) << 1) | 1 };
    ($($a:tt $even:tt)*) => { count_tts!($($a)*) << 1 };
}
macro count_tts {
    () => { 0 },
    ($odd:tt $($a:tt $b:tt)*) => { (count_tts!($($a)*) << 1) | 1 },
    ($($a:tt $even:tt)*) => { count_tts!($($a)*) << 1 },
}
```

As can be seen, they look very similar, with just a few differences as well as that `macro`s have two different forms.

Let's inspect the `count_tts` macro first, as that one looks more like what we are used to. As can be seen, it practically looks identical to the `macro_rules` version with two exceptions, it uses the `macro` keyword and the rule separator is a `,` instead of a `;`.

There is a second form to this though, which is a shorthand for macros that only have one rule. Taking a look at `replace_expr` we can see that in this case we can write the definition in a way that more resembles an ordinary function. We can write the matcher directly after the name followed by the transcriber, dropping a pair of braces and the `=>` token.

Syntax for invoking `macro`s is the same as for `macro_rules` and function-like procedural macros,

the name followed by a `!` followed by the macro input token tree.

`macro` **are proper items**

Unlike with `macro_rules` macros, which are textually scoped and require `#[macro_export]` (and potentially a re-export) to be treated as an item, `macro` macros behave like proper rust items by default.

As such, you can properly qualify them with visibility specifiers like `pub`, `pub(crate)`, `pub(in path)` and the like.

Hygiene

Hygiene is by far the biggest difference between the two declarative macro systems. Unlike `macro_rules` which have mixed site hygiene, `macro` have definition site hygiene, meaning they do not leak identifiers outside of their invocation.

As such the following compiles with a `macro_rules` macro, but fails with a `macro` definition:

```
#![feature(decl_macro)]
// try uncommenting the following line, and commenting out the line right after

macro_rules! foo {
// macro foo {
    ($name: ident) => {
        pub struct $name;

        impl $name {
            pub fn new() -> $name {
                $name
            }
        }
    }
}

foo!(Foo);

fn main() {
    // this fails with a `macro`, but succeeds with a `macro_rules`
    let foo = Foo::new();
}
```

There may be plans to allow escaping hygiene for identifiers(hygiene bending) in the future.

Procedural Macros

Note: This section is still very incomplete!

This chapter will introduce Rust's second syntax extension type, *procedural macros*.

As with the declarative macros chapter, this one is also split into a methodical and a (WIP) practical subchapter with the former being a more formal introduction and the latter being a more practical oriented one.

A lot of the basic information covered has been sourced from the rust reference, as most knowledge about procedural macros is currently located there.

A Methodical Introduction

This chapter will introduce Rust's procedural macro system by explaining the system as a whole.

Unlike a declarative macro, a procedural macro takes the form of a Rust function taking in a token stream(or two) and outputting a token stream.

A proc-macro is at its core just a function exported from a crate with the `proc-macro` crate type, so when writing multiple proc macros you can have them all live in one crate.

Note: When using Cargo, to define a `proc-macro` crate you define and set the `lib.proc-macro` key in the `Cargo.toml` to true.

```
[lib]
proc-macro = true
```

A `proc-macro` type crate implicitly links to the compiler-provided proc_macro crate, which contains all the things you need to get going with developing procedural macros. The two most important types exposed by the crate are the `TokenStream`, which are the proc-macro variant of the already familiar token trees as well as the `Span`, which describes a part of source code used primarily for error reporting and hygiene. See the Hygiene and Spans chapter for more information.

As proc-macros therefore are functions living in a crate, they can be addressed as all the other items in a Rust project. All that is required is to add the crate to the dependency graph of a project and bring the desired item into scope.

Note: Procedural macros invocations still run at the same stage in the compiler expansion-wise as declarative macros, just that they are standalone Rust programs that the compiler compiles, runs, and finally either replaces or appends to.

Types of procedural macros

With procedural macros, there actually exists 3 different kinds with each having slightly different properties.

- *function-like* proc-macros which are used to implement `$name ! $arg` invocable macros
- *attribute* proc-macros which are used to implement `#[$arg]` attributes
- *derive* proc-macros which are used to implement a derive, an *input* to a `#[derive(…)]` attribute

At their core, all 3 work almost the same with a few differences in their inputs and output reflected by their function definition. As mentioned all a procedural macro really is, is a function that maps a token stream so let's take a quick look at each basic definition and their differences.

function-like

```
#[proc_macro]
pub fn my_proc_macro(input: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

attribute

```
#[proc_macro_attribute]
pub fn my_attribute(input: TokenStream, annotated_item: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

derive

```
#[proc_macro_derive(MyDerive)]
pub fn my_derive(annotated_item: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

As shown, the basic structure is the same for each, a public function marked with an attribute defining its procedural macro type returning a `TokenStream` . Note how the return type is a `TokenStream` and not a result or something else that gives the notion of being fallible. This does not mean that proc-macros cannot fail though, in fact they have two ways of reporting errors, the first one being to panic and the second to emit a `compile_error!` invocation. If a proc-macro panics the compiler will catch it and emit the payload as an error coming from the macro invocation.

Beware: The compiler will happily hang on endless loops spun up inside proc-macros causing the compilation of crates using the proc-macro to hang as well.

Function-like

Function-like procedural macros are invoked like declarative macros that is `makro!(…)`.

This type of macro is the simplest of the three though. It is also the only one which you can't differentiate from declarative macros when solely looking at the invocation.

A simple skeleton of a function-like procedural macro looks like the following:

```
use proc_macro::TokenStream;

#[proc_macro]
pub fn tlborm_fn_macro(input: TokenStream) -> TokenStream {
    input
}
```

As one can see this is in fact just a mapping from one `TokenStream` to another where the `input` will be the tokens inside of the invocation delimiters, e.g. for an example invocation `foo!(bar)` the input token stream would consist of the `bar` token. The returned token stream will **replace** the macro invocation.

For this macro type the same placement and expansion rules apply as for declarative macros, that is the macro must output a correct token stream for the invocation location. Unlike with declarative macros though, function-like procedural macros do not have certain restrictions imposed on their inputs though. That is the restrictions for what may follow fragment specifiers listed in the Metavariables and Expansion Redux chapter listed is not applicable here, as the procedural macros work on the tokens directly instead of matching them against fragment specifiers or similar.

With that said it is apparent that the procedural counter part to these macros is more powerful as they can arbitrarily modify their input, and produce any output desired as long as its within the bounds of the language syntax.

Usage example:

```
use tlborm_proc::tlborm_attribute;

fn foo() {
    tlborm_attribute!(be quick; time is mana);
}
```

Attribute

Attribute procedural macros define new *outer* attributes which can be attached to items. This type can be invoked with the `#[attr]` or `#[attr(…)]` syntax where `…` is an arbitrary token tree.

A simple skeleton of an attribute procedural macro looks like the following:

```
use proc_macro::TokenStream;

#[proc_macro_attribute]
pub fn tlborm_attribute(input: TokenStream, annotated_item: TokenStream) ->
TokenStream {
    annotated_item
}
```

Of note here is that unlike the other two procedural macro kinds, this one has two input parameters instead of one.

- The first parameter is the delimited token tree following the attribute's name, excluding the delimiters around it. It is empty if the attribute is written bare, that is just a name without a `(TokenTree)` following it, e.g. `#[attr]`.
- The second token stream is the item the attribute is attached to *without* the attribute this proc macro defines. As this is an `active` attribute, the attribute will be stripped from the item before it is being passed to the proc macro.

The returned token stream will **replace** the annotated item fully. Note that the replacement does not have to be a single item, it can be 0 or more.

Usage example:

```
use tlborm_proc::tlborm_attribute;

#[tlborm_attribute]
fn foo() {}

#[tlborm_attribute(attributes are pretty handsome)]
fn bar() {}
```

Derive

Derive procedural macros define new inputs for the `derive` attribute. This type can be invoked by feeding it to a derive attribute's input, e.g. `#[derive(TlbormDerive)]` .

A simple skeleton of a derive procedural macro looks like the following:

```rust
use proc_macro::TokenStream;

#[proc_macro_derive(TlbormDerive)]
pub fn tlborm_derive(item: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

The `proc_macro_derive` is a bit more special in that it requires an extra identifier, this identifier will become the actual name of the derive proc macro. The input token stream is the item the derive attribute is attached to, that is, it will always be an `enum` , `struct` or `union` as these are the only items a derive attribute can annotate. The returned token stream will be **appended** to the containing block or module of the annotated item with the requirement that the token stream consists of a set of valid items.

Usage example:

```rust
use tlborm_proc::TlbormDerive;

#[derive(TlbormDerive)]
struct Foo;
```

Helper Attributes

Derive proc macros are a bit more special in that they can add additional attributes visible only in the scope of the item definition. These attributes are called *derive macro helper attributes* and are inert. Their purpose is to give derive proc macros additional customizability on a per field or variant basis, that is these attributes can be used to annotate fields or enum variants while having no effect on their own. As they are `inert` they will not be stripped and are visible to all macros.

They can be defined by adding an `attributes(helper0, helper1, ..)` argument to the `proc_macro_derive` attribute containing a comma separated list of identifiers which are the names of the helper attributes.

Thus a simple skeleton of a derive procedural macro with helper attributes looks like the following:

```rust
use proc_macro::TokenStream;

#[proc_macro_derive(TlbormDerive, attributes(tlborm_helper))]
pub fn tlborm_derive(item: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

That is all there is to helper attributes, to consume them in the proc macro the implementation will then have to check the attributes of fields and variants to see whether they are attributed with the corresponding helper. It is an error to use a helper attribute if none of the used derive macros of the given item declare it as such, as the compiler will then instead try to resolve it as a normal attribute.

Usage example:

```
use tlborm_proc::TlbormDerive;

#[derive(TlbormDerive)]
struct Foo {
    #[tlborm_helper]
    field: u32
}

#[derive(TlbormDerive)]
enum Bar {
    #[tlborm_helper]
    Variant { #[tlborm_helper] field: u32 }
}
```

Third-Party Crates

Note: Crates beyond the automatically linked `proc_macro` crate are not required to write procedural macros. The crates listed here merely make writing them simpler and more concise, while potentially adding to the compilation time of the procedural macro due to added dependencies.

As procedural macros live in a crate they can naturally depend on (crates.io) crates. turns out the crate ecosystem has some really helpful crates tailored towards procedural macros that this chapter will quickly go over, most of which will be used in the following chapters to implement the example macros. As these are merely quick introductions it is advised to look at each crate's documentation for more in-depth information if required.

`proc-macro2`

`proc-macro2`, the successor of the `proc_macro` crate! Or so you might think but that is of course not correct, the name might be a bit misleading. This crate is actually just a wrapper around the `proc_macro` crate serving two specific purposes, taken from the documentation:

- Bring proc-macro-like functionality to other contexts like build.rs and main.rs.
- Make procedural macros unit testable.

As the `proc_macro` crate is exclusive to `proc_macro` type crates, making them unit testable or accessing them from non-proc macro code is next to impossible. With that in mind the `proc-macro2` crate mimics the original `proc_macro` crate's api, acting as a wrapper in proc-macro crates and standing on its own in non-proc-macro crates. Hence it is advised to build libraries targeting proc-macro code to be built against `proc-macro2` instead as that will enable those libraries to be unit testable, which is also the reason why the following listed crates take and emit `proc-macro2::TokenStream`s instead. When a `proc_macro` token stream is required, one can simply `.into()` the `proc-macro2` token stream to get the `proc_macro` version and vice-versa.

Procedural macros using the `proc-macro2` crate will usually import the `proc-macro2::TokenStream` in an aliased form like `use proc-macro2::TokenStream as TokenStream2`.

`quote`

The `quote` crate mainly exposes just one macro, the `quote!` macro.

This little macro allows you to easily create token streams by writing the actual source out as syntax while also giving you the power of interpolating tokens right into the written syntax. Interpolation can be done by using the `#local` syntax where local refers to a local in the current scope. Likewise `#( #local )*` can be used to interpolate over an iterator of types that implement `ToTokens`, this works similar to declarative `macro_rules!` repetitions in that they allow a separator as well as extra tokens inside the repetition.

```
let name = /* some identifier */;
let exprs = /* an iterator over expressions tokenstreams */;
let expanded = quote! {
    impl SomeTrait for #name { // #name interpolates the name local from above
        fn some_function(&self) -> usize {
            #( #exprs )* // #name interpolates exprs by iterating the iterator
        }
    }
};
```

This a very useful tool when preparing macro output avoiding the need of creating a token stream by inserting tokens one by one.

Note: As stated earlier, this crate makes use of `proc_macro2` and thus the `quote!` macro returns a `proc-macro2::TokenStream`.

---

syn

The `syn` crate is a parsing library for parsing a stream of Rust tokens into a syntax tree of Rust source code. It is a very powerful library that makes parsing proc-macro input quite a bit easier, as the `proc_macro` crate itself does not expose any kind of parsing capabilities, merely the tokens. As the library can be a heavy compilation dependency, it makes heavy use of feature gates to allow users to cut it as small as required.

So what does it offer? A bunch of things.

First of all it has definitions and parsing for all standard Rust syntax nodes(when the `full` feature is enabled), as well as a `DeriveInput` type which encapsulates all the information a derive macro gets passed as an input stream as a structured input(requires the `derive` feature, enabled by default). These can be used right out of the box with the `parse_macro_input!` macro(requires the `parsing` and `proc-macro` features, enabled by default) to parse token streams into these types.

If Rust syntax doesn't cut it, and instead one wishes to parse custom non-Rust syntax the crate also offers a generic parsing API, mainly in the form of the `Parse` trait(requires the `parsing` feature, enabled by default).

Aside from this the types exposed by the library keep location information and spans which allows procedural macros to emit detailed error messages pointing at the macro input at the points of interest.

As this is again a library for procedural macros, it makes use of the `proc_macro2` token streams and spans and as such, conversions may be required.

Hygiene and Spans

This chapter talks about procedural macro hygiene and the type that encodes it, `Span`.

Every token in a `TokenStream` has an associated `Span` holding some additional info. A span, as its documentation states, is `A region of source code, along with macro expansion information`. It points into a region of the original source code(important for displaying diagnostics at the correct places) as well as holding the kind of *hygiene* for this location. The hygiene is relevant mainly for identifiers, as it allows or forbids the identifier from referencing things or being referenced by things defined outside of the invocation.

There are 3 kinds of hygiene(which can be seen by the constructors of the `Span` type):

- `definition site` (***unstable***): A span that resolves at the macro definition site. Identifiers with this span will not be able to reference things defined outside or be referenced by things outside of the invocation. This is what one would call "hygienic".
- `mixed site`: A span that has the same hygiene as `macro_rules` declarative macros, that is it may resolve to definition site or call site depending on the type of identifier. See here for more information.
- `call site`: A span that resolves to the invocation site. Identifiers in this case will behave as if written directly at the call site, that is they freely resolve to things defined outside of the invocation and can be referenced from the outside as well. This is what one would call "unhygienic".

Glossary

A place for obscure words and their descriptions. If you feel like there is an important word missing here, please open an issue or a pull request.

Function-like macro

A function like macro describes a syntax extension that can be invoked via the form `identifier! (...)`. It is called this way due to its resemblance of a function call.

Syntax Extension

The mechanism Rust's `macro_rules!` and procedural macros are built on.