The Little Book of Rust Macros

Daniel Keep

2015-10-14

Contents

1 Introduction **5**

Thanks 5

License 5

2 Macros, A Methodical Introduction **7**

Syntax Extensions 7

Source Analysis 7

Macros in the AST 10

Expansion 12

macro_rules! 14

Minutiae 17

Captures and Expansion Redux 17

Hygiene 21

Non-Identifier Identifiers 21

Debugging 25

Scoping 27

Import/Export 30

3 Macros, A Practical Introduction **33**

4 Patterns **59**

Callbacks 59

Incremental TT Munchers 60

Internal Rules 61

Push-Down Accumulation 62

Repetition Replacement 63

Trailing Separators 64

TT Bundling 64
Provisional 66

5 Building Blocks **69**
AST Coercion 69
Counting 69
Enum Parsing 72

6 Annotated Examples **75**
Ook! 75

1 Introduction

Note: this is a work in progress.

This book is an attempt to distil the Rust community's collective knowledge of Rust macros. As such, both additions (in the form of pull requests) and requests (in the form of issues) are welcome.

If you wish to contribute, see the GitHub repository[1].

Thanks

Thanks to the following for suggestions and corrections: IcyFoxy, Rym, TheMicroWorm, Yurume, akavel, cmr, ogham, and snake_case.

License

This work is licensed under both the Creative Commons Attribution-ShareAlike 4.0 International License[2] and the MIT license[3].

[1] https://github.com/DanielKeep/tlborm/
[2] http://creativecommons.org/licenses/by-sa/4.0/
[3] http://opensource.org/licenses/MIT

2

Macros, A Methodical Introduction

This chapter will introduce Rust's Macro-By-Example system: `macro_rules!`. Rather than trying to cover it based on practical examples, it will instead attempt to give you a complete and thorough explanation of *how* the system works. As such, this is intended for people who just want the system as a whole explained, rather than be guided through it.

There is also the Macros chapter of the Rust Book[1] which is a more approachable, high-level explanation, and the practical introduction[2] chapter of this book, which is a guided implementation of a single macro.

Syntax Extensions

Before talking about *macros*, it is worthwhile to discuss the general mechanism they are built on: *syntax extensions*. To do *that*, we must discuss how Rust source is processed by the compiler, and the general mechanisms on which user-defined macros are built.

Source Analysis

The first stage of compilation for a Rust program is tokenisation. 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`, …
- Integers: `42`, `72u32`, `0_______0`, …
- Keywords: `_`, `fn`, `self`, `match`, `yield`, `macro`, …
- Lifetimes: `'a`, `'b`, `'a_rare_long_lifetime_name`, …

[1] http://doc.rust-lang.org/book/macros.html
[2] https://danielkeep.github.io/practical-intro-to-macros.html

- Strings: `""`, `"Leicester"`, `r##"venezuelan beaver"##`, …
- Symbols: `[`, `:`, `::`, `->`, `@`, `<-`, …

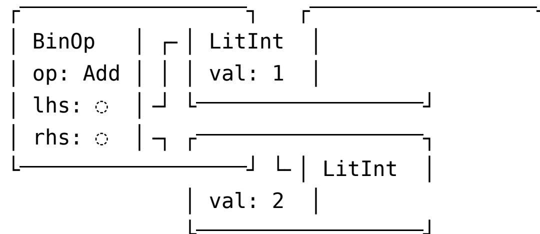
…among others. There are some things to note about the above: first, `self` is both an identifier *and* a keyword. In almost all cases, `self` is a keyword, but it *is* possible for it to be *treated* as an identifier, which will come up later (along with much cursing). Secondly, the list of keywords 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. [3]

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.[4] This is why the following code works:[5]

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

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

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.

[3] `@` 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. Even a member of the Rust core team, proof-reading this chapter, who *specifically* brought up this section, didn't remember that `@` has a purpose. Poor, poor swirly.

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

[5] *Whether* it should work is an entirely *different* question.

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 give 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:

```
«a» «+» «b» «+» «(   )» «+» «e»
          ┌────────┴──────────┐
           «c» «+» «d» «[   ]»
                        ┌─┴─┐
                         «0»
```

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

```
               ┌─────────┐
               │ BinOp   │
               │ op: Add │
             ┌╴│ lhs: ◌  │
┌─────────┐  │ │ rhs: ◌  │╶┐ ┌─────────┐
│ Var     │╶┘ └─────────┘ └╴│ BinOp   │
│ name: a │                 │ op: Add │
└─────────┘               ┌╴│ lhs: ◌  │
              ┌─────────┐ │ │ rhs: ◌  │╶┐ ┌─────────┐
              │ Var     │╶┘ └─────────┘ └╴│ BinOp   │
              │ name: b │                 │ op: Add │
              └─────────┘               ┌╴│ lhs: ◌  │
                            ┌─────────┐ │ │ rhs: ◌  │╶┐ ┌─────────┐
                            │ BinOp   │╶┘ └─────────┘ └╴│ Var     │
                            │ op: Add │                 │ name: e │
                          ┌╴│ lhs: ◌  │                 └─────────┘
              ┌─────────┐ │ │ rhs: ◌  │╶┐ ┌─────────┐
              │ Var     │╶┘ └─────────┘ └╴│ Index   │
              │ name: c │               ┌╴│ arr: ◌  │
              └─────────┘   ┌─────────┐ │ │ ind: ◌  │╶┐ ┌─────────┐
                            │ Var     │╶┘ └─────────┘ └╴│ LitInt  │
                            │ name: d │                 │ val: 0  │
                            └─────────┘                 └─────────┘
```

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 paren, bracket or brace; nor is it possible to have incorrectly nested groups in a token tree.

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 forms (by way of examples):

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

The first two are "attributes", and are shared between both language-specific constructs (such as `#[repr(C)]` which is used to request a C-compatible ABI for user-defined types) and syntax extensions (such as `#[derive(Clone)]`). There is currently no way to define a macro that uses these forms.

The third is the one of interest to us: it is the form available for use with macros. Note that this form is not *limited* to macros: it is a generic syntax extension form. For example, whilst `format!` is a macro, `format_args!` (which is used to *implement* `format!`) is *not*.

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

Disregarding all but the third form (`$name ! $arg`), the question becomes: how does the Rust parser know what `$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! {
    flags 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() {
    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.

The important takeaways are:

- There are multiple kinds of syntax extension in Rust. We will *only* be talking about macros defined by the `macro_rules!` construct.
- Just because you see something of the form `$name! $arg`, doesn't mean it's actually a macro; it might be another kind of syntax extension.
- The input to every macro is a single non-leaf token tree.
- Macros (really, syntax extensions in general) are parsed as *part* of the abstract syntax tree.

> **Aside**: due to the first point, some of what will be said below (including the next paragraph) will apply to syntax extensions *in general*.[6]

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

- Patterns

[6]This is rather convenient as "macro" is much quicker and easier to type than "syntax extension".

- Statements
- Expressions
- Items
- `impl` Items

Some things *not* on this list:

- Identifiers
- Match arms
- Struct fields
- Types

There is absolutely, definitely *no way* to use macros 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 macros.

This involves traversing the AST, locating macro invocations and replacing them with their expansion. In the case of non-macro syntax extensions, *how* this happens is up to the particular syntax extension. That said, syntax extensions go through *exactly* the same process that macros do once their expansion is complete.

Once the compiler has run a syntax extension, it expects the result to be parseable as one of a limited set of syntax elements, based on context. For example, if you invoke a macro at module scope, the compiler will parse the result into an AST node that represents an item. If you invoke a macro 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,
- zero or more items,
- zero or more `impl` items, or
- zero or more statements.

In other words, *where* you can invoke a macro determines what its result will be interpreted as.

The compiler will take this AST node and completely replace the macro's invocation node with the output node. *This is a structural operation*, not a textural one!

For example, consider the following:

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

We can visualise this partial AST as follows:

```
┌───────────────────────────┐
│ Let         │
│ name: eight │   ┌──────────────────┐
│ init: ◌     │──│ BinOp    │
└───────────────────────────┘   │ op: Mul │
               ┌│ lhs: ◌   │
     ┌────────────────┐ │ │ rhs: ◌  │┐ ┌────────────────────────┐
     │ LitInt │┘ └──────────────────┘ └│ Macro       │
     │ val: 2 │                │ name: four │
     └────────────────┘                │ body: ()   │
                               └────────────────────────┘
```

From context, `four!()` *must* expand to an expression (the initialiser 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: ◌  │┐ ┌──────────────────┐
     │ LitInt │┘ └──────────────────┘ └│ BinOp    │
     │ val: 2 │                │ op: Add │
     └────────────────┘                ┌│ lhs: ◌   │
                   ┌────────────────┐ │ │ rhs: ◌  │┐ ┌────────────────┐
                   │ LitInt │┘ └──────────────────┘ └│ LitInt │
                   │ val: 1 │                │ val: 3 │
                   └────────────────┘                └────────────────┘
```

This can be written out like so:

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

Note that we added parens *despite* them not being in the expansion. Remember that the compiler always treats the expansion of a macro 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 macro expansions are treated as AST nodes, as this design has two further implications:

- In addition to there being a limited number of invocation *positions*, macros can *only* expand to the kind of AST node the parser *expects* at that position.
- As a consequence of the above, macros *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 macro 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 macro recursion limit and defaults to 32. If the 32nd expansion contains a macro 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 macros below this limit wherever possible.

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 form:

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

There must be *at least* one rule, and you can omit the semicolon after the last rule.

Each "`rule`" looks like so:

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

Actually, the parens and braces can be any kind of group, but parens around the pattern and braces around the expansion are somewhat conventional.

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 macro. As such, you can *technically* use `macro_rules!` in any position where an empty expansion is valid.

Matching

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

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

The simplest example is of an empty pattern:

```
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 macro *are not* matched. 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.

Patterns 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 use:

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

You can use any token tree that you can write.

Captures

Patterns can also contain captures. These allow input to be matched based on some general grammar category, with the result captured to a variable 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 must be one of the following:

- `item`: an item, like a function, struct, module, etc.
- `block`: a block (i.e. a block of statments and/or an expression, surrounded by braces)
- `stmt`: a statement
- `pat`: a pattern
- `expr`: an expression
- `ty`: a type
- `ident`: an identifier
- `path`: a path (e.g. `foo`, `::std::mem::replace`, `transmute::<_, int>`, …)
- `meta`: a meta item; the things that go inside `#[...]` and `#![...]` attributes
- `tt`: a single token tree

For example, here is a macro which captures its input as an expression:

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

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

You can mix literal token trees and captures, within limits (explained below).

A capture `$name:kind` can be substituted into the expansion by writing `$name`. For example:

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

Much like macro expansion, captures 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 captures in a single pattern:

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

Repetitions

Patterns 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 pattern being repeated.
- `sep` is an *optional* separator token. Common examples are `,`, and `;`.
- `rep` is the *required* repeat control. Currently, this can be *either* `*` (indicating zero or more repeats) or `+` (indicating one or more repeats). You cannot write "zero or one" or any other more specific counts or ranges.

Repetitions can contain any other valid pattern, including literal token trees, captures, and other repetitions.

Repetitions use the same syntax in the expansion.

For example, below is a 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.

```
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
        }
    };
}
```

Minutiae

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

Captures and Expansion Redux

Once the parser begins consuming tokens for a capture, *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 captures. The complete list, as of Rust 1.3 is as follows:

- `item`: anything.
- `block`: anything.
- `stmt`: `=> , ;`
- `pat`: `=> , = if in`
- `expr`: `=> , ;`
- `ty`: `, => : = > ; as`
- `ident`: anything.
- `path`: `, => : = > ; as`
- `meta`: anything.
- `tt`: anything.

Additionally, `macro_rules!` generally forbids a repetition to be followed by another repetition, even if the contents do not conflict.

One aspect of substitution that often surprises people is that substitution is *not* token-based, despite very much *looking* like it. Here is a simple demonstration:

```
macro_rules! capture_expr_then_stringify {
    ($e:expr) => {
        stringify!($e)
    };
}

fn main() {
    println!("{:?}", stringify!(dummy(2 * (1 + (3)))));
    println!("{:?}", capture_expr_then_stringify!(dummy(2 * (1 + (3)))));
}
```

Note that `stringify!` is a built-in syntax extension which simply takes all tokens it is given and concatenates them into one big string.

The output when run is:

```
"dummy ( 2 * ( 1 + ( 3 ) ) )"
"dummy(2 * (1 + (3)))"
```

Note that *despite* having the same input, the output is different. This is because the first invocation is stringifying a sequence of token trees, whereas the second is stringifying *an AST expression node*.

To visualise the difference another way, here is what the `stringify!` macro gets invoked with in the first case:

```
«dummy» «(   )»
   ┌───────────┴───────────┐
    «2» «*» «(   )»
       ┌───────────┴───────────┐
        «1» «+» «(   )»
                ┌─┴─┐
                 «3»
```

...and here is what it gets invoked with in the second case:

```
« »
 │ ┌─────────────────────┐
 └ │ Call          │
   │ fn: dummy     │   ┌───────────────┐
   │ args: ◌       │ ─── │ BinOp     │
   └─────────────────────┘    │ op: Mul  │
                  ┌ │ lhs: ◌   │
       ┌─────────────┐ │ │ rhs: ◌  │┐ ┌───────────────┐
       │ LitInt │ ┘ └───────────────┘ └ │ BinOp     │
       │ val: 2 │                 │ op: Add │
       └─────────────┘                  ┌ │ lhs: ◌   │
                  ┌─────────────┐ │ │ rhs: ◌   │┐ ┌─────────────┐
                  │ LitInt │ ┘ └───────────────┘ └ │ LitInt │
                  │ val: 1 │              │ val: 3 │
                  └─────────────┘                └─────────────┘
```

As you can see, there is exactly *one* token tree, which contains the AST which was parsed from the input to the `capture_expr_then_stringify!` invocation. Hence, what you see in the output is not the stringified tokens, it's the stringified *AST node*.

This has further implications. 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` or `ident` 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.

Hygiene

Macros in Rust are *partially* hygienic. Specifically, they are hygienic when it comes to most identifiers, but *not* when it comes to generic type parameters or lifetimes.

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

To illustrate this, consider the following code:

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

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:

```
<anon>:11:21: 11:22 error: unresolved name `a`
<anon>:11 let four = using_a!(a / 10);
```

Note that the background colour (*i.e.* syntax context) for the expanded macro *changes* as part of expansion. Each macro expansion is given a new, unique syntax context for its contents. As a result, there are *two different* `as` 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:

Which, upon expansion becomes:

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

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 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
    }
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

This fails to compile with:

```
<anon>:2:28: 2:30 error: expected identifier, found keyword `self`
<anon>:2     ($i:ident) => {let mut $i = $i;};
                                    ^~
```

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?

```
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
    }
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

This fails with:

```
<anon>:2:33: 2:37 error: `self` is not available in a static method. Maybe a `self` ar
↳ gument is missing? [E0424]
<anon>:2     ($i:ident) => {let mut $i = self;};
                                         ^~~~
```

That doesn't make any sense, either. It's *not* in a static method. 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.

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

struct Dummy(i32);

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

```
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

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}
}
#
# fn main() {
#     println!("{:?}", Dummy(4).double().0);
# }
```

```
<anon>:12:21: 12:22 error: expected ident, found _
<anon>:12     double_method! {_, 0}
                              ^
```

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

`rustc` provides a number of tools to debug macros. One of the most useful is `trace_macros!`, which is a directive to the compiler instructing it to dump every 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:

```
each_tt! { spim wak plee whum }
each_tt! { wak plee whum }
each_tt! { plee whum }
each_tt! { whum }
each_tt! {  }
```

This is *particularly* invaluable when debugging deeply recursive 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!`.

Sometimes, it is what the macro *expands to* that proves problematic. For this, the `--pretty` argument to the compiler can be used. Given the following code:

```
// Shorthand for initialising 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 -Z unstable-options --pretty expanded hello.rs
```

produces the following output (modified for formatting):

```
#![feature(no_std, prelude_import)]
#![no_std]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std as std;
// Shorthand for initialising a `String`.
fn main() {
    let world = String::from("World");
    ::std::io::_print(::std::fmt::Arguments::new_v1(
        {
            static __STATIC_FMTSTR: &'static [&'static str]
                = &["Hello, ", "!\n"];
            __STATIC_FMTSTR
        },
        &match (&world,) {
             (__arg0,) => [
```

```
                ::std::fmt::ArgumentV1::new(__arg0, ::std::fmt::Display::fmt)
            ],
        }
    ));
}
```

Other options to `--pretty` can be listed using `rustc -Z unstable-options --help -v`; a full list is not provided since, as implied by the name, any such list would be subject to change at any time.

Scoping

The way in which macros are scoped can be somewhat unintuitive. Firstly, unlike everything else in the languages, 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 behaviour* when the module contents are in separate files.

Secondly, *also* unlike everything else in the language, macros are only accessible *after* their definition. Also note that this example demonstrates how 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!
}
```

Macros can be exported from a module using the `#[macro_use]` attribute.

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

```
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 `mac`, the following holds:

```
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!
}
# fn main() {}
```

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

```
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 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*.

Import/Export

There are two ways to expose a macro to a wider scope. The first is the `#[macro_use]` attribute. This can be applied to *either* modules or external crates. For example:

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

X!();
```

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!();
#
# fn main() {}
```

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

Finally, when importing 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 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 this does *not* work for macros, since macros do not interact with regular name resolution in any way. That is, 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.

3

Macros, A Practical Introduction

This chapter will introduce the Rust 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[1] which is another high-level explanation, and the methodical introduction (chapter 2, page 7) chapter of this book, which explains the macro system in detail.

A Little Context

> **Note**: don't panic! What follows is the only math 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[2] can be defined by the relation:

```
<span class="katex"><span class="katex-inner"><span style="height: 0.68333em;" class="strut"></span><span style
```

Thus, the first two numbers in the sequence are 0 and 1, with the third being F0 + F1 = 0 + 1 = 1, the fourth F1 + F2 = 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.

[1] http://doc.rust-lang.org/book/macros.html
[2] https://en.wikipedia.org/wiki/Fibonacci_number

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

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

Construction

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

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

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

From that, we can take a stab at how the 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 macro, using the `macro_rules!` system, called `recurrence!`. This macro has a single parsing rule. That rule says the input to the macro must match:

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

Finally, the rule says that *if* the input matches this rule, then the macro 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 `+`. There is no support for "zero or one" or more specific numbers of repetitions.

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.

```
<thead>
    <tr>
        <th>Position</th>
        <th>Input</th>
        <th><code>inits</code></th>
        <th><code>recur</code></th>
    </tr>
</thead>
<tbody class="small-code">
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>⌂</code></td>
        <td><code>a[n] = 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code> ⌂</code></td>
        <td><code>[n] = 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>  ⌂</code></td>
        <td><code>n] = 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>   ⌂</code></td>
        <td><code>] = 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>    ⌂</code></td>
        <td><code>= 0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>     ⌂</code></td>
        <td><code>0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
```

```
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>         ⌂</code></td>
        <td><code>0, 1, ..., a[n-1] + a[n-2]</code></td>
        <td></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>                     ⌂  ⌂</code></td>
        <td><code>, 1, ..., a[n-1] + a[n-2]</code></td>
        <td><code>0</code></td>
        <td></td>
    </tr>
    <tr>
        <td colspan="4" style="font-size:.7em;">
```

Note: there are two here, because the next input token might match either the comma separator betweenem> 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.

```
        </td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>         ⌂                ⌂</code></td>
        <td><code>1, ..., a[n-1] + a[n-2]</code></td>
        <td><code>0</code></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>                     ⌂  ⌂ <s>⌂</s></code></td>
        <td><code>, ..., a[n-1] + a[n-2]</code></td>
        <td><code>0</code>, <code>1</code></td>
        <td></td>
    </tr>
    <tr>
        <td colspan="4" style="font-size:.7em;">
```

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.

```
        </td>
    </tr>
    <tr>
```

```
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>        ⌂               ⌂</code></td>
        <td><code>..., a[n-1] + a[n-2]</code></td>
        <td><code>0</code>, <code>1</code></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>        <s>⌂</s>                   ⌂</code></td>
        <td><code>, a[n-1] + a[n-2]</code></td>
        <td><code>0</code>, <code>1</code></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>                              ⌂</code></td>
        <td><code>a[n-1] + a[n-2]</code></td>
        <td><code>0</code>, <code>1</code></td>
        <td></td>
    </tr>
    <tr>
        <td><code>a[n] = $($inits:expr),+ , ... , $recur:expr</code>
            <code>                                         ⌂</code></td>
        <td></td>
        <td><code>0</code>, <code>1</code></td>
        <td><code>a[n-1] + a[n-2]</code></td>
    </tr>
    <tr>
        <td colspan="4" style="font-size:.7em;">
```

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.

```
        </td>
    </tr>
</tbody>
```

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;

        #[inline]
        fn next(&mut self) -> Option<u64> {
            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-1] + a[n-2]);

                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 summarise, the complete expansion is:

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

```
        pos: usize,
    }

    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 a = /* something */;
                let n = self.pos;
                let next_val = (a[n-1] + a[n-2]);

                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 macro invocation. Most of this will optimise away in the final binary, with some judicious use of `#[inline]` attributes.

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 macro syntax, you should work out where to introduce it. In this case, we've added `u64`, but that's not neccesarily 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-1] + a[n-2]];

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

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

> **Aside**: if you're wondering, the bit after the colon in a capture 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!` (Captures) (section 2, page 15).
>
> 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; Captures and Expansion Redux (section 2, page 17).

Indexing and Shuffling

I will skim a bit over this part, since it's effectively tangential to the macro 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:

```
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]
    }
}
```

> **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* parameterised 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:

```
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:

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

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

    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-1] + a[n-2])
            };

            {
                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 }
    };

    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
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:

```
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;

    #[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-1] + a[n-2])
            };

            {
                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-1] + a[n-2]];

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

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

```
recurrence.rs:69:45: 69:48 error: local ambiguity: multiple parsing options: built-in
↳ NTs expr ('inits') or 1 other options.
recurrence.rs:69     let fib = recurrence![a[n]: u64 = 0, 1, ..., a[n-1] + a[n-2]];
                                                             ^~~
```

Here, we've run into a limitation of `macro_rules`. 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-1] + a[n-2]];
//                                         ^~~ changed

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

Success! We can now start replacing things in the *expansion* with things we've *captured*.

Substitution

Substituting something you've captured in a macro is quite simple; you can insert the contents of a capture `$sty:ty` by using `$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: [1, 1], pos: 0 }
        }
    };
}

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

    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. 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
}
```

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

```
// 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-1] + a[n-2])
                        };

                        {
                            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.

```
# macro_rules! count_exprs {
#     () => (0);
#     ($head:expr $(, $tail:expr)*) => (1 + count_exprs!($($tail),*));
# }
# macro_rules! recurrence {
#     ( a[n]: $sty:ty = $($inits:expr),+ ... $recur:expr ) => {
#         {
#             const MEMORY: uint = count_exprs!($($inits),+);
#             struct Recurrence {
#                 mem: [$sty; MEMORY],
#                 pos: uint,
#             }
#             struct IndexOffset<'a> {
#                 slice: &'a [$sty; MEMORY],
#                 offset: uint,
#             }
#             impl<'a> Index<uint, $sty> for IndexOffset<'a> {
#                 #[inline(always)]
#                 fn index<'b>(&'b self, index: &uint) -> &'b $sty {
#                     let real_index = *index - self.offset + MEMORY;
#                     &self.slice[real_index]
#                 }
#             }
#             impl Iterator<u64> for Recurrence {
/* ... */
                #[inline]
                fn next(&mut self) -> Option<u64> {
                    if self.pos < MEMORY {
                        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 };
```

```
                            $recur
//                          ^~~~~~ changed
                        };
                        {
                            use std::mem::swap;
                            let mut swap_tmp = next_val;
                            for i in range(0, MEMORY).rev() {
                                swap(&mut swap_tmp, &mut self.mem[i]);
                            }
                        }
                        self.pos += 1;
                        Some(next_val)
                    }
                }
/* ... */
#             }
#             Recurrence { mem: [$($inits),+], pos: 0 }
#         }
#     };
# }
# fn main() {
#     let fib = recurrence![a[n]: u64 = 1, 1 ... a[n-1] + a[n-2]];
#     for e in fib.take(10) { println!("{}", e) }
# }
```

And, when we compile our finished macro...

```
recurrence.rs:77:48: 77:49 error: unresolved name `a`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
                                                                ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:50: 77:51 error: unresolved name `n`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
                                                                  ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:57: 77:58 error: unresolved name `a`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
                                                                         ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
recurrence.rs:77:59: 77:60 error: unresolved name `n`
recurrence.rs:77     let fib = recurrence![a[n]: u64 = 0, 1 ... a[n-1] + a[n-2]];
                                                                           ^
recurrence.rs:7:1: 74:2 note: in expansion of recurrence!
recurrence.rs:77:15: 77:64 note: expansion site
```

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

```
$ rustc -Z unstable-options --pretty expanded recurrence.rs
```

The `--pretty expanded` argument tells `rustc` to perform macro expansion, then turn the resulting AST back into source code. Because this option isn't considered stable yet, we also need `-Z unstable-options`. The output (after cleaning up some formatting) is shown below; in particular, note the place in the code where `$recur` was substituted:

```
#![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_SIZ
↳ E).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 stabilised.

Being Hygienic

The issue here is that identifiers in Rust macros 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 = 42i;
            $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:

Now, let's expand the invocation.

As you can see, the a that's defined by the macro 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 macros: macros can produce ASTs which will not compile, but which *will* compile if written out by hand, or dumped using `--pretty 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:

This now expands to:

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:

```
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
                            $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-1] + a[n-2]];

    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!

4

Patterns

Parsing and expansion patterns.

Callbacks

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

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

macro_rules! recognise_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?") };
}

fn main() {
    recognise_tree!(expand_to_larch!());
    call_with_larch!(recognise_tree);
}
```

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. This can make modularising macros very

difficult.

An alternative is to use recursion and pass a callback. Here is a trace of the above example to demonstrate how this takes place:

```
recognise_tree! { expand_to_larch ! (  ) }
println! { "I don't know; some kind of birch maybe?" }
// ...

call_with_larch! { recognise_tree }
recognise_tree! { larch }
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.

A "TT muncher" is a recursive 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 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 optimisation. 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.

Internal Rules

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

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

Because macros do not interact with regular item privacy or lookup, any public macro *must* bring with it all other macros that it depends on. This can lead to pollution of the global macro namespace, or even conflicts with macros from other crates. It may also cause confusion to users who attempt to *selectively* import macros: they must transitively import *all* macros, including ones that may not be publicly documented.

A good solution is to conceal what would otherwise be other public macros *inside* the macro being exported. The above example shows how the common `as_expr!` macro could be moved *into* the publicly exported macro that is using it.

The reason for using `@` is that, as of Rust 1.2, the `@` token is *not* used in prefix position; as such, it cannot conflict with anything. 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 code.

> **Note**: 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.

If exporting at least one internal macro is unavoidable (*e.g.* you have many macros that depend on a common set of utility rules), you can use this pattern to combine *all* internal macros into a single uber-macro.

```
macro_rules! crate_name_util {
    (@as_expr $e:expr) => {$e};
    (@as_item $i:item) => {$i};
    (@count_tts) => {0usize};
    // ...
}
```

Push-Down Accumulation

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

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

All macros 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 macro expand to a partial construct.

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

```
macro_rules! init_array {
    (@accum 0, $_e:expr) => {/* empty */};
    (@accum 1, $e:expr) => {$e};
    (@accum 2, $e:expr) => {$e, init_array!(@accum 1, $e)};
    (@accum 3, $e:expr) => {$e, init_array!(@accum 2, $e)};
    [$e:expr; $n:tt] => {
        {
            let e = $e;
            [init_array!(@accum $n, e)]
        }
    };
}
```

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

```
            [init_array!(@accum 3, e)]
            [e, init_array!(@accum 2, e)]
            [e, e, init_array!(@accum 1, e)]
            [e, e, e]
```

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 macro invocations proceeds as follows:

```
init_array! { String:: from ( "hi!" ) ; 3 }
init_array! { @ accum ( 3 , e . clone (  ) ) -> (  ) }
init_array! { @ accum ( 2 , e.clone() ) -> ( e.clone() , ) }
init_array! { @ accum ( 1 , e.clone() ) -> ( e.clone() , e.clone() , ) }
init_array! { @ accum ( 0 , e.clone() ) -> ( e.clone() , e.clone() , e.clone() , ) }
init_array! { @ as_expr [ 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 behaviour of such macros.

Push-down accumulation is frequently used as part of incremental TT munchers (**??**, page **??**), as it allows arbitrarily complex intermediate results to be constructed.

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 instead 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.

Trailing Separators

```
macro_rules! match_exprs {
    ($($exprs:expr),* $(,)*) => {...};
}
```

There are various places in the Rust grammar where trailing commas are permitted. The two common ways of matching (for example) a list of expressions (`$($exprs:expr),*` and `$($exprs:expr,)*`) can deal with *either* no trailing comma *or* a trailing comma, but *not both*.

Placing a `$(,)*` repetition *after* the main list, however, will capture any number (including zero or one) of trailing commas, or any other separator you may be using.

Note that this cannot be used in all contexts. If the compiler rejects this, you will likely need to use multiple arms and/or incremental matching.

TT Bundling

```
macro_rules! call_a_or_b_on_tail {
    ((a: $a:expr, b: $b:expr), call a: $($tail:tt)*) => {
```

```
        $a(stringify!($($tail)*))
    };

    ((a: $a:expr, b: $b:expr), 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 doesn't 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's make it return
            some ninety one!
        ),
        Some(91)
    );
}
```

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.

Provisional

This section is for patterns or techniques which are of dubious value, or which might be *too* niche for inclusion.

Abacus Counters

> **Provisional**: needs a more compelling example. Although an important part of the `Ook!` macro, 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 (**??**, page **??**) and incremental TT munchers (**??**, page **??**).

```
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 (**??**, page **??**). 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 whit 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[4]. For the example above, the terminal rules can be replaced with the following:

[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[2] today!
In fairness, it could *also* have been called "unary counting"[3].

[4]../blk/README.html#counting

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

    // This extracts the counter as an integer expression.
    (() -> ()) => {0};
    (() -> (- $($count:tt)*)) => {
        {(-1i32) $(- replace_expr!($count 1i32))*}
    };
    (() -> (+ $($count:tt)*)) => {
        {(1i32) $(+ replace_expr!($count 1i32))*}
    };
}

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

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))*
    }
}
```

5

Building Blocks

Reusable snippets of 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} }
```

These coercions are often used with push-down accumulation (section 4, page 62) 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. That is, because macros cannot appear in type position[1], you cannot have an `as_ty!` macro.

Counting

Repetition with replacement

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

[1]See Issue #27336[2].

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

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

This is much better, 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 *grevious* 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. The *downside* is that as of Rust 1.2, this *cannot* be used to produce a constant expression. Although the result can be optimised to a simple constant (in debug builds it compiles down to a load from memory), it still cannot be used in constant positions (such as the value of `consts`, or a fixed array's size).

However, if a non-constant count is acceptable, this is very much the preferred method.

Enum Parsing

```
macro_rules! parse_unitary_variants {
    (@as_expr $e:expr) => {$e};
    (@as_item $($i:item)+) => {$($i)+};

    // Exit rules.
    (
        @collect_unitary_variants ($callback:ident ( $($args:tt)* )),
        ($(,)*) -> ($($var_names:ident,)*)
    ) => {
        parse_unitary_variants! {
            @as_expr
            $callback!{ $($args)* ($($var_names),*) }
        }
    };

    (
        @collect_unitary_variants ($callback:ident { $($args:tt)* }),
        ($(,)*) -> ($($var_names:ident,)*)
    ) => {
        parse_unitary_variants! {
            @as_item
            $callback!{ $($args)* ($($var_names),*) }
        }
    };

    // Consume an attribute.
    (
        @collect_unitary_variants $fixed:tt,
        (#[$_attr:meta] $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        parse_unitary_variants! {
            @collect_unitary_variants $fixed,
            ($($tail)*) -> ($($var_names)*)
        }
    };

    // Handle a variant, optionally with an with initialiser.
```

```
    (
        @collect_unitary_variants $fixed:tt,
        ($var:ident $(= $_val:expr)*, $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        parse_unitary_variants! {
            @collect_unitary_variants $fixed,
            ($($tail)*) -> ($($var_names)* $var,)
        }
    };

    // Abort on variant with a payload.
    (
        @collect_unitary_variants $fixed:tt,
        ($var:ident $_struct:tt, $($tail:tt)*) -> ($($var_names:tt)*)
    ) => {
        const _error: () = "cannot parse unitary variants from enum with non-unitary v
↳ ariants";
    };

    // Entry rule.
    (enum $name:ident {$($body:tt)*} => $callback:ident $arg:tt) => {
        parse_unitary_variants! {
            @collect_unitary_variants
            ($callback $arg), ($($body)*,) -> ()
        }
    };
}
```

This macro shows how you can use an [incremental tt muncher] and push-down accumulation (section 4, page 62) to parse the variants of an `enum` where all variants are unitary (*i.e.* they have no payload). Upon completion, `parse_unitary_variants!` invokes a [callback] macro with the list of variants (plus any other arbitrary arguments supplied).

This can be modified to also parse `struct` fields, compute tag values for the variants, or even extract the names of *all* variants in an arbitrary `enum`.

6

Annotated Examples

This section contains real-world[1] macros which have been annotated to explain their design and construction.

Ook!

This macro is an implementation of the Ook! esoteric language[2], which is isomorphic to the Brainfuck esoteric language[3].

The execution model for the language is very simple: memory is represented as an array of "cells" (typically at least 8-bits) of some indeterminate number (usually at least 30,000). There is a pointer into memory which starts off at position 0. Finally, there is an execution stack (used to implement looping) and pointer into the program, although these last two are not exposed to the running program; they are properties of the runtime itself.

The language itself is comprised of just three tokens: `Ook.`, `Ook?`, and `Ook!`. These are combined in pairs to form the eight different operations:

- `Ook. Ook?` - increment pointer.
- `Ook? Ook.` - decrement pointer.
- `Ook. Ook.` - increment pointed-to memory cell.
- `Ook! Ook!` - decrement pointed-to memory cell.
- `Ook! Ook.` - write pointed-to memory cell to standard output.
- `Ook. Ook!` - read from standard input into pointed-to memory cell.
- `Ook! Ook?` - begin a loop.
- `Ook? Ook!` - jump back to start of loop if pointed-to memory cell is not zero; otherwise, continue.

[1] For the most part.
[2] http://www.dangermouse.net/esoteric/ook.html
[3] http://www.muppetlabs.com/~breadbox/bf/

Ook! is interesting because it is known to be Turing-complete, meaning that any environment in which you can implement it must *also* be Turing-complete.

Implementation

```
#![recursion_limit = "158"]
```

This is, in fact, the lowest possible recursion limit for which the example program provided at the end will actually compile. If you're wondering what could be so fantastically complex that it would *justify* a recursion limit nearly five times the default limit… take a wild guess[4].

```
type CellType = u8;
const MEM_SIZE: usize = 30_000;
```

These are here purely to ensure they are visible to the macro expansion.[5]

```
macro_rules! Ook {
```

The name should *probably* have been `ook!` to match the standard naming convention, but the opportunity was simply too good to pass up.

The rules for this macro are broken up into sections using the internal rules[6] pattern.

The first of these will be a `@start` rule, which takes care of setting up the block in which the rest of our expansion will happen. There is nothing particularly interesting in this: we define some variables and helper functions, then do the bulk of the expansion.

A few small notes:

- We are expanding into a function largely so that we can use `try!` to simplify error handling.
- The use of underscore-prefixed names is so that the compiler will not complain about unused functions or variables if, for example, the user writes an Ook! program that does no I/O.

```
    (@start $($Ooks:tt)*) => {
        {
            fn ook() -> ::std::io::Result<Vec<CellType>> {
                use ::std::io;
                use ::std::io::prelude::*;

                fn _re() -> io::Error {
                    io::Error::new(
                        io::ErrorKind::Other,
```

[4] https://en.wikipedia.org/wiki/Hello_world_program
[5] They *could* have been defined within the macro, but then they would have to have been explicitly passed around (due to hygiene). To be honest, by the time I realised I *needed* to define these, the macro was already mostly written and… well, would *you* want to go through and fix this thing up if you didn't *absolutely need* to?
[6] ../pat/README.html#internal-rules

```
                String::from("ran out of input"))
        }

        fn _inc(a: &mut [u8], i: usize) {
            let c = &mut a[i];
            *c = c.wrapping_add(1);
        }

        fn _dec(a: &mut [u8], i: usize) {
            let c = &mut a[i];
            *c = c.wrapping_sub(1);
        }

        let _r = &mut io::stdin();
        let _w = &mut io::stdout();

        let mut _a: Vec<CellType> = Vec::with_capacity(MEM_SIZE);
        _a.extend(::std::iter::repeat(0).take(MEM_SIZE));
        let mut _i = 0;
        {
            let _a = &mut *_a;
            Ook!(@e (_a, _i, _inc, _dec, _r, _w, _re); ($($Ooks)*));
        }
        Ok(_a)
    }
    ook()
}
};
```

Opcode parsing

Next are the "execute" rules, which are used to parse opcodes from the input.

The general form of these rules is `(@e $syms; ($input))`. As you can see from the `@start` rule, `$syms` is the collection of symbols needed to actually implement the program: input, output, the memory array, *etc.*. We are using TT bundling[7] to simplify forwarding of these symbols through later, intermediate rules.

First, is the rule that terminates our recursion: once we have no more input, we stop.

```
(@e $syms:tt; ()) => {};
```

Next, we have a single rule for *almost* each opcode. For these, we strip off the opcode, emit the corresponding Rust code, then recurse on the input tail: a textbook TT muncher[8].

```
// Increment pointer.
(@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
```

[7] ../pat/README.html#tt-bundling
[8] ../pat/README.html#incremental-tt-munchers

```
        (Ook. Ook? $($tail:tt)*))
    => {
        $i = ($i + 1) % MEM_SIZE;
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Decrement pointer.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook? Ook. $($tail:tt)*))
    => {
        $i = if $i == 0 { MEM_SIZE } else { $i } - 1;
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Increment pointee.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook. Ook. $($tail:tt)*))
    => {
        $inc($a, $i);
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Decrement pointee.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook! Ook! $($tail:tt)*))
    => {
        $dec($a, $i);
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Write to stdout.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook! Ook. $($tail:tt)*))
    => {
        try!($w.write_all(&$a[$i .. $i+1]));
        Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
    };

    // Read from stdin.
    (@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
        (Ook. Ook! $($tail:tt)*))
    => {
        try!(
            match $r.read(&mut $a[$i .. $i+1]) {
                Ok(0) => Err($re()),
                ok @ Ok(..) => ok,
                err @ Err(..) => err
            }
        );
```

```
    Ook!(@e ($a, $i, $inc, $dec, $r, $w, $re); ($($tail)*));
};
```

Here is where things get more complicated. This opcode, `Ook! Ook?`, marks the start of a loop. Ook! loops are translated to the following Rust code:

> **Note**: this is *not* part of the larger code.
>
> ```
> while memory[ptr] != 0 {
>     // Contents of loop
> }
> ```

Of course, we cannot *actually* emit an incomplete loop. This *could* be solved by using pushdown[9], were it not for a more fundamental problem: we cannot *write* `while memory[ptr] != {`, at all, *anywhere*. This is because doing so would introduce an unbalanced brace.

The solution to this is to actually split the input into two parts: everything *inside* the loop, and everything *after* it. The `@x` rules handle the first, `@s` the latter.

```
(@e ($a:expr, $i:expr, $inc:expr, $dec:expr, $r:expr, $w:expr, $re:expr);
    (Ook! Ook? $($tail:tt)*))
=> {
    while $a[$i] != 0 {
        Ook!(@x ($a, $i, $inc, $dec, $r, $w, $re); (); (); ($($tail)*));
    }
    Ook!(@s ($a, $i, $inc, $dec, $r, $w, $re); (); ($($tail)*));
};
```

Loop extraction

Next are the `@x`, or "extraction", rules. These are responsible for taking an input tail and extracting the contents of a loop. The general form of these rules is: `(@x $sym; $depth; $buf; $tail)`.

The purpose of `$sym` is the same as above. `$tail` is the input to be parsed, whilst `$buf` is a push-down accumulation buffer[10] into which we will collect the opcodes that are inside the loop. But what of `$depth`?

A complication to all this is that loops can be *nested*. Thus, we must have some way of keeping track of how many levels deep we currently are. We must track this accurately enough to not stop parsing too early, nor too late, but when the level is *just right*.[11]

Since we cannot do arithmetic in macros, and it would be infeasible to write out explicit integer-matching rules (imagine the following rules all copy & pasted for a non-trivial number of positive integers), we will instead fall back on one of the most ancient and venerable counting methods in history: counting on our fingers.

[9] ../pat/README.html#push-down-accumulation

[10] ../pat/README.html#push-down-accumulation

[11] It is a little known fact[^aeg-ook.md--fact] that the story of Goldie Locks was actually an allegory for accurate lexical parsing techniques.

But as macros don't *have* fingers, we'll use a token abacus counter[12] instead. Specifically, we will use `@`s, where each `@` represents one additional level of depth. If we keep these `@`s contained in a group, we can implement the three important operations we need:

- Increment: match `($($depth:tt)*)`, substitute `(@ $($depth)*)`.
- Decrement: match `(@ $($depth:tt)*)`, substitute `($($depth)*)`.
- Compare to zero: match `()`.

First is a rule to detect when we find the matching `Ook? Ook!` sequence that closes the loop we're parsing. In this case, we feed the accumulated loop contents to the previously defined `@e` rules.

Note that we *do not* need to do anything with the remaining input tail (that will be handled by the `@s` rules).

```
(@x $syms:tt; (); ($($buf:tt)*);
    (Ook? Ook! $($tail:tt)*))
=> {
    // Outer-most loop is closed.  Process the buffered tokens.
    Ook!(@e $syms; ($($buf)*));
};
```

Next, we have rules for entering and exiting nested loops. These adjust the counter and add the opcodes to the buffer.

```
(@x $syms:tt; ($($depth:tt)*); ($($buf:tt)*);
    (Ook! Ook? $($tail:tt)*))
=> {
    // One level deeper.
    Ook!(@x $syms; (@ $($depth)*); ($($buf)* Ook! Ook?); ($($tail)*));
};

(@x $syms:tt; (@ $($depth:tt)*); ($($buf:tt)*);
    (Ook? Ook! $($tail:tt)*))
=> {
    // One level higher.
    Ook!(@x $syms; ($($depth)*); ($($buf)* Ook? Ook!); ($($tail)*));
};
```

Finally, we have a rule for "everything else". Note the `$op0` and `$op1` captures: as far as Rust is concerned, our Ook! tokens are always *two* Rust tokens: the identifier `Ook`, and another token. Thus, we can generalise over all non-loop opcodes by matching `!`, `?`, and `.` as `tt`s.

Here, we leave `$depth` untouched and just add the opcodes to the buffer.

```
(@x $syms:tt; $depth:tt; ($($buf:tt)*);
    (Ook $op0:tt Ook $op1:tt $($tail:tt)*))
=> {
    Ook!(@x $syms; $depth; ($($buf)* Ook $op0 Ook $op1); ($($tail)*));
};
```

[12] ../pat/README.html#abacus-counters

Loop Skipping

This is *broadly* the same as loop extraction, except we don't care about the *contents* of the loop (and as such, don't need the accumulation buffer). All we need to know is when we are *past* the loop. At that point, we resume processing the input using the `@e` rules.

As such, these rules are presented without further exposition.

```
    // End of loop.
    (@s $syms:tt; ();
        (Ook? Ook! $($tail:tt)*))
    => {
        Ook!(@e $syms; ($($tail)*));
    };

    // Enter nested loop.
    (@s $syms:tt; ($($depth:tt)*);
        (Ook! Ook? $($tail:tt)*))
    => {
        Ook!(@s $syms; (@ $($depth)*); ($($tail)*));
    };

    // Exit nested loop.
    (@s $syms:tt; (@ $($depth:tt)*);
        (Ook? Ook! $($tail:tt)*))
    => {
        Ook!(@s $syms; ($($depth)*); ($($tail)*));
    };

    // Not a loop opcode.
    (@s $syms:tt; ($($depth:tt)*);
        (Ook $op0:tt Ook $op1:tt $($tail:tt)*))
    => {
        Ook!(@s $syms; ($($depth)*); ($($tail)*));
    };
```

Entry point

This is the only non-internal rule.

It is worth noting that because this formulation simply matches *all* tokens provided to it, it is *extremely dangerous*. Any mistake can cause an invocation to fail to match all the above rules, thus falling down to this one and triggering an infinite recursion.

When you are writing, modifying, or debugging a macro like this, it is wise to temporarily prefix rules such as this one with something, such as `@entry`. This prevents the infinite recursion case, and you are more likely to get matcher errors at the appropriate place.

```
    ($($Ooks:tt)*) => {
        Ook!(@start $($Ooks)*)
    };
}
```

Usage

Here, finally, is our test program.

```
fn main() {
    let _ = Ook!(
        Ook. Ook?  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook! Ook?  Ook? Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook?  Ook! Ook!  Ook? Ook!  Ook? Ook.
        Ook! Ook.  Ook. Ook?  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook?
        Ook! Ook!  Ook? Ook!  Ook? Ook.  Ook. Ook.
        Ook! Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook! Ook.  Ook! Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook.  Ook. Ook?  Ook. Ook?
        Ook. Ook?  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook?
        Ook! Ook!  Ook? Ook!  Ook? Ook.  Ook! Ook.
        Ook. Ook?  Ook. Ook?  Ook. Ook?  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook! Ook?  Ook? Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook?  Ook! Ook!  Ook? Ook!  Ook? Ook.
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook.
        Ook? Ook.  Ook? Ook.  Ook? Ook.  Ook? Ook.
        Ook! Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook! Ook.  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook.
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook!  Ook! Ook!  Ook! Ook!  Ook! Ook!
        Ook! Ook.  Ook. Ook?  Ook. Ook?  Ook. Ook.
        Ook! Ook.  Ook! Ook?  Ook! Ook!  Ook? Ook!
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook. Ook.  Ook. Ook.
        Ook. Ook.  Ook. Ook.  Ook! Ook.
    );
}
```

The output when run (after a considerable pause for the compiler to do hundreds of recursive macro expansions) is:

```
Hello World!
```

With that, we have demonstrated the horrifying truth that `macro_rules!` is Turing-complete!

An aside

This was based on a macro implementing an isomorphic language called “Hodor!”. Manish Goregaokar then implemented a Brainfuck interpreter using the Hodor! macro[13]. So that is a Brainfuck interpreter written in Hodor! which was itself implemented using `macro_rules!`.

Legend has it that after raising the recursion limit to *three million* and allowing it to run for *four days*, it finally finished.

…by overflowing the stack and aborting. To this day, esolang-as-macro remains a decidedly *non-viable* method of development with Rust.

[13]https://www.reddit.com/r/rust/comments/39wvrm/hodor_esolang_as_a_rust_macro/cs76rqk?context=10000