Rust Macros

Ryan Eberhardt and Armin Namavari
June 2, 2020

Logistics

- CS110L shouldn’t be your priority right now
- Project 2 is out and we’ve updated our policy on it with regards to current circumstances — please check out Ryan’s Slack post.
- Please fill out Week 8 survey tonight: https://forms.gle/PEmptvXLx5TdTm4A9

Today

- The Plan
  - Preliminaries
  - Rust Macros
    - Declarative Macros
    - Procedural Macros (of which there are three kinds)
- Goal: understand **what Rust macros are and how they work.**
- This is one of the strangest concepts we'll cover (yes, maybe even weirder than nonblocking I/O and futures). **Please ask questions.**
- Next week we'll have a guest speaker who will talk about some exciting systems work he's done with Rust and how that work draws on the power of Rust macros.
  - You may want to review this lecture before next Tuesday!

What are Macros? (in C)

- Basically fancy find-and-replace
- When found, the macro is replaced with some chunk of code
- It's almost like there aren't any rules (see the example on the bottom)
- What about:
  - `#define MAX(X, Y) (((X) > (Y)) ? (X) : (Y))`

```
#define max(a,b) \
  ({ __typeof__ (a) _a = (a); \
      __typeof__ (b) _b = (b); \
    _a > _b ? _a : _b; })
```

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

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

https://stackoverflow.com/questions/3437404/min-and-max-in-c, https://danielkeep.github.io/tlborm/book/mbe-syn-source-analysis.html

Why Macros?

- Because it’s cool to write code that writes other code
- Because code reuse is nice
  - i.e. Having to write boilerplate code over and over again is bad. Why?
- Rust does macros pretty differently from C and this has some cool implications for the kind of code you can write.
  - **Rust macros can let you execute arbitrary code at compile-time**
  - Could you imagine doing something like derive with C macros?

You have already used macros in Rust

- `println!(“hello {}!”, name);`
- `vec![1, 2, 3];`
- `#[derive(Clone, Copy)]`
- `#[tokio::main]`

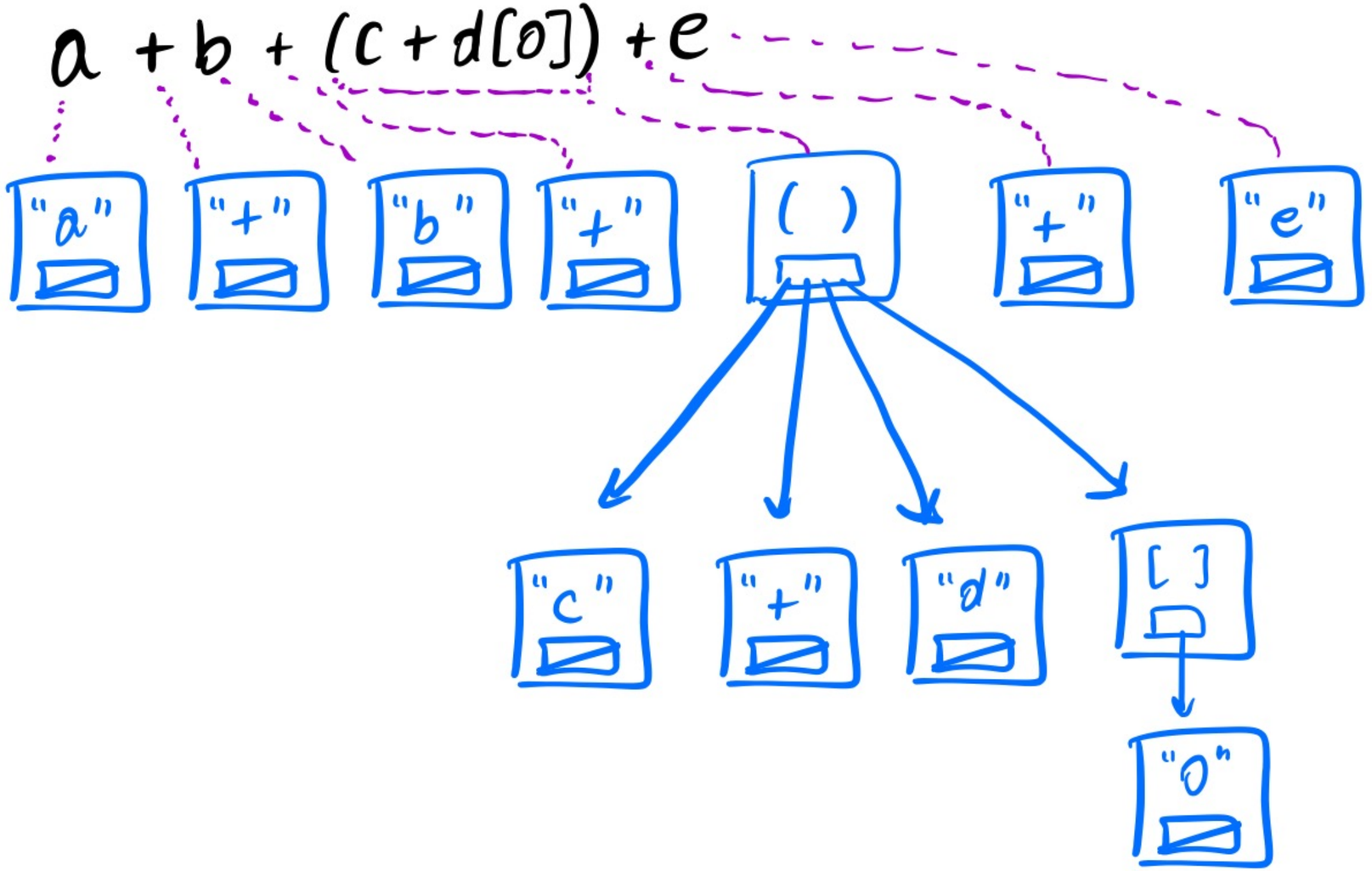
First, a little bit about languages and compilers

- Processors on your computer don’t speak Rust
- The rust compiler (rustc) must take your Rust code and translate it into assembly language
- Compilers usually operate in four steps:
  - **Lexing** — find the tokens e.g. “fn” “if” “struct” “trait” “pub" etc.
  - **Parsing** — understand the structure of these tokens e.g. what part of code corresponds to this if statement? produce an **abstract syntax tree (AST)**
  - **Type-checking/Semantic Analysis** — Make sure the code makes sense e.g. you can’t pass in a String to a function that expects a u32, borrow-checking
  - **Code generation** — convert your type-labeled AST into assembly.
  - If you’d like to learn more and build your very own compiler, take CS143!

Abstract Syntax Trees and Token Trees

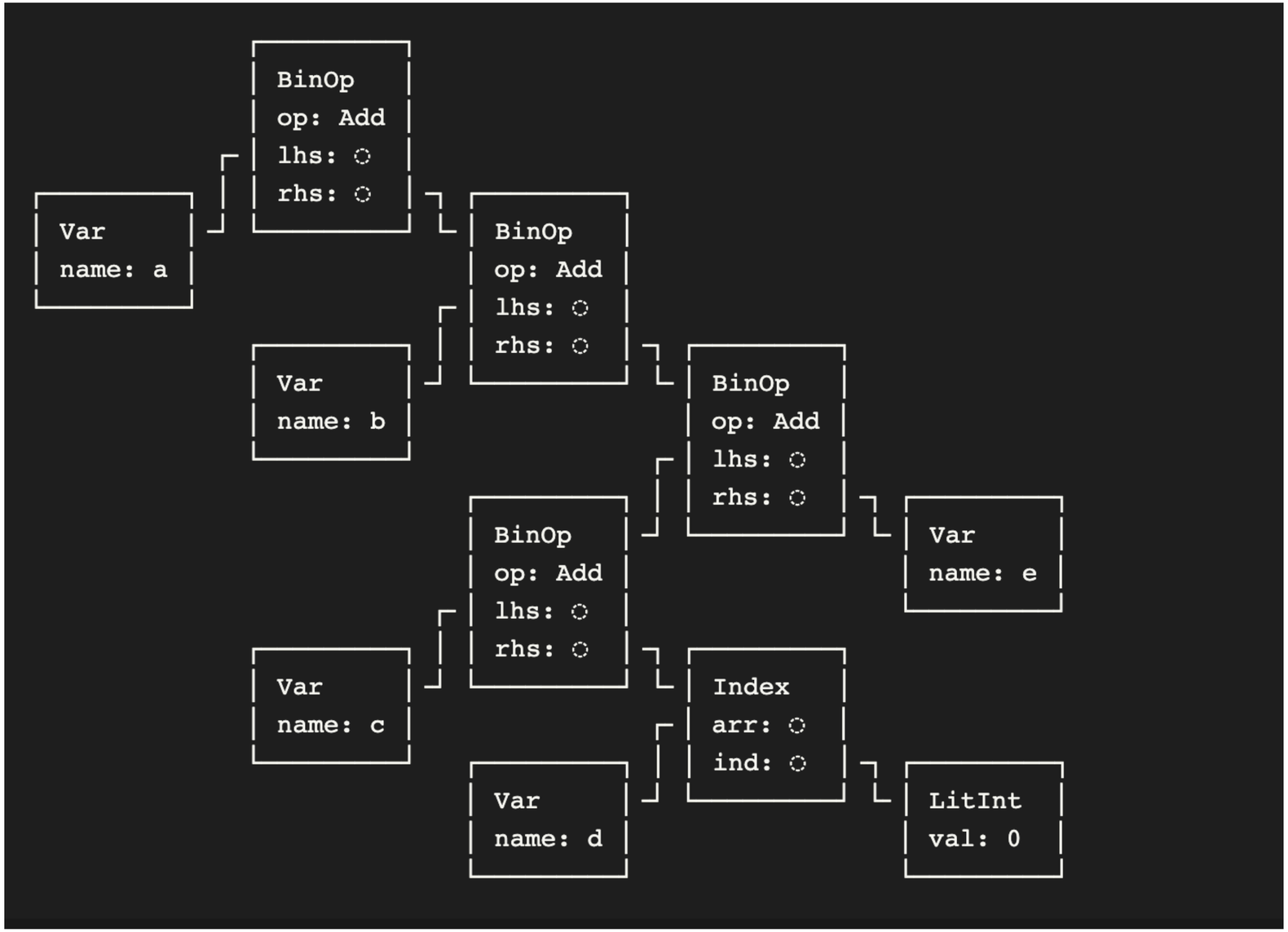
- Rust macros operate over **token trees** which are somewhere between the abstract syntax tree and the raw tokens themselves.
  - Identifiers (variable names, keywords), literals (e.g. int and string literals), punctuation (not a delimiter, e.g. “.”), and groups.
- An AST provides us full info about the expression as a whole
- The token-tree tells us about how tokens are grouped together with (…), {…}, and […]
  - We’ll see pictures of this in the following slides

Token Tree(s) Example



AST Example

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



https://danielkeep.github.io/tlborm/book/mbe-syn-source-analysis.html

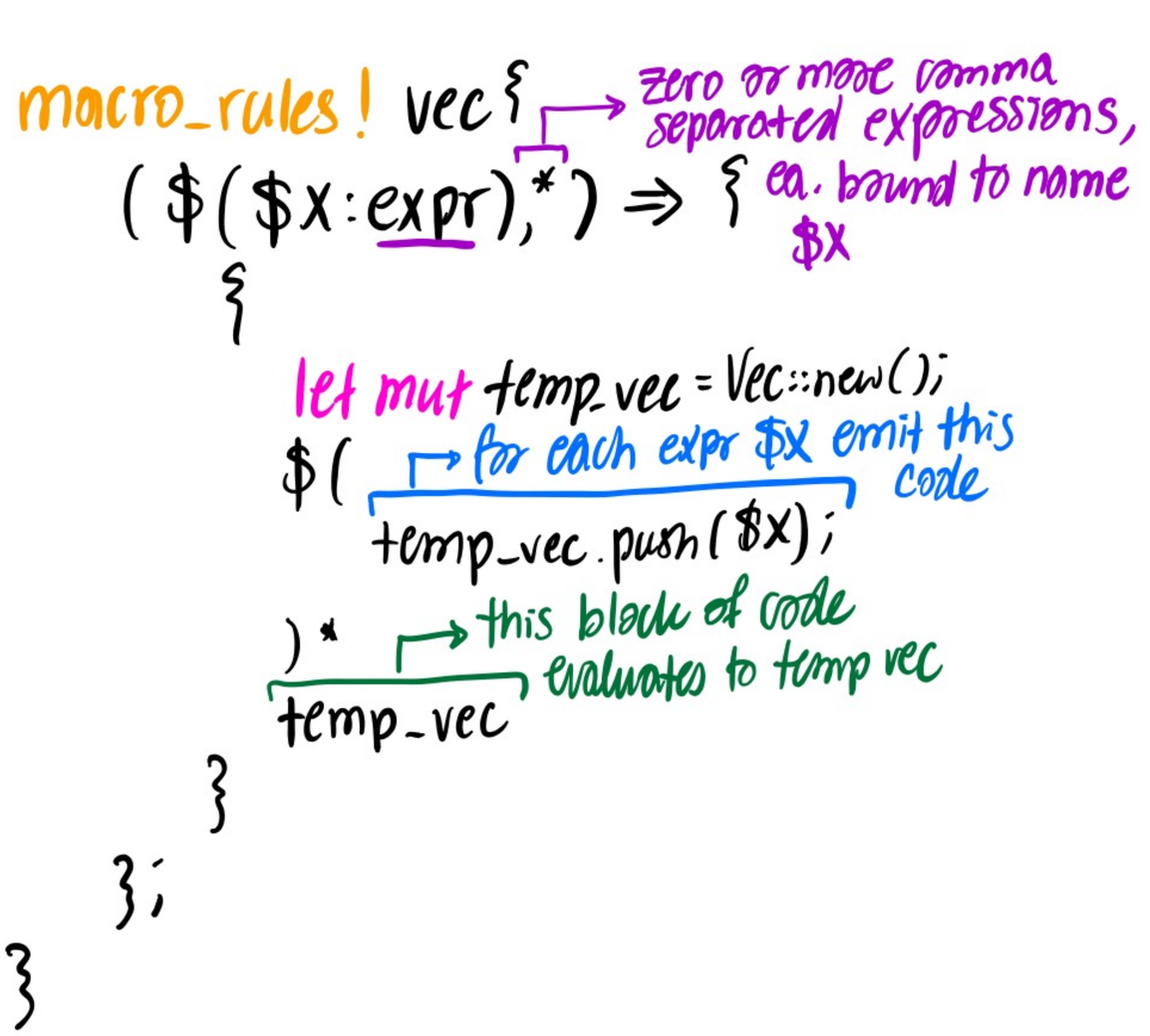
Declarative Macros with `macro_rules!`

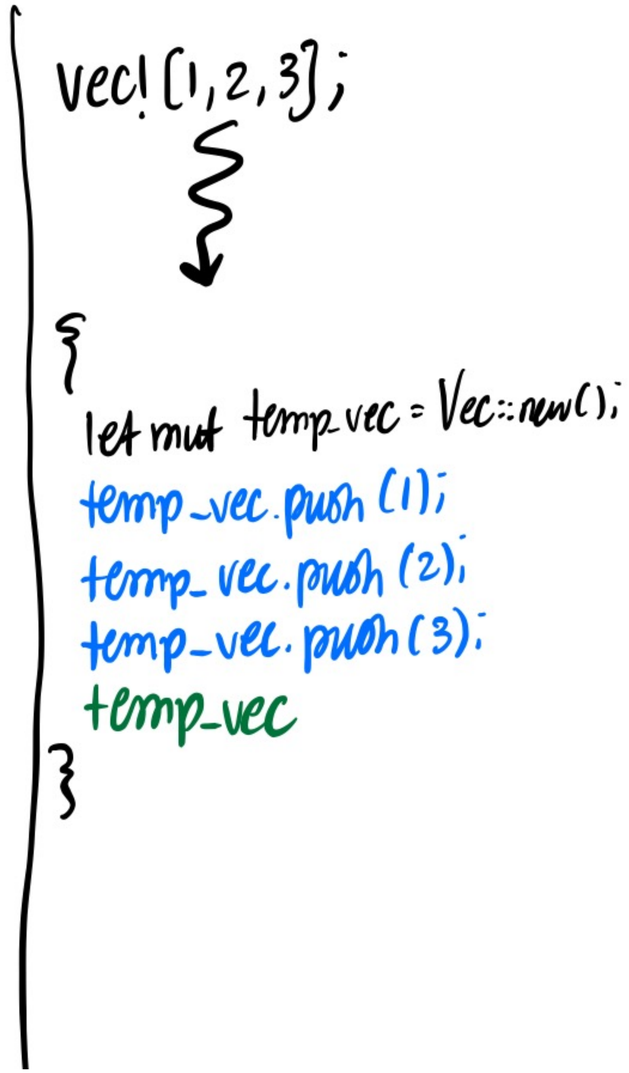
- Very fancy pattern matching. Sort of like C macros on steroids
- Patterns look like this:
  - `{$pattern} => {expansion}`
- Tries to find match (over token tree) and expand to the code indicated by that case of the match (we'll see an example in the next slide)
- If you'd like to learn more about all the possible patterns/rules, take a look through the links on the last slide.

Peeking under the hood of `vec!`

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Peeking under the hood of `vec!`





Procedural Macros

- Functions that take in code as input and produce code as output
  - Declarative macros feel more like match statements than they do like functions.
  - Procedural macros are more powerful than declarative macros but often harder to use (not to imply that `macro_rules!` is easy!)
    - the power vs. simplicity tradeoff is a common theme
- Three kinds:
  - Derive-type macros
  - Attribute-like macros
  - Function-like macros

“Derive” Macros

- Recall that we can automatically derive traits for structs we define
- We’ll take a look at an example from the Rust book for how we can automatically generate code that implements traits for a given type
- We’ll have to deal with `TokenStreams`: stream of **token trees**

```
#[derive(Clone, Copy, Debug)]
pub struct Point {
    x: i32,
    y: i32,
}
```

“Derive” Macros — The Plan

- We’re going to walk through an example from the Rust Book.
- We will define a function that takes in the struct as input as a `TokenStream`
- It will then parse the `TokenStream` as an AST
- It will use the AST to figure out the name of the struct
- We will then use another macro called quote! to define a trait implementation for our struct and output this implementation as a `TokenStream`

```
#[derive(Clone, Copy, Debug)]
pub struct Point {
    x: i32,
    y: i32,
}
```

“Derive” Macros — Code Example

```
// Client of the macro
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

“Derive” Macros — Code Example

```
extern crate proc_macro;

use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}
```

“Derive” Macros — Code Example

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

Attribute-like procedural macros

- Like the derive macros but more general
- You can apply these macros to other syntactic entities e.g. functions
- You can write an attribute macro that verifies that you write your enum variants in sorted order (check out the project link on the last slide)
- You can write an attribute macro that packages a struct into a bitfield (also on the same project link)
- You can write an attribute macro that generates code for an HTTP request handler function (our guest speaker might talk about a project related to this next Tuesday!)

Attribute-like procedural macros (example)

```
#[bitfield]
pub struct MyFourBytes {
    a: B1,
    b: B3,
    c: B4,
    d: B24,
}
// Emits the code below (and rewrites struct definition to contain a private byte array)
impl MyFourBytes {
    // Initializes all fields to 0.
    pub fn new() -> Self;

    // Field getters and setters:
    pub fn get_a(&self) -> u8;
    pub fn set_a(&mut self, val: u8);
    pub fn get_b(&self) -> u8;
    pub fn set_b(&mut self, val: u8);
    pub fn get_c(&self) -> u8;
    pub fn set_c(&mut self, val: u8);
    pub fn get_d(&self) -> u32;
    pub fn set_d(&mut self, val: u32);
}
```

Function-like procedural macros

- Macro that looks like a function call
- e.g. sql! Macro from the Rust book — will construct some sort of SQL query object from SQL syntax.

```
let sql = sql!(SELECT * FROM posts WHERE id=1);

#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
 …
}
```

Recursive Macros

- Macros can invoke other macros
- Macros can invoke themselves
- This can happen with declarative macros and with procedural macros
- We’ll see an example on the next slide

A Declarative Recursive Macro

```
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}
// Usage:
write_html!(&mut out,
    html[
        head[title["Macros guide"]]
        body[h1["Macros are the best!"]]
    ]);

// https://doc.rust-lang.org/1.7.0/book/macros.html
```

Summary

- Declarative macros
  - `macro_rules!`
  - Match expressions and expand out, emitting code accordingly
- Procedural macros
  - Procedures that take in `TokenStreams` and emit `TokenStreams`
  - More powerful than declarative macros but trickier to use
  - Derive
  - Attribute
  - Function-like

Resources

- The Rust Book on Macros
- The Little Book of Rust Macros
- A Great Blog Post about Procedural Macros by Alex Crichton
- A Great Blog Post About Macros
- A Workshop on Procedural Macros
- A Blog Post about Recursive Macros