RUSTikales Rust for advanced coders

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. syn + quote

1. Recap

1. Recap

- Rust has two very powerful macro systems

1. Recap

- Rust has two very powerful macro systems
- Macros are useful for metaprogramming

1. Recap

- Rust has two very powerful macro systems
- Macros are useful for metaprogramming
- Declarative Macros are declared using the keyword macro_rules!

1. Recap

- Rust has two very powerful macro systems
- Macros are useful for metaprogramming
- Declarative Macros are declared using the keyword macro_rules!
- Procedural Macros are special crates
  - proc-macro is a type of library

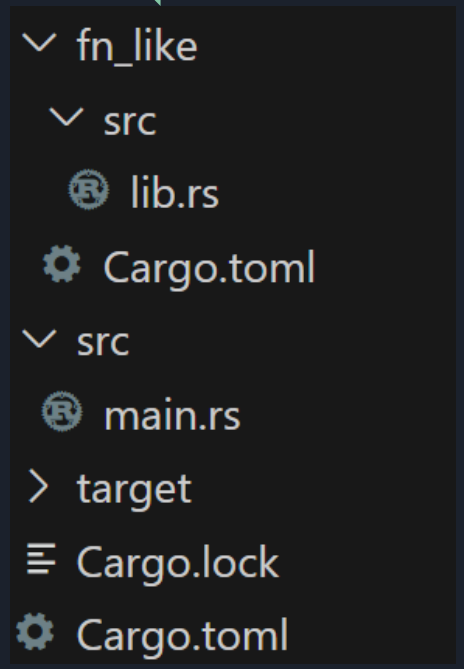
1. Recap

- Rust has two very powerful macro systems
- Macros are useful for metaprogramming
- Declarative Macros are declared using the keyword macro_rules!
- Procedural Macros are special crates
- Procedural Macros allow us to write powerful macros in normal Rust

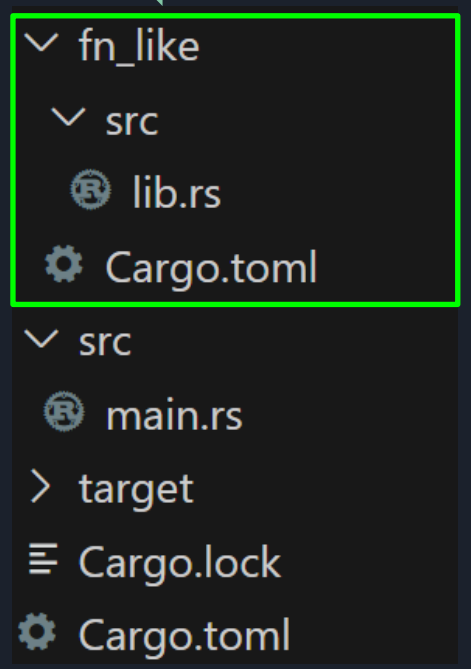
1. Recap

- Rust has two very powerful macro systems
- Macros are useful for metaprogramming
- Declarative Macros are declared using the keyword macro_rules!
- Procedural Macros are special crates
- Procedural Macros allow us to write powerful macros in normal Rust
- There are three types of procedural macros
  - proc_macro → Function-like macros
  - proc_macro_derive → Derive macros
  - proc_macro_attribute → Attribute macros

1. Recap

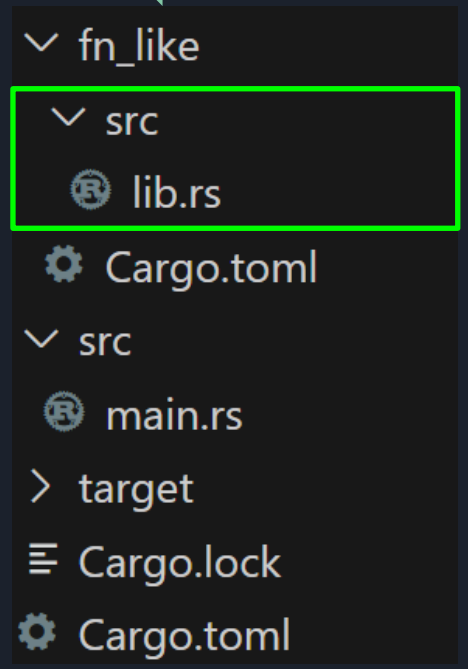

fn_like
src
lib.rs
Cargo.toml
src
main.rs
target
Cargo.lock
Cargo.toml

1. Recap



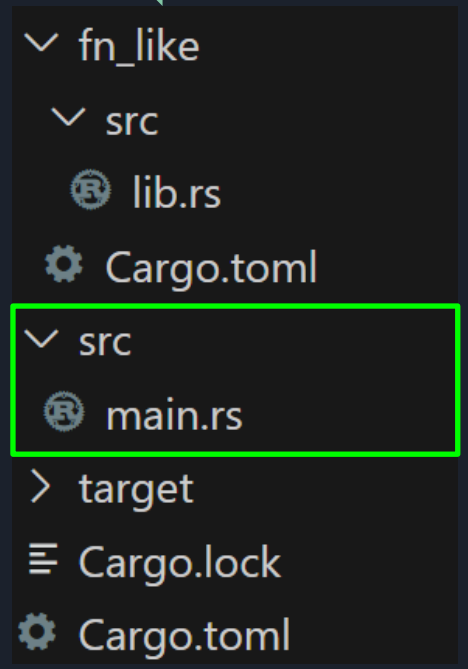
Idea:
→ fn_like is a proc-macro crate

1. Recap



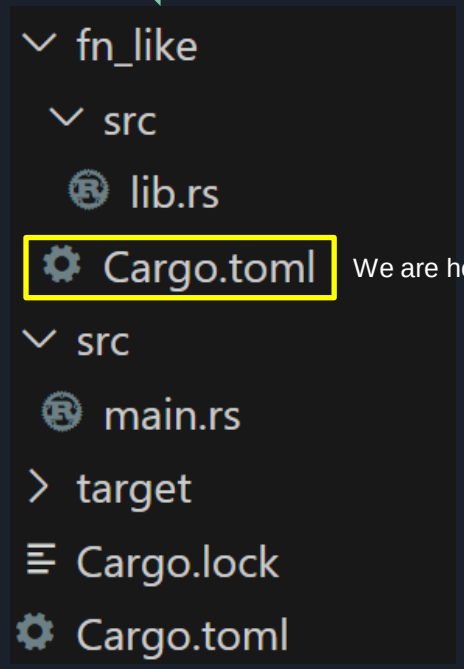
Idea:
→ fn_like is a proc-macro crate
→ In there, we define a macro which we want to use

1. Recap



Idea:
→ fn_like is a proc-macro crate
→ In there, we define a macro which we want to use
→ We later use that macro in our main application

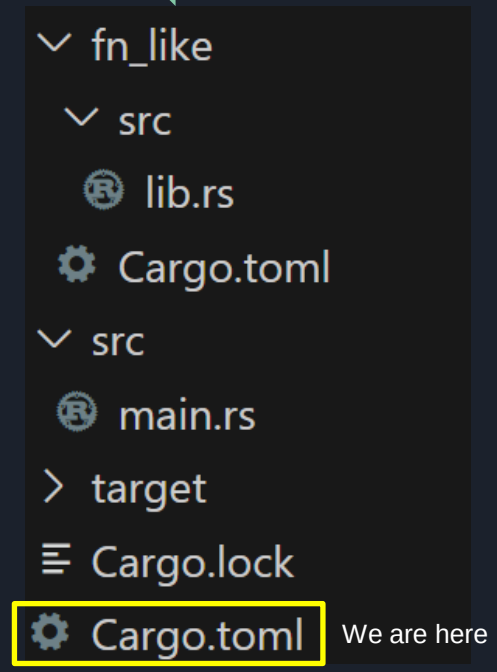
1. Recap




```
[package]
name = "fn_like"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
itertools = "0.13.0"
```

Mark crate as proc-macro

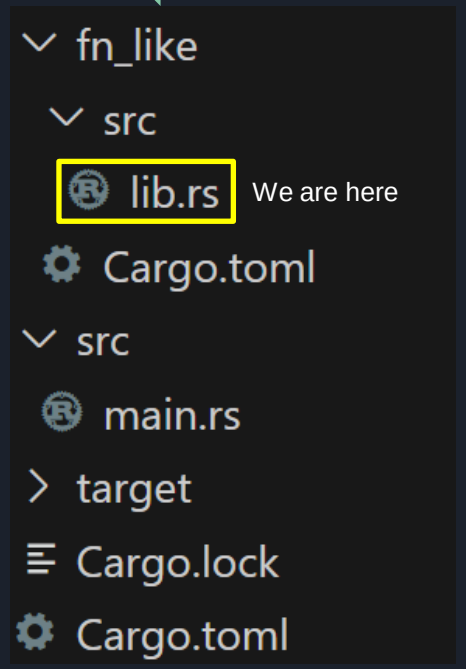
1. Recap




```toml
[package]
name = "recap"
version = "0.1.0"
edition = "2021"

[dependencies]
fn_like = { path="./fn_like" }
```

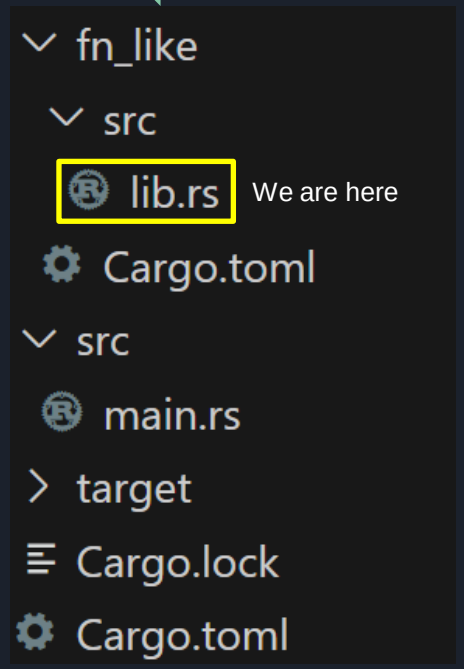
use proc-macro in our main application

1. Recap




```rust
use proc_macro::TokenStream;
use itertools::Itertools;
#[proc_macro]
pub fn print_type_info(input: TokenStream) -> TokenStream {
    let typ: String = input.into_iter().map(|t: TokenTree|t.to_string()).join(sep: "");
    format!("
    {{
        println!(\"Provided type: {typ}\");
        println!(\"Size: {{}} bytes\", std::mem::size_of::<{typ}>());
        println!(\"Alignment: {{}} bytes\", std::mem::align_of::<{typ}>());
        println!(\"\");
    }}
    ").parse().unwrap()
}
```

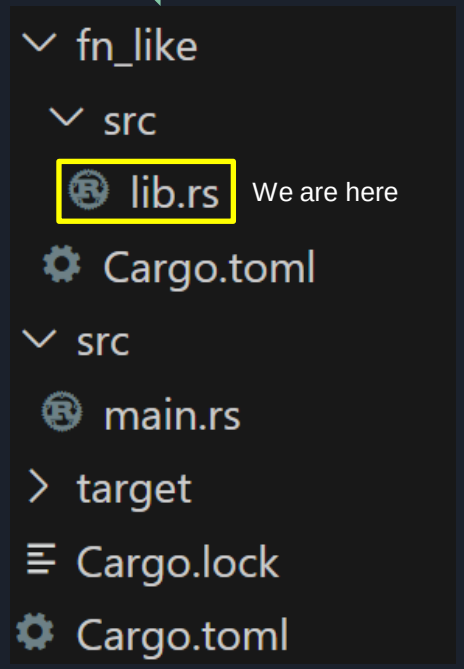
1. Recap




```rust
use proc_macro::TokenStream;
use itertools::Itertools;
#[proc_macro]
pub fn print_type_info(input: TokenStream) -> TokenStream {
    let typ: String = input.into_iter().map(|t: TokenTree|t.to_string()).join(sep: "");
    format!("
    {{
        println!(\"Provided type: {typ}\");
        println!(\"Size: {{}} bytes\", std::mem::size_of::<{typ}>());
        println!(\"Alignment: {{}} bytes\", std::mem::align_of::<{typ}>());
        println!(\"\");
    }}
    ").parse().unwrap()
}
```

The attribute defines the macro type
Here: We declare a Function-like macro

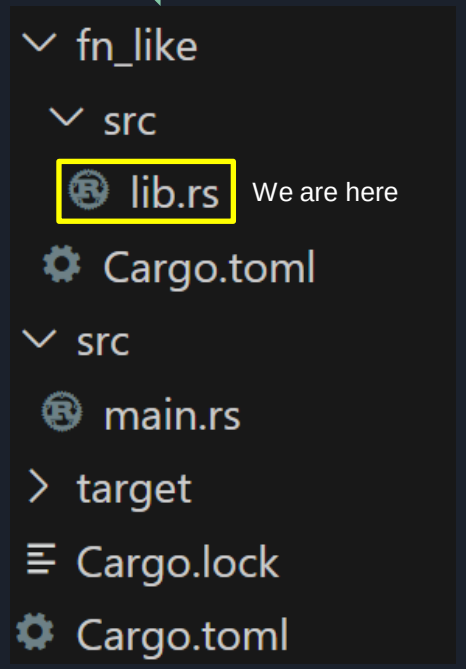
1. Recap




```rust
use proc_macro::TokenStream;
use itertools::Itertools;
#[proc_macro]
pub fn print_type_info(input: TokenStream) -> TokenStream {
    let typ: String = input.into_iter().map(|t: TokenTree|t.to_string()).join(sep: "");
    format!("
    {{
        println!(\"Provided type: {typ}\");
        println!(\"Size: {{}} bytes\", std::mem::size_of::<{typ}>());
        println!(\"Alignment: {{}} bytes\", std::mem::align_of::<{typ}>());
        println!(\"\");
    }}
    ").parse().unwrap()
}
```

Procedural macros work on a TokenStream
instead of matching rules like macro_rules!

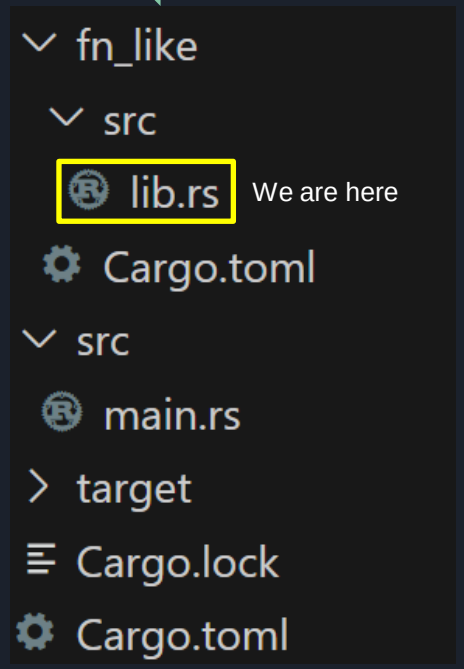
1. Recap




```rust
use proc_macro::TokenStream;
use itertools::Itertools;
#[proc_macro]
pub fn print_type_info(input: TokenStream) -> TokenStream {
    let typ: String = input.into_iter().map(|t: TokenTree|t.to_string()).join(sep: "");
    format!("
    {{
        println!(\"Provided type: {typ}\");
        println!(\"Size: {{}} bytes\", std::mem::size_of::<{typ}>());
        println!(\"Alignment: {{}} bytes\", std::mem::align_of::<{typ}>());
        println!(\"\");
    }}
    ").parse().unwrap()
}
```

Procedural macros are normal functions
which are called at compile time
→ Plugins for the compiler

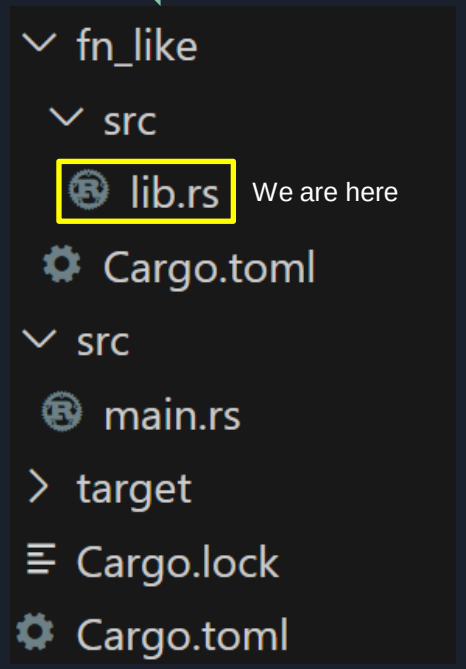
1. Recap




```rust
use proc_macro::TokenStream;
use itertools::Itertools;
#[proc_macro]
pub fn print_type_info(input: TokenStream) -> TokenStream {
    let typ: String = input.into_iter().map(|t: TokenTree|t.to_string()).join(sep: "");
    format!("
    {{
        println!(\"Provided type: {typ}\");
        println!(\"Size: {{}} bytes\", std::mem::size_of::<{typ}>());
        println!(\"Alignment: {{}} bytes\", std::mem::align_of::<{typ}>());
        println!(\"\");
    }}
    ").parse().unwrap()
}
```

Procedural macros are written in normal Rust
→ You can also import other crates!

1. Recap

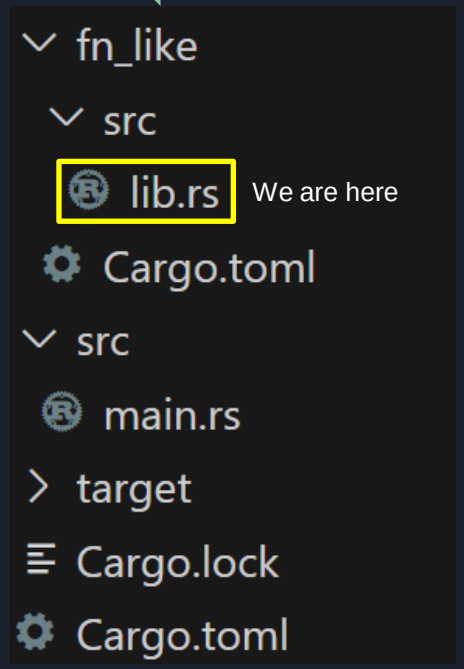



```rust
use proc_macro::TokenStream;
use itertools::Itertools;
#[proc_macro]
pub fn print_type_info(input: TokenStream) -> TokenStream {
    let typ: String = input.into_iter().map(|t: TokenTree|t.to_string()).join(sep: "");
    format!("
    {{
        println!(\"Provided type: {typ}\");
        println!(\"Size: {{}} bytes\", std::mem::size_of::<{typ}>());
        println!(\"Alignment: {{}} bytes\", std::mem::align_of::<{typ}>());
        println!(\"\");
    }}
    ").parse().unwrap()
}
```

Take all input tokens, and turn them into a single string

& + str → &str

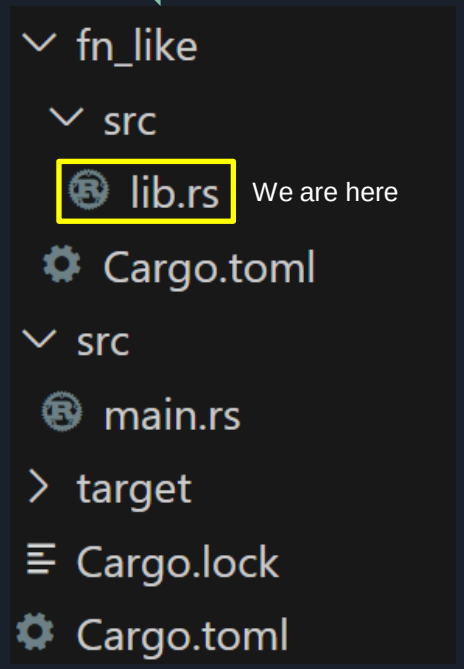
1. Recap




```rust
use proc_macro::TokenStream;
use itertools::Itertools;
#[proc_macro]
pub fn print_type_info(input: TokenStream) -> TokenStream {
    let typ: String = input.into_iter().map(|t: TokenTree|t.to_string()).join(sep: "");
    format!("
    {{
        println!(\"Provided type: {typ}\");
        println!(\"Size: {{}} bytes\", std::mem::size_of::<{typ}>());
        println!(\"Alignment: {{}} bytes\", std::mem::align_of::<{typ}>());
        println!(\"\");
    }}
    ").parse().unwrap()
}
```

The macro can‘t check if the generated String is a valid type.
It only generates code.
The Type Checker will have to figure that out.

1. Recap

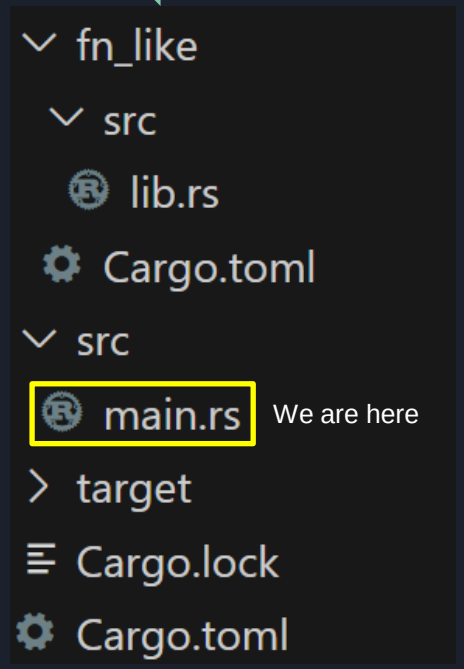



```rust
use proc_macro::TokenStream;
use itertools::Itertools;
#[proc_macro]
pub fn print_type_info(input: TokenStream) -> TokenStream {
    let typ: String = input.into_iter().map(|t: TokenTree|t.to_string()).join(sep: "");
    format!("
    {{
        println!(\"Provided type: {typ}\");
        println!(\"Size: {{}} bytes\", std::mem::size_of::<{typ}>());
        println!(\"Alignment: {{}} bytes\", std::mem::align_of::<{typ}>());
        println!(\"\");
    }}
    ").parse().unwrap()
}
```

Generate code!

Here: Print (at runtime) some information about the provided type

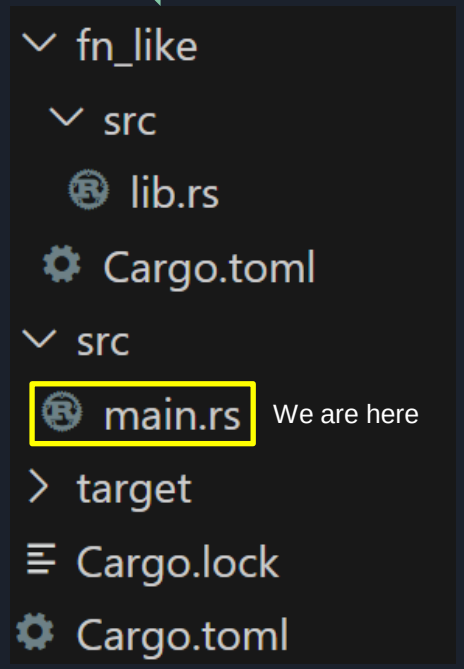
1. Recap




```rust
use fn_like::print_type_info;
0 implementations
struct Custom {
    field: u32,
    other: Option<&'static str>
}
▶ Run | Debug
fn main() {
    print_type_info!(u32);
    print_type_info!(String);
    print_type_info!(Custom);
    print_type_info!((&str, u8, bool));
}
```

proc-macros are used like any other crate

1. Recap




```rust
use fn_like::print_type_info;
0 implementations
struct Custom {
    field: u32,
    other: Option<&'static str>
}
▶ Run | Debug
fn main() {
    print_type_info!(u32);
    print_type_info!(String);
    print_type_info!(Custom);
    print_type_info!((&str, u8, bool));
}
```

At compile time, we call this macro four times and generate new code at those locations

1. Recap

```rust
use fn_like::print_type_info;
0 implementations
struct Custom {
    field: u32,
    other: Option<&'static str>
}
▶ Run | Debug
fn main() {
    print_type_info!(u32);
    print_type_info!(String);
    print_type_info!(Custom);
    print_type_info!((&str, u8, bool));
}
```

cargo expand

```rust
#[prelude_import]
use std::prelude::rust_2021::*;
#[macro_use]
extern crate std;
use fn_like::print_type_info;
struct Custom {
    field: u32,
    other: Option<&'static str>,
}
fn main() {
    {
        {
            ::std::io::_print(format_args!("Provided type: u32\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<u32>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<u32>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: String\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<String>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<String>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: Custom\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<Custom>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<Custom>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: (&str, u8, bool)\n"));
        };
        {
            ::std::io::_print(
                format_args!(
                    "Size: {0} bytes\n",
                    std::mem::size_of::<(&str, u8, bool)>(),
                ),
            );
        };
        {
            ::std::io::_print(
                format_args!(
                    "Alignment: {0} bytes\n",
                    std::mem::align_of::<(&str, u8, bool)>(),
                ),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
}
```

1. Recap

```rust
use fn_like::print_type_info;
0 implementations
struct Custom {
    field: u32,
    other: Option<&'static str>
}
▶ Run | Debug
fn main() {
    print_type_info!(u32);
    print_type_info!(String);
    print_type_info!(Custom);
    print_type_info!((&str, u8, bool));
}
```

Macros can easily generate a lot of code, which can result in worse compile times

→ Use macros responsibly

cargo expand

```rust
#[prelude_import]
use std::prelude::rust_2021::*;
#[macro_use]
extern crate std;
use fn_like::print_type_info;
struct Custom {
    field: u32,
    other: Option<&'static str>,
}
fn main() {
    {
        {
            ::std::io::_print(format_args!("Provided type: u32\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<u32>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<u32>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: String\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<String>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<String>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: Custom\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<Custom>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<Custom>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: (&str, u8, bool)\n"));
        };
        {
            ::std::io::_print(
                format_args!(
                    "Size: {0} bytes\n",
                    std::mem::size_of::<(&str, u8, bool)>(),
                ),
            );
        };
        {
            ::std::io::_print(
                format_args!(
                    "Alignment: {0} bytes\n",
                    std::mem::align_of::<(&str, u8, bool)>(),
                ),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
}
```

1. Recap

```rust
use fn_like::print_type_info;
0 implementations
struct Custom {
    field: u32,
    other: Option<&'static str>
}
▶ Run | Debug
fn main() {
    print_type_info!(u32);
    print_type_info!(String);
    print_type_info!(Custom);
    print_type_info!((&str, u8, bool));
}
```

Macros can easily generate a lot of code, which can result in worse compile times

→ Use macros responsibly

When trying to de-duplicate code, this order is recommended:

Functions → Generics + Traits → Macros

cargo expand

```rust
#[prelude_import]
use std::prelude::rust_2021::*;
#[macro_use]
extern crate std;
use fn_like::print_type_info;
struct Custom {
    field: u32,
    other: Option<&'static str>,
}
fn main() {
    {
        {
            ::std::io::_print(format_args!("Provided type: u32\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<u32>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<u32>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: String\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<String>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<String>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: Custom\n"));
        };
        {
            ::std::io::_print(
                format_args!("Size: {0} bytes\n", std::mem::size_of::<Custom>()),
            );
        };
        {
            ::std::io::_print(
                format_args!("Alignment: {0} bytes\n", std::mem::align_of::<Custom>()),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
    {
        {
            ::std::io::_print(format_args!("Provided type: (&str, u8, bool)\n"));
        };
        {
            ::std::io::_print(
                format_args!(
                    "Size: {0} bytes\n",
                    std::mem::size_of::<(&str, u8, bool)>(),
                ),
            );
        };
        {
            ::std::io::_print(
                format_args!(
                    "Alignment: {0} bytes\n",
                    std::mem::align_of::<(&str, u8, bool)>(),
                ),
            );
        };
        {
            ::std::io::_print(format_args!("\n"));
        };
    };
}
```

1. Recap

```rust
use fn_like::print_type_info;
0 implementations
struct Custom {
    field: u32,
    other: Option<&'static str>
}
▶ Run | Debug
fn main() {
    print_type_info!(u32);
    print_type_info!(String);
    print_type_info!(Custom);
    print_type_info!((&str, u8, bool));
}
```

cargo run

```
     Running `target\debug\reca
Provided type: u32
Size: 4 bytes
Alignment: 4 bytes

Provided type: String
Size: 24 bytes
Alignment: 8 bytes

Provided type: Custom
Size: 24 bytes
Alignment: 8 bytes

Provided type: (&str, u8, bool)
Size: 24 bytes
Alignment: 8 bytes
```

2. syn + quote

2. syn + quote

- As we've seen last time, proc-macros are really difficult to manage properly

2. syn + quote

- As we've seen last time, proc-macros are really difficult to manage properly
- You can use proc-macros at many different locations, with different inputs, and generate anything you want

2. syn + quote

- As we've seen last time, proc-macros are really difficult to manage properly
- You can use proc-macros at many different locations, with different inputs, and generate anything you want
- At the same time, the compiler doesn't offer much utility except a stream of tokens

2. syn + quote

- As we've seen last time, proc-macros are really difficult to manage properly
- You can use proc-macros at many different locations, with different inputs, and generate anything you want
- At the same time, the compiler doesn't offer much utility except a stream of tokens
- But thanks to contributors all around the world, we don't need to suffer

2. syn + quote

- As we've seen last time, proc-macros are really difficult to manage properly
- You can use proc-macros at many different locations, with different inputs, and generate anything you want
- At the same time, the compiler doesn't offer much utility except a stream of tokens
- But thanks to contributors all around the world, we don't need to suffer
- To showcase the power of third-party libraries, we will do two things today
  - We will implement a Derive macro that also works on types with generic types and lifetime parameters
  - We will implement an Attribute macro that replaces identifiers

2. syn + quote

- syn is a very useful crate for developing procedural macros
  - It's so useful, at some point you just *have* to use it

2. syn + quote

- syn is a very useful crate for developing procedural macros
- syn allows us to parse TokenStreams into data structures, which are more convenient to use

2. syn + quote

- syn is a very useful crate for developing procedural macros
- syn allows us to parse TokenStreams into data structures, which are more convenient to use
- Additionally, syn offers some useful macros and traits to make development even easier

2. syn + quote

- syn is a very useful crate for developing procedural macros
- syn allows us to parse TokenStreams into data structures, which are more convenient to use
- Additionally, syn offers some useful macros and traits to make development even easier
- syn offers two main ways of working with its Parser
  - By using the parse_macro_input!() macro
  - By using the Parse trait

2. syn + quote

- quote is another useful crate for developing procedural macros
  - It's so convenient, at some point you just *have* to use it

2. syn + quote

- quote is another useful crate for developing procedural macros
- quote has a macro quote! which performs quasi-quoting

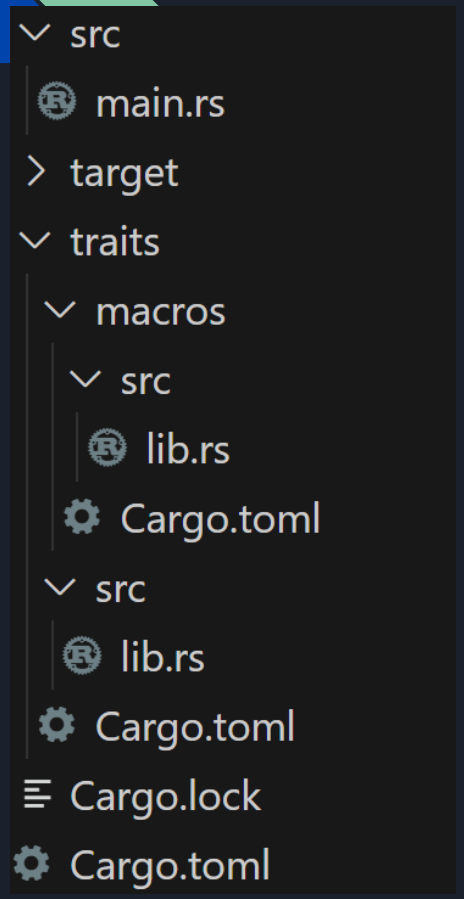
2. syn + quote

- quote is another useful crate for developing procedural macros
- quote has a macro quote! which performs quasi-quoting
- Quasi-quoting means writing code that looks like code to our IDE, but treating it as data

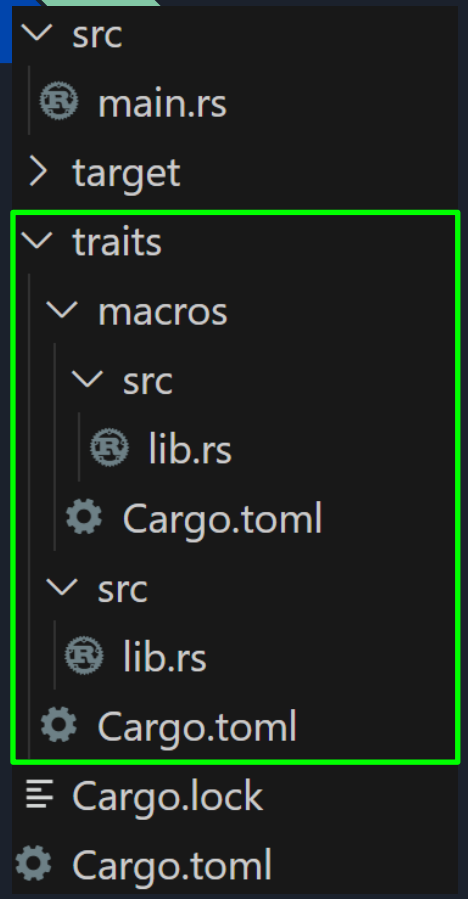
2. syn + quote

- quote is another useful crate for developing procedural macros
- quote has a macro quote! which performs quasi-quoting
- Quasi-quoting means writing code that looks like code to our IDE, but treating it as data
- With quote! we no longer have to format String literals, but instead write normal code

2. syn + quote

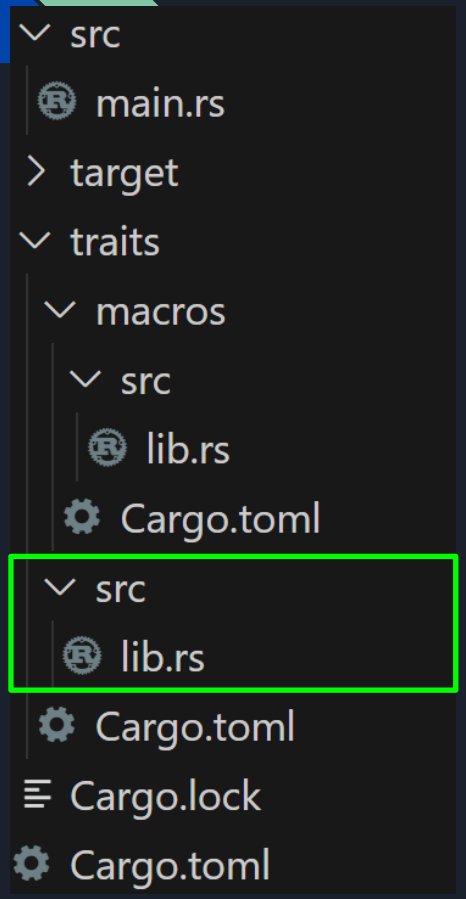

src
main.rs
target
traits
macros
src
lib.rs
Cargo.toml
src
lib.rs
Cargo.toml
Cargo.lock
Cargo.toml

2. syn + quote



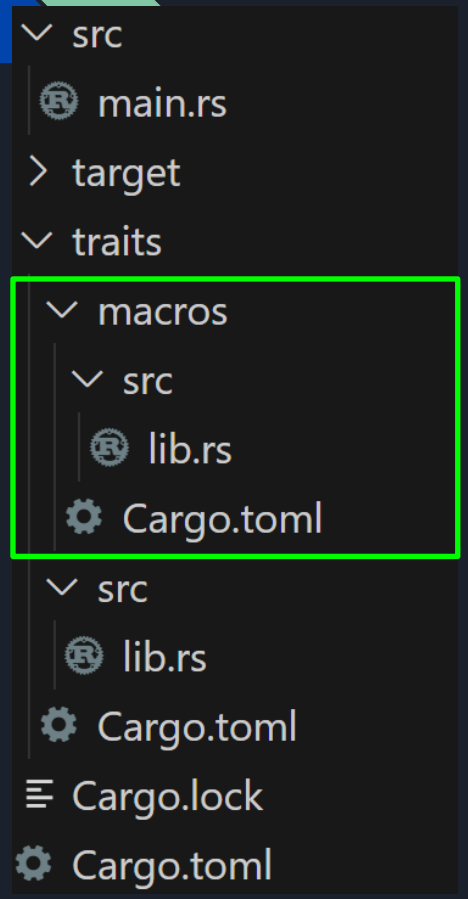
Idea:
→ traits is a normal library crate

2. syn + quote



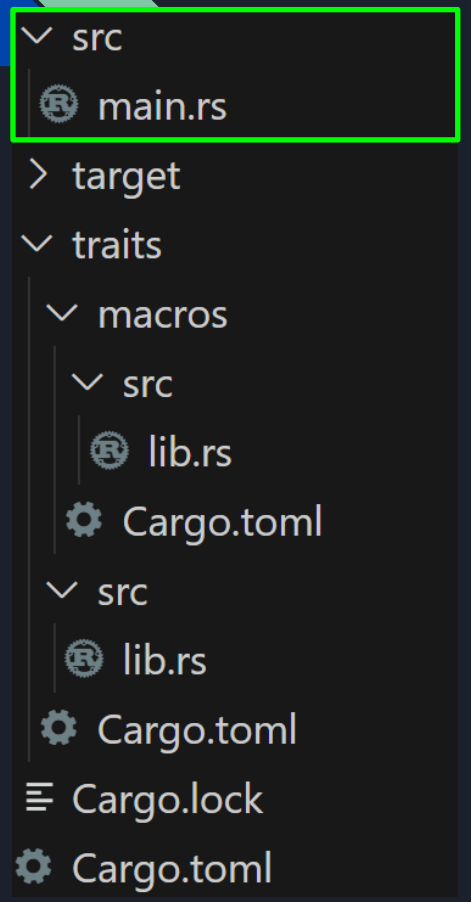
Idea:
→ traits is a normal library crate
→ In there, we define a trait which we want to derive

2. syn + quote



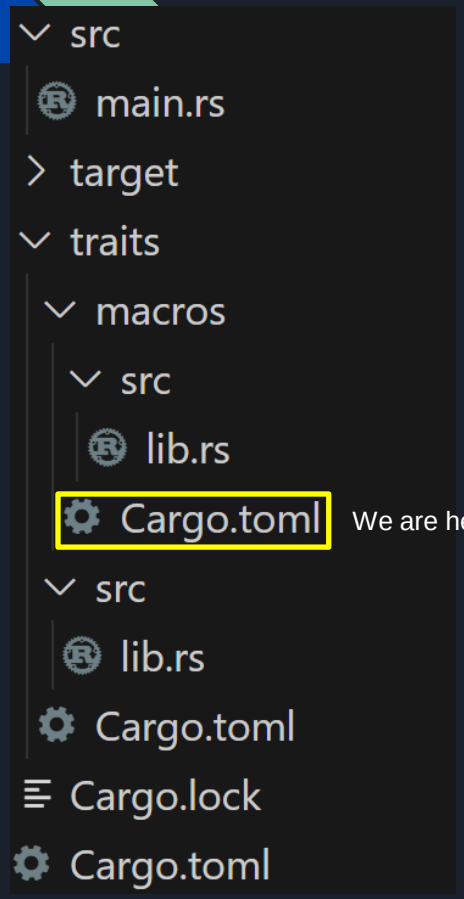
Idea:
→ traits is a normal library crate
→ In there, we define a trait which we want to derive
→ The proc-macro is implemented inside that crate

2. syn + quote



Idea:
→ traits is a normal library crate
→ In there, we define a trait which we want to derive
→ The proc-macro is implemented inside that crate
→ We later derive the trait in our main application

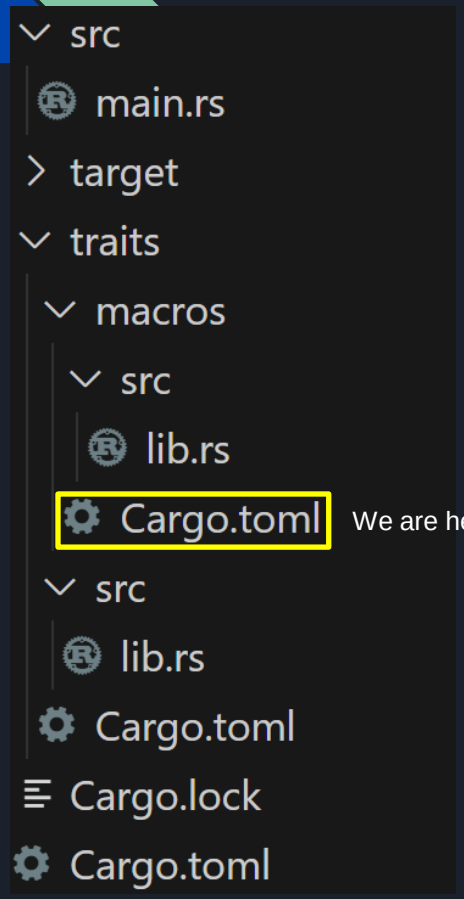
2. syn + quote




```toml
[package]
name = "macros"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
syn = { version="2.0.67", features=["full", "extra-traits"] }
quote = "1.0.36"
proc-macro2 = "1.0.86"
```

Mark crate as proc-macro

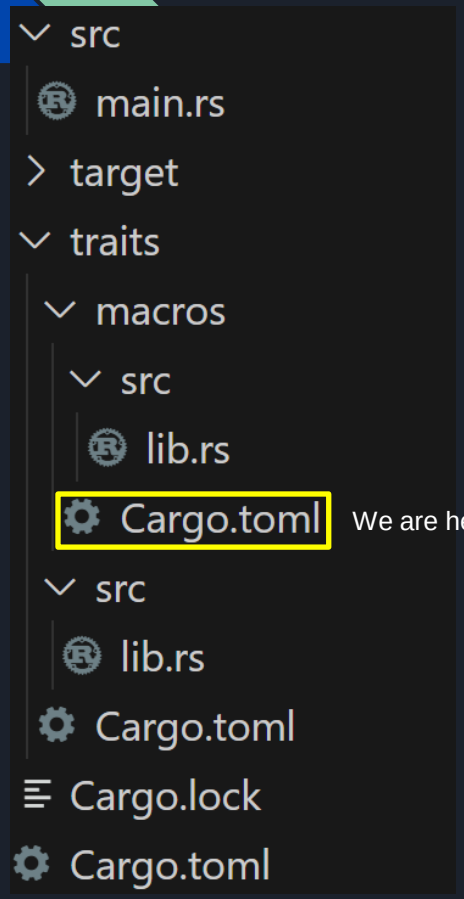
2. syn + quote




```toml
[package]
name = "macros"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
syn = { version="2.0.67", features=["full", "extra-traits"] }
quote = "1.0.36"
proc-macro2 = "1.0.86"
```

syn and quote can be imported like any other crate, additionally we will also need some extra features

2. syn + quote

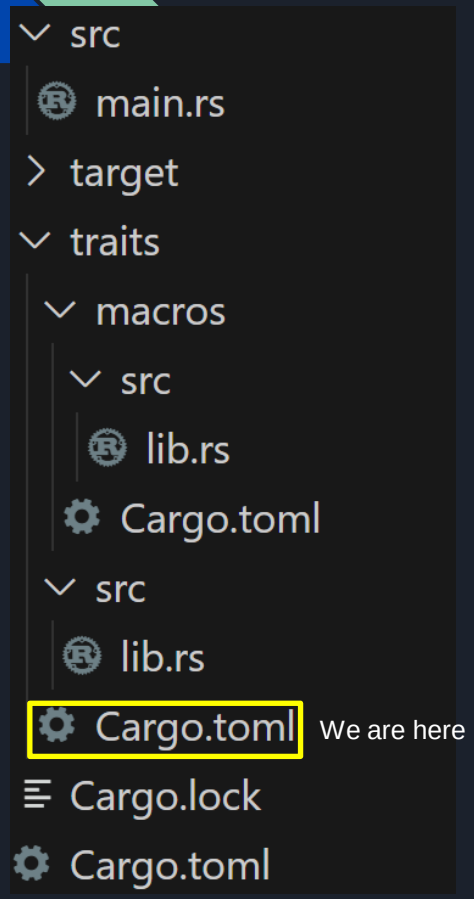



```toml
[package]
name = "macros"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
syn = { version="2.0.67", features=["full", "extra-traits"] }
quote = "1.0.36"
proc-macro2 = "1.0.86"
```

syn and quote can be imported like any other crate, additionally we will also need some extra features

proc-macro2 is needed because syn and quote prefer those types over the normal proc-macro types

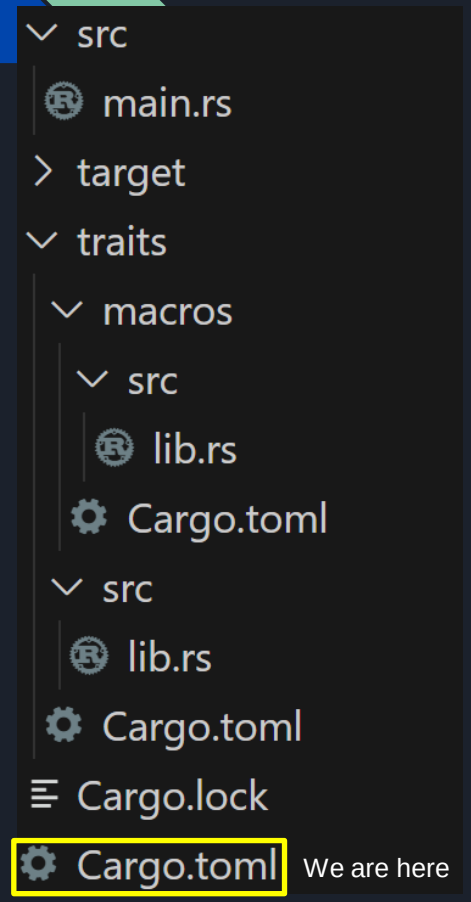
2. syn + quote




```
[package]
name = "traits"
version = "0.1.0"
edition = "2021"

[dependencies]
macros = { path="./macros" }
```

use proc-macro in our library

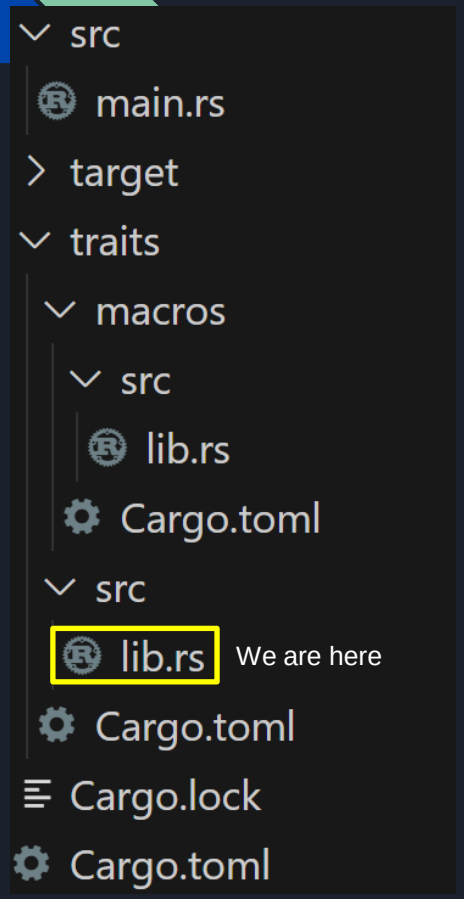
2. syn + quote




```
[package]
name = "derive"
version = "0.1.0"
edition = "2021"

[dependencies]
traits = { path="./traits" }
```

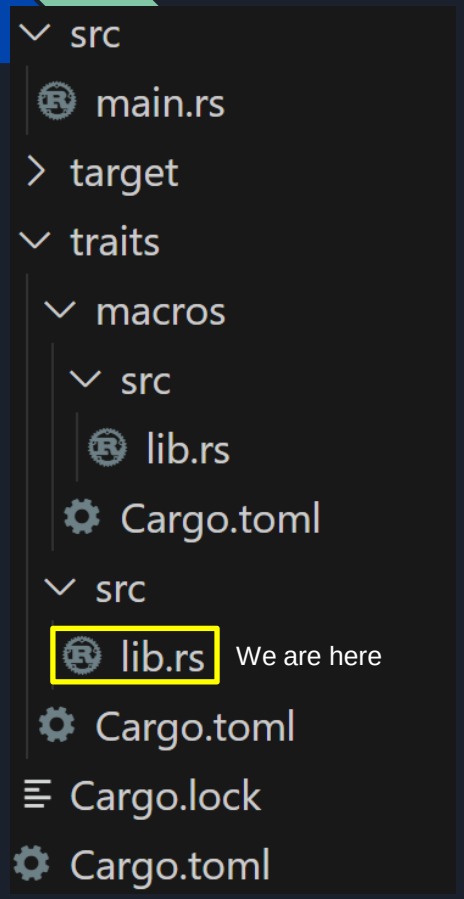
use our library in our main application

2. syn + quote




```rust
pub use macros::TypeInfo as TypeInfo;

1 implementation
pub trait TypeInfo {
    fn get_info() -> (usize, usize);
}
impl TypeInfo for u32 {
    fn get_info() -> (usize, usize) {
        (4, 4)
    }
}
```

2. syn + quote

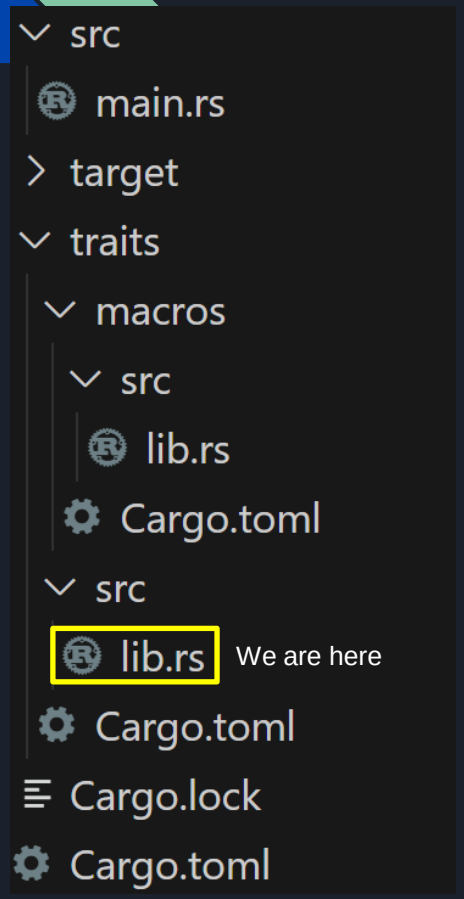



```rust
pub use macros::TypeInfo as TypeInfo;
```

Simple re-export so we can write use traits::TypeInfo instead of use traits::macros::TypeInfo

```rust
1 implementation
pub trait TypeInfo {
    fn get_info() -> (usize, usize);
}
impl TypeInfo for u32 {
    fn get_info() -> (usize, usize) {
        (4, 4)
    }
}
```

2. syn + quote



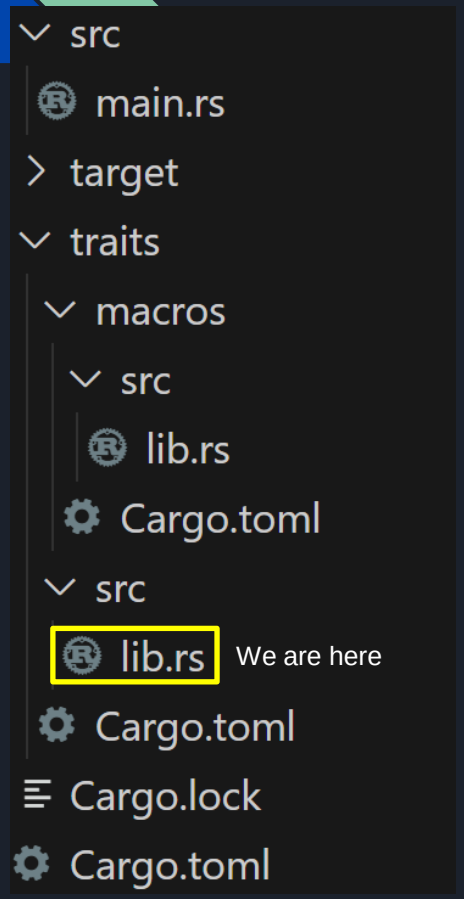

```rust
pub use macros::TypeInfo as TypeInfo;
```

The trait we want to derive, does the same as the macro in the recap, but returns the values instead of printing them

1 implementation

```rust
pub trait TypeInfo {
    fn get_info() -> (usize, usize);
}
impl TypeInfo for u32 {
    fn get_info() -> (usize, usize) {
        (4, 4)
    }
}
```

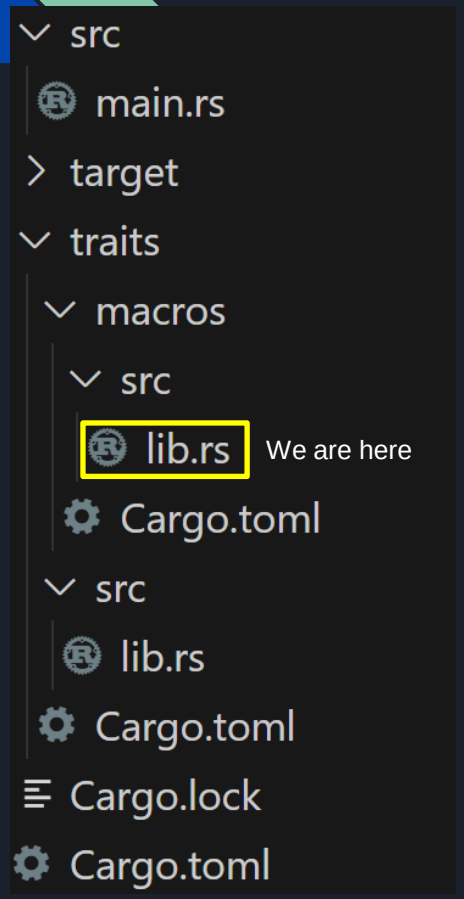
2. syn + quote




```rust
pub use macros::TypeInfo as TypeInfo;

1 implementation
pub trait TypeInfo {
    fn get_info() -> (usize, usize);
}
impl TypeInfo for u32 {
    fn get_info() -> (usize, usize) {
        (4, 4)
    }
}
```

Implementing the trait for all types is tiring, we need derive :^)

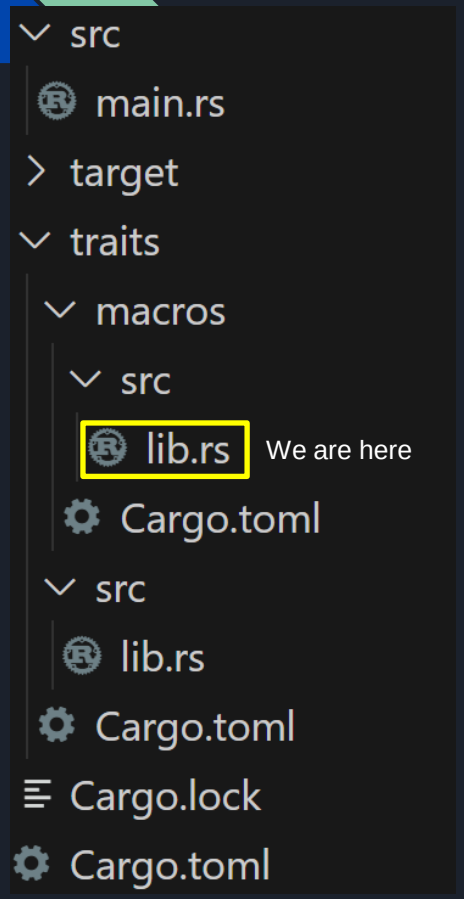
2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
```

```rust
#[derive(TypeInfo)]
0 implementations
struct Custom<'a, T> {
    value: T,
    #[log]
    other: &'a str
}
```

2. syn + quote

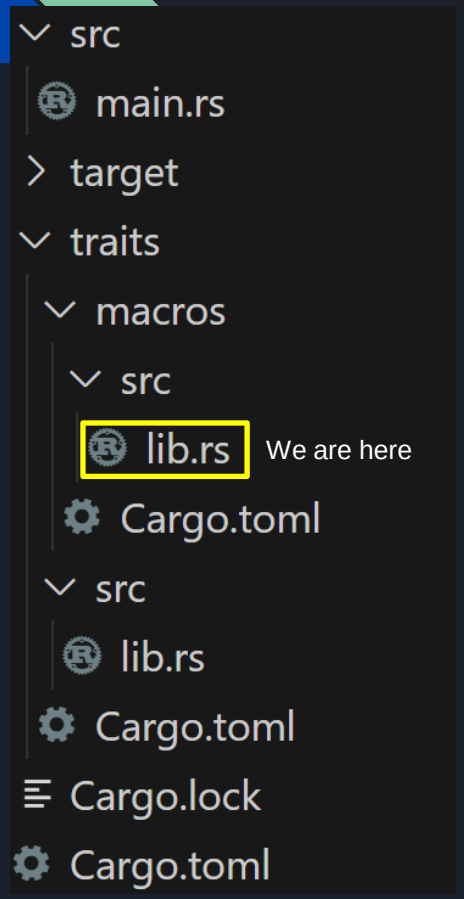


Specify name which we later need to provide

```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
```

```rust
#[derive(TypeInfo)]
0 implementations
struct Custom<'a, T> {
    value: T,
    #[log]
    other: &'a str
}
```

2. syn + quote

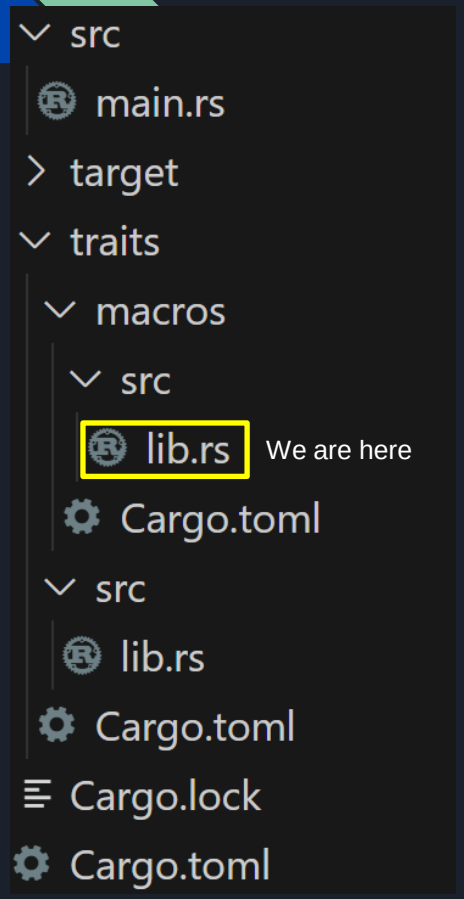



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
```

```rust
#[derive(TypeInfo)]
0 implementations
struct Custom<'a, T> {
    value: T,
    #[log]
    other: &'a str
}
```

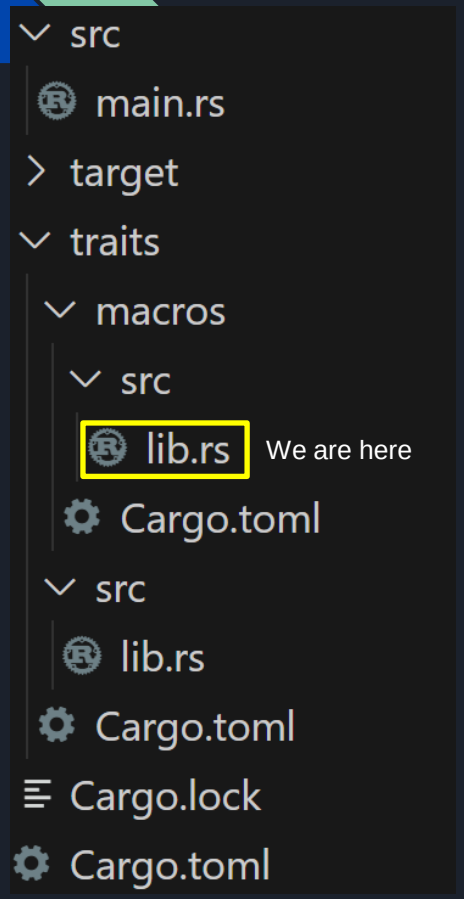
We can also provide extra attributes in derive macros
Here:
When the trait method is called, print Logging field <name> for every tagged field

2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let strukt: DeriveInput = parse_macro_input!(input);
    // or:
    // let strukt = parse_macro_input!(input as DeriveInput);
    "".parse().unwrap()
}
```

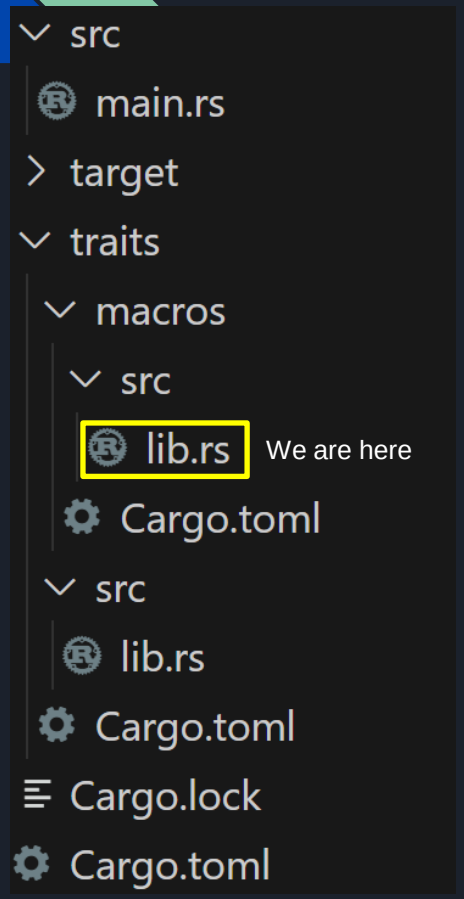
2. syn + quote



parse_macro_input!() tries to parse the desired construct, and triggers a compiler error if it fails

```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let strukt: DeriveInput = parse_macro_input!(input);
    // or:
    // let strukt = parse_macro_input!(input as DeriveInput);
    "".parse().unwrap()
}
```

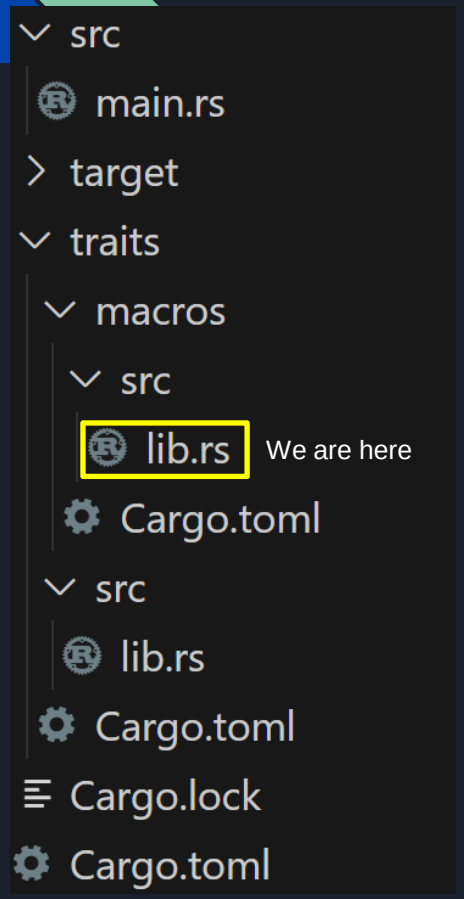
2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let strukt: DeriveInput = parse_macro_input!(input);
    // or:
    // let strukt = parse_macro_input!(input as DeriveInput);
    "".parse().unwrap()
}
```

The desired output can be provided via two means:
→ Specify variable type
→ Use pseudo-typecast

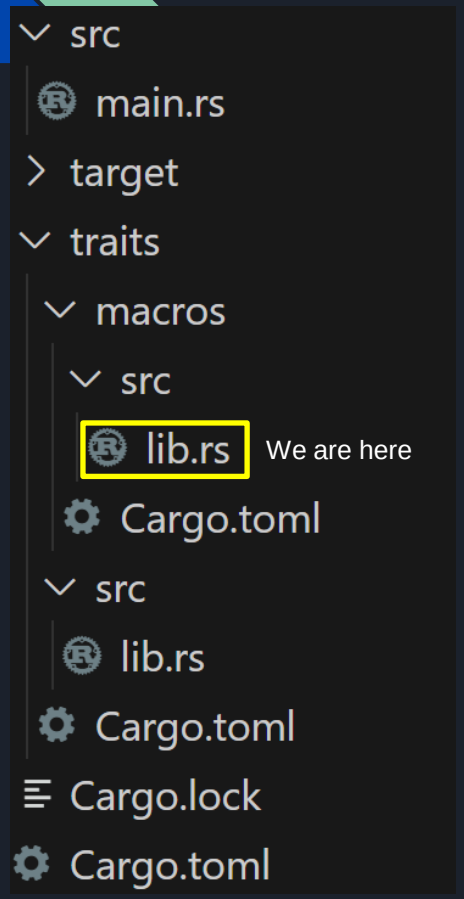
2. syn + quote



Downside of parse_macro_input!():
You can only use it in functions that return a TokenStream

```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let strukt: DeriveInput = parse_macro_input!(input);
    // or:
    // let strukt = parse_macro_input!(input as DeriveInput);
    "".parse().unwrap()
}
```

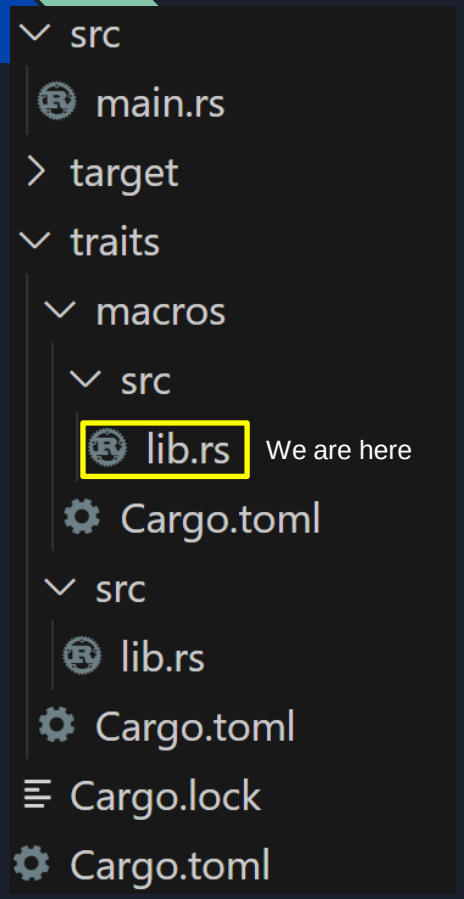
2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let Ok(strukt: DeriveInput) = syn::parse::<DeriveInput>(tokens: input) else {
        panic!("Could not parse input!")
    };
    "".parse().unwrap()
}
```

You can also use the Parser directly, if you want to
Benefit: You have more control, and you can even implement the Parse-trait for your own structs

2. syn + quote

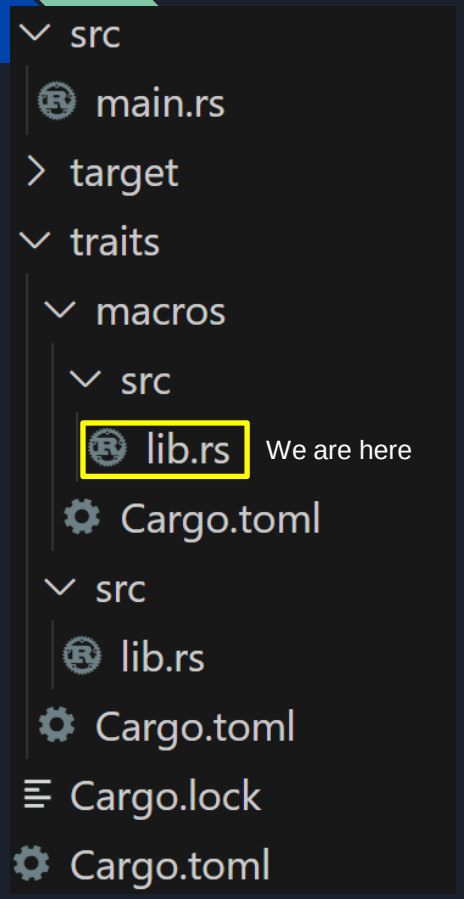



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput {
        attrs: Vec<Attribute>,
        vis: Visibility,
        ident: Ident,
        generics: Generics,
        data: Data
    } = parse_macro_input!(input);
    "".parse().unwrap()
}
```

syn allows us to easily parse the input into any form we want, so we can easily work with it

DeriveInput includes:

2. syn + quote

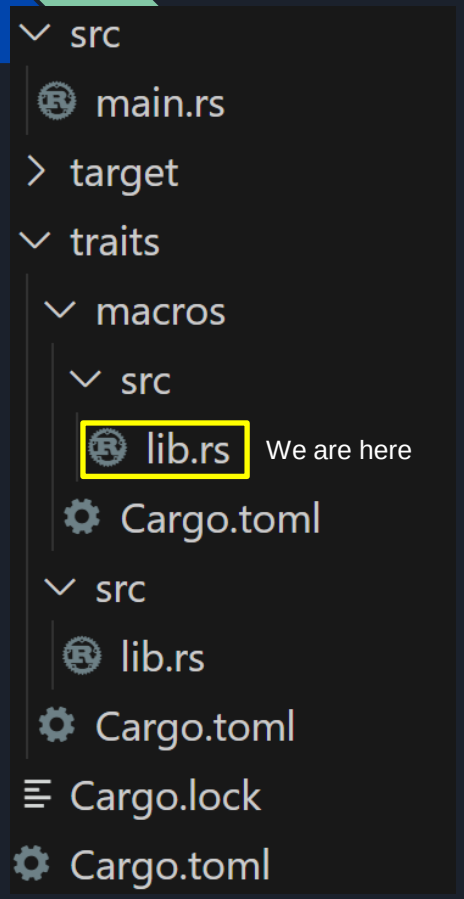



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput {
        attrs: Vec<Attribute>,
        vis: Visibility,
        ident: Ident,
        generics: Generics,
        data: Data
    } = parse_macro_input!(input);
    "".parse().unwrap()
}
```

syn allows us to easily parse the input into any form we want, so we can easily work with it

DeriveInput includes:
→ Attributes that came after the derive call

2. syn + quote

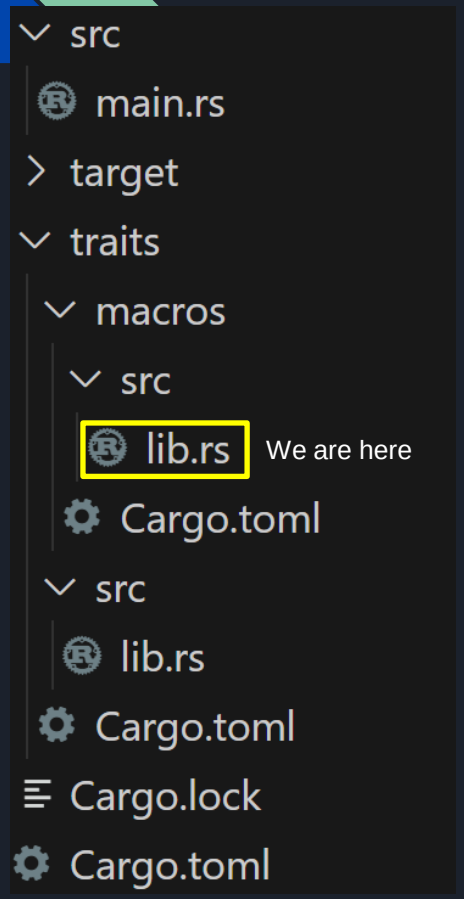



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput {
        attrs: Vec<Attribute>,
        vis: Visibility,
        ident: Ident,
        generics: Generics,
        data: Data
    } = parse_macro_input!(input);
    "".parse().unwrap()
}
```

syn allows us to easily parse the input into any form we want, so we can easily work with it

DeriveInput includes:
→ Attributes that came after the derive call
→ Visibility (public, private)

2. syn + quote

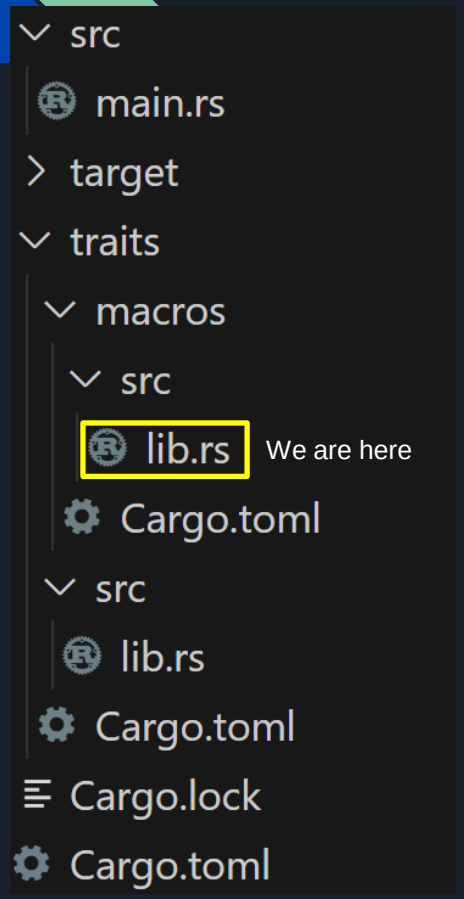



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput {
        attrs: Vec<Attribute>,
        vis: Visibility,
        ident: Ident,
        generics: Generics,
        data: Data
    } = parse_macro_input!(input);
    "".parse().unwrap()
}
```

syn allows us to easily parse the input into any form we want, so we can easily work with it

DeriveInput includes:
→ Attributes that came after the derive call
→ Visibility (public, private)
→ Name of the input (here: struct name)

2. syn + quote

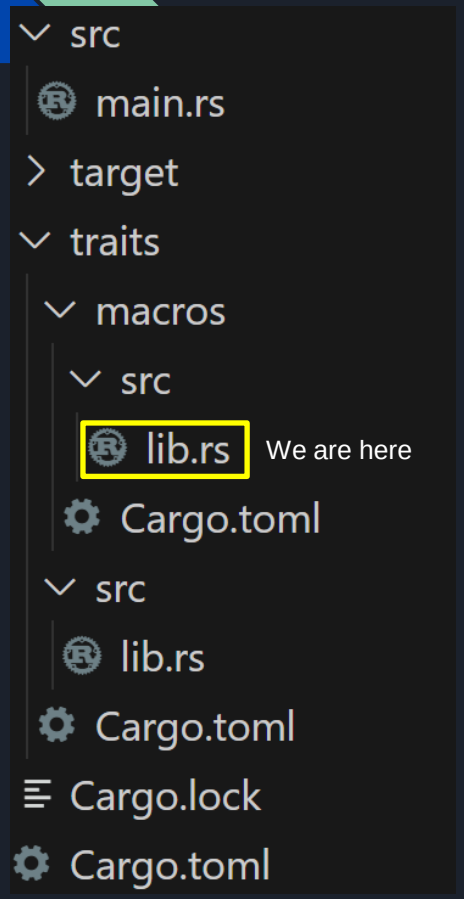



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput {
        attrs: Vec<Attribute>,
        vis: Visibility,
        ident: Ident,
        generics: Generics,
        data: Data
    } = parse_macro_input!(input);
    "".parse().unwrap()
}
```

syn allows us to easily parse the input into any form we want, so we can easily work with it

DeriveInput includes:
→ Attributes that came after the derive call
→ Visibility (public, private)
→ Name of the input (here: struct name)
→ Generic types and lifetime parameters

2. syn + quote

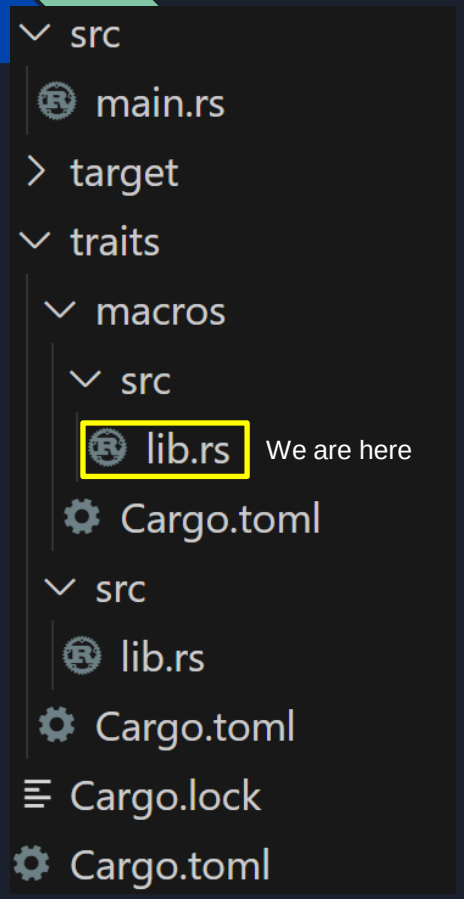



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput {
        attrs: Vec<Attribute>,
        vis: Visibility,
        ident: Ident,
        generics: Generics,
        data: Data
    } = parse_macro_input!(input);
    "".parse().unwrap()
}
```

syn allows us to easily parse the input into any form we want, so we can easily work with it

DeriveInput includes:
→ Attributes that came after the derive call
→ Visibility (public, private)
→ Name of the input (here: struct name)
→ Generic types and lifetime parameters
→ Enum of: Struct Fields, Enum Variants, Union Fields

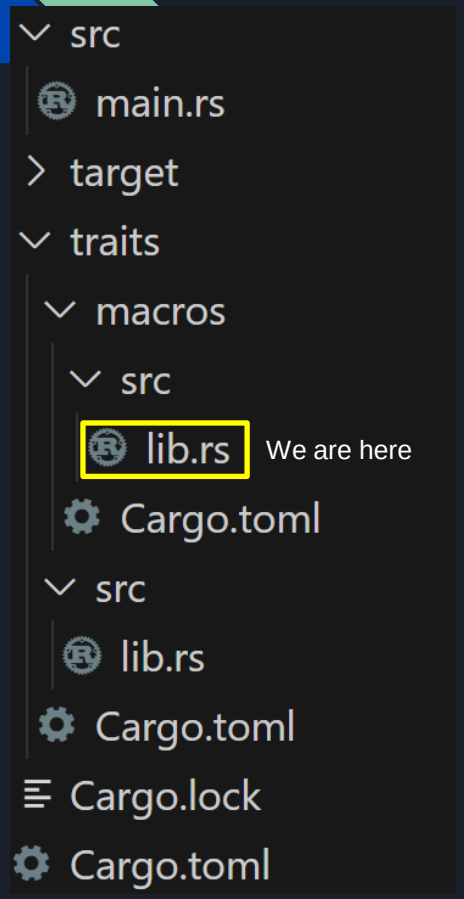
2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility,
    let logged: Vec<&Ident> = gather_fields(&data);
```

We can now easily work with that data

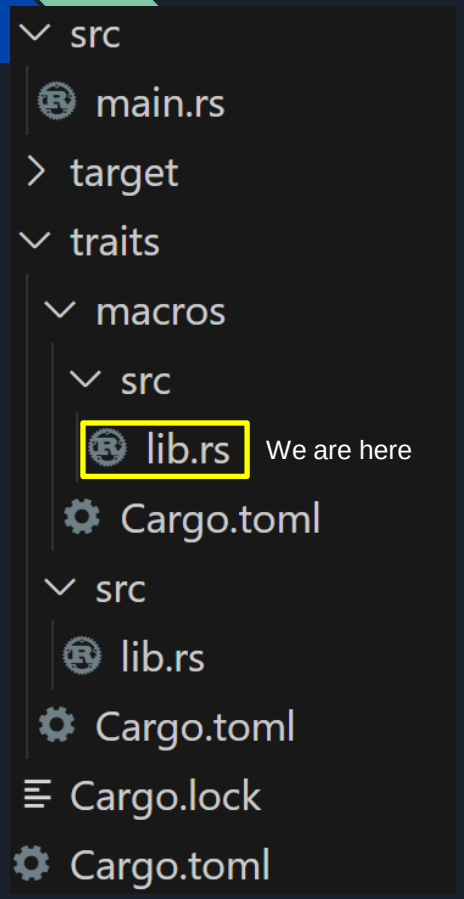
2. syn + quote




```rust
fn gather_fields(data: &Data) -> Vec<&Ident> {
    let Data::Struct(strukt: &DataStruct) = data else {
        panic!("Only structs are supported for now.");
    };
    let mut logged: Vec<&Ident> = Vec::new();
    for field: &Field in &strukt.fields {
        if let Some(attr: &Attribute) = field.attrs.get(index: 0) {
            let path: &Path = attr.path();
            let Ok(ident: &Ident) = path.require_ident() else {
                panic!("Only `log` is a valid attribute")
            };
            let name: String = ident.to_string();
            assert!(name == "log", "Only `log` is a valid attribute.");
            let Some(ident: &Ident) = field.ident.as_ref() else {
                panic!("Tupled structs are not supported yet.")
            };
            logged.push(ident);
        }
    }
    logged
}
```

Using real data structures instead of token streams now makes the whole thing a breeze

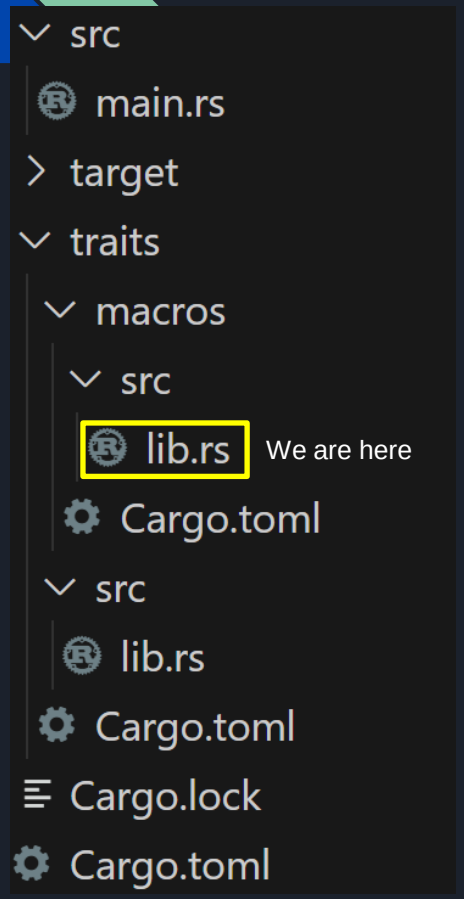
2. syn + quote




```rust
fn gather_fields(data: &Data) -> Vec<&Ident> {
    let Data::Struct(strukt: &DataStruct) = data else {
        panic!("Only structs are supported for now.");
    };
    let mut logged: Vec<&Ident> = Vec::new();
    for field: &Field in &strukt.fields {
        if let Some(attr: &Attribute) = field.attrs.get(index: 0) {
            let path: &Path = attr.path();
            let Ok(ident: &Ident) = path.require_ident() else {
                panic!("Only `log` is a valid attribute")
            };
            let name: String = ident.to_string();
            assert!(name == "log", "Only `log` is a valid attribute.");
            let Some(ident: &Ident) = field.ident.as_ref() else {
                panic!("Tupled structs are not supported yet.")
            };
            logged.push(ident);
        }
    }
    logged
}
```

Expect a struct

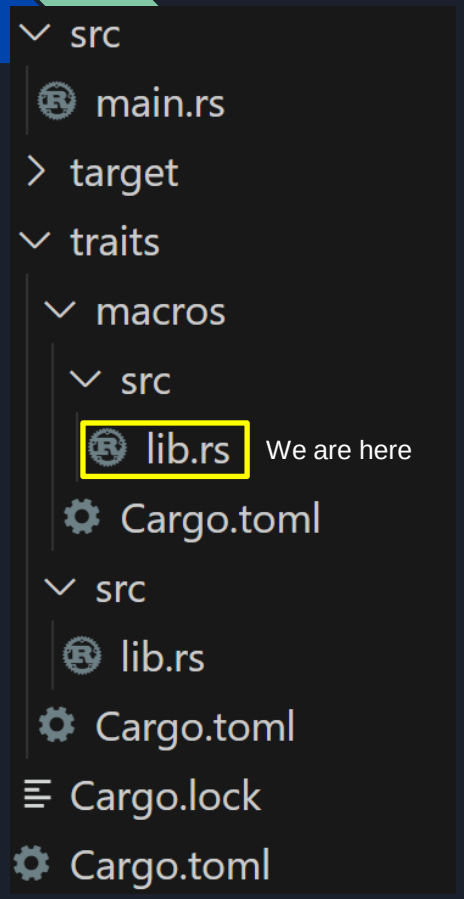
2. syn + quote




```rust
fn gather_fields(data: &Data) -> Vec<&Ident> {
    let Data::Struct(strukt: &DataStruct) = data else {
        panic!("Only structs are supported for now.");
    };
    let mut logged: Vec<&Ident> = Vec::new();
    for field: &Field in &strukt.fields {
        if let Some(attr: &Attribute) = field.attrs.get(index: 0) {
            let path: &Path = attr.path();
            let Ok(ident: &Ident) = path.require_ident() else {
                panic!("Only `log` is a valid attribute")
            };
            let name: String = ident.to_string();
            assert!(name == "log", "Only `log` is a valid attribute.");
            let Some(ident: &Ident) = field.ident.as_ref() else {
                panic!("Tupled structs are not supported yet.")
            };
            logged.push(ident);
        }
    }
    logged
}
```

Iterate over all fields

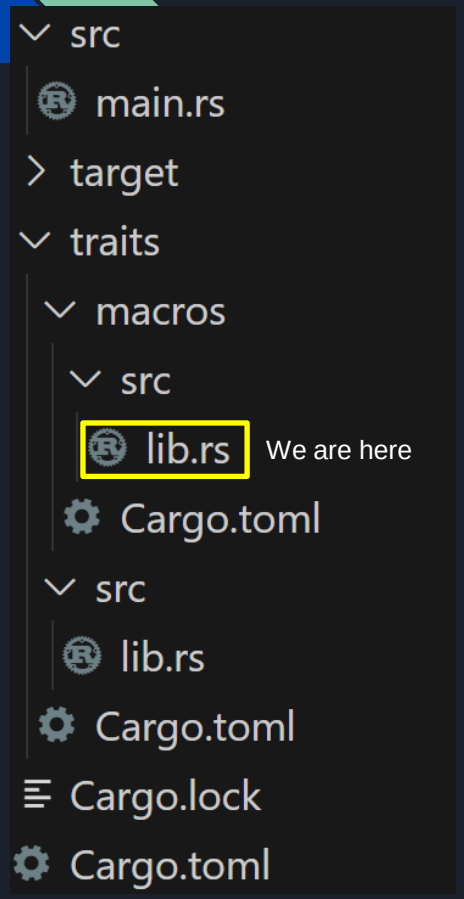
2. syn + quote




```rust
fn gather_fields(data: &Data) -> Vec<&Ident> {
    let Data::Struct(strukt: &DataStruct) = data else {
        panic!("Only structs are supported for now.");
    };
    let mut logged: Vec<&Ident> = Vec::new();
    for field: &Field in &strukt.fields {
        if let Some(attr: &Attribute) = field.attrs.get(index: 0) {
            let path: &Path = attr.path();
            let Ok(ident: &Ident) = path.require_ident() else {
                panic!("Only `log` is a valid attribute")
            };
            let name: String = ident.to_string();
            assert!(name == "log", "Only `log` is a valid attribute.");
            let Some(ident: &Ident) = field.ident.as_ref() else {
                panic!("Tupled structs are not supported yet.")
            };
            logged.push(ident);
        }
    }
    logged
}
```

If the field has an attribute, check if it is the one we want

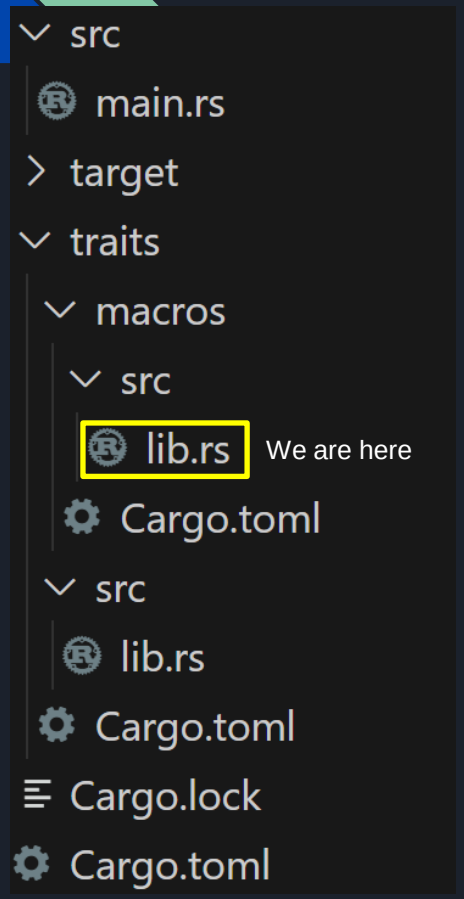
2. syn + quote




```rust
fn gather_fields(data: &Data) -> Vec<&Ident> {
    let Data::Struct(strukt: &DataStruct) = data else {
        panic!("Only structs are supported for now.");
    };
    let mut logged: Vec<&Ident> = Vec::new();
    for field: &Field in &strukt.fields {
        if let Some(attr: &Attribute) = field.attrs.get(index: 0) {
            let path: &Path = attr.path();
            let Ok(ident: &Ident) = path.require_ident() else {
                panic!("Only `log` is a valid attribute")
            };
            let name: String = ident.to_string();
            assert!(name == "log", "Only `log` is a valid attribute.");
            let Some(ident: &Ident) = field.ident.as_ref() else {
                panic!("Tupled structs are not supported yet.")
            };
            logged.push(ident);
        }
    }
    logged
}
```

A Path is a sequence of identifiers, like traits::macros::TypeInfo or path

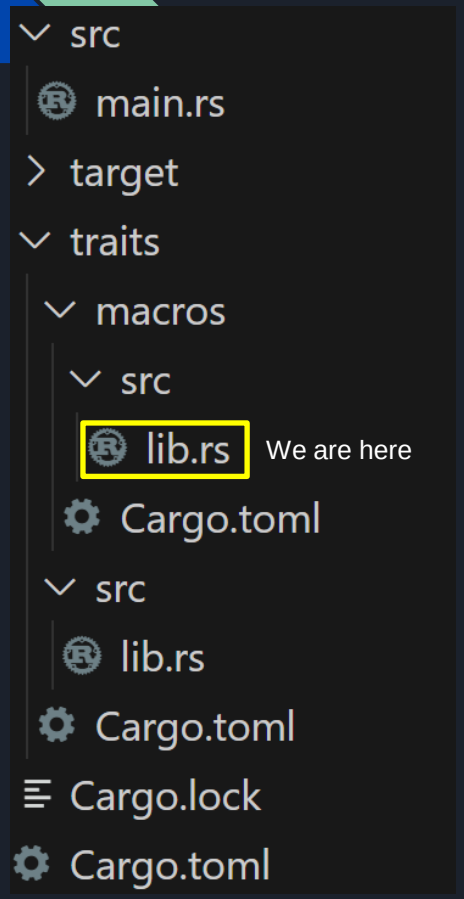
2. syn + quote




```rust
fn gather_fields(data: &Data) -> Vec<&Ident> {
    let Data::Struct(strukt: &DataStruct) = data else {
        panic!("Only structs are supported for now.");
    };
    let mut logged: Vec<&Ident> = Vec::new();
    for field: &Field in &strukt.fields {
        if let Some(attr: &Attribute) = field.attrs.get(index: 0) {
            let path: &Path = attr.path();
            let Ok(ident: &Ident) = path.require_ident() else {
                panic!("Only `log` is a valid attribute")
            };
            let name: String = ident.to_string();
            assert!(name == "log", "Only `log` is a valid attribute.");
            let Some(ident: &Ident) = field.ident.as_ref() else {
                panic!("Tupled structs are not supported yet.")
            };
            logged.push(ident);
        }
    }
    logged
}
```

Make sure it‘s the correct attribute

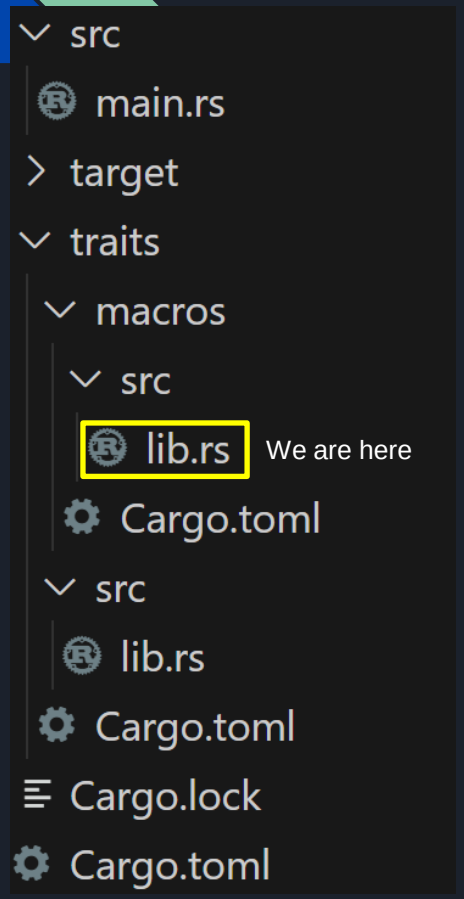
2. syn + quote




```rust
fn gather_fields(data: &Data) -> Vec<&Ident> {
    let Data::Struct(strukt: &DataStruct) = data else {
        panic!("Only structs are supported for now.");
    };
    let mut logged: Vec<&Ident> = Vec::new();
    for field: &Field in &strukt.fields {
        if let Some(attr: &Attribute) = field.attrs.get(index: 0) {
            let path: &Path = attr.path();
            let Ok(ident: &Ident) = path.require_ident() else {
                panic!("Only `log` is a valid attribute")
            };
            let name: String = ident.to_string();
            assert!(name == "log", "Only `log` is a valid attribute.");
            let Some(ident: &Ident) = field.ident.as_ref() else {
                panic!("Tupled structs are not supported yet.")
            };
            logged.push(ident);
        }
    }
    logged
}
```

Add field name to list of fields that need to be logged

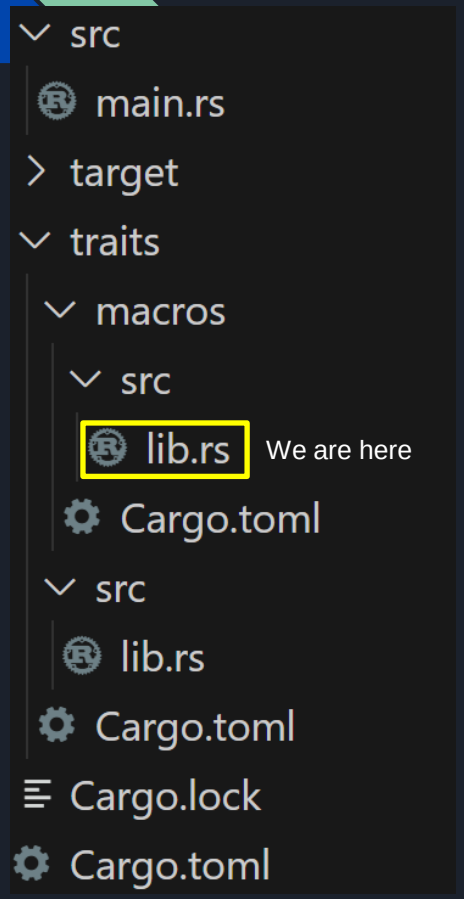
2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged));)*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

logged now contains all field names which were tagged with #[log]

2. syn + quote

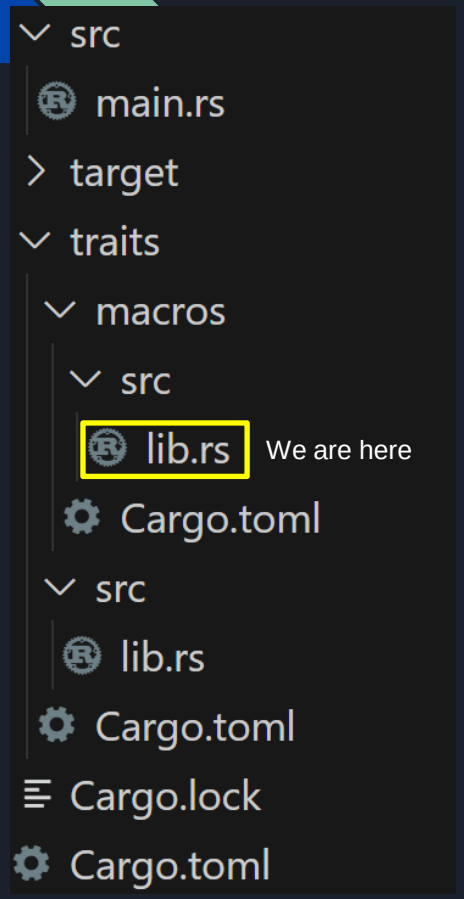



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged)););*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

Now we can focus on this macro

→ As we can see, this macro is a bit more advanced than formatting Strings :^)

2. syn + quote

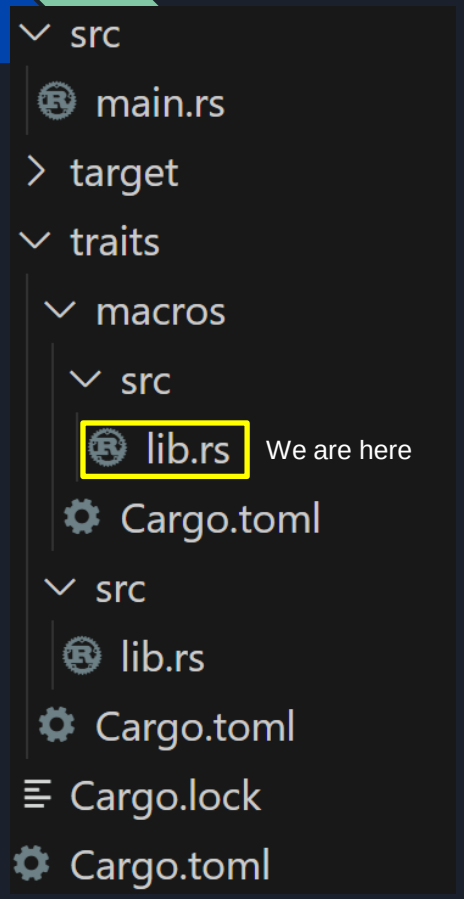



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged));)*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

Quasi-quoting means the code is the data

→ We write code, the IDE can highlight it, but the macro operates on it

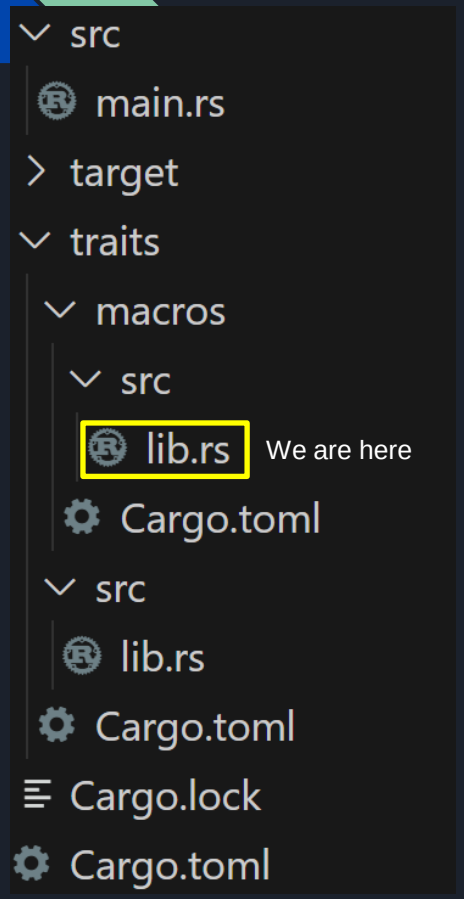
2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged)););*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

quote! is a declarative macro that takes the provided input and turns it into a TokenStream

2. syn + quote

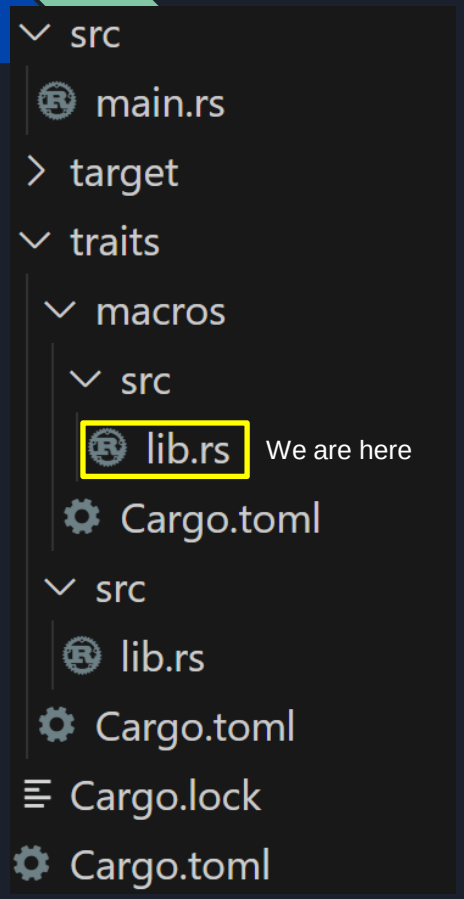



```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged));)*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

quote! is a declarative macro that takes the provided input and turns it into a TokenStream

→ Every normal token is simply passed along

2. syn + quote



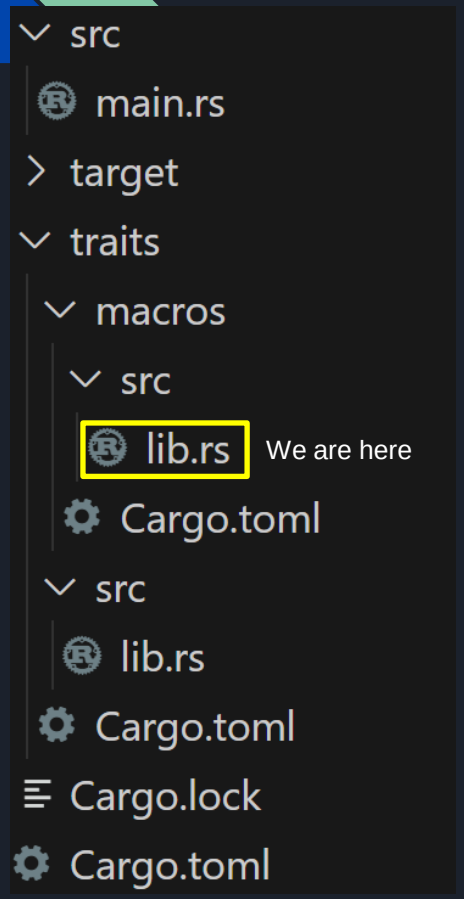

```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged));)*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

quote! is a declarative macro that takes the provided input and turns it into a TokenStream

→ Every normal token is simply passed along

→ quote! interpolates #, inserting the specific value of a variable into the Stream

2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged));)*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

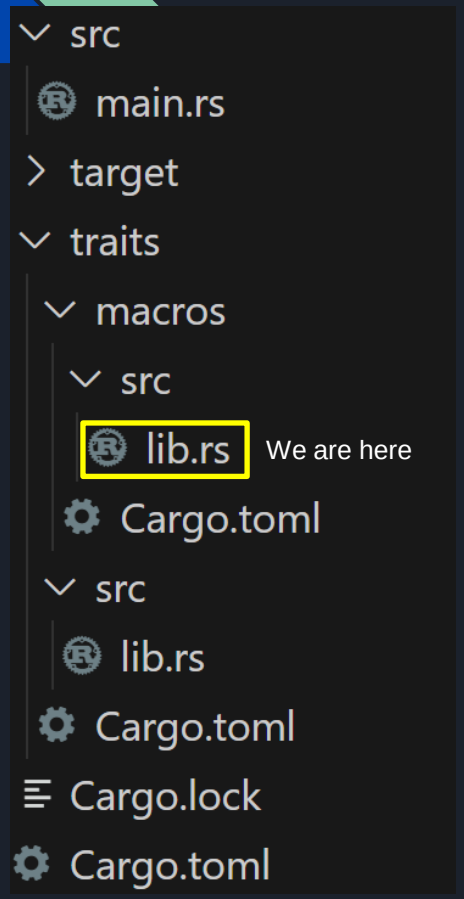
quote! is a declarative macro that takes the provided input and turns it into a TokenStream

→ Every normal token is simply passed along

→ quote! interpolates #, inserting the specific value of a variable into the Stream

→ typ is a token concatenation of the name plus any generics the struct may have

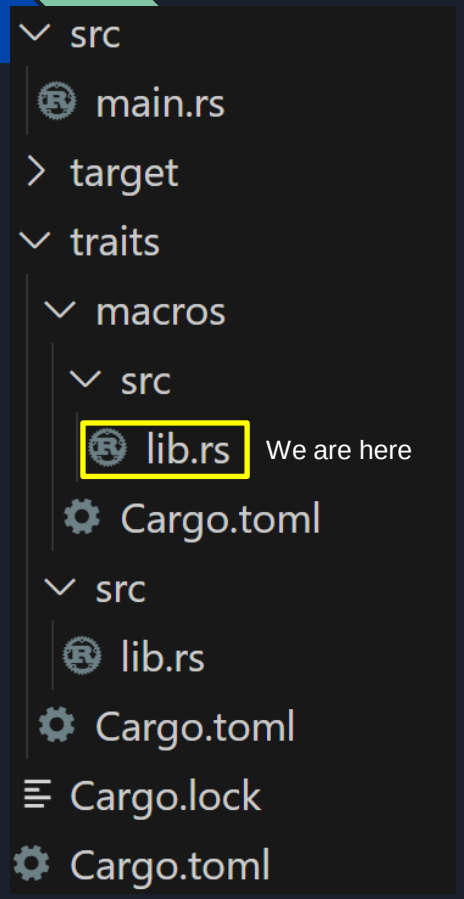
2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged));)*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

quote! also allows repetition, for example when we want to generate code for every element in a collection
→ Here: Emit one println! for every logged field

2. syn + quote




```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged)););*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

→ quote! returns proc_macro2::TokenStream
→ Procedural macros require proc_macro::TokenStream
→ We need to convert the stream first before returning :^)

2. syn + quote

```rust
use traits::TypeInfo;

#[derive(TypeInfo)]
1 implementation
struct Custom<'a, T> {
    value: T,
    #[log]
    other: &'a str
}
```

cargo expand

```rust
use traits::TypeInfo;
struct Custom<'a, T> {
    value: T,
    #[log]
    other: &'a str,
}
impl<'a, T> TypeInfo for Custom<'a, T> {
    fn get_info() -> (usize, usize) {
        {
            ::std::io::_print(format_args!("Logging field `{0}`\n", "other"));
        };
        (std::mem::size_of::<Custom<'a, T>>(), std::mem::align_of::<Custom<'a, T>>())
    }
}
```

```rust
#[proc_macro_derive(TypeInfo, attributes(log))]
pub fn derive_type_info(input: TokenStream) -> TokenStream {
    let DeriveInput { attrs: Vec<Attribute>, vis: Visibility, ident: Ident, g
    let logged: Vec<&Ident> = gather_fields(&data);
    let typ: TokenStream = quote! { #ident #generics };
    quote! {
        impl #generics TypeInfo for #typ {
            fn get_info() -> (usize, usize) {
                #(println!("Logging field `{}`", stringify!(#logged));)*
                ( std::mem::size_of::<#typ>(), std::mem::align_of::<#typ>() )
            }
        }
    }.into()
}
```

2. syn + quote

```rust
#[derive(TypeInfo)]
1 implementation
struct Custom<'a, T> {
    value: T,
    #[log]
    other: &'a str
}
▶ Run | Debug
fn main() {
    println!("{:?}", Custom::<u128>::get_info());
}
```

cargo run

```
Logging field `other`
(32, 8)
```

2. syn + quote

- For the Attribute macro, we will go for two things

2. syn + quote

- For the Attribute macro, we will go for two things
  - An attribute that allows us to replace identifiers in functions
  - An attribute that hijacks string literals, and replaces them with a different text

2. syn + quote

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

2. syn + quote

The macro invocation

```
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

2. syn + quote

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

Replace every string literal inside the function with the given string

2. syn + quote

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

Replace the identifier on the left side with the sequence on the right side

2. syn + quote

```
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

Yes, this code will compile :^)

2. syn + quote

```
#[inject(replace(u8, i8))]
0 implementations
struct Person {
    age: u8,
    name: &'static str
}
0 implementations
struct Cat;
0 implementations
struct Dog;
#[inject(replace(Cat, i32))]
0 implementations
enum Animal {
    Cat(Cat),
    Dog(Dog)
}
```

The attribute will also work on structs and enums

2. syn + quote

```rust
enum Action {
    Payload(LitStr),
    Replace(Ident, ListOfIdents),
}
fn commit_actions(input: TokenStream, actions: &[Action]) -> TokenStream {
    /* TODO */
}
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    /* TODO */
}
#[proc_macro_attribute]
pub fn inject(attr: proc_macro::TokenStream, item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let actions: Vec<Action> = parse_actions(attr: attr.into());
    let output: TokenStream = commit_actions(input: item.into(), &actions);
    output.into()
}
```

2. syn + quote

```rust
enum Action {
    Payload(LitStr),
    Replace(Ident, ListOfIdents),
}
fn commit_actions(input: TokenStream, actions: &[Action]) -> TokenStream {
    /* TODO */
}
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    /* TODO */
}
#[proc_macro_attribute]
pub fn inject(attr: proc_macro::TokenStream, item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let actions: Vec<Action> = parse_actions(attr: attr.into());
    let output: TokenStream = commit_actions(input: item.into(), &actions);
    output.into()
}
```

Our macro will run in two stages:
→ Collect all actions we provided
→ Perform those actions on the given input

2. syn + quote

```rust
enum Action {
    Payload(LitStr),
    Replace(Ident, ListOfIdents),
}
fn commit_actions(input: TokenStream, actions: &[Action]) -> TokenStream {
    /* TODO */
}
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    /* TODO */
}
#[proc_macro_attribute]
pub fn inject(attr: proc_macro::TokenStream, item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let actions: Vec<Action> = parse_actions(attr: attr.into());
    let output: TokenStream = commit_actions(input: item.into(), &actions);
    output.into()
}
```

We‘re doing it right from the start:
→ No strings, instead we‘re using proper AST nodes
→ We‘re using an Enum so we can easily extend the code later

2. syn + quote

```rust
enum Action {
    Payload(LitStr),
    Replace(Ident, ListOfIdents),
}
fn commit_actions(input: TokenStream, actions: &[Action]) -> TokenStream {
    /* TODO */
}
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    /* TODO */
}
#[proc_macro_attribute]
pub fn inject(attr: proc_macro::TokenStream, item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let actions: Vec<Action> = parse_actions(attr: attr.into());
    let output: TokenStream = commit_actions(input: item.into(), &actions);
    output.into()
}
```

Note: Those TokenStreams are from proc_macro2

2. syn + quote

```rust
enum Action {
    Payload(LitStr),
    Replace(Ident, ListOfIdents),
}
fn commit_actions(input: TokenStream, actions: &[Action]) -> TokenStream {
    /* TODO */
}
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    /* TODO */
}
#[proc_macro_attribute]
pub fn inject(attr: proc_macro::TokenStream, item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let actions: Vec<Action> = parse_actions(attr:attr.into());
    let output: TokenStream = commit_actions(input:item.into(), &actions);
    output.into()
}
```

Note: Those TokenStreams are from proc_macro2

We need to do some conversions because the compiler expects proc_macro

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

2. syn + quote

This time we can‘t use parse_macro_input!(),
because we‘re not returning a TokenStream

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

syn again allows us to easily parse whatever we need

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

syn again allows us to easily parse whatever we need
Punctuated → A list of AST nodes, with separators in between

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

syn again allows us to easily parse whatever we need
Punctuated → A list of AST nodes, with separators in between
Meta → One of Path, List, NameValue

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
```

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

syn again allows us to easily parse whatever we need
Punctuated → A list of AST nodes, with separators in between
Meta → One of Path, List, NameValue

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
```

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

List

syn again allows us to easily parse whatever we need
Punctuated → A list of AST nodes, with separators in between
Meta → One of Path, List, NameValue

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
```

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

NameValue

syn again allows us to easily parse whatever we need
Punctuated → A list of AST nodes, with separators in between
Meta → One of Path, List, NameValue

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

syn again allows us to easily parse whatever we need
Punctuated → A list of AST nodes, with separators in between
Meta → One of Path, List, NameValue
Token![,] → The nodes are comma-separated

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

syn again allows us to easily parse whatever we need
Punctuated → A list of AST nodes, with separators in between
Meta → One of Path, List, NameValue
Token![,] → The nodes are comma-separated
→ consume the whole TokenStream

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

syn again allows us to easily parse whatever we need
Punctuated → A list of AST nodes, with separators in between
Meta → One of Path, List, NameValue
Token![,] → The nodes are comma-separated
→ consume the whole TokenStream
→ parse2 works on proc_macro2::TokenStream → Input to parse

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

We now have a parsed data structure which we can work with

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

We now have a parsed data structure which we can work with
Well, after checking if it didn‘t fail :^)

2. syn + quote

```
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

```
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

```
Meta::NameValue { path: Path
Ident { ident: "payload", sp
 }] }, eq_token: Eq, value:
} }
Meta::List { path: Path { le
 { ident: "replace", span: #
, delimiter: MacroDelimiter
, span: #0 bytes(292..295) }
Meta::List { path: Path { le
 { ident: "replace", span: #
, delimiter: MacroDelimiter
t", span: #0 bytes(310..315)
.316) }, Ident { ident: "let
n: #0 bytes(321..324) }] }
Meta::List { path: Path { le
 { ident: "replace", span: #
, delimiter: MacroDelimiter
, span: #0 bytes(339..342) }
43) }, Ident { ident: "u16",
```

2. syn + quote

```
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

```
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    println!("{:?}", attributes);
    todo!()
}
```

We can now easily go over all attributes, and collect them.

```
Meta::NameValue { path: Path
Ident { ident: "payload", sp
 }] }, eq_token: Eq, value:
} }
Meta::List { path: Path { le
 { ident: "replace", span: #
, delimiter: MacroDelimiter
, span: #0 bytes(292..295) }
Meta::List { path: Path { le
 { ident: "replace", span: #
, delimiter: MacroDelimiter
t", span: #0 bytes(310..315)
.316) }, Ident { ident: "let
n: #0 bytes(321..324) }] }
Meta::List { path: Path { le
 { ident: "replace", span: #
, delimiter: MacroDelimiter
, span: #0 bytes(339..342) }
43) }, Ident { ident: "u16",
```

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    let mut actions: Vec<Action> = Vec::with_capacity(attributes.len());
    for attr: Meta in attributes {
        let action: Action = evaluate_attribute(attr);
        actions.push(action);
    }
    actions
}
```

2. syn + quote

```rust
fn parse_actions(attr: TokenStream) -> Vec<Action> {
    let attr_list: Result<Punctuated<Meta, Comma>, …> =
        Punctuated::<Meta, Token![,]>::parse_terminated.parse2(tokens: attr);
    let Ok(attributes: Punctuated<Meta, Comma>) = attr_list else {
        panic!("Could not parse attributes!")
    };
    let mut actions: Vec<Action> = Vec::with_capacity(attributes.len());
    for attr: Meta in attributes {
        let action: Action = evaluate_attribute(attr);
        actions.push(action);
    }
    actions
}
```

For every attribute, turn it into an action and push it to the list

2. syn + quote

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            println!("{:?}", path);
            println!("{:?}", value);
            todo!()
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            println!("{:?}", path);
            println!("{:?}", tokens);
            todo!()
        }
        Meta::Path(..) => panic!("No support for paths yet")
    }
}
```

2. syn + quote

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            println!("{:?}", path);
            println!("{:?}", value);
            todo!()
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            println!("{:?}", path);
            println!("{:?}", tokens);
            todo!()
        }
        Meta::Path(..) => panic!("No support for paths yet")
    }
}
```

Thanks to syn we now work with proper data structures
→ Getting what we want is now a simple thing of working with those structures

2. syn + quote

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            println!("{:?}", path);
            println!("{:?}", value);
            todo!()
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            println!("{:?}", path);
            println!("{:?}", tokens);
            todo!()
        }
        Meta::Path(..) => panic!("No support for paths yet")
    }
}
```

Meta is an enum and comes with three variants,
two of which we'll need

2. syn + quote

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            println!("{:?}", path);
            println!("{:?}", value);
            todo!()
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            println!("{:?}", path);
            println!("{:?}", tokens);
            todo!()
        }
        Meta::Path(..) => panic!("No support for paths yet")
    }
}
```

Thanks to Rust‘s pattern matching, we can easily grab everything we need

2. syn + quote

```
Path { leading_colon: None, segments: [PathSegment { ident: Ident { ident: "replace", span: #0 bytes(284..291) }, arguments: PathArguments::None }] }
TokenStream [Ident { ident: "mut", span: #0 bytes(292..295) }]
Path { leading_colon: None, segments: [PathSegment { ident: Ident { ident: "replace", span: #0 bytes(302..309) }, arguments: PathArguments::None }] }
TokenStream [Ident { ident: "const", span: #0 bytes(310..315) }, Punct { ch: ',', spacing: Alone, span: #0 bytes(315..316) }, Ident { ident: "let", spa
n: #0 bytes(317..320) }, Ident { ident: "mut", span: #0 bytes(321..324) }]
Path { leading_colon: None, segments: [PathSegment { ident: Ident { ident: "replace", span: #0 bytes(331..338) }, arguments: PathArguments::None }] }
TokenStream [Ident { ident: "i32", span: #0 bytes(339..342) }, Punct { ch: ',', spacing: Alone, span: #0 bytes(342..343) }, Ident { ident: "u16", span:
 #0 bytes(344..347) }]
```

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            println!("{:?}", path);
            println!("{:?}", value);
            todo!()
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            println!("{:?}", path);
            println!("{:?}", tokens);
            todo!()
        }
        Meta::Path(..) => panic!("No support for paths yet")
    }
}
```

2. syn + quote

```
Path { leading_colon: None, segments: [PathSegment { ident: Ident { ident: "replace", span: #0 bytes(284..291) }, arguments: PathArguments::None }] }
TokenStream [Ident { ident: "mut", span: #0 bytes(292..295) }]
Path { leading_colon: None, segments: [PathSegment { ident: Ident { ident: "replace", span: #0 bytes(302..309) }, arguments: PathArguments::None }] }
TokenStream [Ident { ident: "const", span: #0 bytes(310..315) }, Punct { ch: ',', spacing: Alone, span: #0 bytes(315..316) }, Ident { ident: "let", spa
n: #0 bytes(317..320) }, Ident { ident: "mut", span: #0 bytes(321..324) }]
Path { leading_colon: None, segments: [PathSegment { ident: Ident { ident: "replace", span: #0 bytes(331..338) }, arguments: PathArguments::None }] }
TokenStream [Ident { ident: "i32", span: #0 bytes(339..342) }, Punct { ch: ',', spacing: Alone, span: #0 bytes(342..343) }, Ident { ident: "u16", span:
 #0 bytes(344..347) }]
```

What we're after is only one more step away and is part of the Path

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            println!("{:?}", path);
            println!("{:?}", value);
            todo!()
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            println!("{:?}", path);
            println!("{:?}", tokens);
            todo!()
        }
        Meta::Path(..) => panic!("No support for paths yet")
    }
}
```

2. syn + quote

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            let ident: &Ident = path.require_ident().unwrap_or_else(op: |e: Error|{ panic!("{e}") });
            match ident.to_string().as_str() {
                "payload" => evaluate_payload(expr: value),
                s: &str => panic!("Unknown key `{s}`."),
            }
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            let command: &Ident = path.require_ident().unwrap_or_else(op: |e: Error|{ panic!("{e}") });
            match command.to_string().as_str() {
                "replace" => evaluate_replace(tokens),
                s: &str => panic!("Unknown command `{s}`."),
            }
        }
        m: Meta => todo!("{m:?}"),
    }
}
```

2. syn + quote

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            let ident: &Ident = path.require_ident().unwrap_or_else(op: |e: Error|{ panic!("{e}") });
            match ident.to_string().as_str() {
                "payload" => evaluate_payload(expr: value),
                s: &str => panic!("Unknown key `{s}`."),
            }
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            let command: &Ident = path.require_ident().unwrap_or_else(op: |e: Error|{ panic!("{e}") });
            match command.to_string().as_str() {
                "replace" => evaluate_replace(tokens),
                s: &str => panic!("Unknown command `{s}`."),
            }
        }
        m: Meta => todo!("{m:?}"),
    }
}
```

Paths are sequences of identifiers, like traits::macros::TypeInfo or payload

require_ident() causes an error if the path is not a single identifier

2. syn + quote

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            let ident: &Ident = path.require_ident().unwrap_or_else(op: |e: Error|{ panic!("{e}") });
            match ident.to_string().as_str() {
                "payload" => evaluate_payload(expr: value),
                s: &str => panic!("Unknown key `{s}`."),
            }
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            let command: &Ident = path.require_ident().unwrap_or_else(op: |e: Error|{ panic!("{e}") });
            match command.to_string().as_str() {
                "replace" => evaluate_replace(tokens),
                s: &str => panic!("Unknown command `{s}`."),
            }
        }
        m: Meta => todo!("{m:?}"),
    }
}
```

We‘re now at a point where we can work with the identifiers!

2. syn + quote

```rust
fn evaluate_attribute(attribute: Meta) -> Action {
    match attribute {
        Meta::NameValue(MetaNameValue { path: Path, eq_token: _, value: Expr }) => {
            let ident: &Ident = path.require_ident().unwrap_or_else(op: |e: Error|{ panic!("{e}") });
            match ident.to_string().as_str() {
                "payload" => evaluate_payload(expr: value),
                s: &str => panic!("Unknown key `{s}`."),
            }
        }
        Meta::List(MetaList { path: Path, delimiter: _, tokens: TokenStream }) => {
            let command: &Ident = path.require_ident().unwrap_or_else(op: |e: Error|{ panic!("{e}") });
            match command.to_string().as_str() {
                "replace" => evaluate_replace(tokens),
                s: &str => panic!("Unknown command `{s}`."),
            }
        }
        m: Meta => todo!("{m:?}"),
    }
}
```

We‘re now at a point where we can work with the identifiers!
For now only payload and replace, but we can always add more later :^)

2. syn + quote

```rust
enum Action {
    Payload(LitStr),
    Replace(Ident, ListOfIdents),
}
fn evaluate_payload(expr: Expr) -> Action {
    let Expr::Lit(ExprLit { lit: Lit::Str(lit: LitStr), .. } ) = expr else {
        panic!("Expected string literal for payload!");
    };
    Action::Payload(lit.clone())
}
```

2. syn + quote

```rust
enum Action {
    Payload(LitStr),
    Replace(Ident, ListOfIdents),
}
fn evaluate_payload(expr: Expr) -> Action {
    let Expr::Lit(ExprLit { lit: Lit::Str(lit: LitStr), .. } ) = expr else {
        panic!("Expected string literal for payload!");
    };
    Action::Payload(lit.clone())
}
```

The payload is pretty simple → All we need is a string literal, which we can easily check thanks to pattern matching and syn

2. syn + quote

```rust
enum Action {
    Payload(LitStr),
    Replace(Ident, ListOfIdents),
}
fn evaluate_payload(expr: Expr) -> Action {
    let Expr::Lit(ExprLit { lit: Lit::Str(lit: LitStr), .. } ) = expr else {
        panic!("Expected string literal for payload!");
    };
    Action::Payload(lit.clone())
}
```

We‘re generating our first actions!

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one identifier.");
    assert!(list.len() <= 2, "Expected at most two identifiers.");
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to replace.");
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

replace is a bit more complicated because of some edge cases

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one identifier.");
    assert!(list.len() <= 2, "Expected at most two identifiers.");
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to replace.");
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

Here we‘re actually calling to our own Parser!
ListOfIdents is a struct declared by us, not by syn.

2. syn + quote

```rust
struct ListOfIdents {
    idents: Vec<Ident>
}
impl Parse for ListOfIdents {
    fn parse(input: syn::parse::ParseStream) -> syn::Result<Self> {
        let mut idents: Vec<Ident> = Vec::new();
        while !(input.is_empty() || input.peek(Token![,])) {
            idents.push(input.call(function: Ident::parse_any)?);
        }
        Ok(Self {
            idents
        })
    }
}
```

2. syn + quote

```rust
struct ListOfIdents {
    idents: Vec<Ident>
}
impl Parse for ListOfIdents {
    fn parse(input: syn::parse::ParseStream) -> syn::Result<Self> {
        let mut idents: Vec<Ident> = Vec::new();
        while !(input.is_empty() || input.peek(Token![,])) {
            idents.push(input.call(function: Ident::parse_any)?);
        }
        Ok(Self {
            idents
        })
    }
}
```

This struct is needed for a simple reason
→ syn does not accept keywords when parsing identifiers
→ let and mut are normal identifiers, but they would be rejected!

2. syn + quote

```rust
struct ListOfIdents {
    idents: Vec<Ident>
}
impl Parse for ListOfIdents {
    fn parse(input: syn::parse::ParseStream) -> syn::Result<Self> {
        let mut idents: Vec<Ident> = Vec::new();
        while !(input.is_empty() || input.peek(Token![,])) {
            idents.push(input.call(function: Ident::parse_any)?);
        }
        Ok(Self {
            idents
        })
    }
}
```

syn allows us to implement custom parsers

2. syn + quote

```rust
struct ListOfIdents {
    idents: Vec<Ident>
}
impl Parse for ListOfIdents {
    fn parse(input: syn::parse::ParseStream) -> syn::Result<Self> {
        let mut idents: Vec<Ident> = Vec::new();
        while !(input.is_empty() || input.peek(Token![,])) {
            idents.push(input.call(function: Ident::parse_any)?);
        }
        Ok(Self {
            idents
        })
    }
}
```

Our parser is pretty simple:
It keeps collecting identifiers until it sees a comma or reaches the end

2. syn + quote

```rust
struct ListOfIdents {
    idents: Vec<Ident>
}
impl Parse for ListOfIdents {
    fn parse(input: syn::parse::ParseStream) -> syn::Result<Self> {
        let mut idents: Vec<Ident> = Vec::new();
        while !(input.is_empty() || input.peek(Token![,])) {
            idents.push(input.call(function: Ident::parse_any)?);
        }
        Ok(Self {
            idents
        })
    }
}
```

This way, we‘re able to call other methods of the parser
→ Ident::parse_any does accept keywords

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one identifier.");
    assert!(list.len() <= 2, "Expected at most two identifiers.");
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to replace.");
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

At this point, list is a parsed comma-separated sequence of list of identifiers

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one identifier.");
    assert!(list.len() <= 2, "Expected at most two identifiers.");
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to replace.");
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

Checking if the provided input is in the correct format

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one identifier.");
    assert!(list.len() <= 2, "Expected at most two identifiers.");
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to replace.");
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

Generate an action!

Note: In the else-branch we technically delete the identifier

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one ident
    assert!(list.len() <= 2, "Expected at most two ident
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to r
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one ident
    assert!(list.len() <= 2, "Expected at most two ident
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to r
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one ident
    assert!(list.len() <= 2, "Expected at most two ident
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to r
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

2. syn + quote

```rust
fn evaluate_replace(tokens: TokenStream) -> Action {
    let maybe: Result<Punctuated<ListOfIdents, …>, …> =
        Punctuated::<ListOfIdents, Token![,]>::parse_terminated.parse2(tokens);
    let Ok(list: Punctuated<ListOfIdents, …>) = maybe else {
        panic!("Error: {}", maybe.err().unwrap());
    };
    assert!(list.len() > 0, "Expected at least one ident
    assert!(list.len() <= 2, "Expected at most two ident
    let src: ListOfIdents = list[0].clone();
    if src.idents.len() != 1 {
        panic!("`replace()` requires one identifier to r
    }
    if list.len() == 2 {
        Action::Replace(src.idents[0].clone(), list[1].clone())
    } else {
        Action::Replace(src.idents[0].clone(), ListOfIdents::empty())
    }
}
```

```rust
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
```

We don‘t replace mut with anything
→ We delete all occurences of mut (except the ones we inserted)

2. syn + quote

```rust
#[proc_macro_attribute]
pub fn inject(attr: proc_macro::TokenStream, item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let actions: Vec<Action> = parse_actions(attr: attr.into());
```

We‘re done parsing the attributes!
After this step, our Vector looks like this:

```
Replace(Ident { ident: "u8", span: #0 bytes(89..91) }, ListOfIdents { idents: [Ident {
Replace(Ident { ident: "Cat", span: #0 bytes(194..197) }, ListOfIdents { idents: [Ident
Payload(LitStr { token: "meow!" })
Replace(Ident { ident: "mut", span: #0 bytes(292..295) }, ListOfIdents { idents: [] })
Replace(Ident { ident: "const", span: #0 bytes(310..315) }, ListOfIdents { idents: [Ide
Replace(Ident { ident: "i32", span: #0 bytes(339..342) }, ListOfIdents { idents: [Ident
```

2. syn + quote

```rust
#[proc_macro_attribute]
pub fn inject(attr: proc_macro::TokenStream, item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let actions: Vec<Action> = parse_actions(attr: attr.into());
    let output: TokenStream = commit_actions(input: item.into(), &actions);
    output.into()
}
```

We can now focus on applying those rules to the input

```
Replace(Ident { ident: "u8", span: #0 bytes(89..91) }, ListOfIdents { idents: [Ident {
Replace(Ident { ident: "Cat", span: #0 bytes(194..197) }, ListOfIdents { idents: [Ident
Payload(LitStr { token: "meow!" })
Replace(Ident { ident: "mut", span: #0 bytes(292..295) }, ListOfIdents { idents: [] })
Replace(Ident { ident: "const", span: #0 bytes(310..315) }, ListOfIdents { idents: [Ide
Replace(Ident { ident: "i32", span: #0 bytes(339..342) }, ListOfIdents { idents: [Ident
```

2. syn + quote

```
fn commit_actions(input: TokenStream, actions: &[Action]) -> TokenStream {
    let mut output: TokenStream = TokenStream::new();
    for tkn: TokenTree in input {
        match &tkn {
            TokenTree::Ident(ident: &Ident) => {…
            TokenTree::Literal(lit: &Literal) => {…
            TokenTree::Punct(_) => output.append(token: tkn),
            TokenTree::Group(group: &Group) => {…
        }
    }
    output
} fn commit_actions
```

2. syn + quote

```rust
fn commit_actions(input: TokenStream, actions: &[Action]) -> TokenStream {
    let mut output: TokenStream = TokenStream::new();
    for tkn: TokenTree in input {
        match &tkn {
            TokenTree::Ident(ident: &Ident) => {…
            TokenTree::Literal(lit: &Literal) => {…
            TokenTree::Punct(_) => output.append(token: tkn),
            TokenTree::Group(group: &Group) => {…
        }
    }
    output
} fn commit_actions
```

Idea: We'll slowly build up a TokenStream ourselves.

As we're only taking the input and slightly modify the tokens, we don't care whether it's a function or a struct, we don't need to parse anything.

Similarly, quote! is also not needed here.

2. syn + quote

```rust
fn commit_actions(input: TokenStream, actions: &[Action]) -> TokenStream {
    let mut output: TokenStream = TokenStream::new();
    for tkn: TokenTree in input {
        match &tkn {
            TokenTree::Ident(ident: &Ident) => {…
            TokenTree::Literal(lit: &Literal) => {…
            TokenTree::Punct(_) => output.append(token: tkn),
            TokenTree::Group(group: &Group) => {…
        }
    }
    output
} fn commit_actions
```

Easy case: We never replace any Puncts, as they are only simple characters such as & % =

2. syn + quote

```rust
TokenTree::Ident(ident: &Ident) => {
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if let Action::Replace(s: &Ident, _) = a {
            if s == ident { possible.push(a); }
        }
    }
    if possible.is_empty() {
        output.append(token: tkn);
    } else {
        assert!(possible.len() == 1, "Multiple replacements are not supported.");
        let Action::Replace(_, ListOfIdents { idents: &Vec<Ident> }) = possible[0] else {
            unreachable!()
        };
        for id: &Ident in idents {
            output.append(token: id.clone());
        }
    }
}
```

Identifiers are special, they can be replaced

2. syn + quote

```rust
TokenTree::Ident(ident: &Ident) => {
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if let Action::Replace(s: &Ident, _) = a {
            if s == ident { possible.push(a); }
        }
    }
    if possible.is_empty() {
        output.append(token: tkn);
    } else {
        assert!(possible.len() == 1, "Multiple replacements are not supported.");
        let Action::Replace(_, ListOfIdents { idents: &Vec<Ident> }) = possible[0] else {
            unreachable!()
        };
        for id: &Ident in idents {
            output.append(token: id.clone());
        }
    }
}
```

Find possible replacements for our identifier

2. syn + quote

```rust
TokenTree::Ident(ident: &Ident) => {
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if let Action::Replace(s: &Ident, _) = a {
            if s == ident { possible.push(a); }
        }
    }
    if possible.is_empty() {
        output.append(token: tkn);
    } else {
        assert!(possible.len() == 1, "Multiple replacements are not supported.");
        let Action::Replace(_, ListOfIdents { idents: &Vec<Ident> }) = possible[0] else {
            unreachable!()
        };
        for id: &Ident in idents {
            output.append(token: id.clone());
        }
    }
}
```

No replacement? Just add the token to the output

2. syn + quote

```rust
TokenTree::Ident(ident: &Ident) => {
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if let Action::Replace(s: &Ident, _) = a {
            if s == ident { possible.push(a); }
        }
    }
    if possible.is_empty() {
        output.append(token: tkn);
    } else {
        assert!(possible.len() == 1, "Multiple replacements are not supported.");
        let Action::Replace(_, ListOfIdents { idents: &Vec<Ident> }) = possible[0] else {
            unreachable!()
        };
        for id: &Ident in idents {
            output.append(token: id.clone());
        }
    }
}
```

Otherwise, replace with the given identifiers

2. syn + quote

```
TokenTree::Literal(lit: &Literal) => { Literals are special, they can be replaced too
    let Ok(_) = syn::parse2::<LitStr>(tokens: lit.into_token_stream()) else {
        output.append(token: tkn);
        continue;
    };
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if matches!(a, Action::Payload(_)) {
            possible.push(a);
        }
    }
    assert!(possible.len() <= 1, "Multiple payloads are not supported.");
    if let Some(Action::Payload(payload: &LitStr)) = possible.get(index: 0) {
        output.append(payload.token());
    } else {
        output.append(token: tkn);
    }
}
```

2. syn + quote

```
TokenTree::Literal(lit: &Literal) => {
    let Ok(_) = syn::parse2::<LitStr>(tokens: lit.into_token_stream()) else {
        output.append(token: tkn);
        continue;
    };
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if matches!(a, Action::Payload(_)) {
            possible.push(a);
        }
    }
    assert!(possible.len() <= 1, "Multiple payloads are not supported.");
    if let Some(Action::Payload(payload: &LitStr)) = possible.get(index: 0) {
        output.append(payload.token());
    } else {
        output.append(token: tkn);
    }
}
```

But not all literals! We only replace string literals.

2. syn + quote

```
TokenTree::Literal(lit: &Literal) => {
    let Ok(_) = syn::parse2::<LitStr>(tokens: lit.into_token_stream()) else {
        output.append(token: tkn);
        continue;
    };
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if matches!(a, Action::Payload(_)) {
            possible.push(a);
        }
    }
    assert!(possible.len() <= 1, "Multiple payloads are not supported.");
    if let Some(Action::Payload(payload: &LitStr)) = possible.get(index: 0) {
        output.append(payload.token());
    } else {
        output.append(token: tkn);
    }
}
```

Find a payload

2. syn + quote

```
TokenTree::Literal(lit: &Literal) => {
    let Ok(_) = syn::parse2::<LitStr>(tokens: lit.into_token_stream()) else {
        output.append(token: tkn);
        continue;
    };
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if matches!(a, Action::Payload(_)) {
            possible.push(a);
        }
    }
    assert!(possible.len() <= 1, "Multiple payloads are not supported.");
    if let Some(Action::Payload(payload: &LitStr)) = possible.get(index: 0) {
        output.append(payload.token());
    } else {
        output.append(token: tkn);
    }
}
```

No payload found? Just add the literal back to the output

2. syn + quote

```
TokenTree::Literal(lit: &Literal) => {
    let Ok(_) = syn::parse2::<LitStr>(tokens: lit.into_token_stream()) else {
        output.append(token: tkn);
        continue;
    };
    let mut possible: Vec<&Action> = Vec::new();
    for a: &Action in actions {
        if matches!(a, Action::Payload(_)) {
            possible.push(a);
        }
    }
    assert!(possible.len() <= 1, "Multiple payloads are not supported.");
    if let Some(Action::Payload(payload: &LitStr)) = possible.get(index: 0) {
        output.append(payload.token());
    } else {
        output.append(token: tkn);
    }
}
```

Otherwise, insert the given string!

2. syn + quote

Groups require recursion → We want to replace *everything*

```
TokenTree::Group(group: &Group) => {
    let delim: Delimiter = group.delimiter();
    let elems: TokenStream = group.stream();
    let gr_out: TokenStream = commit_actions(input: elems, actions);
    let new: Group = Group::new(delimiter: delim, stream: gr_out);
    output.append(token: new);
}
```

2. syn + quote

```rust
#[proc_macro_attribute]
pub fn inject(attr: proc_macro::TokenStream, item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let actions: Vec<Action> = parse_actions(attr: attr.into());
    let output: TokenStream = commit_actions(input: item.into(), &actions);
    output.into()
}
```

And we‘re done! We can now test this macro!

```rust
#[inject(replace(u8, i8))]
0 implementations
struct Person {
    age: u8,
    name: &'static str
}
0 implementations
struct Cat;
0 implementations
struct Dog;
#[inject(replace(Cat, i32))]
0 implementations
enum Animal {
    Cat(Cat),
    Dog(Dog)
}
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

```rust
#[inject(replace(u8, i8))]
0 implementations
struct Person {
    age: u8,
    name: &'static str
}
0 implementations
struct Cat;
0 implementations
struct Dog;
#[inject(replace(Cat, i32))]
0 implementations
enum Animal {
    Cat(Cat),
    Dog(Dog)
}
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

cargo expand

```rust
struct Person {
    age: i8,
    name: &'static str,
}
struct Cat;
struct Dog;
enum Animal {
    i32(i32),
    Dog(Dog),
}
fn main() {
    let person: Person = Person { age: 24, name: "meow!" };
    let mut a: u16 = 1;
    a = 300;
    {
        ::std::io::_print(format_args!("meow!\n"));
    };
    {
        ::std::io::_print(format_args!("meow!\n"));
    };
}
```

```rust
#[inject(replace(u8, i8))]
0 implementations
struct Person {
    age: u8,
    name: &'static str
}
0 implementations
struct Cat;
0 implementations
struct Dog;
#[inject(replace(Cat, i32))]
0 implementations
enum Animal {
    Cat(Cat),
    Dog(Dog)
}
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

cargo expand

```rust
struct Person {
    age: i8,
    name: &'static str,
}
struct Cat;
struct Dog;
enum Animal {
    i32(i32),
    Dog(Dog),
}
fn main() {
    let person: Person = Person { age: 24, name: "meow!" };
    let mut a: u16 = 1;
    a = 300;
    {
        ::std::io::_print(format_args!("meow!\n"));
    };
    {
        ::std::io::_print(format_args!("meow!\n"));
    };
}
```

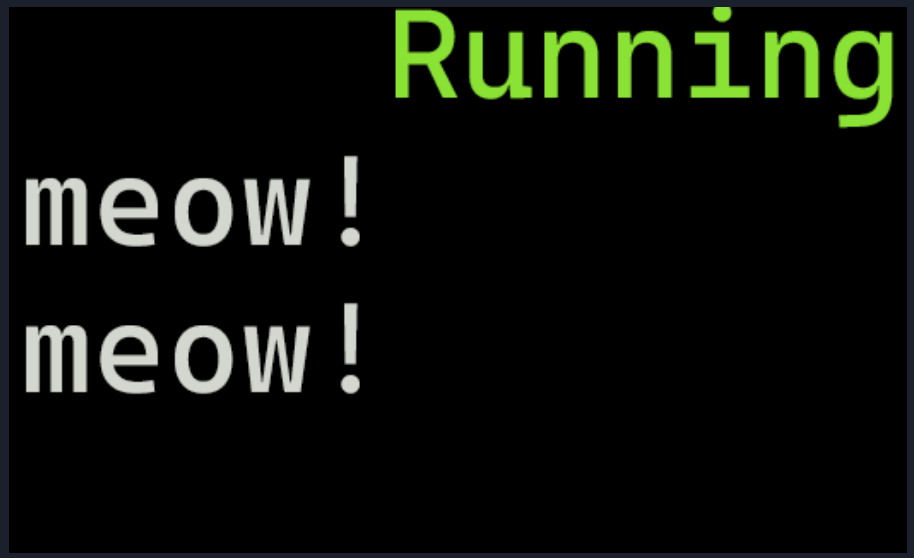
```rust
#[inject(replace(u8, i8))]
0 implementations
struct Person {
    age: u8,
    name: &'static str
}
0 implementations
struct Cat;
0 implementations
struct Dog;
#[inject(replace(Cat, i32))]
0 implementations
enum Animal {
    Cat(Cat),
    Dog(Dog)
}
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

cargo expand

```rust
struct Person {
    age: i8,
    name: &'static str,
}
struct Cat;
struct Dog;
enum Animal {
    i32(i32),
    Dog(Dog),
}
fn main() {
    let person: Person = Person { age: 24, name: "meow!" };
    let mut a: u16 = 1;
    a = 300;
    {
        ::std::io::_print(format_args!("meow!\n"));
    };
    {
        ::std::io::_print(format_args!("meow!\n"));
    };
}
```

```rust
#[inject(replace(u8, i8))]
0 implementations
struct Person {
    age: u8,
    name: &'static str
}
0 implementations
struct Cat;
0 implementations
struct Dog;
#[inject(replace(Cat, i32))]
0 implementations
enum Animal {
    Cat(Cat),
    Dog(Dog)
}
#[inject(
    payload="meow!",
    replace(mut),
    replace(const, let mut),
    replace(i32, u16),
)]
▶ Run | Debug
fn main() {
    let mut person: Person = Person {
        age: 24,
        name: "Max"
    };
    // person.age = 12;
    const a: i32 = 1;
    a = 300;
    println!("{a}");
    println!("{a - 5}");
}
```

cargo run



3. Next time

- Functional Programming