RUSTikales Rust for advanced coders

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. tt-munchers

Plan for today

1. Recap
2. tt-munchers
3. Procedural Macros

1. Recap

1. Recap

- Rust has a very powerful macro system

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output
- Declarative macros are declared using the keyword macro_rules!

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output
- Declarative macros are declared using the keyword macro_rules!
- Declarative macros are expanded at compile time, in the parsing stage
  - Depending on the context, they are interpreted differently

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output
- Declarative macros are declared using the keyword macro_rules!
- Declarative macros are expanded at compile time, in the parsing stage
- macro_rules! matches AST nodes to metavariables, which we can use to implement logic

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output
- Declarative macros are declared using the keyword macro_rules!
- Declarative macros are expanded at compile time, in the parsing stage
- macro_rules! matches AST nodes to metavariables, which we can use to implement logic
  - ident matches an identifier
  - expr matches an expression
  - ty matches a type

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output
- Declarative macros are declared using the keyword macro_rules!
- Declarative macros are expanded at compile time, in the parsing stage
- macro_rules! matches AST nodes to metavariables, which we can use to implement logic
  - ident matches an identifier
  - expr matches an expression
  - ty matches a type
  - lifetime matches a lifetime
  - stmt matches a statement
  - pat matches a pattern

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output
- Declarative macros are declared using the keyword macro_rules!
- Declarative macros are expanded at compile time, in the parsing stage
- macro_rules! matches AST nodes to metavariables, which we can use to implement logic
  - ident matches an identifier
  - expr matches an expression
  - ty matches a type
  - lifetime matches a lifetime
  - stmt matches a statement
  - pat matches a pattern
  - tt matches any single tree node

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output
- Declarative macros are declared using the keyword macro_rules!
- Declarative macros are expanded at compile time, in the parsing stage
- macro_rules! matches AST nodes to metavariables, which we can use to implement logic
- macro_rules! can also match repetitions of nodes
  - $($t:ty),* matches a comma-separated list of types

1. Recap

- Rust has a very powerful macro system
- Macros are useful for metaprogramming
  - They take Rust code as an input, and produce Rust code as an output
- Declarative macros are declared using the keyword macro_rules!
- Declarative macros are expanded at compile time, in the parsing stage
- macro_rules! matches AST nodes to metavariables, which we can use to implement logic
- macro_rules! can also match repetitions of nodes
- The first matched rule is always chosen
  - Important when matching different types:
    - Every identifier is an expression
    - Not every expression is an identifier
    - Better to match identifiers first

1. Recap

```rust
macro_rules! calculator {
    ($e:expr) => {
        println!("{} = {}", stringify!($e), $e);
    };
    () => {
        println!("Nothing was provided.");
    };
}
▶ Run | Debug
fn main() {
    calculator!(5 + 4);
    calculator!();
}
```

1. Recap

```rust
macro_rules! calculator { 
    ($e:expr) => {
        println!("{} = {}", stringify!($e), $e);
    };
    () => {
        println!("Nothing was provided.");
    };
}
▶ Run | Debug
fn main() {
    calculator!(5 + 4);
    calculator!();
}
```

Name of the macro

1. Recap

```rust
macro_rules! calculator {
    ($e:expr) => {
        println!("{} = {}", stringify!($e), $e);
    };
    () => {
        println!("Nothing was provided.");
    };
}
▶ Run | Debug
fn main() {
    calculator!(5 + 4);
    calculator!();
}
```

Cases to match against

1. Recap

```
macro_rules! calculator {
    ($e:expr) => {
        println!("{} = {}", stringify!($e), $e);
    };
    () => {
        println!("Nothing was provided.");
    };
}
▶ Run | Debug
fn main() {
    calculator!(5 + 4);
    calculator!();
}
```

We expect an expression, which we bind to the metavariable $e

1. Recap

```
macro_rules! calculator {
    ($e:expr) => {
        println!("{} = {}", stringify!($e), $e);
    };
    () => {
        println!("Nothing was provided.");
    };
}
▶ Run | Debug
fn main() {
    calculator!(5 + 4);
    calculator!();
}
```

If we matched an expression, we insert this code at the macro call site

Using cargo expand we can see what the code looks like without macros (cargo install cargo-expand)

```rust
macro_rules! calculator {
    ($e:expr) => {
        println!("{} = {}", stringify!($e), $e);
    };
    () => {
        println!("Nothing was provided.");
    };
}
▶ Run | Debug
fn main() {
    calculator!(5 + 4);
    calculator!();
}
```

```rust
fn main() {
    {
        ::std::io::_print(format_args!("{0} = {1}\n", "5 + 4", 5 + 4));
    };
    {
        ::std::io::_print(format_args!("Nothing was provided.\n"));
    };
}
```

Using cargo expand we can see what the code looks like without macros
(cargo install cargo-expand)

```rust
macro_rules! calculator {
    ($e:expr) => {
        println!("{} = {}", stringify!($e), $e);
    };
    () => {
        println!("Nothing was provided.");
    };
}
▶ Run | Debug
fn main() {
    calculator!(5 + 4); we matched an expression, and replaced it with the call to println!
    calculator!();
}
```

```rust
fn main() {
    {
        ::std::io::_print(format_args!("{0} = {1}\n", "5 + 4", 5 + 4));
    };
    {
        ::std::io::_print(format_args!("Nothing was provided.\n"));
    };
}
```

```rust
macro_rules! count_ident {
    () => { 0 };
    ($id:ident) => { 1 };
    ($id:ident, $($rest:ident),*) => {
        1 + count_ident!($($rest),*)
    };
}
▶ Run | Debug
fn main() {
    let count: i32 = count_ident!(a, b, c);
    println!("We have {count} identifiers!");
}
```

```rust
macro_rules! count_ident {
    () => { 0 };
    ($id:ident) => { 1 };
    ($id:ident, $($rest:ident),*) => {
        1 + count_ident!($($rest),*)
    };
}
▶ Run | Debug
fn main() {
    let count: i32 = count_ident!(a, b, c);
    println!("We have {count} identifiers!");
}
```

Macro invocations can be recursive

Here: Count all identifiers in the list

```rust
macro_rules! count_ident {
    () => { 0 };
    ($id:ident) => { 1 };
    ($id:ident, $($rest:ident),*) => {
        1 + count_ident!($($rest),*)
    };
}

▶ Run | Debug
fn main() {
    let count: i32 = count_ident!(a, b, c);
    println!("We have {count} identifiers!");
}
```

Macro invocations can be recursive

Here: Count all identifiers in the list

Idea: Split list into start and rest, and call macro again on the rest

```rust
macro_rules! count_ident {
    () => { 0 };
    ($id:ident) => { 1 };
    ($id:ident, $($rest:ident),*) => {
        1 + count_ident!($($rest),*)
    };
}
▶ Run | Debug
fn main() {
    let count: i32 = count_ident!(a, b, c);
    println!("We have {count} identifiers!");
}
```

We need some base cases to stop recursion

```
macro_rules! count_ident {
    () => { 0 };
    ($id:ident) => { 1 };
    ($id:ident, $($rest:ident),*) => {
        1 + count_ident!($($rest),*)
    };
}
▶ Run | Debug
fn main() {
    let count: i32 = count_ident!(a, b, c);
    println!("We have {count} identifiers!");
}
```

The order is not important, as rule 3 also expects a comma

```rust
macro_rules! count_ident {
    () => { 0 };
    ($id:ident) => { 1 };
    ($id:ident, $($rest:ident),*) => {
        1 + count_ident!($($rest),*)
    };
}
▶ Run | Debug
fn main() {
    let count: i32 = count_ident!(a, b, c);
    println!("We have {count} identifiers!");
}
```

```rust
fn main() {
    let count = 1 + (1 + 1);
    {
        ::std::io::_print(format_args!("We have {0} identifiers!\n", count));
    };
}
```

```rust
macro_rules! count_ident {
    () => { 0 };
    ($id:ident) => { 1 };
    ($id:ident, $($rest:ident),*) => {
        1 + count_ident!($($rest),*)
    };
}
▶ Run | Debug
fn main() {
    let count: i32 = count_ident!(a, b, c);
    println!("We have {count} identifiers!");
}
```

```rust
fn main() {
    let count = 1 + (1 + 1);
    {
        ::std::io::_print(format_args!("We have {0} identifiers!\n", count));
    };
}
```

Important:
The count is computed at runtime!
We only generated the formula at compile time.

2. tt-munchers

2. tt-munchers

- macro_rules! is very powerful and allows us to generate all sorts of code

2. tt-munchers

- macro_rules! is very powerful and allows us to generate all sorts of code
- Very often we encounter the same patterns when declaring macros

2. tt-munchers

- macro_rules! is very powerful and allows us to generate all sorts of code
- Very often we encounter the same patterns when declaring macros
- To showcase one of the most important ones, we'll do something fancy today:
  - We'll implement a calculator for expressions in reverse polish notation using only macro_rules!

2. tt-munchers

```rust
fn main() {
    // 2 3 + 4 *
    // 15 7 1 1 + - / 3 * 2 1 1 + + -
}
```

2. tt-munchers

```rust
fn main() {
    // 2 3 + 4 *
    // 15 7 1 1 + - / 3 * 2 1 1 + + -
}
```

Stack

2. tt-munchers

```
fn main() {
    // 2 3 + 4 *
    // 15 7 1 1 + − / 3 * 2 1 1 + + −
}
```

Stack
2

2. tt-munchers

```
fn main() {
    // 2 3 + 4 *
    // 15 7 1 1 + - / 3 * 2 1 1 + + -
}
```

Stack
2
3

2. tt-munchers

```
fn main() {
    // 2 3 + 4 *
    // 15 7 1 1 + - / 3 * 2 1 1 + + -
}
```

Stack
2
3

2. tt-munchers

```
fn main() {
    // 2 3 + 4 *
    // 15 7 1 1 + - / 3 * 2 1 1 + + -
}
```

Stack
5

2. tt-munchers

```
fn main() {
    // 2 3 + 4 *
    // 15 7 1 1 + - / 3 * 2 1 1 + + -
}
```

Stack
5
4

2. tt-munchers

```
fn main() {
    // 2 3 + 4 *
    // 15 7 1 1 + - / 3 * 2 1 1 + + -
}
```

Stack
20

2. tt-munchers

- macro_rules! is very powerful and allows us to generate all sorts of code
- Very often we encounter the same patterns when declaring macros
- To showcase one of the most important ones, we'll do something fancy today:
  - We'll implement a calculator for expressions in reverse polish notation using only macro_rules!
- We'll start from the end, and slowly build it all up

2. tt-munchers

```rust
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
    let b = rpn!(15 7 1 1 + - / 3 * 2 1 1 + + -); // expect: 5
}
```

We want our macro to return the result so we can freely use it later
(for example, to print it in the console)

2. tt-munchers

```rust
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
    let b = rpn!(15 7 1 1 + - / 3 * 2 1 1 + + -); // expect: 5
}
```

We want our macro to return the result so we can freely use it later
(for example, to print it in the console)

Let‘s start simple: Pushing a number to the stack

2. tt-munchers

```
macro_rules! rpn {
    ($num:literal) => {

    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *);
```

2. tt-munchers

```rust
macro_rules! rpn {
    ($num:literal) => {

    };
}
fn main() {
    let a = rpn!(2 3 + 4 *);
```

Matching a number is easy – We simply match a literal

2. tt-munchers

```rust
macro_rules! rpn {
    ($num:literal) => {

    };
}

fn main() {
    let a = rpn!(2 3 + 4 *);
```

Matching a number is easy – We simply match a literal
However, we also need a stack to keep track of all numbers

2. tt-munchers

```
macro_rules! rpn {
    ($num:literal) => {

    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *);
```

Matching a number is easy – We simply match a literal
However, we also need a stack to keep track of all numbers
→ Problem: Our macro can‘t use real variables, we need the stack at compile time

2. tt-munchers

```rust
macro_rules! rpn {
    ($num:literal) => {

    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *);
```

Matching a number is easy – We simply match a literal
However, we also need a stack to keep track of all numbers
→ Problem: Our macro can‘t use real variables, we need the stack at compile time
→ But we can work with token sequences

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal) => {
        rpn!([ $num $(, $stack)* ])
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal) => {
        rpn!([ $num $(, $stack)* ])
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

Idea:
→ The stack is a list of expressions

2. tt-munchers

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal) => {
        rpn!([ $num $(, $stack)* ])
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

Idea:
→ The stack is a list of expressions
→ Square brackets so we don‘t accidentally confuse it with input

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal) => {
        rpn!([ $num $(, $stack)* ])
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

Idea:
→ The stack is a list of expressions
→ Square brackets so we don‘t accidentally confuse it with input
→ Elements in the stack are expressions such as 2 + 3

2. tt-munchers

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal) => {
        rpn!([ $num $(, $stack)* ])
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

Idea:

→ The stack is a list of expressions

→ Square brackets so we don‘t accidentally confuse it with input

→ Elements in the stack are expressions such as 2 + 3

→ Later on we‘ll have rules to match operators against the stack

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal) => {
        rpn!([ $num $(, $stack)* ])
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

There are no limits to what we can emit with macros
→ We can‘t directly modify the stack, but we can create a modified copy
→ Here: Push the number at the front

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal) => {
        rpn!([ $num $(, $stack)* ])
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

The final code must be valid, but we‘re not actually generating anything!

The stack is only temporary and passed between macro calls.

2. tt-munchers

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal) => {
        rpn!([ $num $(, $stack)* ])
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

The final code must be valid, but we‘re not actually generating anything!

The stack is only temporary and passed between macro calls.

Okay, numbers are done! Now we need to work on the rest

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

This is a tt-muncher – A pattern which comes up all the time with declarative macros

2. tt-munchers

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

This is a tt-muncher – A pattern which comes up all the time with declarative macros
→ We‘re processing one token at a time

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

This is a tt-muncher – A pattern which comes up all the time with declarative macros
→ We‘re processing one token at a time, and pass along the rest

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

This is a tt-muncher – A pattern which comes up all the time with declarative macros
→ We‘re processing one token at a time, and pass along the rest
→ We don‘t care what tree type the rest has, we eat it all

2. tt-munchers

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 + 4 *); // expect: 20
```

This is a tt-muncher – A pattern which comes up all the time with declarative macros
→ We‘re processing one token at a time, and pass along the rest
→ We don‘t care what tree type the rest has, we eat it all

In our case: This macro now accepts a sequence of numbers, and puts them into the stack!

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 4);
```

2. tt-munchers

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 4);
```

Our final result should be the only element

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 4);
```

Entrypoint so we don‘t have to specify []

2. tt-munchers

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    let a = rpn!(2 3 4);
```

The compiler is not happy...

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    let a = rpn!(2 3 4);
    trace_macros!(false);
```

```
note: trace_macro
  --> src\main.rs:13:13
   |
13 |     let a = rpn!(2 3 4);
   |             ^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 4 }`
   = note: to `rpn! ([] 2 3 4)`
   = note: expanding `rpn! { [] 2 3 4 }`
   = note: to `rpn! ([2] 3 4)`
   = note: expanding `rpn! { [2] 3 4 }`
   = note: to `rpn! ([3, 2] 4)`
   = note: expanding `rpn! { [3, 2] 4 }`
   = note: to `rpn! ([4, 3, 2])`
   = note: expanding `rpn! { [4, 3, 2] }`
   = note: to `rpn! ([] [4, 3, 2])`
   = note: expanding `rpn! { [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [] [4, 3, 2])`
```

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    let a = rpn!(2 3 4);
    trace_macros!(false);
```

```
note: trace_macro
  --> src\main.rs:13:13
   |
13 |     let a = rpn!(2 3 4);
   |             ^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 4 }`
   = note: to `rpn! ([] 2 3 4)`
   = note: expanding `rpn! { [] 2 3 4 }`
   = note: to `rpn! ([2] 3 4)`
   = note: expanding `rpn! { [2] 3 4 }`
   = note: to `rpn! ([3, 2] 4)`
   = note: expanding `rpn! { [3, 2] 4 }`
   = note: to `rpn! ([4, 3, 2])`
   = note: expanding `rpn! { [4, 3, 2] }`
   = note: to `rpn! ([] [4, 3, 2])`
   = note: expanding `rpn! { [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [] [4, 3, 2])`
```

The important bit first: We collected the numbers into the stack!

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    let a = rpn!(2 3 4);
    trace_macros!(false);
```

```
note: trace_macro
  --> src\main.rs:13:13
   |
13 |     let a = rpn!(2 3 4);
   |             ^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 4 }`
   = note: to `rpn! ([] 2 3 4)`
   = note: expanding `rpn! { [] 2 3 4 }`
   = note: to `rpn! ([2] 3 4)`
   = note: expanding `rpn! { [2] 3 4 }`
   = note: to `rpn! ([3, 2] 4)`
   = note: expanding `rpn! { [3, 2] 4 }`
   = note: to `rpn! ([4, 3, 2])`
   = note: expanding `rpn! { [4, 3, 2] }`
   = note: to `rpn! ([] [4, 3, 2])`
   = note: expanding `rpn! { [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [] [4, 3, 2])`
```

The important bit first: We collected the numbers into the stack!
This is where it went wrong though...

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    let a = rpn!(2 3 4);
    trace_macros!(false);
```

```
note: trace_macro
  --> src\main.rs:13:13
   |
13 |     let a = rpn!(2 3 4);
   |             ^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 4 }`
   = note: to `rpn! ([] 2 3 4)`
   = note: expanding `rpn! { [] 2 3 4 }`
   = note: to `rpn! ([2] 3 4)`
   = note: expanding `rpn! { [2] 3 4 }`
   = note: to `rpn! ([3, 2] 4)`
   = note: expanding `rpn! { [3, 2] 4 }`
   = note: to `rpn! ([4, 3, 2])`
   = note: expanding `rpn! { [4, 3, 2] }`
   = note: to `rpn! ([] [4, 3, 2])`
   = note: expanding `rpn! { [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [] [4, 3, 2])`
```

The important bit first: We collected the numbers into the stack!
This is where it went wrong though...
→ The stack does not match rule 1

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    let a = rpn!(2 3 4);
    trace_macros!(false);
```

```
note: trace_macro
  --> src\main.rs:13:13
   |
13 |     let a = rpn!(2 3 4);
   |             ^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 4 }`
   = note: to `rpn! ([] 2 3 4)`
   = note: expanding `rpn! { [] 2 3 4 }`
   = note: to `rpn! ([2] 3 4)`
   = note: expanding `rpn! { [2] 3 4 }`
   = note: to `rpn! ([3, 2] 4)`
   = note: expanding `rpn! { [3, 2] 4 }`
   = note: to `rpn! ([4, 3, 2])`
   = note: expanding `rpn! { [4, 3, 2] }`
   = note: to `rpn! ([] [4, 3, 2])`
   = note: expanding `rpn! { [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [] [4, 3, 2])`
```

The important bit first: We collected the numbers into the stack!
This is where it went wrong though...
→ The stack does not match rule 1
→ The stack does not contain a single expression → not rule 2

```
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    let a = rpn!(2 3 4);
    trace_macros!(false);
```

```
note: trace_macro
  --> src\main.rs:13:13
   |
13 |     let a = rpn!(2 3 4);
   |             ^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 4 }`
   = note: to `rpn! ([] 2 3 4)`
   = note: expanding `rpn! { [] 2 3 4 }`
   = note: to `rpn! ([2] 3 4)`
   = note: expanding `rpn! { [2] 3 4 }`
   = note: to `rpn! ([3, 2] 4)`
   = note: expanding `rpn! { [3, 2] 4 }`
   = note: to `rpn! ([4, 3, 2])`
   = note: expanding `rpn! { [4, 3, 2] }`
   = note: to `rpn! ([] [4, 3, 2])`
   = note: expanding `rpn! { [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [] [4, 3, 2])`
```

The important bit first: We collected the numbers into the stack!
This is where it went wrong though...
→ The stack does not match rule 1
→ The stack does not contain a single expression → not rule 2
Rule 3 is a black hole, it matches *everything* – including our stack

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    let a = rpn!(2 3 4);
    trace_macros!(false);
```

```
note: trace_macro
  --> src\main.rs:13:13
   |
13 |     let a = rpn!(2 3 4);
   |             ^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 4 }`
   = note: to `rpn! ([] 2 3 4)`
   = note: expanding `rpn! { [] 2 3 4 }`
   = note: to `rpn! ([2] 3 4)`
   = note: expanding `rpn! { [2] 3 4 }`
   = note: to `rpn! ([3, 2] 4)`
   = note: expanding `rpn! { [3, 2] 4 }`
   = note: to `rpn! ([4, 3, 2])`
   = note: expanding `rpn! { [4, 3, 2] }`
   = note: to `rpn! ([] [4, 3, 2])`
   = note: expanding `rpn! { [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [] [4, 3, 2])`
```

The important bit first: We collected the numbers into the stack!
This is where it went wrong though...
→ The stack does not match rule 1
→ The stack does not contain a single expression → not rule 2
Rule 3 is a black hole, it matches *everything* – including our stack
And now we‘re stuck in infinite recursion...

```rust
macro_rules! rpn {
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    let a = rpn!(2 3 4);
    trace_macros!(false);
```

```
note: trace_macro
  --> src\main.rs:13:13
   |
13 |     let a = rpn!(2 3 4);
   |             ^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 4 }`
   = note: to `rpn! ([] 2 3 4)`
   = note: expanding `rpn! { [] 2 3 4 }`
   = note: to `rpn! ([2] 3 4)`
   = note: expanding `rpn! { [2] 3 4 }`
   = note: to `rpn! ([3, 2] 4)`
   = note: expanding `rpn! { [3, 2] 4 }`
   = note: to `rpn! ([4, 3, 2])`
   = note: expanding `rpn! { [4, 3, 2] }`
   = note: to `rpn! ([] [4, 3, 2])`
   = note: expanding `rpn! { [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [4, 3, 2])`
   = note: expanding `rpn! { [] [] [] [4, 3, 2] }`
   = note: to `rpn! ([] [] [] [] [4, 3, 2])`
```

But that‘s okay for now. We just wanted to put numbers into the stack.

Let‘s continue with operators.

```rust
macro_rules! rpn {
    ([ $b:expr, $a:expr $(, $stack:expr)* ] + $($rest:tt)*) => {
        rpn!([ $a + $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] - $($rest:tt)*) => {
        rpn!([ $a - $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] * $($rest:tt)*) => {
        rpn!([ $a * $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] / $($rest:tt)*) => {
        rpn!([ $a / $b $(, $stack)* ] $($rest)*)
    };
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
       rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
```

```rust
macro_rules! rpn {
    ([ $b:expr, $a:expr $(, $stack:expr)* ] + $($rest:tt)*) => {
        rpn!([ $a + $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] - $($rest:tt)*) => {
        rpn!([ $a - $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] * $($rest:tt)*) => {
        rpn!([ $a * $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] / $($rest:tt)*) => {
        rpn!([ $a / $b $(, $stack)* ] $($rest)*)
    };
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
      rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
```

Sadly there is no operator-type, we need to match manually and write duplicate code

```rust
macro_rules! rpn {
    ([ $b:expr, $a:expr $(, $stack:expr)* ] + $($rest:tt)*) => {
        rpn!([ $a + $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] - $($rest:tt)*) => {
        rpn!([ $a - $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] * $($rest:tt)*) => {
        rpn!([ $a * $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] / $($rest:tt)*) => {
        rpn!([ $a / $b $(, $stack)* ] $($rest)*)
    };
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
      rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
```

All operators expect 2 operands, which we explicitly match here

→ If the stack does not have two elements, we fall through

```rust
macro_rules! rpn {
    ([ $b:expr, $a:expr $(, $stack:expr)* ] + $($rest:tt)*) => {
        rpn!([ $a + $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] - $($rest:tt)*) => {
        rpn!([ $a - $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] * $($rest:tt)*) => {
        rpn!([ $a * $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] / $($rest:tt)*) => {
        rpn!([ $a / $b $(, $stack)* ] $($rest)*)
    };
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
```

Then we push the result to the stack

```rust
macro_rules! rpn {
    ([ $b:expr, $a:expr $(, $stack:expr)* ] + $($rest:tt)*) => {
        rpn!([ $a + $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] - $($rest:tt)*) => {
        rpn!([ $a - $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] * $($rest:tt)*) => {
        rpn!([ $a * $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] / $($rest:tt)*) => {
        rpn!([ $a / $b $(, $stack)* ] $($rest)*)
    };
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
       rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
```

The stack shrinks because we capture all but the first two elements

```
macro_rules! rpn {
    ([ $b:expr, $a:expr $(, $stack:expr)* ] + $($rest:tt)*) => {
        rpn!([ $a + $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] - $($rest:tt)*) => {
        rpn!([ $a - $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] * $($rest:tt)*) => {
        rpn!([ $a * $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] / $($rest:tt)*) => {
        rpn!([ $a / $b $(, $stack)* ] $($rest)*)
    };
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
        rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
```

And finally, we proceed with eating tokens until we‘re done

2. tt-munchers

```rust
fn main() {
    let a: i32 = rpn!(2 3 + 4 *); // expect: 20
    let b: i32 = rpn!(15 7 1 1 + - / 3 * 2 1 1 + + -); // expect: 5
    println!("a = {:?}", a);
    println!("b = {:?}", b);
}
```

We can now put the macro to the test!

```
note: trace_macro
  --> src\main.rs:25:13
   |
25 |     let a = rpn!(2 3 + 4 *); // expect:
   |             ^^^^^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 + 4 * }`
   = note: to `rpn! ([] 2 3 + 4 *)`
   = note: expanding `rpn! { [] 2 3 + 4 * }`
   = note: to `rpn! ([2] 3 + 4 *)`
   = note: expanding `rpn! { [2] 3 + 4 * }`
   = note: to `rpn! ([3, 2] + 4 *)`
   = note: expanding `rpn! { [3, 2] + 4 * }`
   = note: to `rpn! ([2 + 3] 4 *)`
   = note: expanding `rpn! { [2 + 3] 4 * }`
   = note: to `rpn! ([4, 2 + 3] *)`
   = note: expanding `rpn! { [4, 2 + 3] * }`
   = note: to `rpn! ([2 + 3 * 4])`
   = note: expanding `rpn! { [2 + 3 * 4] }`
   = note: to `(2 + 3) * 4`
```

```rust
fn main() {
    let a: i32 = rpn!(2 3 + 4 *); // expect: 20
    let b: i32 = rpn!(15 7 1 1 + - / 3 * 2 1 1 + + -); // expect: 5
    println!("a = {:?}", a);
    println!("b = {:?}", b);
}
```

```
note: trace_macro
  --> src\main.rs:25:13
   |
25 |     let a = rpn!(2 3 + 4 *); // expect:
   |             ^^^^^^^^^^^^^^^
   |
   = note: expanding `rpn! { 2 3 + 4 * }`
   = note: to `rpn! ([] 2 3 + 4 *)`
   = note: expanding `rpn! { [] 2 3 + 4 * }`
   = note: to `rpn! ([2] 3 + 4 *)`
   = note: expanding `rpn! { [2] 3 + 4 * }`
   = note: to `rpn! ([3, 2] + 4 *)`
   = note: expanding `rpn! { [3, 2] + 4 * }`
   = note: to `rpn! ([2 + 3] 4 *)`
   = note: expanding `rpn! { [2 + 3] 4 * }`
   = note: to `rpn! ([4, 2 + 3] *)`
   = note: expanding `rpn! { [4, 2 + 3] * }`
   = note: to `rpn! ([2 + 3 * 4])`
   = note: expanding `rpn! { [2 + 3 * 4] }`
   = note: to `(2 + 3) * 4`
```

```rust
fn main() {
    let a: i32 = rpn!(2 3 + 4 *); // expect: 20
    let b: i32 = rpn!(15 7 1 1 + - / 3 * 2 1 1 + + -); // expect: 5
    println!("a = {:?}", a);
    println!("b = {:?}", b);
}
```

```
a = 20
b = 5
```

```rust
fn main() {
    let a = (2 + 3) * 4;
    let b = 15 / (7 - (1 + 1)) * 3 - (2 + (1 + 1));
    {
        ::std::io::_print(format_args!("a = {0:?}\n", a));
    };
    {
        ::std::io::_print(format_args!("b = {0:?}\n", b));
    };
}
```

As expected, the macro properly evaluated the expression

```rust
fn main() {
    let a: i32 = rpn!(2 3 + 4 *); // expect: 20
    let b: i32 = rpn!(15 7 1 1 + - / 3 * 2 1 1 + + -); // expect: 5
    println!("a = {:?}", a);
    println!("b = {:?}", b);
}
```

```
a = 20
b = 5
```

```rust
macro_rules! rpn {
    ([ $b:expr, $a:expr $(, $stack:expr)* ] + $($rest:tt)*) => {
        rpn!([ $a + $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] - $($rest:tt)*) => {
        rpn!([ $a - $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] * $($rest:tt)*) => {
        rpn!([ $a * $b $(, $stack)* ] $($rest)*)
    };
    ([ $b:expr, $a:expr $(, $stack:expr)* ] / $($rest:tt)*) => {
        rpn!([ $a / $b $(, $stack)* ] $($rest)*)
    };
    ([ $($stack:expr),* ] $num:literal $($rest:tt)*) => {
       rpn!([ $num $(, $stack)* ] $($rest)*)
    };
    ([$result:expr]) => { $result };
    ($($t:tt)*) => { rpn!([] $($t)*) }
}
```

2/3

The macro currently crashes for invalid expressions, for example if there aren‘t enough operands on the stack, or not enough operators in the equation
→ How could we fix that?

3. Procedural Macros

3. Procedural Macros

– Procedural Macros are the second macro system that Rust offers

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library

First, create a normal library

```
cargo new --lib showcase
```

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library

First, create a normal library

```
cargo new --lib showcase
```

Then declare it to be a proc-macro

lib.rs

```
[lib]
proc-macro = true
```

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library

First, create a normal library

```
cargo new --lib showcase
```

Then declare it to be a proc-macro

lib.rs

```
[lib]
proc-macro = true
```

Now you can use it like a normal library

main.rs

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

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library
- Because of that, Procedural Macros are written in normal Rust, which makes them more flexible than macro_rules!

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library
- Because of that, Procedural Macros are written in normal Rust, which makes them more flexible than macro_rules!
- Procedural Macros are often used to generate complex code which macro_rules! isn‘t able to handle

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library
- Because of that, Procedural Macros are written in normal Rust, which makes them more flexible than macro_rules!
- Procedural Macros are often used to generate complex code which macro_rules! isn‘t able to handle
- There are three types of Procedural Macros

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library
- Because of that, Procedural Macros are written in normal Rust, which makes them more flexible than macro_rules!
- Procedural Macros are often used to generate complex code which macro_rules! isn‘t able to handle
- There are three types of Procedural Macros
  - Function-like macros – Just like declarative macros, invoked using macro_name!()

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library
- Because of that, Procedural Macros are written in normal Rust, which makes them more flexible than macro_rules!
- Procedural Macros are often used to generate complex code which macro_rules! isn‘t able to handle
- There are three types of Procedural Macros
  - Function-like macros – Just like declarative macros, invoked using macro_name!()
  - Derive macros – Reduces boilerplate, used to implement traits for structs and enums

3. Procedural Macros

- Procedural Macros are the second macro system that Rust offers
- Procedural Macros operate over token streams instead of matching patterns at compile time
- Procedural Macros are a special crate type in Rust, which you can use like any other library
- Because of that, Procedural Macros are written in normal Rust, which makes them more flexible than macro_rules!
- Procedural Macros are often used to generate complex code which macro_rules! isn‘t able to handle
- There are three types of Procedural Macros
  - Function-like macros – Just like declarative macros, invoked using macro_name!()
  - Derive macros – Reduces boilerplate, used to implement traits for structs and enums
  - Attribute macros – Allows us to define outer attributes, which we can attach to items

3. Procedural Macros – proc_macro

lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn function_like(_item: TokenStream) -> TokenStream {
    todo!()
}
```

3. Procedural Macros – proc_macro

lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn function_like(_item: TokenStream) -> TokenStream {
    todo!()
}
```

Procedural macros are normal functions

3. Procedural Macros – proc_macro

lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn function_like(_item: TokenStream) -> TokenStream {
    todo!()
}
```

Procedural macros are normal functions
The attribute in front declares the type of the macro:
proc_macro → Function-like macro
proc_macro_derive → Derive macro
proc_macro_attribute → Attribute macro

3. Procedural Macros – proc_macro

lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn function_like(_item: TokenStream) -> TokenStream {
    todo!()
}
```

proc_macros are used like this:

```rust
function_like!(1, 2);
```

3. Procedural Macros – proc_macro

lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn function_like(_item: TokenStream) -> TokenStream {
    todo!()
}
```

Each type comes with a function signature that needs to be satisfied
Here:
→ proc_macro requires one argument of type TokenStream
→ proc_macro requires us to return a value of type TokenStream

3. Procedural Macros – proc_macro

lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn function_like(_item: TokenStream) -> TokenStream {
    todo!()
}
```

Each type comes with a function signature that needs to be satisfied
Here:
→ proc_macro requires one argument of type TokenStream
→ proc_macro requires us to return a value of type TokenStream

If the signature doesn‘t match, the library will not compile.

3. Procedural Macros – proc_macro

lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn function_like(_item: TokenStream) -> TokenStream {
    todo!()
}
```

Procedural Macros don‘t work with tree nodes, they work with a TokenStream

3. Procedural Macros – proc_macro

lib.rs

```
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn function_like(_item: TokenStream) -> TokenStream {
    todo!()
}
```

TokenStream is a type defined in the crate proc_macro, which is part of the standard library, no third-party libraries required

3. Procedural Macros – proc_macro

```rust
struct TokenStream(Vec<TokenTree>);
enum TokenTree {
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
    Group(Group),
}
```

3. Procedural Macros – proc_macro

```rust
struct TokenStream(Vec<TokenTree>);
enum TokenTree {
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
    Group(Group),
}
```

The final TokenStream we receive (and must return) is
a sequence of TokenTrees, or a sequence of tokens

3. Procedural Macros – proc_macro

```rust
struct TokenStream(Vec<TokenTree>);
enum TokenTree {
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
    Group(Group),
}
```

Fundamental tokens in Rust:
→ Ident: Identifiers and keywords
→ Punct: Single characters such as + or –
→ Literal: “Hello“, ‘a‘, 123u16

3. Procedural Macros – proc_macro

```rust
struct TokenStream(Vec<TokenTree>);
enum TokenTree {
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
    Group(Group),
}
```

TokenStream which is surrounded by delimiters
→ Delimiter = () {} []
→ (4 + 5)
→ { let x = 0; }

lib.rs

```rust
#[proc_macro]
pub fn function_like(item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

```rust
#[proc_macro]
pub fn function_like(item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

We can now write normal Rust code, and work with the given TokenStream

Here we just iterate over the stream, and print each entry

lib.rs

```rust
#[proc_macro]
pub fn function_like(item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

We can now write normal Rust code, and work with the given TokenStream

Here we just iterate over the stream, and print each entry

and return nothing. This macro did not generate any code :^)

```rust
use showcase::function_like;

function_like!(
    struct Person {
        field: u32
    }
);
```

main.rs

```rust
#[proc_macro]
pub fn function_like(item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

```rust
use showcase::function_like;

function_like!(
    struct Person {
        field: u32
    }
);
```

main.rs

```rust
#[proc_macro]
pub fn function_like(item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

```
   Compiling proc_showcase v0.1.0 (C:\Users\pfhau\GithubProje
t-advanced\06 - Procedural Macros\proc_showcase)
Ident { ident: "struct", span: #0 bytes(50..56) }
Ident { ident: "Person", span: #0 bytes(57..63) }
Brace
  Ident { ident: "field", span: #0 bytes(74..79) }
  Punct { ch: ':', spacing: Alone, span: #0 bytes(79..80) }
  Ident { ident: "u32", span: #0 bytes(81..84) }
    Finished dev [unoptimized + debuginfo] target(s) in 0.18s
     Running `target\debug\proc_showcase.exe`
```

```rust
use showcase::function_like;

function_like!(
    struct Person {
        field: u32
    }
);
```

main.rs

```rust
#[proc_macro]
pub fn function_like(item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

```
   Compiling proc_showcase v0.1.0 (C:\Users\pfhau\GithubProje
t-advanced\06 - Procedural Macros\proc_showcase)
Ident { ident: "struct", span: #0 bytes(50..56) }
Ident { ident: "Person", span: #0 bytes(57..63) }
Brace
  Ident { ident: "field", span: #0 bytes(74..79) }
  Punct { ch: ':', spacing: Alone, span: #0 bytes(79..80) }
  Ident { ident: "u32", span: #0 bytes(81..84) }
    Finished dev [unoptimized + debuginfo] target(s) in 0.18s
     Running `target\debug\proc_showcase.exe`
```

Procedural macros are called at compile time

```rust
use showcase::function_like;

function_like!(
    struct Person {
        field: u32
    }
);
```

Function-like macros capture everything between the brackets

main.rs

```rust
#[proc_macro]
pub fn function_like(item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

```
   Compiling proc_showcase v0.1.0 (C:\Users\pfhau\GithubProje
t-advanced\06 - Procedural Macros\proc_showcase)
Ident { ident: "struct", span: #0 bytes(50..56) }
Ident { ident: "Person", span: #0 bytes(57..63) }
Brace
  Ident { ident: "field", span: #0 bytes(74..79) }
  Punct { ch: ':', spacing: Alone, span: #0 bytes(79..80) }
  Ident { ident: "u32", span: #0 bytes(81..84) }
    Finished dev [unoptimized + debuginfo] target(s) in 0.18s
     Running `target\debug\proc_showcase.exe`
```

```rust
function_like!(1, 2);
```

For multiple arguments, we still get a single TokenStream

```rust
#[proc_macro]
pub fn function_like(item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

```
Literal { kind: Integer, symbol: "1", suffix: None, span: #0 bytes(45..46) }
Punct { ch: ',', spacing: Alone, span: #0 bytes(46..47) }
Literal { kind: Integer, symbol: "2", suffix: None, span: #0 bytes(48..49) }
    Finished dev [unoptimized + debuginfo] target(s) in 0.17s
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

Usage:

```rust
create_struct!(Person age u8 name String);
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

Usage:

```
create_struct!(Person age u8 name String);
```

lib.rs

```
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

Turn TokenStream into iterator so we can work with it

Usage:

```
create_struct!(Person age u8 name String);
```

lib.rs

```
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

Expect name of the struct

Usage:

```rust
create_struct!(Person age u8 name String);
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

Panics in procedural macros are caught by the compiler, and turned into a normal compiler error

Usage:

```rust
create_struct!(Person age u8 name String);
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

We now generate code!

Usage:

```rust
create_struct!(Person age u8 name String);
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

While there are tokens, expect a pair of two identifiers every time

Usage:

```rust
create_struct!(Person age u8 name String);
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

We do not have information if this type is actually valid, that‘s the job of the Type Checker

Usage:

```rust
create_struct!(Person age u8 name String);
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

Add a new field to the struct!

Usage:

```rust
create_struct!(Person age u8 name String);
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}'); Close struct brackets
    result.parse().unwrap()
}
```

Usage:

```rust
create_struct!(Person age u8 name String);
```

lib.rs

```rust
#[proc_macro]
pub fn create_struct(values: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = values.into_iter().peekable();
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected identifier, found {:?}", tokens.peek());
    };
    let mut result: String = format!("struct {name} {{");
    tokens.next();
    while let Some(TokenTree::Ident(field: Ident)) = tokens.next() {
        let Some(TokenTree::Ident(typ: &Ident)) = tokens.peek() else {
            panic!("Expected type, found {:?}", tokens.peek());
        };
        result.push_str(string: &format!("{field}: {typ},"));
        tokens.next();
    }

    result.push(ch: '}');
    result.parse().unwrap()
}
```

parse() converts the String into a TokenStream due to type inference

main.rs

```rust
use showcase::function_like;
use showcase::create_struct;

function_like!(1, 2);

create_struct!(Person age u8 name String);

▶ Run | Debug
fn main() {
}
```

```rust
use showcase::function_like;
use showcase::create_struct;

function_like!(1, 2);

create_struct!(Person age u8 name String);

▶ Run | Debug
fn main() {
}
```

main.rs

```rust
use showcase::function_like;
use showcase::create_struct;
struct Person {
    age: u8,
    name: String,
}
fn main() {}
```

main.rs

```rust
use showcase::function_like;
use showcase::create_struct;

function_like!(1, 2);

create_struct!(Person age u8 name String);

▶ Run | Debug
fn main() {
}
```

This macro always returns an empty stream

```rust
use showcase::function_like;
use showcase::create_struct;
struct Person {
    age: u8,
    name: String,
}
fn main() {}
```

```rust
use showcase::function_like;
use showcase::create_struct;

function_like!(1, 2);

create_struct!(Person age u8 name String);

▶ Run | Debug
fn main() {
}
```

main.rs

We successfully generated a struct Person, which we can now use in our code!

```rust
use showcase::function_like;
use showcase::create_struct;
struct Person {
    age: u8,
    name: String,
}
fn main() {}
```

3. Procedural Macros – proc_macro_derive

```
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(_input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

3. Procedural Macros – proc_macro_derive

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(_input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

proc_macro_derives are used like this:

```rust
use showcase::DeriveTrait;
#[derive(DeriveTrait)]
0 implementations
struct Dog;
#[derive(DeriveTrait)]
0 implementations
struct Cat;
```

3. Procedural Macros – proc_macro_derive

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(_input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

proc_macro_derives are used like this:

```rust
use showcase::DeriveTrait;
#[derive(DeriveTrait)]
0 implementations
struct Dog;
#[derive(DeriveTrait)]
0 implementations
struct Cat;
```

calls our macro in the background

3. Procedural Macros – proc_macro_derive

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(_input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

proc_macro_derive accepts one argument, and returns a TokenStream

3. Procedural Macros – proc_macro_derive

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(_input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

proc_macro_derive accepts one argument, and returns a TokenStream

Important:
proc_macro and proc_macro_attribute eat the input, proc_macro_derive leaves it untouched!
The input is the struct/enum you want to derive the trait for.

3. Procedural Macros – proc_macro_derive

```
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(_input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

The name here is not the trait you derive, it‘s what you later call #[derive()] with

→ DeriveTrait → #[derive(DeriveTrait)]

→ ABC → #[derive(ABC)]

3. Procedural Macros – proc_macro_derive

You can name it whatever you want

```
#[derive(ausidsad)]
0 implementations
struct Dog;
```

```
#[proc_macro_derive(ausidsad)]
pub fn trait_derive(_input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

3. Procedural Macros – proc_macro_derive

```rust
use showcase::DeriveTrait;
#[derive(DeriveTrait)]
0 implementations
struct Dog;
#[derive(DeriveTrait)]
0 implementations
struct Cat;
```

main.rs

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(input: TokenStream) -> TokenStream {
    for tt: TokenTree in input.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

```rust
use showcase::DeriveTrait;
#[derive(DeriveTrait)]
0 implementations
struct Dog;
#[derive(DeriveTrait)]
0 implementations
struct Cat;
```

main.rs

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(input: TokenStream) -> TokenStream {
    for tt: TokenTree in input.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

```
Ident { ident: "struct", span: #0 bytes(239..245) }
Ident { ident: "Dog", span: #0 bytes(246..249) }
Punct { ch: ';', spacing: Alone, span: #0 bytes(249..250) }
Ident { ident: "struct", span: #0 bytes(274..280) }
Ident { ident: "Cat", span: #0 bytes(281..284) }
Punct { ch: ';', spacing: Alone, span: #0 bytes(284..285) }
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
```

```rust
use showcase::DeriveTrait;
#[derive(DeriveTrait)]
0 implementations
struct Dog;
#[derive(DeriveTrait)]
0 implementations
struct Cat;
```

main.rs

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(input: TokenStream) -> TokenStream {
    for tt: TokenTree in input.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

Derive macros capture the item after the derive invocation

```
Ident { ident: "struct", span: #0 bytes(239..245) }
Ident { ident: "Dog", span: #0 bytes(246..249) }
Punct { ch: ';', spacing: Alone, span: #0 bytes(249..250) }
Ident { ident: "struct", span: #0 bytes(274..280) }
Ident { ident: "Cat", span: #0 bytes(281..284) }
Punct { ch: ';', spacing: Alone, span: #0 bytes(284..285) }
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
```

The trait we‘re deriving:

```rust
pub trait OurTrait {
    fn greet(&self);
}
```

The trait we're deriving:

```rust
pub trait OurTrait {
    fn greet(&self);
}
```

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(input: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = input.into_iter().peekable();
    tokens.next(); // Skip `struct` or `enum`
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected struct with name, got {:?}", tokens.peek());
    };
    let name: String = name.to_string();
    tokens.next();
    format!("
    impl OurTrait for {name} {{
        fn greet(&self) {{
            println!(\"Hello {name}!\");
        }}
    }}
    ").parse().unwrap()
}
```

The trait we're deriving:

```rust
pub trait OurTrait {
    fn greet(&self);
}
```

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(input: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = input.into_iter().peekable();
    tokens.next(); // Skip `struct` or `enum`
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected struct with name, got {:?}", tokens.peek());
    };
    let name: String = name.to_string();
    tokens.next();
    format!("
    impl OurTrait for {name} {{
        fn greet(&self) {{
            println!(\"Hello {name}!\");
        }}
    }}
    ").parse().unwrap()
}
```

Get struct/enum name

The trait we‘re deriving:

```rust
pub trait OurTrait {
    fn greet(&self);
}
```

```rust
#[proc_macro_derive(DeriveTrait)]
pub fn trait_derive(input: TokenStream) -> TokenStream {
    let mut tokens: impl Iterator<Item = TokenTree> = input.into_iter().peekable();
    tokens.next(); // Skip `struct` or `enum`
    let Some(TokenTree::Ident(name: &Ident)) = tokens.peek() else {
        panic!("Expected struct with name, got {:?}", tokens.peek());
    };
    let name: String = name.to_string();
    tokens.next();
    format!("
impl OurTrait for {name} {{
    fn greet(&self) {{
        println!(\"Hello {name}!\");
    }}
}}
").parse().unwrap()
}
```

Here we generate and return the trait implementation, where we greet the type

```rust
use showcase::DeriveTrait;
#[derive(DeriveTrait)]
1 implementation
struct Dog;
#[derive(DeriveTrait)]
1 implementation
struct Cat;

▶ Run | Debug
fn main() {
    Dog.greet();
    Cat.greet();
}
```

```rust
pub trait OurTrait {
    fn greet(&self);
}
```

```rust
impl OurTrait for {name} {{
    fn greet(&self) {{
        println!(\"Hello {name}!\");
    }}
}}
```

```rust
use showcase::DeriveTrait;
#[derive(DeriveTrait)]
1 implementation
struct Dog;
#[derive(DeriveTrait)]
1 implementation
struct Cat;

▶ Run | Debug
fn main() {
    Dog.greet();
    Cat.greet();
}
```

```rust
pub trait OurTrait {
    fn greet(&self);
}
```

```rust
impl OurTrait for {name} {{
    fn greet(&self) {{
        println!(\"Hello {name}!\");
    }}
}}
```

```
Running
Hello Dog!
Hello Cat!
```

3. Procedural Macros – proc_macro_attribute

```rust
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

3. Procedural Macros – proc_macro_attribute

```rust
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

proc_macro_attributes are used like this:

```rust
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
#[log_call]
fn bye() {
    println!("Bye!");
}
```

3. Procedural Macros – proc_macro_attribute

```rust
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

proc_macro_attributes are used like this:

```rust
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
#[log_call]
fn bye() {
    println!("Bye!");
}
```

Calls our macro in the background

3. Procedural Macros – proc_macro_attribute

```
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

Attribute Macros are different: They expect two arguments

→ The first argument is the token tree following the attribute‘s name

→ The second argument is the rest of the item, including other attributes

3. Procedural Macros – proc_macro_attribute

```rust
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

```rust
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
#[log_call]
fn bye() {
    println!("Bye!");
}
```

3. Procedural Macros – proc_macro_attribute

```rust
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

```rust
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
#[log_call]
fn bye() {
    println!("Bye!");
}
```

Attribute stream is empty here

3. Procedural Macros – proc_macro_attribute

```rust
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

```rust
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
#[log_call]
fn bye() {
    println!("Bye!");
}
```

Attribute stream is empty here

main.rs

```rust
#[log_call]
#[derive(DeriveTrait)]
struct Cat;
#[log_call]
enum Animal {
    Dog(Dog),
    Cat(Cat),
}
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
```

```rust
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

main.rs

```rust
#[log_call]
#[derive(DeriveTrait)]
struct Cat;
#[log_call]
enum Animal {
    Dog(Dog),
    Cat(Cat),
}
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
```

Many items accept attributes

```rust
#[proc_macro_attribute]
pub fn log_call(_attr: TokenStream, item: TokenStream) -> TokenStream {
    for tt: TokenTree in item.into_iter() {
        match tt {
            TokenTree::Ident(id: Ident) => println!("{:?}", id),
            TokenTree::Punct(pt: Punct) => println!("{:?}", pt),
            TokenTree::Literal(lit: Literal) => println!("{:?}", lit),
            TokenTree::Group(grp: Group) => {
                println!("{:?}", grp.delimiter());
                for gtt: TokenTree in grp.stream() {
                    println!("  {:?}", gtt);
                }
            }
        }
    }
    "".parse().unwrap()
}
```

lib.rs

3. Procedural Macros – proc_macro_attribute

```rust
#[log_call]
#[derive(DeriveTrait)]
struct Cat;
#[log_call]
enum Animal {
    Dog(Dog),
    Cat(Cat),
}
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
```

```
Punct { ch: '#', spacing: Alone, span: #0 bytes(287..288) }
Bracket
  Ident { ident: "derive", span: #0 bytes(289..295) }
  Group { delimiter: Parenthesis, stream: TokenStream [Ident { ident: "DeriveTrait", sp
an: #0 bytes(296..307) }], span: #0 bytes(295..308) }
Ident { ident: "struct", span: #0 bytes(310..316) }
Ident { ident: "Cat", span: #0 bytes(317..320) }
Punct { ch: ';', spacing: Alone, span: #0 bytes(320..321) }
```

3. Procedural Macros – proc_macro_attribute

```rust
#[log_call]
#[derive(DeriveTrait)]
struct Cat;
#[log_call]
enum Animal {
    Dog(Dog),
    Cat(Cat),
}
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
```

```
Ident { ident: "enum", span: #0 bytes(334..338) }
Ident { ident: "Animal", span: #0 bytes(339..345) }
Brace
  Ident { ident: "Dog", span: #0 bytes(352..355) }
  Group { delimiter: Parenthesis, stream: TokenStream [Ident { ident: "Dog", span: #0 b
ytes(356..359) }], span: #0 bytes(355..360) }
  Punct { ch: ',', spacing: Alone, span: #0 bytes(360..361) }
  Ident { ident: "Cat", span: #0 bytes(366..369) }
  Group { delimiter: Parenthesis, stream: TokenStream [Ident { ident: "Cat", span: #0 b
ytes(370..373) }], span: #0 bytes(369..374) }
  Punct { ch: ',', spacing: Alone, span: #0 bytes(374..375) }
```

3. Procedural Macros – proc_macro_attribute

```rust
#[log_call]
#[derive(DeriveTrait)]
struct Cat;
#[log_call]
enum Animal {
    Dog(Dog),
    Cat(Cat),
}
#[log_call(greet="yes")]
fn hello() {
    println!("Hello!");
}
```

```
Ident { ident: "fn", span: #0 bytes(403..405) }
Ident { ident: "hello", span: #0 bytes(406..411) }
Parenthesis
Brace
  Ident { ident: "println", span: #0 bytes(420..427) }
  Punct { ch: '!', spacing: Alone, span: #0 bytes(427..428) }
  Group { delimiter: Parenthesis, stream: TokenStream [Literal { kind: Str, symbol: "He
llo!", suffix: None, span: #0 bytes(429..437) }], span: #0 bytes(428..438) }
  Punct { ch: ';', spacing: Alone, span: #0 bytes(438..439) }
```

3. Procedural Macros – proc_macro_attribute

- We can call procedural macros in many different contexts

3. Procedural Macros – proc_macro_attribute

- We can call procedural macros in many different contexts
- We need to make sure that the macro is always correct, for any input

3. Procedural Macros – proc_macro_attribute

- We can call procedural macros in many different contexts
- We need to make sure that the macro is always correct, for any input
- That's *a lot* of complexity, given that we're only working with token streams

3. Procedural Macros – proc_macro_attribute

- We can call procedural macros in many different contexts
- We need to make sure that the macro is always correct, for any input
- That's *a lot* of complexity, given that we're only working with token streams
- But: Because we're using normal Rust, we can use everything that Rust provides

3. Procedural Macros – proc_macro_attribute

- We can call procedural macros in many different contexts
- We need to make sure that the macro is always correct, for any input
- That's *a lot* of complexity, given that we're only working with token streams
- But: Because we're using normal Rust, we can use everything that Rust provides
- We can use crates which parse streams into convenient structures, to make writing procedural macros easier

3. Procedural Macros – proc_macro_attribute

- We can call procedural macros in many different contexts
- We need to make sure that the macro is always correct, for any input
- That's *a lot* of complexity, given that we're only working with token streams
- But: Because we're using normal Rust, we can use everything that Rust provides
- We can use crates which parse streams into convenient structures, to make writing procedural macros easier
- The most important crates, syn and quote, will be introduced next week

4. Next time

- syn + quote
- Functional Programming in Rust