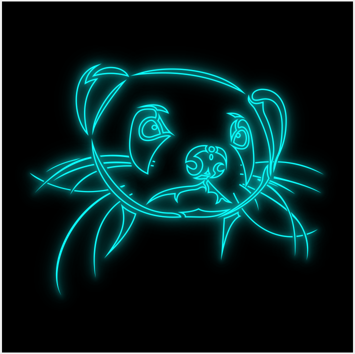
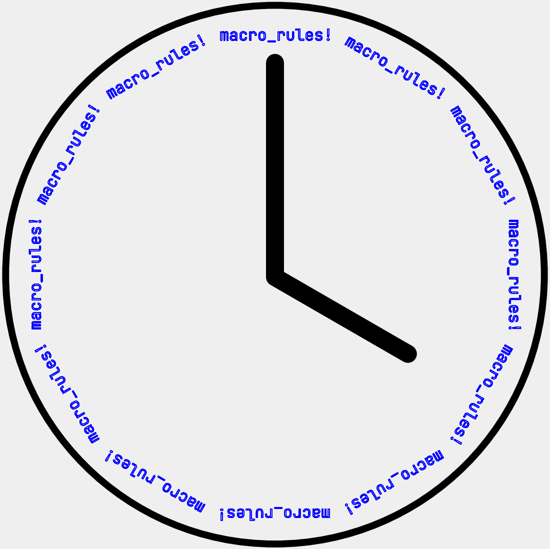
Anything you can do, I can do worse with `macro_rules!`

Aurorans Solis

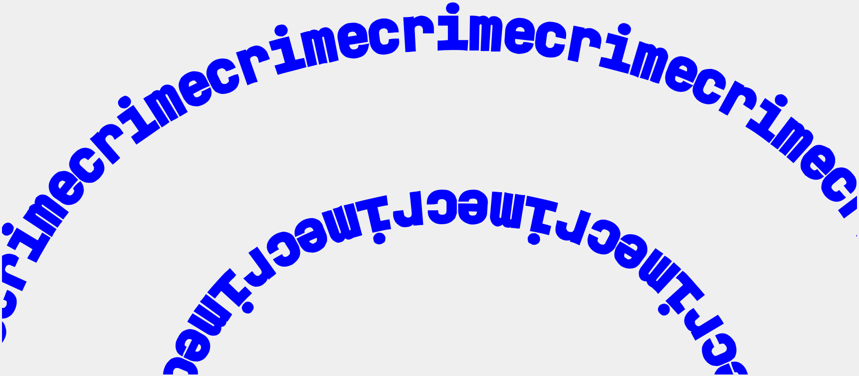


(:



macro_rules!
macro_rules!
macro_rules!
macro_rules!
macro_rules!
macro_rules!
macro_rules!
macro_rules!
macro_rules!
macro_rules!
macro_rules!
macro_rules!

macro_rules!


crimecrimecrimecrimecrimecrimec

It’s not snacktime anymore. It’s crimetime.

It’s not snacktime anymore. It’s crimetime.
We’re going to define traits using XML.

The Horrors

Start of the evil arc

We’re going to take this trait:

```rust
pub trait Foo<const BAR: usize>: Baz {
    type Baq: Qux;
    const QUUX: Self::Baq;
    fn corge<Grault, Garply>(waldo: Grault) -> Garply;
}
```

And turn it into this XML:

```xml
<trait>
    <name>Foo</name>
    <vis>pub</vis>
    <bounds>
        <const>
            <name>BAR</name>
            <type>usize</type>
        </const>
        <req>Baz</req>
    </bounds>
    <assoctype>
        <name>Baq</name>
        <bounds>
            <req>Qux</req>
        </bounds>
    </assoctype>
    <assocconst>
        <name>QUUX</name>
        <type>Self::Baq</type>
    </assocconst>
    <assocfn>
        <name>corge</name>
        <bounds>
            <type>
                <name>Grault</name>
            </type>
            <type>
                <name>Garply</name>
            </type>
        </bounds>
        <args>
            <arg>
                <name>waldo</name>
                <type>Grault</type>
            </arg>
        </args>
        <ret>Garply</ret>
    </assocfn>
</trait>
```

And turn it into this XML:

```xml
<trait>
    <name>Foo</name>
    <vis>pub</vis>
    <bounds>
        <const>
            <name>BAR</name>
            <type>usize</type>
        </const>
        <req>Baz</req>
    </bounds>
    <assoctype>
        <name>Baq</name>
        <bounds>
            <req>Qux</req>
        </bounds>
    </assoctype>
    <assocconst>
        <name>QUUX</name>
        <type>Self::Baq</type>
    </assocconst>
    <assocfn>
        <name>corge</name>
        <bounds>
            <type>
                <name>Grault</name>
            </type>
            <type>
                <name>Garply</name>
            </type>
        </bounds>
        <args>
            <arg>
                <name>waldo</name>
                <type>Grault</type>
            </arg>
        </args>
        <ret>Garply</ret>
    </assocfn>
</trait>
```

If you’re asking, “Why? Why would you do this to us?” you’re asking an excellent question.

And turn it into this XML:

```xml
<trait>
    <name>Foo</name>
    <vis>pub</vis>
    <bounds>
        <const>
            <name>BAR</name>
            <type>usize</type>
        </const>
        <req>Baz</req>
    </bounds>
    <assoctype>
        <name>Baq</name>
        <bounds>
            <req>Qux</req>
        </bounds>
    </assoctype>
    <assocconst>
        <name>QUUX</name>
        <type>Self::Baq</type>
    </assocconst>
    <assocfn>
        <name>corge</name>
        <bounds>
            <type>
                <name>Grault</name>
            </type>
            <type>
                <name>Garply</name>
            </type>
        </bounds>
        <args>
            <arg>
                <name>waldo</name>
                <type>Grault</type>
            </arg>
        </args>
        <ret>Garply</ret>
    </assocfn>
</trait>
```

If you’re asking, “Why? Why would you do this to us?” you’re asking an excellent question. I will not answer it.

The Horrors

WHAT

AURO NO PLEASE STOP

See the better question is, “Ooh, how did you do that?”

See the better question is, “Ooh, how did you do that?”

Let’s take a look, shall we?

These are the places we'll look at:

- Internal rules:
  - `trait_xml_macro.rs:9`, `trait_xml_macro.rs:37`
  - `assoc_fn.rs:1618`, `assoc_fn.rs:1649`, `assoc_fn.rs:1680`, `assoc_fn.rs:1711`

These are the places we'll look at:

- Internal rules:
  - `trait_xml_macro.rs:9`, `trait_xml_macro.rs:37`
  - `assoc_fn.rs:1618`, `assoc_fn.rs:1649`, `assoc_fn.rs:1680`, `assoc_fn.rs:1711`
- Incremental TT muncher: `vis.rs:58`

These are the places we'll look at:

- Internal rules:
  - `trait_xml_macro.rs:9`, `trait_xml_macro.rs:37`
  - `assoc_fn.rs:1618`, `assoc_fn.rs:1649`, `assoc_fn.rs:1680`, `assoc_fn.rs:1711`
- Incremental TT muncher: `vis.rs:58`
- Push-down accumulation: `type_ty.rs:116`

These are the places we'll look at:

- Internal rules:
  - `trait_xml_macro.rs:9`, `trait_xml_macro.rs:37`
  - `assoc_fn.rs:1618`, `assoc_fn.rs:1649`, `assoc_fn.rs:1680`, `assoc_fn.rs:1711`
- Incremental TT muncher: `vis.rs:58`
- Push-down accumulation: `type_ty.rs:116`
- TT bundling (and unbundling): `assoc_type.rs:88`, `trait_xml_macro.rs:336`, `trait_xml_macro.rs:823`

These are the places we'll look at:

- Internal rules:
  - `trait_xml_macro.rs:9`, `trait_xml_macro.rs:37`
  - `assoc_fn.rs:1618`, `assoc_fn.rs:1649`, `assoc_fn.rs:1680`, `assoc_fn.rs:1711`
- Incremental TT muncher: `vis.rs:58`
- Push-down accumulation: `type_ty.rs:116`
- TT bundling (and unbundling): `assoc_type.rs:88`, `trait_xml_macro.rs:336`, `trait_xml_macro.rs:823`
- Callback framework: `trait_xml_macro.rs:87`, `name_ident.rs:92`, `trait_xml_macro.rs:261`

Review of `macro_rules!`

Review of `macro_rules!`

Why macros?

Why macros?

Turns out that metaprogramming is pretty cool actually

- Repeat lots of similar but not quite identical things

Turns out that metaprogramming is pretty cool actually

- Repeat lots of similar but not quite identical things
- Define new grammars that get expanded to valid Rust

Why macros?

Turns out that metaprogramming is pretty cool actually

- Repeat lots of similar but not quite identical things
- Define new grammars that get expanded to valid Rust
- Confuse everyone (including yourself!)

Review of `macro_rules!`

What is a `macro_rules!`?

What is a `macro_rules!`?

Declarative macros (the ones we care about for this presentation) are sort of like functions on the AST.

What is a `macro_rules!`?

Declarative macros (the ones we care about for this presentation) are sort of like functions on the AST.

We try to match on certain AST patterns (rules) against the input.

What is a `macro_rules!`?

Declarative macros (the ones we care about for this presentation) are sort of like functions on the AST.

We try to match on certain AST patterns (rules) against the input.

Rules are tried in order from top to bottom.

What is a `macro_rules!`?

Declarative macros (the ones we care about for this presentation) are sort of like functions on the AST.

We try to match on certain AST patterns (rules) against the input.

Rules are tried in order from top to bottom.

We have fragments and fragment specifiers to allow for general AST node input but constrain what form that input is allowed to take.

What is a `macro_rules!`?

Declarative macros (the ones we care about for this presentation) are sort of like functions on the AST.

We try to match on certain AST patterns (rules) against the input.

Rules are tried in order from top to bottom.

We have fragments and fragment specifiers to allow for general AST node input but constrain what form that input is allowed to take.

We also have repetition specifiers for:

- zero or one (`?`)
- zero or more (`*`)
- one or more (`+`)

Review of `macro_rules!`

What are our AST node types, a.k.a. fragment specifiers?

Fragment specifier types

Rust has these fragment specifier types:

:item	:block	:stmt	:pat_param
:pat	:expr	:ty	:ident
:path	💕:tt💕	:meta	:lifetime
	:vis	:literal	

Each of these, except `:tt` are subject to regular Rust parsing rules.

Fragment specifier types

Rust has these fragment specifier types:

:item	:block	:stmt	:pat_param
:pat	:expr	:ty	:ident
:path	💕:tt💕	:meta	:lifetime
	:vis	:literal	

Each of these, except `:tt` are subject to regular Rust parsing rules.

There are also some limitations on what can come after certain fragment specifiers – follow set ambiguity restrictions

- `:expr` and `:stmt` can only be followed by =>, ,, or ;
- `:pat_param` can only be followed by =>, ,, =, |, `if`, or `in`
- etc.

Review of `macro_rules!`

Fragment Specifier Composition

Composition

Fragment specifiers can be composed into other fragment specifiers. For example, a `:ident` and `:expr` can be composed into a `:stmt`.

Fragment specifiers can be composed into other fragment specifiers. For example, a `:ident` and `:expr` can be composed into a `:stmt`.

The reverse is **NOT** true.

`:ident, :expr => :stmt`	=	✓
`:stmt => :ident, :expr`	=	✗

Fragment specifiers can be composed into other fragment specifiers. For example, a `:ident` and `:expr` can be composed into a `:stmt`.

The reverse is **NOT** true.

`:ident, :expr => :stmt`	=	✓
`:stmt => :ident, :expr`	=	✗

However, `:tt` tends to be the most flexible option for these sorts of operations.

```rust
macro_rules! me_reaping {
    ($let:tt $lhs:tt $equal:tt $rhs:tt) => {
        // compose `:tt`s into a `:stmt`
        me_reaping!(@matchstmt $let $lhs $equal $rhs)
    };
    (@matchstmt $stmt:stmt) => {
        $stmt
    };
}

macro_rules! me_sowing {
    ($stmt:stmt) => {
        // attempt to break a `:stmt` back into component `:tt`s
        me_reaping!($stmt);
    }
}

fn main() {
    me_reaping!(let haha = "yes!!!");
    println!("{haha}");
    me_sowing!(let well_this = "sucks ):");
    println!("{well_this}");
}
```

That gives the following error message:

```
error: unexpected end of macro invocation
  --> src/main.rs:14:26
   |
 1 | macro_rules! me_reaping {
   | ----------------------- when calling this macro
...
14 | me_reaping!($stmt);
   |                   ^ missing tokens in macro arguments
   |
note: while trying to match meta-variable `$lhs:tt`
  --> src/main.rs:2:14
   |
 2 |     ($let:tt $lhs:tt $equal:tt $rhs:tt) => {
   |              ^^^^^^^
```

That gives the following error message:

```
error: unexpected end of macro invocation
  --> src/main.rs:14:26
   |
 1 | macro_rules! me_reaping {
   | ----------------------- when calling this macro
...
14 | me_reaping!($stmt);
   |                  ^ missing tokens in macro arguments
   |
note: while trying to match meta-variable `$lhs:tt`
  --> src/main.rs:2:14
   |
 2 |     ($let:tt $lhs:tt $equal:tt $rhs:tt) => {
   |              ^^^^^^^
```

This is definitely all the magic stuff we will do with token composition (lies!)

Main Restrictions

There's two main restrictions that I've come across that aren't super obvious at first:

Main Restrictions

There's two main restrictions that I've come across that aren't super obvious at first:

1. No significant whitespace

Main Restrictions

There's two main restrictions that I've come across that aren't super obvious at first:

1. No significant whitespace
2. No matching tokens (generically, at least, without a bunch more macros)

Main Useful Patterns

Main Useful Patterns

Overview

There are ~~four~~six Big Lads of the Macropalypse:

- Recursion
  - of course macros can call themselves!

Overview

There are ~~four~~six Big Lads of the Macropalypse:

- Recursion
  - of course macros can call themselves!
- Internal rules
  - these are branches that generally shouldn't be called by users

There are ~~four~~six Big Lads of the Macropalypse:

- Recursion
  - of course macros can call themselves!
- Internal rules
  - these are branches that generally shouldn't be called by users
- Incremental TT munchers
  - grabs chunks off the front end of the list of inputs

There are ~~four~~six Big Lads of the Macropalypse:

- Recursion
  - of course macros can call themselves!
- Internal rules
  - these are branches that generally shouldn't be called by users
- Incremental TT munchers
  - grabs chunks off the front end of the list of inputs
- Push-down accumulation
  - holds tokens in a list for later expansion

There are ~~four~~six Big Lads of the Macropalypse:

- Recursion
  - of course macros can call themselves!
- Internal rules
  - these are branches that generally shouldn't be called by users
- Incremental TT munchers
  - grabs chunks off the front end of the list of inputs
- Push-down accumulation
  - holds tokens in a list for later expansion
- TT bundling
  - boils down to grouping multiple tokens into a single list

There are ~~four~~six Big Lads of the Macropalypse:

- Recursion
  - of course macros can call themselves!
- Internal rules
  - these are branches that generally shouldn't be called by users
- Incremental TT munchers
  - grabs chunks off the front end of the list of inputs
- Push-down accumulation
  - holds tokens in a list for later expansion
- TT bundling
  - boils down to grouping multiple tokens into a single list
- Callbacks
  - workaround to let you pass the expansion of one macro as input to another*

Let’s look at each of these in turn, but first, disclaimers.

Let's look at each of these in turn, but first, disclaimers.

- You're going to hear and see, "You've seen this already in this presentation!" a lot.

Let’s look at each of these in turn, but first, disclaimers.

- You’re going to hear and see, “You’ve seen this already in this presentation!” a lot.
- Yes, these patterns (in combination) will let you parse just about anything.

Let's look at each of these in turn, but first, disclaimers.

- You're going to hear and see, "You've seen this already in this presentation!" a lot.
- ~~Yes, these patterns (in combination) will let you parse just about anything.~~ STOP STOP STOP
- Recursion is **THE** building block for macros using the aforementioned patterns, so for big inputs you may end up needing `#![recursion_limit = "a very big number"]`.

Let’s look at each of these in turn, but first, disclaimers.

- You’re going to hear and see, “You’ve seen this already in this presentation!” a lot.
- ~~Yes, these patterns (in combination) will let you parse just about anything.~~ STOP STOP STOP
- Recursion is **THE** building block for macros using the aforementioned patterns, so for big inputs you may end up needing `#![recursion_limit = "a very big number"]`. And a long time to compile.

Let's look at each of these in turn, but first, disclaimers.

- You're going to hear and see, "You've seen this already in this presentation!" a lot.
- ~~Yes, these patterns (in combination) will let you parse just about anything.~~ STOP STOP STOP
- Recursion is **THE** building block for macros using the aforementioned patterns, so for big inputs you may end up needing `#![recursion_limit = "a very big number"]`. And a long time to compile. And a lot of memory.

Let's look at each of these in turn, but first, disclaimers.

- Declarative macros can be (and often are) very difficult to debug.

Overview

Let’s look at each of these in turn, but first, disclaimers.

- Declarative macros can be (and often are) very difficult to debug.
- Maintenance of big macros is... oh boy.

Let’s look at each of these in turn, but first, disclaimers.

- Declarative macros can be (and often are) very difficult to debug.
- Maintenance of big macros is… oh boy.
- All that said, these patterns can be leveraged to simplify some things quite a lot.

Let's look at each of these in turn, but first, disclaimers.

- Declarative macros can be (and often are) very difficult to debug.
- Maintenance of big macros is... oh boy.
- All that said, these patterns can be leveraged to simplify some things quite a lot.
- This talk is mostly going to be cursed stuff, however, I'll also talk about a few ways I've used these patterns in my own work and also on how to make things a little less cursed.

Main Useful Patterns

Recursion

Recursion my beloved

You’ve seen this already in this presentation!

This is the tool that every other pattern mentioned uses to work.

```rust
macro_rules! me_reaping {
    ($let:tt $lhs:tt $equal:tt $rhs:tt) => {
        // Recursive call to `me_reaping`
        me_reaping!(@matchstmt $let $lhs $equal $rhs)
    };
    (@matchstmt $stmt:stmt) => {
        $stmt
    };
}

fn main() {
    me_reaping!(let haha = "yes!!!");
    println!("{haha}");
    // no sowing (:<
}
```

Main Useful Patterns

Internal Rules

Internal rules my beloved

You've seen this already in this presentation! Two slides prior, even!

As said previously, you generally don't want users calling these rules. Usually they're used as a helper to grab new kinds of tokens or to specify a certain mode of parsing.

```
macro_rules! me_reaping {
    ($let:tt $lhs:tt $equal:tt $rhs:tt) => {
        me_reaping!(@matchstmt $let $lhs $equal $rhs)
    };
    // PATTERNS: internal rule to grab a statement
    (@matchstmt $stmt:stmt) => {
        $stmt
    };
}

fn main() {
    me_reaping!(let haha = "yes!!!");
    println!("{haha}");
    // no sowing (:<
}
```

Internal rules my beloved

Not all internal rules start with `@something`! Internal rules are just any rules that users are not expected to call but are used at some intermediate stage in macro expansion.

Useful in a couple ways

- Help avoid polluting crate namespace
  - Each internal rule could be its own macro, but those would also have to be marked with `#[macro_export]`
- Can be used to set “modes”
  - Useful for parsing context-sensitive things

Main Useful Patterns

Incremental TT munchers

Incremental TT munchers my beloved

You haven't seen this one already in this presentation! Surprise!

With these you typically look to grab an expected pattern including some inputs.

```
macro_rules! munch_and_crunch {
    () => {
        println!("empty!");
    };
    ($first:tt $($rest:tt)*) => {
        println!(concat!("munched: ", stringify!($first)));
        munch_and_crunch!($($rest)*);
    };
}

fn main() {
    munch_and_crunch!(foo bar baz baq);
    munch_and_crunch!(foo bar [baz baq]);
}
```

This one is very simple, but `all-unique` in the repo shows a slightly more interesting example of how to apply a pure TT muncher.

Main Useful Patterns

Push-down Accumulation

Push-down accumulation my beloved

This one hasn't been shown in this presentation yet!

Frequently used with incremental TT munchers for the purpose of holding tokens that have been munched. For example:

```
macro_rules! reverse_tokens {
    (@rev [$first:tt$(, $rest:tt)*] [$($rev:tt),*]) => {
        reverse_tokens! {
            @rev [$($rest),*][$first $(, $rev)*]
        }
    };
    (@rev [] [$($rev:tt),*]) => {
        $($rev)*
    };
    ($($tt:tt)+) => {
        reverse_tokens! {
            @rev [$($tt),+] []
        }
    };
}

fn main() {
    reverse_tokens! {
        ;0 = foo let
    }
    println!("{foo}");
}
```

Main Useful Patterns

TT Bundling

TT bundling my beloved

This one hasn't shown up yet either!

TT bundling is a sort of special case for composition, except this time we *can* actually reverse it!

Multiple tokens ⇒ [ ]-list (:tt)
Let me show you what I mean.

TT BUNDLING

tt-bundling/src/main.rs:

```rust
macro_rules! bundle_and_unbundle {
    ($name:ident, $type:ty, $value:expr) => {
        bundle_and_unbundle! {
            @bundled [$name, $type, $value]
        }
    };
    (@bundled $bundle:tt) => {
        const _: &str = stringify!($bundle);
        bundle_and_unbundle! {
            @unbundle $bundle
        }
    };
    (@unbundle [$name:ident, $type:ty, $value:expr]) => {
        let $name: $type = $value;
    };
}

fn main() {
    bundle_and_unbundle! {
        foo, u8, 0
    }
}
```

Main Useful Patterns

Callbacks

Callbacks my beloved

Very very generally, a callback looks something like this:

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

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

Callbacks my beloved

Very very generally, a callback looks something like this:

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

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

Maybe these don’t seem super useful, but they’re great for being able to reuse one macro in multiple places.

Very very generally, a callback looks something like this:

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

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

Maybe these don’t seem super useful, but they’re great for being able to reuse one macro in multiple places. As long as you have a consistent framework for calling your macros.

Debugging

If you’re defining items outside of functions, how do you debug things?

HOW???

If you're defining items outside of functions, how do you debug things?

What do when no `println!`?

If you’re defining items outside of functions, how do you debug things?

What do when no `println!`?

Introducing your new best friend:

```
const _: &str = stringify!($tokens);
// also
const _: &str = concat!($(stringify($tokens)),*);
```

Pretend println

You can see some residue from me debugging things and creating examples for documentation in `trait_xml_macro.rs`:

```
    // End of trait definition
    (
        @parsetrait {
            input: [</trait>],
            output: $outtoks:tt,
        }
    ) => {
        // const _: &str = stringify!($outtoks);
        $crate::trait_xml_inner! {
            @expand {
                output: $outtoks,
                vis: [],
                unsafe: ,
                name: ,
                gparams: [],
                tpbs: [],
                wc: [],
                assoc types: [],
                assoc consts: [],
                fns: [],
            }
        }
    };
```

No rules expected the token...

"But Auro! My macro just plain doesn't work! I can't use `const _`."

Yes you can.

```
macro_rules! this_fails_somehow {
    // a bunch of rules up here...

    // ...and then at the end:
    ($($all:tt)*) => {
        const _: &str = concat!($(stringify!($all)),*);
    };
}
```

This shows you the input as a string so you can try and figure out what's going wrong.

It's also only matched if no other branch matches.

No rules expected the token...

Another thing you can do to keep `error: no rules expected the token` from coming up is by just... making rules that expect the token/those tokens.

No rules expected the token...

Another thing you can do to keep `error: no rules expected the token` from coming up is by just... making rules that expect the token/those tokens.

My XML parsing macros do this a bunch - if you give them an invalid input, a lot of the time it'll expand to a `compile_error!`, e.g. in `lifetime.rs`:

```
    (
        @parse {
            input: [$unx:tt$($rest:tt)*],
            lifetime: $($lt:lifetime)?,
            bounds: $boundstoks:tt,
            callback: [
                name: $callback:path,
                rule: $ruletoks:tt,
                args: $argstoks:tt,
            ],
        }
    ) => {
        compile_error!(
```

Other things

Some other suggestions to help you on your way (that sound a lot like general programming advice):

- Use descriptive names for rules, helper tokens, and fragments
- Don't worry about excessive complexity in a single rule – just try to keep the depth of recursion down if you can
- Don't be afraid to make another macro, especially if it can be reused in other places
- If you need more steps to finish parsing, add them. Nobody needs to see all your internal rules but you (:

The materials for this talk are available on GitHub and GitLab at `AuroransSolis/rustconf-2023.git`.