Macros in Rust

Syntax Extensions and Metaprogramming

Jan Ferdinand Sauer

Why Extend Rust's Syntax?

- grant convenience
- avoid boilerplate
- fun

```
println!("Hello, Basel!")
#[tokio::main]
#[derive(serde::Deserialize)]
python! { [x**5 for x in range(10)] }
```

The Power of Macros

- variable number of arguments
- embed DSLs

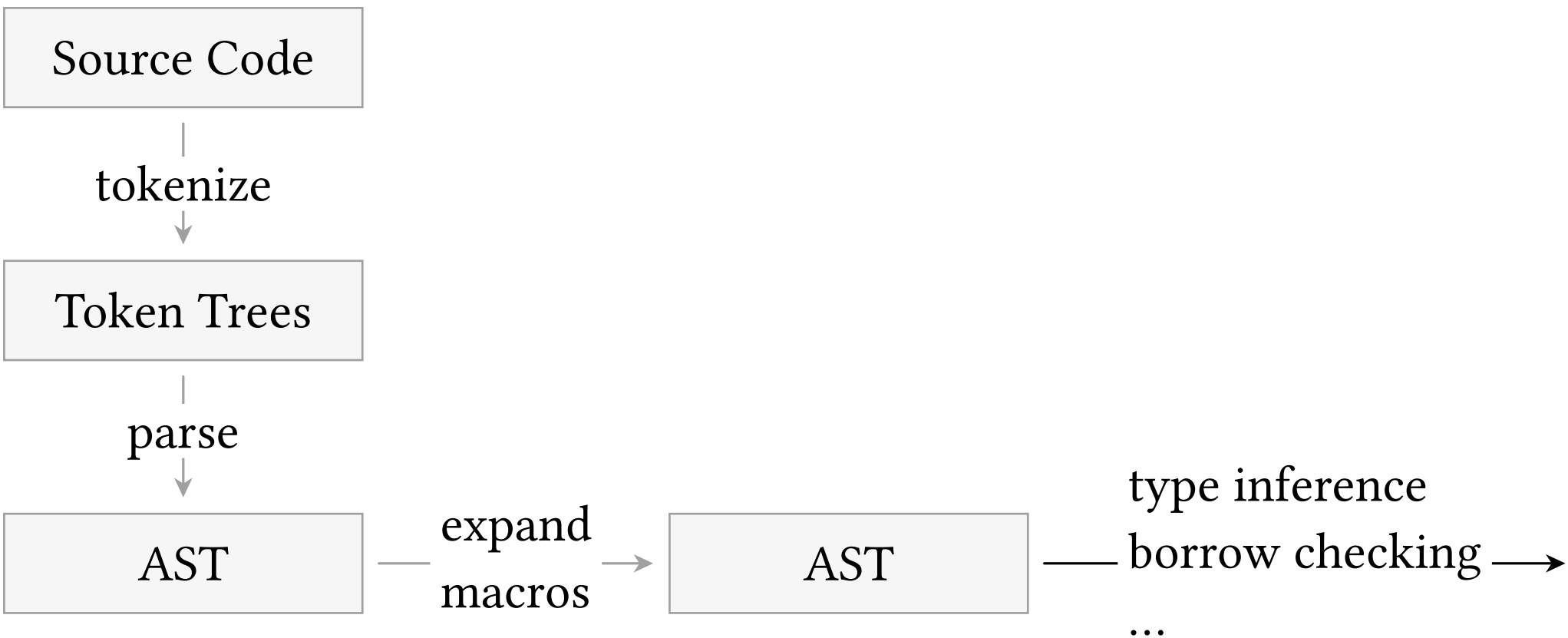
```
vec![0, 1, 2, 4]
```

```
table! { users {
  id -> Integer,
  name -> VarChar,
}}
```

- …

metaprogramming — let code write code

Rust Compiler Pipeline



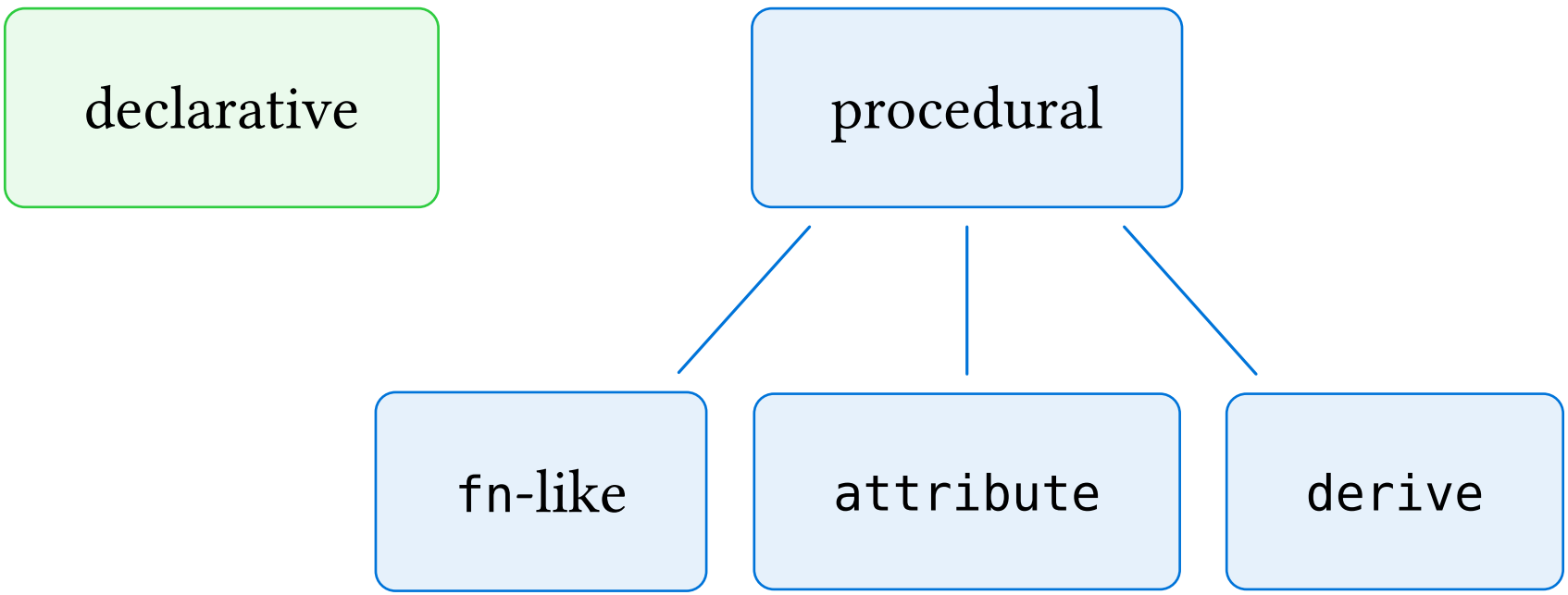
Legal Positions

- items
- statements
- expressions
- patterns
- types

```
my_impl! {
    pub fn foo() -> my_type!() {
        let pat!() = some!(0) else {
            panic!();
        };

        result!()
    }
}
```

Macro Flavors



```
macro_rules! vec { }
^^^^^^^^^^^^^^^^^^^^
error: macros must contain at least one rule
```

```
macro_rules! vec {
    () => {
        Vec::new()
    };
    ($it:expr; $len:expr) => {
        ::std::vec::from_elem($it, $len)
    };
    ($($it:expr),*) => {{
        let mut v = vec![];
        $(v.push($it);)*
        v
    }};
}
```

(actual `vec!` is better)

```
vec![0, 1, 42]
```

⇩

```
let mut v = vec![];
v.push(0);
v.push(1);
v.push(42);
v
```

A Match Made in Order

```
macro_rules! $name {
    ($matcher_0) => {$expansion_0};
// …
    ($matcher_i) => {$expansion_i};
// …
    ($matcher_n) => {$expansion_n};
}
```

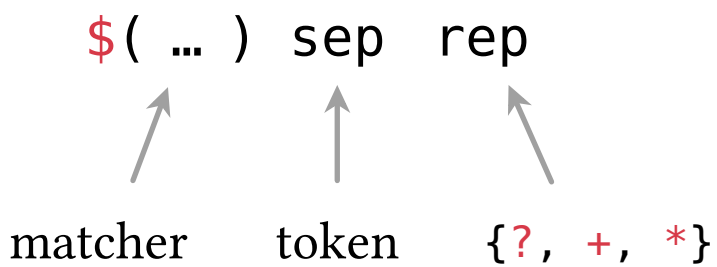
← match?
← match?
← match!
⇩
expand

Metaprogramming Means Metavariables

```
macro_rules! meta {
    ($b:block) => { "a block" };
    ($e:expr)  => { "an expression" };
    ($i:ident) => { "an identifier" };
    ($i:item)  => { "an item (e.g. fn, struct, mod, …)" };
    ($p:pat)   => { "a pattern" };
    ($p:path)  => { "a path (e.g. std::iter::empty)" };
    ($s:stmt)  => { "a statement" };
    ($t:tt)    => { "one token tree" };
    ($t:ty)    => { "a type" };
    (more)     => { "look it up ☺" };
}
```

"The Little Book of Rust Macros"

etition, Repetition, Repetition, Repetition, Repetition, Repeti




```
macro_rules! repetition {
    ($( . )?)      => { () };      // zero or one
    ($( $t:tt ),+) => { $($t)|+ }; // one or more
    ($( $t:tt ).*) => { $($t)^* }; // zero or more
}
```

Hygiene — Oh, Bother!

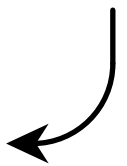
```
macro_rules! create_x {
    () => { let mut x = 0; };
}

create_x!();
x = 42;
```

⇨

```
let mut x = 0;
x = 42;
```

different syntax contexts



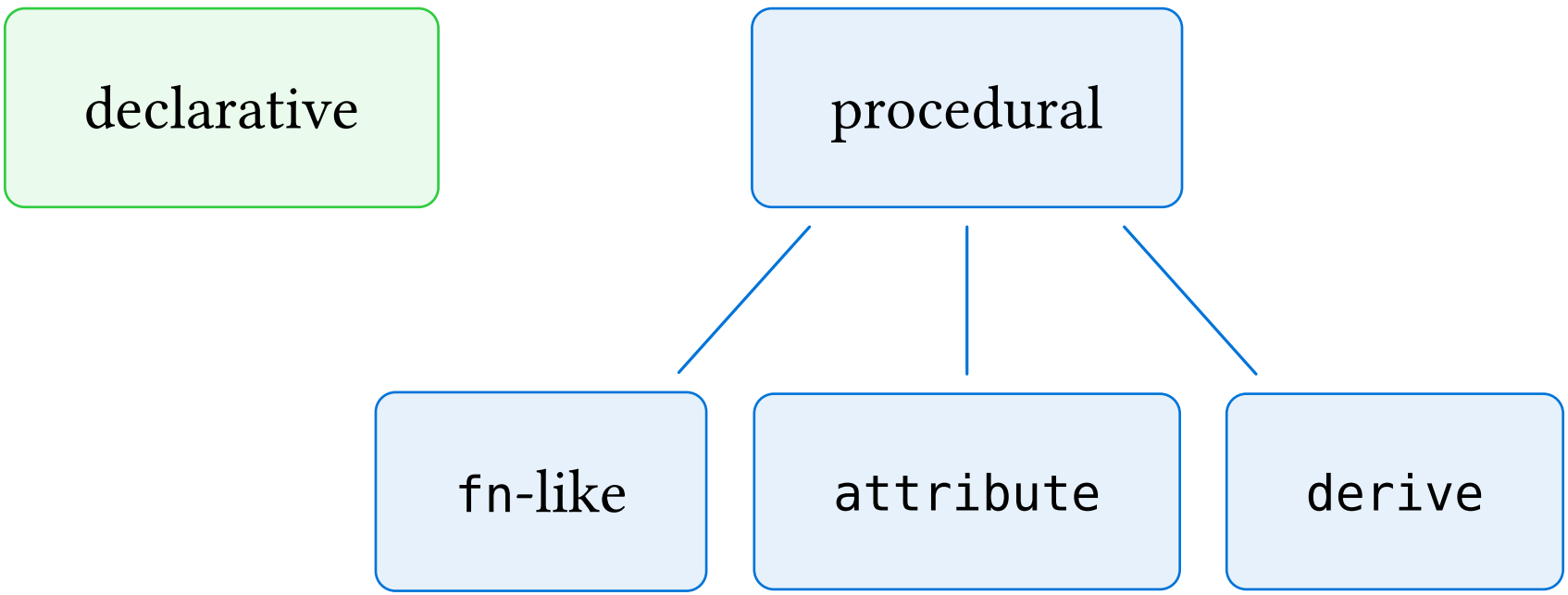
```
^
error[E0425]: cannot find value `x` in this scope

help: an identifier with the same name is defined here,
      but is not accessible due to macro hygiene
```

Macro Standoff

	declarative	procedural
declare	in-line	separate crate
parse	matcher	custom (or `syn`, `unsynn`, …)
transform	substitution	custom
hygiene	mixed-site	none

Macro Flavors



Proc Macro – `fn`-like

```
#[proc_macro]
pub fn my_macro(
    input: TokenStream,
) -> TokenStream {
    "println!(\"Hi!\");".parse().unwrap()
}

// macro usage
my_macro!();          ⇨          println!("Hi");
```

Proc Macro – `fn-like`

```
diesel::table! { users {
-     id -> VarChar
+     id -> Integer
}}
```

```
// generated code
+ impl<Rhs> ::std::ops::Add<Rhs> for id /* … */
+ impl<Rhs> ::std::ops::Sub<Rhs> for id /* … */
+ impl<Rhs> ::std::ops::Div<Rhs> for id /* … */
+ impl<Rhs> ::std::ops::Mul<Rhs> for id /* … */
```

Hygiene — What's That?

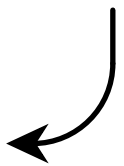
```
// proc_macro
output += "let mut x = 0;";

// macro usage
create_x!();
x = 42;
```

⇨

```
let mut x = 0;
x = 42;
```

same syntax contexts



```
Compiling …
 Finished …
```

Proc Macros – Attribute

```rust
#[proc_macro_attribute]
pub fn my_macro(
    args: TokenStream,
    item: TokenStream,
) -> TokenStream {
    "impl Foo { … }".parse().unwrap()
}

// macro usage
#[my_macro(args)]
item { … }
```

⇨

```rust
impl Foo { … }
```

Proc Macros – Attribute

```
#[tokio::main(flavor = "multi_thread", worker_threads = 10)]
async fn main() { foo().await; }
```

```
fn main() {
    tokio::runtime::Builder::new_multi_thread()
        .worker_threads(10)
        .enable_all()
        .build()
        .unwrap()
        .block_on(async { foo().await; })
}
```

Proc Macros – Derive

```rust
#[proc_macro_derive(MyDerive), attributes(helper_attr)]
pub fn my_derive(input: TokenStream) -> TokenStream {
    "struct FooBuilder { … }".parse().unwrap()
}

// macro usage
#[derive(MyDerive)]
struct Foo{ #[helper_attr] x: () }
```

```rust
struct Foo{ #[helper_attr] x: () }
struct FooBuilder { … }
```

Closing Thoughts

- Macros are powerful
- Macros incur costs
  - Mental
  - Compile times
  - Sometimes: generate *lots* of code
  - Proc macros: separate crate
- Difficult to get “right”
  - Testing
  - Error reporting
  - Documentation of generated items
- Prefer `fn`, traits, generics where possible

Where to Go from Here?

Declarative The Little Book of Rust Macros
Procedural Proc Macro Workshop

Useful Resources

- cargo expand
- astexplorer.net
- proc_macro2
- syn, unsynn, quote
- proc_macro_error
- trybuild
- zyn

Further Reading

- blog.turbo.fish
- How much code does that proc macro generate?
- A daft proc-macro trick

Macros in Rust

Syntax Extensions and Metaprogramming

Jan Ferdinand Sauer

Quiz Time! – Hygiene

```
macro_rules! internal_x {
    ($s:stmt) => {
        let mut x = 1;
        $s
    };
}

let mut x = 0;
internal_x!(x += 1);

println!("{x}"); // 1 or 2?
```

Quiz Time! – Hygiene

```
// proc_macro
let mut_x_1 = format!("let mut x = 1; {input}");
mut_x_1.parse().unwrap()

// macro usage
let mut x = 0;
mut_x_1!(x += 1);

println!("{x}"); // 1 or 2?
```

Quiz Time! – Brace Yourself

```
// proc_macro
let weird = match input {
    "{" => 0,
    _ => 1
};

// macro usage
let x = weird!({});
println!("{x}"); // 0, 1, or panic?
```

Quiz Time! – Double or Nothing

```
// this compiles
match () {
    () => {}
    () => {}
}

// does this compile?
macro_rules! double_empty {
    () => {};
    () => {};
}
```