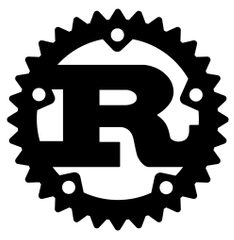


Procedural Macros

vs

Sliced Bread

Alex Crichton

Rise of Procedural Macros

- Jan 2014 - PR #11151 lands, first "loadable syntax extensions"
- Jan 2014 to present - regret #11151 landing
- Feb 2016 - RFC 1561, modularizing macros
- Feb 2016 - RFC 1566, first proc macro specification
- July 2016 - RFC 1681, Derive Macros (serde!)
- Apr 2018 - Call for stabilization of macros other than `#[derive]`
- Oct 2018 - Attribute and function-like macros on stable

Rise of Sliced Bread 

- 1912 - Otto Frederick Rohwedder, of Iowa, prototypes first bread slicing machine
- 1912 - prototype machine destroyed in fire
- 1928 - Chillicothe Baking Company sells first sliced bread
- Jan 1943 - US bans sliced bread as a wartime conservation measure
- Mar 1943 - ban rescinded, sliced bread too popular
- 1912 to present - sliced bread still tasty

Procedural Macros?!

Writing Macros

Future of Macros

Macro Forms

```
println!(...)
```

```
#[derive(Serialize)]
```

```
#[wasm_bindgen]
```

The `proc-macro` crate type

```
[package]
name = "my-macro"
version = "0.1.0"

[lib]
proc-macro = true
```

println!(...)

```
#[proc_macro]
fn println(input: TokenStream)
    -> TokenStream
{
    // ...
}
```

```
println!("wheat");
```

```
std::io::print(format_args!("wheat"));
```

#[derive(Serialize)]

```
#[proc_macro_derive(Serialize)]
fn derive_ser(input: TokenStream)
    -> TokenStream
{
    // ...
}
```

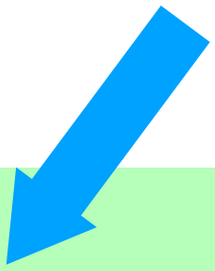
```
#[derive(Serialize)]
struct Rye { grains: i32 }
```

```
impl Serialize for Rye { ... }
```

#[wasm_bindgen]

```rust
#[proc_macro_attribute]
fn wasm_bindgen(
    args: TokenStream,
    input: TokenStream,
) -> TokenStream
{ /* ... */ }
```

```rust
#[wasm_bindgen(start)]
fn bake() { /* ... */ }
```



```rust
#[no_mangle]
#[export_name = "bake"]
pub extern fn __wbg_bake() { bake() }

fn bake() { /* ... */ }
```

`TokenStream`?

- Lexical foundation for Rust syntax
- Provided by `proc_macro` compiler crate
- `"Rc<Vec<TokenTree>>"`

TokenTree?

```
enum TokenTree {
    Group(Group),
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
}

#[derive(Serialize)]
#[serde(rename_all = "kebab-case")]
struct Sourdough {
    sour_factor: u32,
    starter_dynasty: Dynasty,
}
```

TokenTree?

```
enum TokenTree {
    Group(Group),
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
}
```

```
#[derive(Serialize)]
#[serde(rename_all = "kebab-case")]
struct Sourdough {
    sour_factor: u32,
    starter_dynasty: Dynasty,
}
```

TokenTree?

```
enum TokenTree {
    Group(Group),
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
}

#[derive(Serialize)]
#[serde(rename_all = "kebab-case")]
struct Sourdough {
    sour_factor: u32,
    starter_dynasty: Dynasty,
}
```

TokenTree?

```
enum TokenTree {
    Group(Group),
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
}

#[derive(Serialize)]
#[serde(rename_all = "kebab-case")]
struct Sourdough {
    sour_factor: u32,
    starter_dynasty: Dynasty,
}
```

Modularized Macros

Brings in `trait` as a bonus!

```
use std::println;
use serde::Serialize;
use wasm_bindgen::prelude::wasm_bindgen;

// or...

pub use wasm_bindgen_macro::wasm_bindgen;
```

Procedural Macros?!

Writing Macros

Future of Macros

```
#[proc_macro]
pub fn println(input: TokenStream)
    -> TokenStream
{
    // ???
}
```

```
    println!("{} loaves please", count);
```

```
struct MyInvocation {
    format: String,
    args: Vec<Expr>,
}
```

```
#[proc_macro]
pub fn println(input: TokenStream)
    -> TokenStream
{
    let input = MyInvocation::parse(input)?;
    input.validate()?;
    input.expand()
}
```

```
    println!("{} loaves please", count);
```

```
struct MyInvocation {
    format: String,
    args: Vec<syn::Expr>,
}
```

```
syn = "0.15"
quote = "0.6"
```

```
#[proc_macro]
pub fn println(input: TokenStream)
    -> TokenStream
{
    let input = MyInvocation::parse(input)?;
    input.validate()?;
    input.expand()
}
```

Parsing with `syn`

- Provides parsers for all Rust syntax
- Easily define custom recursive descent parsers
- Preserves `Span` information (hard, but important!)
- Aggressively feature gated to compile quickly

Parsing with `syn`

```
use syn::{parse_macro_input, DeriveInput};

#[proc_macro_derive(MyMacro)]
pub fn my_macro(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    // ...
}
```

Expanding with `quote`

- Only way to create a `TokenStream` is `FromIterator`
- The `quote` crate provides a `quote!` macro for quasi quoting
- Custom types interpolated with `ToTokens` trait

```
TokenStream::from_iter(vec![
    TokenTree::from(Ident::new("let", span)),
    TokenTree::from(Ident::new("slices", span)),
    TokenTree::from(Punct::new('=', Spacing::Alone)),
    TokenTree::from(Literal::u32_unsuffixed(42)),
    TokenTree::from(Punct::new(';', Spacing::Alone)),
])
```

vs...

```
quote! {
    let slices = 42;
}
```

Interpolation in `quote`

```
let name: Ident = ...;
let fields: Vec<TokenStream> = ...;

quote! {
    struct #name {
        #(#fields),*
    }

    impl BakeBread for #name {
        fn place_in_oven(&self) {
            // ...
        }
    }
}
```

Working with Span

```
#[bake_at(375)]
pub fn knead(bread: &Bread) {
    let (a, b) = bread.split_in_half();
}
```

```
error[E0599]: no method named `split_in_half` found for type `&Bread` in the current scope
 --> src/main.rs:4:1
  |
4 | #[bake_at(375)]
  | ^^^^^^^^^^^^^^^
```

VS

```
error[E0599]: no method named `split_in_half` found for type `&Bread` in the current scope
 --> src/main.rs:6:24
  |
6 |     let (a, b) = bread.split_in_half();
  |                        ^^^^^^^^^^^^^^
```

Working with Span

```
#[bake_at(375)]
pub fn knead(bread: &Bread) {
    let (a, b) = bread.split_in_half();
}
```

```
error: please specify what kind of bread
 --> src/main.rs:5:22
  |
5 | pub fn knead(bread: &Bread) {
  |                      ^^^^^
  |
```

Procedural Macros?!

Writing Macros

Future of Macros

Hygiene

```
quote! {
    impl MyTrait for #the_type {
        // ...
    }
}
```

Hygiene

```
quote! {
    impl my_crate::MyTrait for #the_type {
        // ...
    }
}
```

Hygiene

```
[dependencies]
other-name = { package = "my-crate", version = "1.0" }
```

```
quote! {
    impl ???::MyTrait for #the_type {
        // ...
    }
}
```

Diagnostics API

```
error[E0308]: mismatched types
   --> src/main.rs:14:10
    |
14  |         bake(bread);
    |              ^^^^^
    |              |
    |              expected &Bread, found struct `Bread`
    |              help: consider borrowing here: `&bread`
    |
    = note: expected type `&Bread`
               found type `Bread`
```

Debugging Macros

```
error: expected one of `!` or `::`, found `<eof>`
 --> src/main.rs:4:1
  |
4 | #[bake_at(375)]
  | ^^^^^^^^^^^^^^^
  |
```

Procedural Macro Gallery

```
#[remain::sorted]
pub enum Bread {
    Focaccia,
    Rye,
    Sourdough,
    Wheat,
}
```

https://github.com/dtolnay/remain

Procedural Macro Gallery

```rust
#[derive(StructOpt)]
#[structopt(name = "bake", about = "Bake some bread.")]
struct Opt {
    #[structopt(short = "t", long = "temperature")]
    temp: u32,
    #[structopt(short = "d", long = "duration", default_value = "3600")]
    dur: u64,
    // ...
}
```

https://github.com/TeXitoi/structopt

Procedural Macro Gallery

```
gobject_gen! {
    class MyBread: GObject {
        slices_left: Cell<i32>,
        consumers: RefCell<Vec<String>>,
    }

    impl MyClass {
        virtual fn eat_slice(&self, who: &str) {
            // ...
        }
    }
}
```

https://gitlab.gnome.org/federico/gnome-class

Procedural Macro Gallery

```
#[no_panic]
fn bake(at: u32) -> Bread {
    assert!(at >= 350);
    // ...
}
```

https://github.com/dtolnay/no-panic

 Questions? 

- https://github.com/dtolnay/proc-macro-workshop
- https://doc.rust-lang.org/reference/procedural-macros.html
- https://docs.rs/syn
- https://docs.rs/quote