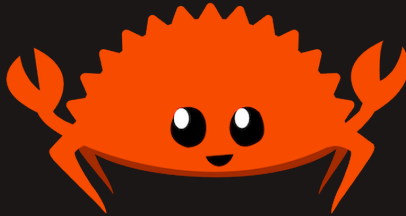




# Skalierbares Design prozeduraler Makros

*Scalable design of procedural macros*





# Content

1. What are proc-macros
2. How do proc-macros work?
3. Common proc-macro crates
4. Scalable design
5. Shortcomings



# Why do we need macros?

## Without macros

```
let a = {  
    let mut v = Vec::new();  
    v.push(1);  
    v.push(2);  
    v.push(3);  
    v  
};
```

## With macros

```
macro_rules! vec {  
    ($($x:expr),+) => ({  
        let mut v = Vec::new();  
        $( v.push($x); )+  
        v  
    });  
}  
  
let a = vec![0, 1, 2];  
let b = vec![3, 4, 5];  
let c = vec![6, 7, 9];  
// ...
```



## **Who uses macros regularly?**



## Commonly used proc-macros

- `serde: #[derive(Serialize)]`
- `thiserror: #[derive(Error)]`
- `async_trait: #[async_trait]`
- `tokio: #[tokio::main]`
- `Relm4:`

```
view! {  
    gtk::Box {  
        gtk::Label {  
            set_label: "This is a label",  
        }  
    }  
}
```

## Some numbers

syn is the most downloaded crate with over  
**13.000.000** downloads per month.





# Proc-macro architecture

- declarative macros      syntax patterns  $\Rightarrow$  Rust code
- procedural macros      tokens  $\xRightarrow{\text{Rust code}}$  Rust code



## Function-like proc-macros

```
#[proc_macro]
pub fn make_answer(_item: TokenStream) → TokenStream {
    "fn answer() → u32 { 42 }".parse().unwrap()
}

// ...

make_answer!()
```





## Derive proc-macros

```
#[proc_macro_derive(AnswerFn)]
pub fn derive_answer_fn(_item: TokenStream) → TokenStream {
    "fn answer() → u32 { 42 }".parse().unwrap()
}

// ...

#[derive(AnswerFn)]
struct Struct {
    // ...
}
```



# Attribute proc-macros

```
#[proc_macro_attribute]
pub fn return_as_is(_attr: TokenStream, item: TokenStream) →
TokenStream {
    item
}
```

```
// ...
```

```
#[return_as_is]
struct MyStruct;
```

```
#[return_as_is(further_attrs)]
enum MyEnum {}
```



## Valid syntax

Invalid

```
#[quick_methods]
struct MyStruct {
    some_value: u8,
    method ⇒ |_| "test",
}
```

Valid

```
quick_methods! {
    struct MyStruct {
        some_value: u8,
        method ⇒ |_| "test",
    }
}
```



## Valid syntax

Invalid: Brackets must match

```
c_code! {  
    #define closing_bracket }  
    {  
        int i = 0;  
        closing_bracket  
    }
```



# The proc-macro crate

- Part of the Rust toolchain
- Link between rustc and proc-macro library
- Very basic
- Only works in proc-macros (no tests)
- Based on TokenTree:

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



## The proc-macro2 crate

- Wrapper on top of proc-macro
- Works outside of proc-macros
- Can be used in in `#[test]` or `build.rs`

## The syn crate

- Convenient parsing
- Pre-defined items (struct, enum, impl, ...)
- Utilities for processing and error handling





## The syn crate - Parse trait

```
struct ItemStruct {  
    struct_token: Token![struct],  
    ident: Ident,  
    brace_token: token::Brace,  
    fields: Punctuated<Field, Token![,]>,  
}
```





## The syn crate - Parse trait

```
impl Parse for ItemStruct {
    fn parse(input: ParseStream) → Result<Self> {
        let content;
        Ok(ItemStruct {
            struct_token: input.parse()?,
            ident: input.parse()?,
            brace_token: braced!(content in input),
            fields:
content.parse_terminated(Field::parse_named, Token![,])?,
        })
    }
}
```



# The quote crate

- Convert tokens and syn data structures into tokens
- Produces TokenStreams for the code generation of the macro

```
let struct_name = "MyStruct";
let field_name = format_ident!("_{}", ident);
quote! {
    struct #struct_name {
        #field_name: u8;
    }
}
```



# Anti-patterns

- Unfamiliar syntax
  - Higher learning curve
  - More custom parsing
- Highly conditional parsing
  - Avoid conditional syntax

```
pub(crate) ? // struct | enum | type ...
```

- Don't use context-dependent parsing

```
#[my_macro()]  
enum MyEnum {} // Works  
#[my_macro("only-structs-please")]  
enum MyEnum2 {} // Error
```



## **Good design patterns**

1. 3-step processing: Parsing -> Logic -> Code-generation
2. Multiple streams
3. Spanned tokens/errors
4. Error recovery/Fallback
5. Visitors



## Good design patterns - 3-step parsing

1. `main.rs`
  1. Entry point
  2. Type definitions for parsing
2. `parse.rs`
  1. Parse implementations
3. `gen.rs`
  1. Logic
  2. Code generation



## Good design patterns - Spanned errors

```
#[crate::track]
struct Test {
    // This is needed → #[tracker::no_eq]
    c: Empty,
}
```

```
error[E0369]: binary operation `!=` cannot be applied to type `test::Empty`
--> src/lib.rs:177:9
```

```
177 |         c: Empty,
    |         ^
```



## Good design patterns - Error recovery

```
match MyStruct::parse(input) {
    Ok(my_struct) => my_struct.generate_code(),
    Err(err) => {
        quote! {
            impl SomeTrait for MyStruct {
                fn some_method() {
                    todo!()
                }
            }
        }
    }
}
```



## Shortcomings

- Difficult to get right
- Hard to read and review
- Black box for the compiler
  - Bad language server integration
- Sandboxing
- Bad hygiene
  - Clean imports only through use `:: crate_name;`
  - Does not work well with re-exports
- Parsing logic can't be used for writing formatters