Metaprogramming - Macros

Lukáš Hozda

2025-04-10

Braiins Systems s.r.o

Metaprogramming

John McCarthy: one of programming's major magic grandpas, Lisp inventor, AI pioneer

Metaprogramming

- Programs as data

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**
    - ORMs, WSDL generation, Protocol buffers/gRPC

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**
    - ORMs, WSDL generation, Protocol buffers/gRPC
- Runtime:

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**
    - ORMs, WSDL generation, Protocol buffers/gRPC
- Runtime:
  - **Reflection**

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**
    - ORMs, WSDL generation, Protocol buffers/gRPC
- Runtime:
  - **Reflection**
  - **Dynamic code evaluation**

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**
    - ORMs, WSDL generation, Protocol buffers/gRPC
- Runtime:
  - **Reflection**
  - **Dynamic code evaluation**
  - **Metaclasses**

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**
    - ORMs, WSDL generation, Protocol buffers/gRPC
- Runtime:
  - **Reflection**
  - **Dynamic code evaluation**
  - **Metaclasses**
    - e.g Python

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**
    - ORMs, WSDL generation, Protocol buffers/gRPC
- Runtime:
  - **Reflection**
  - **Dynamic code evaluation**
  - **Metaclasses**
    - e.g Python
- Both:

Metaprogramming

- Programs as data
- Either modifies program at runtime (self-modifying, extensions) or at compile-time
- Compile-time:
  - **Macros and preprocessors**
  - **Templating**
  - **Codegen tools**
    - ORMs, WSDL generation, Protocol buffers/gRPC
- Runtime:
  - **Reflection**
  - **Dynamic code evaluation**
  - **Metaclasses**
    - e.g Python
- Both:
  - **Multi-stage programming**

Macros in Rust

Macros in Rust

- In Rust:
  - Declarative macros (**decl macros**)
    - Analogue to C/C++ macros
    - At least a bit hygienic
    - Pattern-matching
  - Procedural macros (**proc macros**)
    - Essentially programs that take a stream of token as input and produce a token stream
    - Un-hygienic
    - What you do is up to you
- Attributes are macros too
- Call syntax:
  - Always macro!(), macro![], or macro!{}
  - ↑ parentheses/braces/square brackets interchangable

Macros in Rust

- Strange place:
  - Not exactly first class citizens - placement within file, macro_export/use
- Some compiler built-ins pretend to be macros:

```
// src/core/macros/mod.rs
#[stable(feature = "rust1", since = "1.0.0")]
#[rustc_builtin_macro]
#[macro_export]
macro_rules! concat {
    ($($e:expr),* $(,)?) => {{ /* compiler built-in */ }};
}
...
macro_rules! stringify {
    ($($t:tt)*) => {
        /* compiler built-in */
    };
}
```

Macros in STD

Name	Desc
`line!`, `module_path!`, `file!`, `column!`	builtin position macros
`assert!`, `assert_ne!`, `assert_eq!`	asserts
`matches!`	pattern matching shorthand
`format_args!`	builtin macro for implementing string format macros
`format!`, `(e)print!`, `(e)println!`, `write!`, `writeln!`	string format macros
`dbg!`	prints and returns value - for quick & dirty debugging
`stringify!`, `concat!`, `compile_error!`	helper macros
`vec!`	shorthand for Vec<T> creation

Macros in STD

Name	Desc
`panic!`	builtin for panicking with message
`todo!`, `unimplemented!`, `unreachable!`	semantic panic macros
`env!`, `option_env!`	builtins for env variables
`include!`, `include_str!`, `include_bytes!`	copy contents of file here

Declarative Macros

- Pattern matching
- Can be recursive
  - But a macro invocation cannot be a parameter to another macro invocation
    - Solutions: proposal for macro!!() (eager syntax), hacks with proc macros
- Hygiene:

```
macro_rules! using_a {
    ($e:expr) => {
        {
            let a = 42;
            $e
        }
    }
}
let four = using_a!(a / 10);
```

- ↑ won't compile - why?

Declarative Macros

```
let four = {
    let a = 42;
    /* own syntax context */ a / 10 /**/
};
```

Fix:

```
macro_rules! using_a {
    ($a:ident, $e:expr) => {
        {
            let $a = 42;
            $e
        }
    }
}

let four = using_a!(a, a / 10);
```

Declarative Macros

Hygiene: local variables, labels, `$crate`

Writing our own decl macro

```
macro_rules! assert_min_args {
    (@count) => { 0 };
    (@count $first:expr $(, $rest:expr)*) => {
        1 + assert_min_args!(@count $($rest),*)
    };

    ($min_count:expr, $($args:expr),+ $(,)?) => {
        const _: () = {
            let count: usize = assert_min_args!(@count $($args),+);
            assert!(count >= $min_count, "Not enough arguments provided");
            ()
        };
    };
}
```

Writing our own decl macro

```
fn main() {
    // yay
    assert_min_args!(2, "a", "b", "c");

    // nay
    //assert_min_args!(4, "a", "b");
}
```

Matchers - Shamelessly copied from Rust reference

- `item`: an Item
- `block`: a BlockExpression
- `stmt`: a Statement without the trailing semicolon (except for item statements that require semicolons)
- `pat_param`: a PatternNoTopAlt
- `pat`: at least any PatternNoTopAlt, and possibly more depending on edition
- `expr`: an Expression
- `ty`: a Type
- `ident`: an IDENTIFIER_OR_KEYWORD or RAW_IDENTIFIER
- `path`: a TypePath style path
- `tt`: a TokenTree (a single token or tokens in matching delimiters (), [], or {})
- `meta`: an Attr, the contents of an attribute
- `lifetime`: a LIFETIME_TOKEN
- `vis`: a possibly empty Visibility qualifier
- `literal`: matches LiteralExpression

Repetition et al.:

- `$()` - encloses pattern to be repeated
- `*` - zero or more
- `+` - one or more
- `?` - zero or one (aka optional)
- Smart about delimiters (include your delimiter character before after the `)`)
- `$crate` - refers to the crate macro is defined in - use to call things from your crate
  - Note that it is the absolute path to the root of your crate

```
macro_rules! vec_of_stringified {
    ($($element:expr),*) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push(stringify!($element));
            )*
            temp_vec
        }
    };
}
```

Procedural Macros

- Essentially programs
- Use the `proc_macro` builtin library
  - Available **only** in procedural macro crates
- Every proc macro needs to be its own library crate
  - Technical reasons (compile down to dynamic libraries linked to the compiler)
  - Pattern: Crates for all proc macros, then one top-level crate that re-exports them all
- Commonly used crates:
  - `syn`: Rust parser
  - `quote`: Crate for quasi-quoting (“templating” your output token stream)
  - `proc_macro2`: Substitute for `proc_macro`, so you can write libraries for macro development
- You can do literally anything:
  - inline_python, inline_rust
- Types: **function-like** and **attributes** (special case: **#[derive()]**s)

```rust
use inline_python::python;

fn main() {
    let who = "world";
    let n = 5;
    python! {
        for i in range('n'):
            print(i, "Hello", 'who)
        print("Goodbye")
    }
}
```

```
const CONST_FOR_LOOP: i32 = inline_rust!({
    let mut sum: i32 = 0;
    for n in 0..30 {
        sum += n;
    }
    format!("{}", sum)
});
```

Writing our own proc_macro

```toml
[package]
name = "my_proc_macro"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

Simple proc macro

```
use quote::quote;
use proc_macro::TokenStream;

#[proc_macro]
pub fn make_function(input: TokenStream) -> TokenStream {
    let name = syn::parse::<syn::Ident>(input).unwrap();
    let output = quote! {
        fn #name() -> String {
            String::from("Hello, world!")
        }
    };
    output.into()
}
```

Derive

```
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput};

#[proc_macro_derive(SimpleDebug)]
pub fn simple_debug_derive(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    let name = &input.ident;
    let gen = quote! {
        impl std::fmt::Debug for #name {
            fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
                write!(f, stringify!(#name))
            }
        }
    };
    gen.into()
}
```

Attribute macro

```rust
extern crate proc_macro;

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, AttributeArgs, ItemFn, parse_macro_input,
parse_args};

#[proc_macro_attribute]
pub fn log_execution(args: TokenStream, input: TokenStream) -> TokenStream {
    // Parse the input TokenStream (the function the attribute is applied to)
    let input_fn = parse_macro_input!(input as ItemFn);

    // You can also parse and use args if your macro needs arguments
    let _args = parse_macro_input!(args as AttributeArgs);

    // Retrieve the function's name
    let fn_name = &input_fn.sig.ident;
```

Attribute macro

```
    // Generate the new function, wrapping the original function body
    let output = quote! {
        #input_fn
        fn modified_#fn_name() {
            println!("Executing function: {}", stringify!(#fn_name));
            #fn_name();
            println!("Finished executing: {}", stringify!(#fn_name));
        }
    };

    // Return the generated code
    output.into()
}
```

```rust
#[log_execution]
fn hello() {
    println!("Hello, world!");
}

fn main() {
    modified_hello(); // This is the function generated by the macro.
}
```

Development

- cargo-expand - show expanded macros from a target place
- trace_macros!() - Make rustc print macro expansion as it goes
  - Nightly
- log_syntax!() - Print passed tokens into stdout
  - Nightly
- macro_railroad - Generate SVG diagrams of macros
- The Little Book of Rust Macros

Curses - Maybe keyword, maybe not

```
macro_rules! what_is {
    (self) => {"the keyword `self`"};
    ($i:ident) => {concat!("the identifier `", stringify!($i), "`")};
}

macro_rules! call_with_ident {
    ($c:ident($i:ident)) => {$c!($i)};
}

fn main() {
    println!("{}", what_is!(self));
    println!("{}", call_with_ident!(what_is(self)));
}
```

```
the keyword `self`
the keyword `self`
```

Miscellaneous and curses

- macro_export and legacy macro_use
  - Ignore all visibility
- decl macros are only accessible after their definition
  - Does not apply to macros themselves lol
    - So long as callsite has all macros accessible, it's OK
  - Imports work as expected also :)
    - External crate macros hoisted to the top of importing crate
- Macros can be shadowed
  - Use $crate where possible when calling other of your macros
    - Good luck with proc macros
- Follow-set ambiguity restrictions
  - a nonterminal matched by a metavariable must be followed by a token which has been decided can be safely used after that kind of match

Ignoring visibility

```
mod macros {
    #[macro_export] macro_rules! X { () => { Y!(); } }
    #[macro_export] macro_rules! Y { () => {} }
}
```

Hoisting

```
X!();
#[macro_use] extern crate macs;
X!();
```

Accesibility

```
mod a {
    // X!(); // undefined
}
mod b {
    // X!(); // undefined
    macro_rules! X { () => {}; }
    X!(); // defined
}
mod c {
    // X!(); // undefined
}
```

```
macro_rules! X { (1) => {}; }
X!(1);
macro_rules! X { (2) => {}; }
// X!(1); // Error: no rule matches `1`
X!(2);

mod a {
    macro_rules! X { (3) => {}; }
    // X!(2); // Error: no rule matches `2`
    X!(3);
}
// X!(3); // Error: no rule matches `3`
X!(2);
```

Follow-set rules

- `expr` and `stmt` may only be followed by one of: `=>`, `,`, or `;`.
- `pat_param` may only be followed by one of: `=>`, `,`, `=`, `|`, `if`, or `in`.
- `pat` may only be followed by one of: `=>`, `,`, `=`, `if`, or `in`.
- `path` and `ty` may only be followed by one of: `=>`, `,`, `=`, `|`, `;`, `:`, `>`, `>>`, `[`, `{`, `as`, `where`, or a macro variable of `block` fragment specifier.
- `vis` may only be followed by one of: `,`, an identifier other than a non-raw `priv`, any token that can begin a type, or a metavariable with a `ident`, `ty`, or `path` fragment specifier.
- All other fragment specifiers have no restrictions.

Further reading

- https://veykril.github.io/tlborm/introduction.html
- https://doc.rust-lang.org/reference/macros-by-example.html
- https://doc.rust-lang.org/stable/reference/macros.html#macros

Questions?