

Declarative Macros | Detail

■ Writing Declarative Macros

- ◆ Consist of two parts: **Matchers** and **Transcribers**
- ◆ **Matchers** define input patterns to match upon
 - The input patterns are *different* than patterns used in (for example) **match** and **if let**
 - ▶ Completely different ruleset
 - Multiple *matchers* may be defined for one macro
 - ▶ Checked from top to bottom
- ◆ **Transcribers** read the input captured by the *matchers* and then emit Rust code
 - Code transcribed completely replaces the macro invocation
- ◆ Macros must appear *before* usage in code

■ Matchers

- ◆ Matchers consist of four components:
 - Metavariables
 - Fragment specifiers
 - Repetitions (covered later)
 - Glyphs: anything not listed above
 - ▶ Dollar (\$) is used by metavariables and cannot be used for glyphs
- ◆ Whitespace is ignored
 - Can be used for clarity in macro and by invoker

Metavariables

- ◆ Contain Rust code supplied by macro invoker
- ◆ Used by the transcriber to make substitutions
 - Metavariable will be substituted with the code provided by the invoker
- ◆ Metavariables start with a dollar (\$)

\$fn

\$my_metavar

\$varname

Fragment Specifiers

- ◆ Fragment specifiers determine what kind of data is allowed in a metavariable
- ◆ Available specifiers are:
 - item
 - block
 - stmt
 - pat_param / pat
 - expr
 - ty
 - ident
 - path
 - tt
 - meta
 - lifetime
 - vis
 - literal

■ Creating a Macro

```
macro_rules! your_macro_name {  
    ($metavariable_name:fragment_specifier) => {};  
    ($a:ident, $b:literal, $c:tt) => {  
        // Can use $a $b $c  
    };  
    () => {};  
}
```

Matcher

Transcriber

■ Glyphs

```
macro_rules! demo {  
  [W 0 W! _ any | thing? yes.#meta ( / ^ . ^ ) / ° ] => { };  
}
```

```
demo!(W 0 W! _ any | thing? yes. #meta( / ^ . ^ ) / ° );
```

Fragment Specifier: item

```
macro_rules! demo {
    ($i:item) => { $i };
}

demo!(const a: char = 'g');
demo! {fn hello() {}}
demo! {mod demo {}}
struct MyNum(i32);
demo! {
    impl MyNum {
        pub fn demo(&self) {
            println!("my num is {}", self.0);
        }
    }
}
```

Fragment Specifier: block

```
macro_rules! demo {  
    ($b:block) => { $b };  
}
```

```
let num = demo!(  
    {  
        if 1 == 1 { 1 } else { 2 }  
    }  
);
```

Fragment Specifier: `stmt`

```
macro_rules! demo {  
    ($s:stmt) => { $s };  
}
```

```
demo!( let a = 5 );  
let mut myvec = vec![];  
demo!( myvec.push(a) );
```

Fragment Specifier: pat / pat_param

```
macro_rules! demo {  
    ($p:pat) => {{  
        let num = 3;  
        match num {  
            $p => (),  
            1 => (),  
            _ => (),  
        }  
    }};  
}  
demo!( 2 );
```

Fragment Specifier: `expr`

```
macro_rules! demo {  
    ($e:expr) => { $e };  
}
```

```
demo!( loop {} );
```

```
demo!( 2 + 2 );
```

```
demo!( {  
    panic!();  
} );
```

Fragment Specifier: ty

```
macro_rules! demo {  
    ($t:ty) => {{  
        let d: $t = 4;  
        fn add(lhs: $t, rhs: $t) -> $t {  
            lhs + rhs  
        }  
    }};  
}  
  
demo!(i32);  
demo!(usize);
```

Fragment Specifier: ident

```
macro_rules! demo {  
    ($i:ident, $i2:ident) => {  
        fn $i() {  
            println!("hello");  
        }  
        let $i2 = 5;  
    };  
}  
  
demo!(say_hi, five);  
say_hi();  
assert_eq!(5, five)
```

Fragment Specifier: path

```
macro_rules! demo {  
    ($p:path) => {  
        use $p;  
    };  
}  
  
demo!(std::collections::HashMap);
```

Fragment Specifier: tt

```
macro_rules! demo {  
    ($t:tt) => {  
        $t {}  
    };  
}  
  
demo!(loop);  
  
demo!({  
    println!("hello");  
});
```

Fragment Specifier: meta

```
macro_rules! demo {  
    ($m:meta) => {  
        #[derive($m)]  
        struct MyNum(i32);  
    };  
}  
demo! (Debug);
```

Fragment Specifier: lifetime

```
macro_rules! demo {  
    ($l:lifetime) => {  
        let a: &$l str = "sample";  
    };  
}  
demo!('static);
```

Fragment Specifier: vis

```
macro_rules! demo {  
    ($v:vis) => {  
        $v fn sample() {}  
    };  
}  
demo!(pub);
```

■ Fragment Specifier: literal

```
macro_rules! demo {  
    ($l:literal) => { $l };  
}  
  
let five = demo!(5);  
let hi = demo!("hello");
```

■ Allowed Syntax

- ◆ Some specifiers have restrictions on what can follow
 - Prevent ambiguities between custom syntax and Rust syntax
- ◆ Specifiers with restrictions:
 - **expr, stmt, pat, path, ty, vis**
 - Compiler error will indicate what is allowed

<https://doc.rust-lang.org/reference/macros-by-example.html>

Imports

- ◆ When using **external crates** in a macro, use the full path prefixed with two colons (::)
 - `use ::std::collections::HashMap;`
- ◆ When using modules from the **current crate**, use `$crate:`
 - `$crate::module1::func();`
- ◆ This helps resolve import issues since macros can be invoked from any location

Recap

- ◆ **Matchers** define syntax to match on
 - Some restrictions placed in order to prevent ambiguities
- ◆ **Transcribers** define the code to output
- ◆ **Metavariables** contain data provided by the macro invoker
 - Used as a substitution by transcribers
- ◆ **Fragment specifiers** determine what kinds of data is allowed in a metavariable
- ◆ Use absolute paths when utilizing modules or external crates