RUST MACROS

Sebastian Vișan

University of Helsinki
Faculty of Science
Master of Computer Science

6 Nov 2025

Challenges

Metaprogramming is confusing...

Challenges

Metaprogramming is confusing...

- to implement

Challenges

Metaprogramming is confusing...

- to implement
- to discuss

Challenges

Metaprogramming is confusing...

- to implement
- to discuss
- to follow

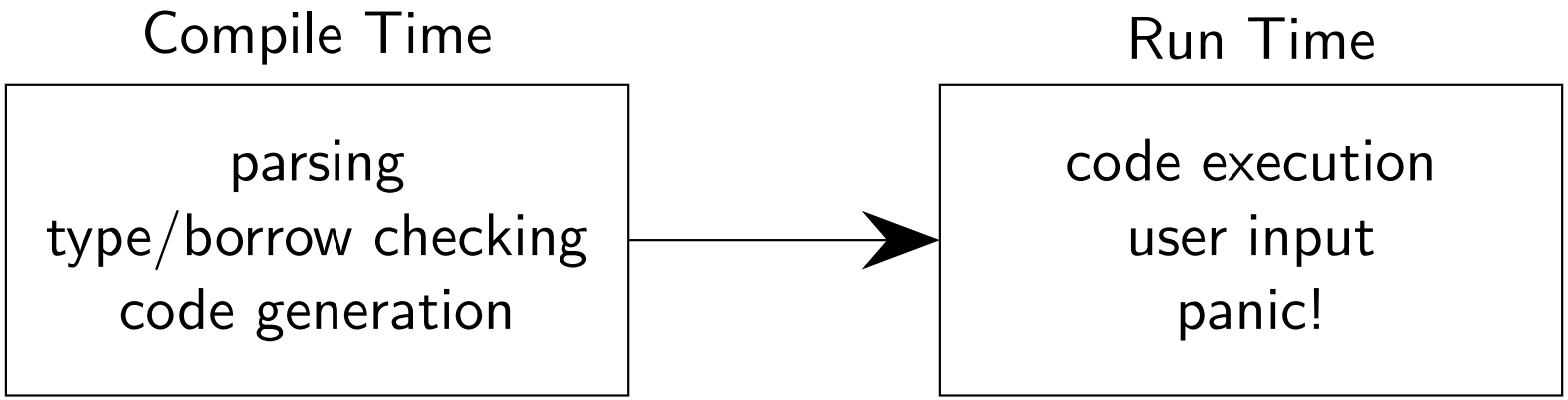
What is a *Macro*?

What is a *Macro*?

A function that outputs code

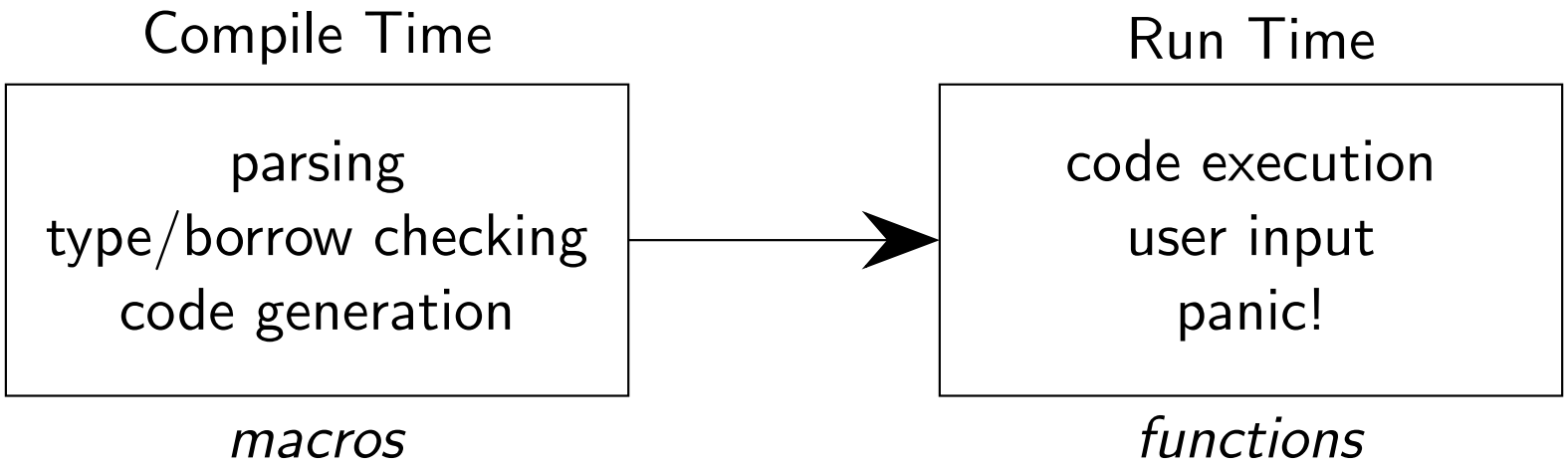
What is a *Macro*?

A function that outputs code



What is a *Macro*?

A function that outputs code



Types of Macros

Declarative

`macro_rules!`

Procedural

- **Derive**
  `#[proc_macro_derive()]`
- **Attribute-Like**
  `#[proc_macro_attribute]`
- **Function-Like**
  `#[proc_macro]`

Declarative Macros

Most common type of macro.
Work with well defined structure.

Declarative Macros

Most common type of macro.
Work with well defined structure.
Some familiar examples:

```
print!("Hello {}!", "world");
println!("{} {} <{}", "hello", "world", 3);
format!("{} * {} = {}", 2, 3, 6); // "2 * 3 = 6"
stringify!(2 * 3 = 6); // 2 * 3 = 6
vec!['l', 'm', 'n', 'o', 'p'];
vec!["abc"; 3]; // ["abc", "abc", "abc"]
assert!(2 + 2 == 4, "{} + {} != {}", 2, 2, 4);
assert_eq!(3 + 2, 4); // assertion left == right failed
panic!("error");
```

Declarative Macros

Most common type of macro.
Work with well defined structure.
Some familiar examples:

```
print!("Hello {}!", "world");
println!("{} {} <{}", "hello", "world", 3);
format!("{} * {} = {}", 2, 3, 6); // "2 * 3 = 6"
stringify!(2 * 3 = 6); // 2 * 3 = 6
vec!['l', 'm', 'n', 'o', 'p'];
vec!["abc"; 3]; // ["abc", "abc", "abc"]
assert!(2 + 2 == 4, "{} + {} != {}", 2, 2, 4);
assert_eq!(3 + 2, 4); // assertion left == right failed
panic!("error");
```

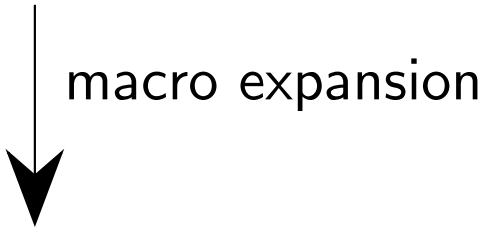
Most of these are implemented directly by the compiler.

Declarative Macros

```
let v = vec![1, 2, 3, 4, 5];
```

Declarative Macros

```
let v = vec![1, 2, 3, 4, 5];
```




```
let mut v = Vec::new();
v.push(1);
v.push(2);
v.push(3);
v.push(4);
v.push(5);
```

Declarative Macros

```
let v = vec![1, 2, 3, 4, 5];
```


```
let mut v = Vec::new();
v.push(1);
v.push(2);
v.push(3);
v.push(4);
v.push(5);
```

```
vec!(6, 7, 8);
vec!{6, 7, 8};
```

Procedural Derive Macros

Used on enums and structs.
Provide automated trait implementations.

Procedural Derive Macros

Used on enums and structs.
Provide automated trait implementations.

```
#[derive(Debug, PartialEq, PartialOrd)]
enum Bit {
    Zero,
    One,
}

let bit = Bit::Zero;
println!("Bit: {:?}", bit); // Bit: Zero
assert_eq!(bit, Bit::Zero);
assert!(Bit::Zero < Bit::One);
```

Procedural Derive Macros

Used on enums and structs.
Provide automated trait implementations.

```
#[derive(Debug, Default, Clone, Copy)]
struct Wrapper {
    val: i32,
}

let wrap = Wrapper::default();
let wrap2 = wrap.clone();
assert_eq!(wrap.val, wrap2.val);
println!("{:?}", wrap2); // Wrapper { val: 0 }
```

The `i32` field also needs to implement the traits.

Procedural Derive Macros

We can use the **cargo expand** crate to look at the generated code after the macro expansion:

```
#[automatically_derived]
impl ::core::clone::Clone for Wrapper {
    #[inline]
    fn clone(&self) -> Wrapper {
        let _: ::core::clone::AssertParamIsClone<i32>;
        *self
    }
}
```

Notice the check that `i32` implements the `Clone` trait.

Procedural Attribute-Like Macros

Very similar to custom derive macros, but also work for other items than structs and enums.

```
#[route(GET, "/")]
pub fn web_page() {}
```

Procedural Attribute-Like Macros

Very similar to custom derive macros, but also work for other items than structs and enums.

```
#[route(GET, "/")]
pub fn web_page() {}
```

Other familiar attributes:

```
#![crate_type = "lib"]

#[test]
#[unsafe(no_mangle)]
#[allow(dead_code, unused_variables)]
#[rustfmt::skip]
fn my_func() {
      let   x = 2 ;
}
```

Procedural Function-Like Macros

Invocation looks like that of declarative macros.
They work on arbitrary token streams as opposed to defined structured patterns.

```rust
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream { ... }

let sql = sql!(SELECT * FROM posts WHERE id=1);
```

Procedural Function-Like Macros

Invocation looks like that of declarative macros.
They work on arbitrary token streams as opposed to defined structured patterns.

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

let sql = sql!(SELECT * FROM posts WHERE id=1);
```

Custom syntax allows us to write embedded DSLs (domain specific languages).
These macros are very powerful, but less readable.

Example1 - timed!

Problem: benchmark how much time a specific block of code takes to run.

Example1 - timed!

Problem: benchmark how much time a specific block of code takes to run.
Solution: measure the time before and after, then subtract.

Example1 - timed!

Problem: benchmark how much time a specific block of code takes to run.
Solution: measure the time before and after, then subtract.

```rust
use std::time::SystemTime;
// Measure the time before
let start = SystemTime::now();
// Run the code we want to benchmark
let res = fibonacci(35);
// Measure the time after
let end = SystemTime::now();
// Calculate the time difference
let diff = end.duration_since(start).expect("std::time error");
// Show the message
println!("result {} took: {} ms", res, diff.as_millis());
// -> result 14930352 took: 64 ms
```

Example1 - timed!

Problem: benchmark how much time a specific block of code takes to run.
Solution: measure the time before and after, then subtract.

```rust
use std::time::SystemTime;
// Measure the time before
let start = SystemTime::now();
// Run the code we want to benchmark
let res = fibonacci(35);
// Measure the time after
let end = SystemTime::now();
// Calculate the time difference
let diff = end.duration_since(start).expect("std::time error");
// Show the message
println!("result {} took: {} ms", res, diff.as_millis());
// -> result 14930352 took: 64 ms
```

Problem: boilerplate
Could we write a function for this?

Example1 - timed!

Problem: benchmark how much time a specific block of code takes to run.
Solution: measure the time before and after, then subtract.

```rust
use std::time::SystemTime;
// Measure the time before
let start = SystemTime::now();
// Run the code we want to benchmark
let res = fibonacci(35);
// Measure the time after
let end = SystemTime::now();
// Calculate the time difference
let diff = end.duration_since(start).expect("std::time error");
// Show the message
println!("result {} took: {} ms", res, diff.as_millis());
// -> result 14930352 took: 64 ms
```

Problem: boilerplate
Could we write a function for this? No, can't pass code block as argument.

Example1 - timed!

Declarative macro that expands our code block into a timed version:

```
#[macro_export]
macro_rules! timed {
    ( $message: expr => $code: block ) => {{
        use std::time::SystemTime;
        // Measure the time before
        let start = SystemTime::now();
        // Run the provided block of code
        let res = $code;
        // Measure the time after
        let end = SystemTime::now();
        // Calculate the time difference
        let diff = end.duration_since(start).expect("std::time error");
        // Show the message
        println!("{} took: {} ms", stringify!($message), diff.as_millis());
        // Return the result of the code block
        res
    }};
}
```

Example1 - timed!

```
timed!(Fibonacci => { fibonacci(35) });
// Fibonacci took: 63 ms
```

Example1 - timed!

```
timed!(Fibonacci => { fibonacci(35) });
// Fibonacci took: 63 ms
```

```
{
    use std::time::SystemTime;
    let start = SystemTime::now();
    let res = { fibonacci(35) };
    let end = SystemTime::now();
    let diff = end.duration_since(start).expect("std::time error");
    {
        ::std::io::_print(format_args!(
            "{0} took: {1} ms\n",
            "Fibonacci",
            diff.as_millis()
        ));
    };
    res
};
```

Example1 - timed!

```
let nums = vec![1; 1000000];
let nums_sum = timed! {
    "vector sum" =>
    {
        println!("Calculating the sum");
        let mut sum = 0;
        for num in nums {
            sum += num;
        }
        println!("Done");
        sum
    }
};
// Calculating the sum
// Done
// "vector sum" took: 4 ms
println!("Sum: {}", nums_sum);
// Sum: 1000000
```

Example2 - comp!

What we'll build: simplified Python List Comprehension

```
[x * 2 for x <- xs]
```

Example2 - comp!

What we'll build: simplified Python List Comprehension

```
[x * 2 for x <- xs]
```

How would we write this in Rust?

```
xs.into_iter().map(|x| x * 2).collect()
```

Example2 - comp!

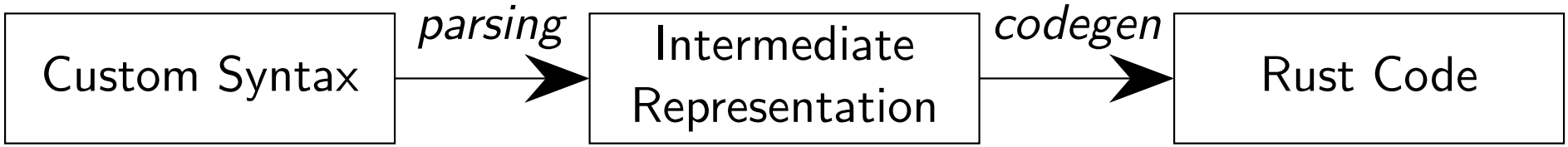
What we'll build: simplified Python List Comprehension

```
[x * 2 for x <- xs]
```

How would we write this in Rust?

```
xs.into_iter().map(|x| x * 2).collect()
```

Architecture: **syn** and **quote** crates



Example2 - comp!

Intermediate Representation:
Using `Expr` and `Pat` types from `syn`

```
// [x * 2 for x <- xs]
// expression 'for' pattern '<-' expression
struct Comp {
    mapping: Expr,
    pattern: Pat,
    sequence: Expr,
}
```

Parse `[x * w for x <- xs]` into:
`Comp {mapping = x * 2, pattern:  x, sequence:  xs}`

Example2 - comp!

Parsing using the syn crate:

```
impl Parse for Comp {
    fn parse(input: ParseStream) -> Result<Self> {
        let mapping = input.parse::<Expr>()?;
        _ = input.parse::<Token![for]>()?;
        let pattern = Pat::parse_single(input)?;
        _ = input.parse::<Token![<-]>()?;
        let sequence = input.parse::<Expr>()?;
        Ok(Comp {
            mapping,
            pattern,
            sequence,
        })
    }
}
```

Example2 - comp!

Codegen using the quote crate:

```rust
#[proc_macro]
pub fn comp(input: TokenStream) -> TokenStream {
    // Custom Syntax Parsing
    let code = parse_macro_input!(input as Comp);
    let Comp {
        mapping,
        pattern,
        sequence,
    } = code;
    // Rust Code Generation
    quote! {
        core::iter::IntoIterator::into_iter(#sequence).map(|#pattern|{
            #mapping
        })
    }
    .into()
}
```

Example2 - comp!

Results:

```
use comp_macro::comp;

let v: Vec<i32> = comp![x * 2 for x <- [1, 2, 3]].collect();
assert_eq!(v, [2, 4, 6]);

let nums = vec![12, 13, 15, 16, 17];
let evens: Vec<bool> = comp![num % 2 == 0 for num <- nums].collect();
assert_eq!(evens, [true, false, false, true, false]);

let sums = comp![x + y for (x, y) <- [(1, 2), (3, 4), (5, 6)]];
println!("Sums: {:?}", sums.collect::<Vec<i32>>());
// Sums: [3, 7, 11]
```

Example2 - comp!

Results:

```
use comp_macro::comp;

let v: Vec<i32> = comp![x * 2 for x <- [1, 2, 3]].collect();
assert_eq!(v, [2, 4, 6]);

let nums = vec![12, 13, 15, 16, 17];
let evens: Vec<bool> = comp![num % 2 == 0 for num <- nums].collect();
assert_eq!(evens, [true, false, false, true, false]);

let sums = comp![x + y for (x, y) <- [(1, 2), (3, 4), (5, 6)]];
println!("Sums: {:?}", sums.collect::<Vec<i32>>());
// Sums: [3, 7, 11]
```

Adding custom syntax would be very confusing in a real world project, but macros can be very fun to write and experiment with!

Resources

- **Code for Examples**
  https://github.com/VSebastian8/rust-macros
- **The Rust Programming Language**
  https://doc.rust-lang.org/book/ch20-05-macros.html
- **The Little Book of Rust Macros**
  https://lukaswirth.dev/tlborm/decl-macros.html
- **Comprehending Proc Macros**
  https://www.youtube.com/watch?v=SMCRQj9Hbx8