Advanced Rust - Lab 3: Rust Macros

Lukáš Hozda

2025

Introduction

It's time to write some macros.

Setup

1. Make sure you have Rust installed with `rustup`
2. For declarative macros: create a project with `cargo new declarative_macros`
3. For procedural macros: create a workspace with two crates:
    - `cargo new proc_macro_workshop --lib`
    - `cargo new proc_macro_tests --lib`

Exercise 1: Declarative Macros - Basic Pattern Matching (20 minutes)

Objective

Implement a declarative macro that allows creating hashmaps with a concise syntax.

Instructions

Create a `hashmap!` macro that supports:

1. Empty hashmap: `hashmap!{}`
2. Single key-value pair: `hashmap!{"key" => "value"}`
3. Multiple pairs: `hashmap!{"a" => 1, "b" => 2}`
4. Different types (with type parameters): `hashmap!{1 => "one", 2 => "two"}`

Requirements

```rust
// Create a hashmap! macro that works like this:
let map1: HashMap<&str, i32> = hashmap!{};
let map2 = hashmap!{"a" => 1, "b" => 2};
let map3 = hashmap!{1 => "one", 2 => "two"};

// TODO: Implement the macro
macro_rules! hashmap {
    // Write your patterns here
}

fn main() {
    use std::collections::HashMap;

    // Empty hashmap
    let empty: HashMap<&str, i32> = hashmap!{};
    assert_eq!(empty.len(), 0);

    // Single key-value pair
    let single = hashmap!{"a" => 1};
    assert_eq!(single.get("a"), Some(&1));

    // Multiple key-value pairs
    let multiple = hashmap!{"a" => 1, "b" => 2, "c" => 3};
    assert_eq!(multiple.get("b"), Some(&2));

    // Different types
    let strs = hashmap!{1 => "one", 2 => "two"};
    assert_eq!(strs.get(&1), Some(&"one"));
}
```

Questions to Consider

1. How do macros handle repetition in patterns?
2. What are the various fragment specifiers and when should you use each?
3. How do you ensure your macro handles different types correctly?

Exercise 2: Declarative Macros - Nested Patterns (25 minutes)

Objective

Create a logging macro with various verbosity levels and optional formatting.

Instructions

Implement a `log!` macro that supports:

1. Different log levels: `debug`, `info`, `warn`, `error`
2. Simple messages: `log!(debug, "Starting process")`
3. Formatted messages: `log!(info, "User {} logged in", username)`

Requirements

```
// Create a log! macro that works like this:
log!(debug, "Loading data");
log!(info, "Process started: {}", process_name);
log!(warn, "Resource usage at {}%", usage);
log!(error, "Failed to connect: {}", error);

// TODO: Implement the macro
macro_rules! log {
    // Write your patterns here
}

// Helper function to demonstrate actual logging
fn write_log(level: &str, message: &str) {
    println!("[{}] {}", level, message);
```

```rust
}

fn main() {
    let username = "alice";
    let score = 95;

    log!(debug, "Starting application");
    log!(info, "User {} logged in", username);
    log!(warn, "High CPU usage");
    log!(error, "Failed to save score: {}", score);
}
```

Questions to Consider

1. What's the difference between forwarding format arguments and building a new format string?
2. How can you minimize code duplication in the macro implementation?

Exercise 3: Declarative Macros - Recursion (25 minutes)

Objective

Create a recursive macro for composing nested function calls.

Instructions

Implement a `compose!` macro that allows chaining function calls from right to left:

- `compose!(f, g, h)(x)` should evaluate to `f(g(h(x)))`

Requirements

```rust
// Create a compose! macro that works like this:
let composed = compose!(double, increment, square);
assert_eq!(composed(2), double(increment(square(2))));

// TODO: Implement the macro
macro_rules! compose {
    // Write your patterns here
}

// Example functions to compose
fn increment(x: i32) -> i32 {
    x + 1
}

fn double(x: i32) -> i32 {
    x * 2
}

fn square(x: i32) -> i32 {
    x * x
}

fn main() {
    // Simple composition of two functions
    let f = compose!(double, increment);
    assert_eq!(f(5), 12); // double(increment(5)) = double(6) = 12

    // More complex composition
    let g = compose!(double, increment, square);
    assert_eq!(g(3), 20); // double(increment(square(3))) = double(increment(9)) = double(10) = 20

    // Single function (edge case)
    let h = compose!(double);
    assert_eq!(h(5), 10);
}
```

Questions to Consider

1. How do macros implement recursion?
2. What are the termination conditions for recursive macros?
3. How would you modify this macro to work with functions that have different parameter types?

Exercise 4: Procedural Macros - Custom Derive (30 minutes)

Objective

Create a custom derive macro that automatically implements builder pattern for structs.

Instructions

Create a '#[derive(Builder)]' macro that:

1. Creates a corresponding builder struct for any struct it's applied to
2. Adds setter methods for each field
3. Adds a build method that returns the original struct

Requirements

First, set up your procedural macro crate:

```toml
# In proc_macro_workshop/Cargo.toml
[package]
name = "proc_macro_workshop"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
syn = { version = "2", features = ["full"] }
quote = "1"
proc-macro2 = "1"
```

Then, implement the custom derive:

```rust
// In proc_macro_workshop/src/lib.rs
extern crate proc_macro;
use proc_macro::TokenStream;
```

```rust
#[proc_macro_derive(Builder)]
pub fn derive_builder(input: TokenStream) -> TokenStream {
    // TODO: Implement the derive macro
}
```

For testing, create a binary or test file:

```rust
// In proc_macro_tests/src/main.rs
use proc_macro_workshop::Builder;

#[derive(Builder)]
pub struct Person {
    name: String,
    age: u32,
    address: Option<String>,
}

fn main() {
    let person = Person::builder()
        .name("John Doe".to_string())
        .age(30)
        .address(Some("123 Main St".to_string()))
        .build();

    println!("Person: {} age {} at {:?}",
             person.name, person.age, person.address);
}
```

Expected Generated Code

The derive macro should generate code similar to:

```rust
pub struct PersonBuilder {
    name: Option<String>,
    age: Option<u32>,
    address: Option<Option<String>>,
}

impl Person {
    pub fn builder() -> PersonBuilder {
        PersonBuilder {
            name: None,
            age: None,
            address: None,
        }
    }
}

impl PersonBuilder {
    pub fn name(&mut self, name: String) -> &mut Self {
        self.name = Some(name);
        self
    }

    pub fn age(&mut self, age: u32) -> &mut Self {
        self.age = Some(age);
        self
    }

    pub fn address(&mut self, address: Option<String>) -> &mut Self {
        self.address = Some(address);
        self
    }

    pub fn build(&self) -> Person {
        Person {
            name: self.name.clone().expect("name is required"),
            age: self.age.expect("age is required"),
            address: self.address.clone().unwrap_or(None),
        }
    }
}
```

Questions to Consider

1. How do you parse and transform Rust code using the ‘syn‘ crate?
2. How do you generate new code using the ‘quote‘ crate?
3. What challenges arise when generating code for different field types?