`macro_rules!`

Florian "Florob" Zeitz

2024-03-06

Outline

1. C Macros
2. Rust Macros
3. Syntax
4. Fragments
5. Hygiene
6. Visibility
7. Examples

C Macros

The C Preprocessor

- only basic string replacement (at least in ISO C)
- takes one of two forms
  - `#define X 23`
  - `#define SUM(a, b) a + b`

Precedence Issues

```c
#define PROD(a, b) a * b

int main(void) {
  printf("%i\n", PROD(1 + 2, 3));
}
```

- this prints “7”
- expansion: `1 + 2 * 3`

Precedence Issues 2

```c
#define SUM(a, b) (a) + (b)

int main(void) {
  printf("%i\n", SUM(1, 2) * 3);
}
```

- still prints “7”
- expansion: `(1) + (2) * 3`

Precedence Resolved

```c
#define SUM(a, b) ((a) + (b))

int main(void) {
  printf("%i\n", SUM(1, 2) * 3);
}
```

- finally prints “9”
- expansion: `((1) + (2)) * 3`
- correct C macros often require lots of parentheses

Rust Macros

Syntax Extensions

- show up in various forms
  - outer attributes: `#[used]`, `#[derive(Debug)]`
  - inner attributes: `#![no_std]`, `#![allow(unused)]`
  - function-like: `println!()`, `vec![]`
- come in various flavors
  - declarative macros
    - can only define function-like macros
  - procedural macros
    - compiler plugins
  - built-ins

Syntax Extensions

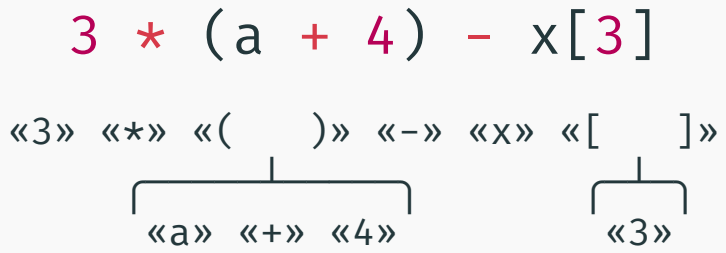
- show up in various forms
  - outer attributes: `#[used]`, `#[derive(Debug)]`
  - inner attributes: `#![no_std]`, `#![allow(unused)]`
  - function-like: `println!()`, `vec![]`
- come in various flavors
  - declarative macros ← **You are here**
    - can only define function-like macros
  - procedural macros
    - compiler plugins
  - built-ins

Rust Macros

- are parsed as part of the AST
- can only appear in place of a few constructs
  - patterns
  - statements
  - expressions
  - items
  - types
- **not** in place of
  - identifiers
  - match arms
  - struct fields

Token Tree

- Rust macros work on *token trees*
- one is consumed, one is produced
- consumed tokens must not be valid Rust, but form a valid tree
- not space-sensitive
  - e.g. matched parentheses



Syntax

Basic Syntax

- match-like syntax
- matches a set of tokens
- produces Rust code
- fragment specifiers `$name:type`
- regular text is matched verbatim

```
macro_rules! sum {
    ($a:expr, $b:expr) ⇒ {
        $a + $b
    };
}

assert_eq!(9, sum!(1, 2) * 3);
```

Repetitions

- repetitions take the form `$ ( … ) sep rep`, where:
- `$` is just a dollar sign
- `( … )` is a matcher in parentheses
- `sep` is an optional separator
- `rep` specifies the type of repeat
  `*`: zero or more repetitions
  `+`: one or more repetitions
  `?`: zero or one repetition

Long Sum

- expansion looks like matching
- separator can be different from the matcher, or absent
- `$(,)?` is pretty common to accept trailing commas

```
macro_rules! sum {
    ($x:expr, $($ys:expr),+ $(,)?) ⇒ {
        $x $(+ $ys)*
    };
}

assert_eq!(9, sum!(1, 2, 3, 3));
```

Fragments

Fragment Types

- matched fragments can be constrained by various fragment specifiers

`block` a code block { }
`expr` an expression
`ident` an identifier or keyword
`item` an item definition
`lifetime` a lifetime
`literal` a literal
`meta` contents of an attribute
`pat` a pattern (including or-patterns)
`pat_param`
a pattern (excluding or-patterns)

`path` a path
`stmt` a statement
`tt` a token tree
`ty` a type
`vis` a visibility qualifier

Statement Fragments

- matches a statement without semicolon
- semicolons are inserted as needed upon expansion

```
macro_rules! statements {
    ($($stmt:stmt)*) ⇒ ($($stmt)*)
}

statements! {
    let x = 3;
    let y = 4
    3
    if false {} else {}
}
```

```
let x = 3;
;
let y = 4;
3;
if false {} else {}
```

Token Tree Fragment

- can be matched on again turning it into other fragments
- very powerful
- e.g. counting:

```
macro_rules! count_tts {
    () ⇒ { 0 };
    ($odd:tt $($a:tt $b:tt)*) ⇒ { count_tts!($($a)*) * 2 + 1 };
    ($($a:tt $even:tt)*) ⇒ { count_tts!($($a)*) * 2 };
}
```

Hygiene

Hygiene

- declarative Rust's macros are partially hygienic
- local variables, labels and `$crate` can not leak in or out

```
macro_rules! compute {
    ($e:expr) ⇒ {{
        let x = 12;
        $e
    }}
}

let three = compute!(x / 4);
```

```
error[E0425]: cannot find value `x` in this scope
 ⟶ src/main.rs:9:26
  |
9 |     let three = compute!(x / 4);
  |                          ^ not found in this scope
```

Accessing Context

- identifiers passed in from outside refer to variables outside

```
macro_rules! compute {
    ($x:ident, $e:expr) ⇒ {{
        let x = 12;
        $e
    }}
}

let x = 8;
let two = compute!(x, x / 4);
```

Visibility

Using Macros

- macro must be defined before use
- even across modules in separate file
  (i.e. the module defining the macro must come first)
- macros can be imported from modules and crates using `#[macro_use]`
- macros exported from crates can also be `use`-d

Exporting macros

- to export a macro from a crate annotate it with `#[macro_export]`
- macros are always exported at the root
- ... even if they are defined in a nested module

Examples

vec![]

```
macro_rules! vec {
    ($($x:expr),*) ⇒ {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

hash_map!{}

```
macro_rules! hash_map {
    ($( ($key:expr, $value:expr) ),* $(,)?) ⇒ {
        {
            let mut map = HashMap::new();
            $(
                map.insert($key, $value);
            )*
            map
        }
    };
}
```