

4-3: Procedural and declarative macros (Theory)

Artem Pavlov, TII, Abu Dhabi, 06.05.2024

Macros

- Rust has 4 kinds of macros:
 - Declarative macros defined with `macro_rules!`
 - Procedural macros:
 - Custom `#[derive]` macros that specify code added with the `derive` attribute used on `structs` and `enums`
 - Attribute-like macros that define custom attributes usable on any item
 - Function-like macros that look like function calls but operate on the tokens specified as their argument
- Macro must be defined or brought into scope before it's called
- All procedural macros accept `TokenStream` and return `TokenStream`

Declarative macros

- Sometimes referred to as “macros by example,” “**macro_rules!** macros,” or just plain “macros”

- **macro_rules!** uses the following form:

```
macro_rules! $name {  
    ($matcher0) => {$expansion0} ;  
    ($matcher1) => {$expansion1} ;  
    // ...  
    ($matcherN) => {$expansionN} ;  
}
```

Primitive macros

```
macro_rules! four {
    () => {
        1 + 3
    };
}

macro_rules! gibberish {
    (4 fn ['spang "whammo"] @_@) => {
        42
    };
}

// note: the macros should be defined BEFORE uses
fn main() {
    let x: u32 = four!();
    let y: u64 = four!();
    let z: u128 = gibberish!(4 fn ['spang "whammo"] @_@);
    println!("{x} {y} {z}")
}
```

Captures

- Matchers can also contain captures.
- Captures are written as a dollar (\$) followed by an identifier, a colon (:), and finally the kind of capture:
 - **block**: a block (i.e. a block of statements and/or an expression, surrounded by braces)
 - **expr**: an expression
 - **ident**: an identifier (this includes keywords)
 - **item**: an item, like a function, struct, module, impl, etc.
 - **lifetime**: a lifetime (e.g. 'foo, 'static, etc.)
 - **literal**: a literal (e.g. "Hello World!", 3.14, etc.)
 - **meta**: a meta item; the things that go inside the #[...] and #![...] attributes
 - **pat**: a pattern
 - **path**: a path (e.g. foo, ::std::mem::replace, transmute::<_, int>, etc.)
 - **stmt**: a statement
 - **tt**: a single token tree
 - **ty**: a type
 - **vis**: a possible empty visibility qualifier (e.g. pub, pub(in crate), etc.)

Simple capture

```
macro_rules! multiply_add {  
|    ($a:expr, $b:expr, $c:expr) => { $a * ($b + $c) };  
}  
  
macro_rules! discard {  
|    ($e:expr) => {};  
}  
  
macro_rules! repeat {  
|    ($e:expr) => { $e; $e; $e; };  
}
```

Repetitions

- Matchers can contain repetitions.
- Repetitions have the general form `$ (...) sep rep`:
 - `$` is a literal dollar token.
 - `(...)` is the paren-grouped matcher being repeated.
 - `sep` is an optional separator token. Common examples are `,` and `;`
 - `rep` is the required repeat operator. Currently, this can be:
 - `?`: indicating at most one repetition
 - `*`: indicating zero or more repetitions
 - `+`: indicating one or more repetitions
- Since `?` represents at most one occurrence, it cannot be used with a separator.

Simple repetition

```
macro_rules! vec_strs {
    (
        $($element:expr),*
    ) => {
        // Enclose the expansion in a block so that we can use
        // multiple statements.
        {
            let mut v = Vec::new();
            $(
                v.push(format!("{}", $element));
            )*
            v
        }
    };
}

fn main() {
    let s = vec_strs![1, "a", true, 4.14159f32];
    assert_eq!(s, &["1", "a", "true", "4.14159"]);
}
```


Multiple repetitions

```
macro_rules! repeat_two {  
    | ($($i:ident)*, $($i2:ident)*) => {  
    |     $( let $i: (); let $i2: (); )*  
    |     }  
    }  
  
    // works  
    repeat_two!( a b c d e f, u v w x y z );  
  
    // does not work  
    repeat_two!( a b c d e f, x y z );
```

Custom derive and **serde**

```
[package]
name = "p14"
version = "0.1.0"
edition = "2021"

[dependencies]
serde = { version = "1", features = ["derive"] }
serde_json = "1"
bincode = "1.3"
```

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug, Eq, PartialEq)]
pub struct Foo {
    a: u64,
    b: Vec<String>,
}

fn main() {
    let orig = Foo {
        a: 42,
        b: vec![String::from("hello"), String::from("world")],
    };

    let json_str = serde_json::to_string(&orig).unwrap();
    println!("{}", json_str);
    let v1: Foo = serde_json::from_str(&json_str).unwrap();
    assert_eq!(orig, v1);

    let bincode_val = bincode::serialize(&orig).unwrap();
    println!("{}", bincode_val);
    let v2 = bincode::deserialize(&bincode_val).unwrap();
    assert_eq!(orig, v2);
}
```

Questions?