Procedural Macros in Rust

Felix Kohlgrüber | Rust Meetup Karlsruhe | 2019/11/6

Macros

- Executed at compile time
- Produce source code
- In Rust:
  - Declarative Macros
  - Procedural Macros

Declarative Macros

- a.k.a. `macro_rules!` macros, „macros by example“
- Usage examples:
  - `vec!( ... )`, `println!( ... )`, `assert!( ... )`, …
- Definition example:

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

Procedural Macros | Usage

- Function-like

```
html!{ <div>“Hello World“</div> }
```

- Custom Derive

```
#[derive(Serialize)]
struct Foo { ... }
```

- Attribute

```
#[get("/")]
fn index() → &'static str { ... }
```

Procedural Macros | Definition

- proc-macro crates:
  - in Cargo.toml:
    ```
    [lib]
    proc-macro = true
    ```
  - May only export procedural macros
- Procedural macro:

  `Fn(TokenStream) → TokenStream`

TokenStream

- `TokenStream ≅ Vec<TokenTree>`
- `Token ≅` „word of code“ (e.g. identifier, string, number, …)
- `TokenTree:` Hierarchy formed by parens, brackets and braces

```
pub enum TokenTree {
    Token(Token),
    Delimited(
        … , DelimToken, TokenStream
    ),
}
```

```
pub enum DelimToken {
    Paren,
    Bracket,
    Brace,
    NoDelim,
}
```

Function-like Procedural Macro

- Definition:

```
#[proc_macro]
pub fn function_macro(input: TokenStream) → TokenStream {
    ...
}
```

- Usage:

```
function_macro!{ “Hello World“ + 123 }
```

Custom Derive Procedural Macro

- Definition:

```
#[proc_macro_derive(DeriveMacro)]
pub fn derive_macro(input: TokenStream) → TokenStream {
    ...
}
```

- Usage:

```
#[derive(DeriveMacro)]
struct Foo { ... }
```

Attribute Procedural Macro

- Definition:

```
#[proc_macro_attribute]
pub fn foo(attr: TokenStream, item: TokenStream) → TokenStream {
    ...
}
```

- Usage:

```
#[foo(„Parameter“ 23 bar)]
struct Baz { ... }
```

DEMO

Summary

- Three kinds of procedural macros
- Procedural macros need to be in separate crate
- `cargo expand` is useful for debugging proc macros

Part 2 - A „real“ project

Motivation

- Advent of Code 2018
- Text input -> Rust data structures
- Example (Day 3):

```
#1 @ 565,109: 14×24
#2 @ 413,723: 16×28
#3 @ 136,229: 27×11
#4 @ 640,187: 10×17
#5 @ 666,879: 15×23
 ...
```

->

```
#[derive(Debug)]
struct Patch {
    id: usize,
    left: usize,
    top: usize,
    width: usize,
    heigth: usize,
}
```

Motivation

```
impl Patch {
    pub fn from_str(s: &str) → Option<Self> {
        lazy_static! {
            static ref RE: Regex =
                Regex::new(r"#(\d+) @ (\d+),(\d+): (\d+)x(\d+)").unwrap();
        }

        RE.captures(s).map(|cap| Self {
            id: cap[1].parse().unwrap(),
            left: cap[2].parse().unwrap(),
            top: cap[3].parse().unwrap(),
            width: cap[4].parse().unwrap(),
            heigth: cap[5].parse().unwrap(),
        })
    }
}
```

Goal

- Generate `from_str` from a simple pattern:

```
#[parse(r"#{} @ {},{}: {}x{}")]
#[derive(Debug)]
struct Patch {
    id: usize,
    left: usize,
    top: usize,
    width: usize,
    heigth: usize,
}
```

Approach

- Attribute procedural macro
- Main steps:
  - Parse input (syn crate)
  - Process
  - Generate code (quote crate)

Parse input

- Parse TokenStreams into Rust AST nodes

```
#[proc_macro_attribute]
pub fn parse(attr: TokenStream, item: TokenStream) → TokenStream {
    let strukt =
        syn::parse_macro_input!(item as syn::ItemStruct);
    let pattern: String =
        syn::parse_macro_input!(attr as syn::LitStr).value();
    ...
}
```

Process information

- For the struct:
  - Name
  - Name and type for each member
- For the list:
  - List of regex strings

```
#[parse(r"#{} @ {},{}: {}x{}")]
#[derive(Debug)]
struct Patch {
    id: usize,
    left: usize,
    top: usize,
    width: usize,
    heigth: usize,
}
```

Process information

```
// extract struct name + name and type for each struct element
let name = &strukt.ident;
let mut items = vec!();
if let syn::Fields::Named(nf) = &strukt.fields {
    for field in &nf.named {
        items.push(
            (field.ident.as_ref().unwrap(),
             field.ty.to_token_stream().to_string()
            )
        )
    }
}

// extract regex strings from pattern
let parts = pattern.split("{}").collect::<Vec<_>>();
```

Process information

„Error handing“

```
// panic if number of brace pairs in pattern doesn't
// match number of struct elements
assert!(parts.len() == items.len()+1);
```

Process information

"foo{}bar" → "foo(\d+)bar"

```
// generate the regex pattern string
let mut regex_str = String::new();
for (pattern_prefix, item) in parts.iter().zip(items.iter()) {
    let regex_for_type = get_regex_for_type(&item.1);
    regex_str.push_str(&format!("{}({})", pattern_prefix, regex_for_type));
}
regex_str.push_str(parts.last().unwrap());


fn get_regex_for_type(ty: &str) -> &'static str {
    match ty {
        "usize" => r"\d+",
        "f64" => r"[0-9]*\.?[0-9]*",
        "char" => r".",
        "bool" => r"true|false",
        t => panic!( ... )
    }
}
```

Code generation

```
quote!(
    #strukt

    impl #name {

        pub fn from_str(s: &str) -> Option<Self> {
            lazy_static! {
                static ref RE: Regex = Regex::new(#regex_str).unwrap();
            }

            Self::get_regex().captures(s).map(|cap| Self {
                #(#inits),*
            })
        }
).into()
```

Code generation

```
// generate initializers for each struct element
let mut inits = vec!();
for (idx, item) in items.iter().enumerate() {
    let name = &item.0;
    inits.push(quote!( #name: cap[#idx+1].parse().unwrap()));
}
```

Usage

```
#[parse(r"#{} @ {},{}: {}x{}")]
#[derive(Debug)]
struct Patch {
    id: usize,
    left: usize,
    top: usize,
    width: usize,
    heigth: usize,
}

fn main(){
    println!("{:?}", Patch::from_str("#1 @ 2,3: 4×5"));
    // Output:
    // Some(Patch { id: 1, left: 2, top: 3, width: 4, heigth: 5 })
}
```

Usage

```rust
#[parse(r"{}\n{},+{}\nFlag: {}")]
#[derive(Debug)]
struct Types {
    name: char,
    int: usize,
    float: f64,
    flag: bool
}

fn main(){
    println!("{:?}", Patch::from_str("X\n2,,,123.456\nFlag: true"));
    // Output:
    // Some(Types { name: 'X', int: 2, float: 123.456, flag: true })
}
```

Conclusion

- Procedural macros <3

End