Macros

Sebastian Agudelo Morales
Miguel Angel Lopez Fernandez

NACION CRYPTO By DECRY

ATOM
ACCELERATOR

Que son?

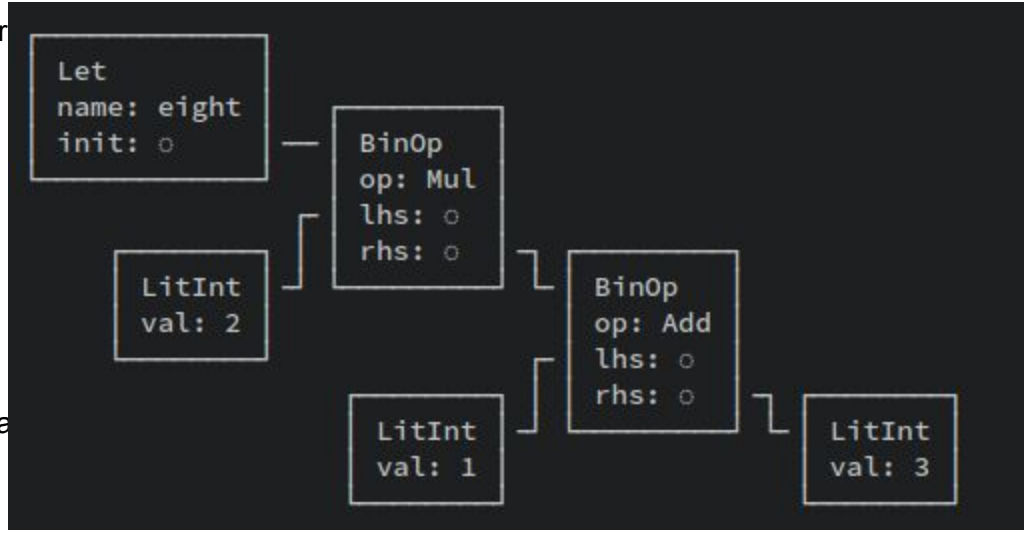
Los macros son una forma de escribir código que escribe otro código, que se conoce como *metaprogramación*.

En rust le permite a los desarrolladores generar código en tiempo de compilación basado en patrones y argumentos proporcionados.

Funcionamiento

Cuando el compilador encuentra una invocación de macro, procesa los argumentos y el cuerpo de la macro para generar nuevo código, que luego se inserta en el AST en lugar de la invocación de la macro original.

```
let eight = 2 * four!();
```



Macros declarativos

Permiten escribir algo similar a una expresión de coincidencia que opera sobre el código de Rust proporcionado como argumentos. Se utilizan para generar código que reemplaza la invocación de la macro

Funcionamiento

Cuando el compilador encuentra una invocación de macro, procesa los argumentos y el cuerpo de la macro para generar nuevo código, que luego se inserta en el AST en lugar de la invocación de la macro original.

```
macro_rules! $name {
  $rule0 ;
  $rule1 ;
  ..
   argumento
  ($matcher) => {$expansion};
  ($matcher) => {$expansion};
  $ruleN ;
}

uso
name_macro!($captura)
```

Metavariables

Argumentos en los macros declarativos

Block-expr-ident-item-liftime- pat- path-stmt- etc

```
($identificador:tipo_de_token)
```

```
macro_rules! repeat_two {
    ($($i:ident)*, $($i2:ident)*) => {
        $( let $i: (); let $i2: (); )*
    }
}

repeat_two!( a b c d e f, u v w x y z );
```

Repeticiones

- ?: indicating at most one repetition
- *: indicating zero or more repetitions
- +: indicating one or more repetitions

```
macro_rules! vec_strs {
    (// Start a repetition:
        $(// Each repeat must contain an expression...
            $element:expr
        )// ...separated by commas...
        ,// ...zero or more times.
        *
    ) => {// Enclose the expansion in a block so that we can use
        // multiple statements.
        {    let mut v = Vec::new();
            // Start a repetition:
            $(  // Each repeat will contain the following statement, with
                // $element replaced with the corresponding expression.
                v.push(format!("{}", $element));
            )*
            v
        }
    };
}

fn main() {
    let s = vec_strs![1, "a", true, 3.14159f32];
    assert_eq!(s, &["1", "a", "true", "3.14159"]);
}
```

ATOM ACCELERATOR

Macros procedurales

Reciben una representación sintáctica de código como entrada (**Ingresa token**), realizan alguna transformación, y luego devuelven una nueva representación de código que se **inserta** en el lugar donde se invocó el macro (**retorna token**).

Como se implementan?

como "crates" especiales que se vinculan al compilador de Rust en tiempo de compilación. Estos crates deben depender de `proc_macro` y usualmente también de `syn` y `quote` para facilitar el manejo y generación de la sintaxis de Rust.

```rust
// En el crate de macros procedural
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, ItemFn};

#[proc_macro_attribute]
pub fn log_entrada(_: TokenStream, input: TokenStream) -> TokenStream {
    let mut funcion = parse_macro_input!(input as ItemFn);
    let nombre_funcion = &funcion.sig.ident;
    funcion.block.stmts.insert(0, syn::parse2(quote! {
        println!("Ingresando a la función: {}", stringify!(#nombre_funcion));
    }).unwrap());
    quote!(#funcion).into()
}
```

```rust
// En el crate del usuario
#[log_entrada]
fn saludar() {
    println!("Hola, mundo!");
}

fn main() {
    saludar();
}
```

```
Ingresando a la función: saludar
Hola, mundo!
```

NACION CRYPTO By DECRY

Tipos de macros y su sintaxis

Macros de función - nombre_del_macro!

```rust
// Importamos TokenStream.
use proc_macro::TokenStream;
// Importamos 'quote' para construir   tokens.
use quote::quote;
// Importamos funciones para analizar la entrada.
use syn::parse_macro_input;
// Importamos el tipo que representa una cadena literal.
use syn::LitStr;
// Definimos un macro de función.
#[proc_macro]
```

```rust
// Función que será llamada en tiempo de compilación.
pub fn generar_funcion(input: TokenStream) -> TokenStream {
// Convertimos la entrada en una cadena literal.
    let nombre_funcion = parse_macro_input!(input as LitStr);
// Construimos el código a generar.
    let salida = quote! {
// Generamos una función pública con el nombre dado.
        pub fn #nombre_funcion() {
// La función imprime un mensaje.
            println!("Mensaje desde la función generada.");
        }
    };
    salida.into()  // Convertimos la salida en un TokenStream.
}
```

Tipos de macros y su sintaxis

Macros de función - nombre_del_macro!

```
// Crate del usuario

fn main() {
// Invocamos el macro para generar una función llamada 'funcion_genera
da'.
  generar_funcion!("funcion_generada");
}

//Salida esperada
Mensaje desde la función generada.
```

ATOM ACCELERATOR

Tipos de macros y su sintaxis

Macros de atributos - #[attribute]

Permiten **adjuntar metadatos adicionales** o alterar el comportamiento de una **función, tipo, o módulo**. Un macro de atributo puede manipular el código adjunto de diversas maneras, cómo añadir **validaciones adicionales** o **modificar la lógica**.

```rust
// Importamos TokenStream.
use proc_macro::TokenStream;
// Importamos 'quote' para construir tokens.
use quote::quote;
// Importamos funciones para analizar la entrada.
use syn::{parse_macro_input, ItemFn};
```

```rust
#[proc_macro_attribute]        // Definimos un macro de atributo.
// Función que será llamada en tiempo de compilación.
pub fn mensaje_antes(_: TokenStream, input: TokenStream)->TokenStream {
// Convertimos la entrada en una estructura ItemFn.
    let mut funcion = parse_macro_input!(input as ItemFn);
// Insertamos una nueva declaración al inicio del bloque de la función.
    funcion.block.stmts.insert(0, syn::parse2(quote! {
// La declaración es una llamada a println!.
        println!("Mensaje antes de la función.");
    }).unwrap());
// Regresamos la función modificada como TokenStream.
    quote!(#funcion).into()
}
```

Tipos de macros y su sintaxis

Macros de atributos - #[attribute]

```
// Crate del usuario
#[mensaje_antes]   // Aplicamos el macro de atributo a 'mi_funcion'.
fn mi_funcion() {  // Definimos una función simple.
    println!("Dentro de la función."); // Imprimimos un mensaje.
}

//Salida esperada
Mensaje antes de la función.
Dentro de la función.
```

Tipos de macros y su sintaxis

Macros de derivación - #[derive]

El token de entrada (`annotated_item`) siempre será una `enum`, `struct` o `union`, ya que estos son los únicos elementos que un atributo de derivación que se pueden anotar

```rust
// Importamos TokenStream, que representa un flujo de tokens.
use proc_macro::TokenStream;
// Importamos la macro 'quote' que nos ayuda a construir tokens.
use quote::quote;
// Importamos funciones para analizar la entrada.
use syn::{parse_macro_input, DeriveInput};
// Definimos un macro de derivación llamado 'MiDebug'.
#[proc_macro_derive(MiDebug)]
```

```rust
// Función que será llamada en tiempo de compilación.
pub fn mi_debug_derive(input: TokenStream) -> TokenStream {
// Convertimos la entrada en una estructura DeriveInput.
    let input = parse_macro_input!(input as DeriveInput);
// Obtenemos el identificador (nombre) del tipo.
    let nombre = &input.ident;
// Construimos el código a generar.
    let salida = quote! {
// Implementamos la trait Debug para el tipo dado.
        impl std::fmt::Debug for #nombre {
// Definimos el método 'fmt'.
            fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
                std::fmt::Result {
// Escribimos el nombre del tipo en la salida.
                write!(f, "Instancia de {:?}", stringify!(#nombre))
            }
        }
    };
    salida.into()  // Convertimos la salida en un TokenStream.
}
```

Tipos de macros y su sintaxis

Macros de derivación - #[derive]

El flujo de tokens devuelto se agrega al bloque o módulo contenedor del elemento anotado con el requisito de que el flujo de tokens esté formado por un conjunto de elementos válidos.

```
// Crate del usuario
// Indicamos que queremos derivar 'MiDebug' para 'MiEstructura'.
#[derive(MiDebug)]
struct MiEstructura; // Definimos una estructura simple.

fn main() {
    let instancia = MiEstructura;
    println!("{:?}", instancia);
}

//Salida esperada
Instancia de "MiEstructura"
```

BIBLIOGRAFIA:

- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html
- https://veykril.github.io/tlborm/introduction.html
- https://doc.rust-lang.org/stable/book/ch19-06-macros.html

Tutorial

- https://blog.logrocket.com/macros-in-rust-a-tutorial-with-examples/

Notion

- https://gold-thing-feb.notion.site/Macros-e32ff8c594904ba9b050eafb785788ff?pvs=4