MACROS RULE

A DIVE INTO RUST'S SYNTAX EXTENSION TOOLBOX

THOTH GUNTER

@TKGGUNTER

A MACRO IS ...

a single instruction that expands into multiple instructions -- The Common Lisp cookbook

(When describing macros in assembly environments.)

```
//HELPME is a macro
let hm = HELPME!();
```

```
//expanded macro
let hm = {let uql = life_universe_everything();
          let mut f = rand();
          bbq(f) % uql };
```

MACROS OFFER MORE THAN JUST TEXTUAL REPLACEMENT.

MACROS IN C/C++ 98

```
for(int i=0; i < l.size(); i++){
```

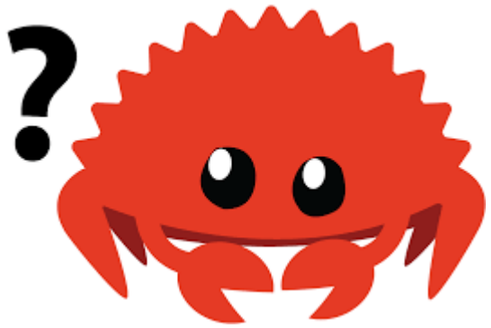
```
#define FOR_IN(iter, arr) \
for(int iter=0; iter < arr.size(); iter++)

main(){
    std::vector myvec = [12, 1, 32, 4];
    FOR_IN(i, myvec){
      printf("iter: %d : %d", i, i + myvec[i]);
    }
}
```

WHY USE MACROS?

- Reduction of Redundant Code
- Fundamentally impact the structure and syntax of the language.

WHAT ABOUT RUST?



MACROS IN RUST

- "Macro is a rule or pattern that specifies how a certain input sequence should be mapped to a replacement output sequence." -- wikipedia Macro_(computer_science)
- "Macros are a way of writing code that writes other code." -- rust-doc 1.30 appendix 04

MACROS IN RUST

- "Macro is a rule or pattern that specifies how a certain input sequence should be mapped to a replacement output sequence." -- wikipedia Macro_(computer_science) ← Declarative Macro
- "Macros are a way of writing code that writes other code." -- rust-doc 1.30 appendix 04 ← Procedural Macro

THE DECLARATIVE MACRO

1. Define how the macro is identified.
2. Set of patterns to be checked against.
   - (), {}, or []
3. Patterns consist of named tokens and
4. typed tokens.
5. Replacement code.
6. Ending in a semicolon.

```
macro_rules! (MACRO_NAME)[1] {
(MATCH STATEMENT)[2] (\{\[
    $(VARIABLE NAME)[3]:(VARIABLE TYPE)[4],\(DELIMITER or
REPETITION or CUSTOM SYMBOL)
]\}\) => {
              $(VARIABLE NAME)[5] //perform some
operation
          };[6]
}
```

STARTING SIMPLE

```
macro_rules! update_and_print {
($x:tt)=>{ println!("previous {}", $x);
           $x += 1;
         };
($x:tt, $y:tt)=>{ println!("previous {} {}", $x, $y);
           $x += 1;
           $y += 1;
         };
}
```

Original Code

```
let mut a = 10;
let mut b = 10;
update_and_print!(a);
update_and_print!(a, b);
```

Expanded Code

```
println!("previous {}", a);
a += 1;
println!("previous {} {}", a, b);
a += 1;
b += 1;
```

ERRORS

```
update_and_print!();
^^^^^^^^^^^^^^^^^^^^ missing tokens in macro arguments

update_and_print!(a);
------------------^--
                  |
                  `CustomType` cannot be formatted with the default formatter
                  in this macro invocation


update_and_print!(a, b, c);
                     ^ no rules expected this token in macro call
```

MACRO TYPES

- item: an item (function, struct, module, etc)
- block: a block e.g. `{}` of statements or expressions, e.g., `{ let x = 5; }`
- stmt: a statement e.g., `let x = 1 + 1;` , `String::new();` or `vec![];`
- pat: a pattern e.g., `Some(true)` , `(17, 'a')` , or `_`
- expr: an expression e.g., `x` , `1 + 1` , `String::new()` or `vec![]`
- ty: a type e.g., `String` , `usize` or `Vec` .
- ident: an identifier e.g. for example in `let x = 0;` the identifier is `x` .
- path: a path e.g. `foo` , `::std::mem::replace` , `transmute::<_, int>` , ...
- meta: a meta item; the things that go inside `#[...]` and `#![...]` attributes
- tt: a single token tree
- lifetime: a lifetime token e.g. `'a`
- vis: visibility qualifier e.g. `pub` , `pub(crate)`
- literal: hard coded floats, ints, chars and strings

REPETITION SIGNIFIER

- * (zero or more repetition)
- + (one or more repetitions)
- ? (zero or one instances)

https://doc.rust-lang.org/1.30.0/book/first-edition/macros.html

YOUR ALL PROS NOW

```
macro_rules! math{
   ($x:literal ^ $y:literal )=>{
       ($x as f32).powf($y as f32)
   };
   ($x:literal ^ $y:literal $sym:tt $($expression:tt)* )=>{
       ($x as f32).powf($y as f32) $sym math!($($expression)*)
   };
   ($x:literal $sym:tt $($expression:tt)* )=>{
       $x as f32 $sym math!($($expression)*)
   };
   ($sym:tt $x:literal $($expression:tt)* )=>{
       $sym $x as f32 math!($($expression)*)
   };
   ($x:literal)=>{
       $x as f32
   };
   ()=>{};
}
```

```
macro_rules! math{
    ($x:literal ^ $y:literal )=>{
        ($x as f32).powf($y as f32)
    };
    ($x:literal ^ $y:literal $sym:tt
$($expression:tt)* )=>{
        ($x as f32).powf($y as f32) $sym
math!($($expression)*)
    };
    ($x:literal $sym:tt $($expression:tt)*
)=>{
        $x as f32 $sym math!
($($expression)*)
    };
    ($sym:tt $x:literal $($expression:tt)*
)=>{
        $sym $x as f32 math!
($($expression)*)
    };
    ($x:literal)=>{
        $x as f32
    };
    ()=>{};
}
```

CHEAT SHEET

- item: an item (function, struct, module, etc)
- block: a block e.g. `{}` of statements or expressions, e.g., `{ let x = 5; }`
- stmt: a statement e.g., `let x = 1 + 1;` , `String::new();` or `vec![];`
- pat: a pattern e.g., `Some(true)` , `(17, 'a')` , or `_`
- expr: an expression e.g., `x` , `1 + 1` , `String::new()` or `vec![]`
- ty: a type e.g., `String` , `usize` or `Vec` .
- ident: an identifier e.g. for example in `let x = 0;` the identifier is `x` .
- path: a path e.g. `foo` , `::std::mem::replace` , `transmute::<_, int>` , ...
- meta: a meta item; the things that go inside `#[...]` and `#![...]` attributes
- tt: a single token tree
- lifetime: a lifetime token e.g. `'a`
- vis: visibility qualifier e.g. `pub` , `pub(crate)`
- literal: hard coded floats, ints, chars and strings
    - * (zero or more repetition)
    - + (one or more repetitions)
    - ? (zero or one instances)

```
macro_rules! math{
    ($x:literal ^ $y:literal )=>{
        ($x as f32).powf($y as f32)
    };
    ($x:literal ^ $y:literal $sym:tt
$($expression:tt)* )=>{
        ($x as f32).powf($y as f32) $sym
math!($($expression)*)
    };
    ($x:literal $sym:tt $($expression:tt)*
)=>{
        $x as f32 $sym math!
($($expression)*)
    };
    ($sym:tt $x:literal $($expression:tt)*
)=>{
        $sym $x as f32 math!
($($expression)*)
    };
    ($x:literal)=>{
        $x as f32
    };
    ()=>{};
}
```

Allows user to input simple mathematical expressions of numeral literals and calculates the results.

It take a mixed of integer and float types and converts all inputs to float.

```
let example = math!(1 + 23.42^2.3);
```

```
//Expansion
let example = 1 as f32 + (23.42 as
f32).powf(2.3 as f32);
```

WHAT HAPPENS HERE?

```
let example = math!( 13.4 * (1.2^23 + 12) + 23.42^2.3);
```

WHAT HAPPENS HERE?

```
let example = math!( 13.4 * (1.2^23 + 12) + 23.42^2.3);
```

```
macro_rules! math{
   ($x:literal ^ $y:literal )=>{
       ($x as f32).powf($y as f32)
   };
   ($x:literal ^ $y:literal $sym:tt $($expression:tt)* )=>{
       ($x as f32).powf($y as f32) $sym math!($($expression)*)
   };
   ($x:literal $sym:tt $($expression:tt)* )=>{
       $x as f32 $sym math!($($expression)*)
   };
   ($sym:tt $x:literal $($expression:tt)* )=>{
       $sym $x as f32 math!($($expression)*)
   };
   ($x:literal)=>{
       $x as f32
   };
   ()=>{};
}
```

WHAT HAPPENS HERE?

```
let example = math!( 13.4 * (1.2^23 + 12) + 23.42^2.3);
```

```
!!ERROR!!
--> main02.rs:32:46
   |
10 | macro_rules! math{
   | ----------------- when calling this macro
...
32 | println!("{}", math!( 13.4 * (1.2^23 + 12) + 23 / 3.14^2));
   |                                              ^ no rules expected this token
in macro call
```

DEBUGGING TOOLS (ONLY USING NIGHTLY COMPILER)

Expanding macros with the nightly compiler and checking source code is probably the best way to go when debugging macros.

```
rustup default nightly
rustup default stable
```

(cargo) rustc

```
(cargo) rustc -Zunstable-options --pretty=expanded
```

AND/OR

```
#![feature(trace_macros)]
fn main(){
    trace_macros!(true);
    ...
    trace_macros!(false);
}
```

DEBUGGING TOOLS (ONLY USING NIGHTLY COMPILER)

```
let example = math!( 13.4 * (1.2^23 + 12) 23.42^2.3);

//Trace macro expansion
= note: expanding `math! { 13.4 + (1.2 ^ 23 + 12) + 23.42 ^ 2.3 }`
= note: to `13.4 as f32 + math ! ((1.2 ^ 23 + 12) + 23.42 ^ 2.3)`
= note: expanding `math! { (1.2 ^ 23 + 12) + 23.42 ^ 2.3 }`
```

PROCEDURAL MACROS

are a way of writing rust code that writes rust code.

PROCEDURAL MACROS

Compiler extension.

DIFFERENCES FROM DECLARATIVE MACROS

- Defined in separate crate.
- Not pattern based.
- Direct interaction with token streams.

PROS

- Custom error messages.
- Working with the language you use everyday

SETTING UP A PROC MARCO

SETUP: cargo toml and rustc

Cargo junky (cargo)

```
#CARGO.TOML
[lib]
proc-macro = true
```

One Off (rustc)

```
// if using rustc be sure
// the following heads your .rs
lib file.
#![crate_type = "proc-macro"]
```

```
//src.rs
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn macro_name( _sts: TokenStream) -> TokenStream {
```

SIMPLE EXAMPLE

```
//testlibproc_macro
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn macro_name( _sts: TokenStream) -> TokenStream {
    "fn test_fn()-> f32 { 23.23 }".parse().unwrap()
}
```

```
extern crate testlibproc_macro;
use proc_lib::macro_name;

macro_name!(); //Expands to fn test_fn()->f32 { 23.23 }

fn main(){
    let test = test_fn();
    println!("test {}", test);
}
```

```
//testlibproc_macro
#[proc_macro]
pub fn macro_name( _sts: TokenStream) -> TokenStream {
    println!("Hello World.")
    "fn test_fn()-> f32 { 23.23 }".parse().unwrap()
}
```

```
Compiling example v0.1.0 (/example)
Hello World
Finished dev [unoptimized + debuginfo] target(s) in 0.61s
Running `target/debug/example`
test 42
```

MESSING WITH TOKENS (PROC_MACRO LIB)

TokenStream is made up of TokenTree(s).

TokenTree is an enum.

- Group(Group)
  A token stream surrounded by bracket delimiters.
- Ident(Ident)
  An identifier.
- Punct(Punct)
  A single punctuation character (+, ,, $, etc.).
- Literal(Literal)
  A literal character ('a'), string ("hello"), number (2.3), etc.

COMPLEX EXAMPLE

```
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn math(expression: TokenStream)->TokenStream{
    use proc_macro::TokenTree;

    let mut output_string = String::new();
    let mut open_brackets = 0;
    let mut pre_index = 0;
    for it in expression.into_iter(){
        match it{
            TokenTree::Group(group)=>{
                output_string += "( ";
                let ts = math(group.stream());
                output_string += &ts.to_string();
                output_string += " ) ";
            },
            TokenTree::Ident(ident)=>{
                pre_index = output_string.len();
                output_string += &format!(" {} as f32 ", ident.to_string() );
                if open_brackets > 0 {
                    output_string += ") ";
                    open_brackets -= 1;
                }
            },
            TokenTree::Literal(literal)=>{
                pre_index = output_string.len();
                output_string += &format!(" {} as f32 ", literal.to_string() );
                if open_brackets > 0 {
                    output_string += ") ";
                    open_brackets -= 1;
                }
            },
            TokenTree::Punct(punct)=>{
                let _char = punct.as_char();
                match _char{
                    '+' => { output_string += "+";},
                    '-' => { output_string += "-";},
                    '/' => { output_string += "/";},
                    '^' => {
                        open_brackets += 1;
                        output_string += " ).powf(";
                        output_string.insert(pre_index, '(');
                    },
                    '*' => { output_string += "*";},
                    _   => {panic!("Not implemented! symbol unknown");}
                }
            },
            _=>{panic!("Error unsupported tokentree");}
        }
    }
    match output_string.parse(){
        Ok(ts) =>  ts,
        Err(e) => panic!(format!("{:?}", e))
    }
}
```

COMPLEX EXAMPLE

```
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn math(expression: TokenStream)->TokenStream{
    use proc_macro::TokenTree;

    let mut output_string = String::new();
    let mut open_brackets = 0;
    let mut pre_index = 0;
    for it in expression.into_iter(){
        match it{
            TokenTree::Group(group)=>{
                output_string += "( ";
                let ts = math(group.stream());
                output_string += &ts.to_string();
                output_string += " ) ";
            },
            TokenTree::Ident(ident)=>{
                pre_index = output_string.len();
                output_string += &format!(" {} as f32 ", ident.to_string() );
                if open_brackets > 0 {
                    output_string += ") ";
                    open_brackets -= 1;
                }
            },
            TokenTree::Literal(literal)=>{
                pre_index = output_string.len();
                output_string += &format!(" {} as f32 ", literal.to_string() );
                if open_brackets > 0 {
                    output_string += ") ";
                    open_brackets -= 1;
                }
            },
            TokenTree::Punct(punct)=>{
                let _char = punct.as_char();
                match _char{
                    '+' => { output_string += "+";},
                    '-' => { output_string += "-";},
                    '/' => { output_string += "/";},
                    '^' => {
                        open_brackets += 1;
                        output_string += " ).powf(";
                        output_string.insert(pre_index, '(');
                    },
                    '*' => { output_string += "*";},
                    _   => {panic!("Not implemented! symbol unknown");}
                }
            },
            _=>{panic!("Error unsupported tokentree");}
        }
    }
    match output_string.parse(){
        Ok(ts) =>  ts,
        Err(e) => panic!(format!("{:?}", e))
    }
}
```

In ~60 lines of code we have a more robust macro that is potentially easier to debug.

```
#![feature(proc_macro_hygiene)]
extern crate proc_lib;
use proc_lib::*;


fn main() {

    let x = math!( 2.3 ^ 10 - 4);
    assert_eq!(2.3_f32.powf(10.0) - 4.0, x);

    let y = 42;
    let x = math!(1 + 1.0 / y);
    assert_eq!(1.0 + 1.0/ 42.0, x);


    let x = math!( 5.2 * (1 + 4) - 43);
    assert_eq!(5.2 * (1.0 + 4.0) - 43.0, x);
}
```

EVERYTHING NOT COVERED

ADDITIONAL CONCEPTS

- pattern precedence
- hygiene
- Derive macros
- Attribute macros
- Proc Macros can use compiler args

ADDITIONAL TOOLS

- proc-macro2 crate
  is a re-implementation of the proc-macro std crate
- quote crate
  is a quasi quote
- syn crate
  is a parsing library that allows easy pattern matching.

QUESTIONS?

- https://danielkeep.github.io/quick-intro-to-macros.html
- https://doc.rust-lang.org/1.30.0/book/first-edition/macros.html
- https://rust-lang.github.io/rustc-guide/macro-expansion.html
- https://rreverser.com/writing-complex-macros-in-rust/
- https://doc.rust-lang.org/1.30.0/book/2018-edition/appendix-04-macros.html
- http://willcrichton.net/notes/type-directed-metaprogramming-in-rust/
- https://danielkeep.github.io/tlborm/book/mbe-syn-source-analysis.html
- https://github.com/dtolnay/cargo-expand