Macros

Cooper Pierce

Carnegie Mellon University

Table of Contents

Macros in Other Languages

What are some other languages with macros?

Macros in Other Languages

What are some other languages with macros?

- C

Macros in Other Languages

What are some other languages with macros?

- C
- C++

Macros in Other Languages

What are some other languages with macros?

- C
- C++
- Lisp

Macros in Other Languages

What are some other languages with macros?

- C
- C++
- Lisp
- Prolog (SWI)

Macros in Other Languages

What are some other languages with macros?

- C
- C++
- Lisp
- Prolog (SWI)
- Julia

Macros in Other Languages

What are some other languages with macros?

- C
- C++
- Lisp
- Prolog (SWI)
- Julia
- Nim

Macros in Other Languages

What are some other languages with macros?

- C
- C++
- Lisp
- Prolog (SWI)
- Julia
- Nim
- Elixir

Macros in Other Languages

What are some other languages with macros?

- C
- C++
- Lisp
- Prolog (SWI)
- Julia
- Nim
- Elixir

What are the uses of macros?

C(++) Macros

There are a couple uses here:

C(++) Macros

There are a couple uses here:

- constant expressions

C(++) Macros

There are a couple uses here:

- constant expressions
- boilerplate reduction

C(++) Macros

There are a couple uses here:

- constant expressions
- boilerplate reduction
- custom iteration

C(++) Macros

There are a couple uses here:

- constant expressions
- boilerplate reduction
- custom iteration
- “polymorphism”

C(++) Macros

There are a couple uses here:

- constant expressions
- boilerplate reduction
- custom iteration
- “polymorphism”
- in variadic functions

C(++) Macros

There are a couple uses here:

- constant expressions
- boilerplate reduction
- custom iteration
- “polymorphism”
- in variadic functions

... but this system is a bit unwieldy

Real C Code, Written by Real C Programmers

```
LOCAL VOID      gsort(from,to)
        STRING          from[], to[];
{
        INT             k, m, n;
        REG INT         i, j;

        IF (n=to-from)<=1 THEN return FI

        FOR j=1; j<=n; j*=2 DONE

        FOR m=2*j-1; m/=2;
        DO  k=n-m;
            FOR j=0; j=0; i-=m
                DO  REG STRING *fromi; fromi = &from[i];
                    IF cf(fromi[m],fromi[0])>0
                    THEN break;
                    ELSE STRING s; s=fromi[m]; fromi[m]=fromi[0]; fromi[0]=s;
                    FI
                OD
            OD
        OD
}
```

Pitfalls

There are a lot of issues:

Pitfalls

There are a lot of issues:

- macros don't have to produce valid code

Pitfalls

There are a lot of issues:

- macros don't have to produce valid code
- replacements might be contextually wrong

Pitfalls

There are a lot of issues:

- macros don't have to produce valid code
- replacements might be contextually wrong
- processing happens by a separate program, prior to the compiler

Pitfalls

There are a lot of issues:

- macros don't have to produce valid code
- replacements might be contextually wrong
- processing happens by a separate program, prior to the compiler

```c
#define TWO 1 + 1

int main() {
    printf("%d\n", 2 * TWO);
    return 0;
}
```

Table of Contents

Lexing

One of the first steps a compiler has to do is to transform the input source:

```
fn main() {
    let x = 7;
}
```

Lexing

One of the first steps a compiler has to do is to transform the input source:

```
fn main() {
    let x = 7;
}
```

into a list of tokens:

```
FN IDENT LPAREN RPAREN LCURLY LET IDENT EQ INT_LIT SEMICOLON RCURLY
```

Syntax Trees

Once we have a list of tokens:

```
FN IDENT LPAREN RPAREN LCURLY LET IDENT EQ INT_LIT SEMICOLON RCURLY
```

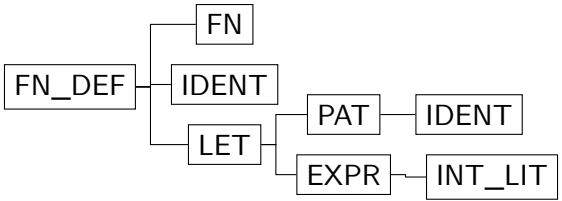
we'll want to turn this into something more structured, essentially a tree:

Syntax Trees

Once we have a list of tokens:

```
FN IDENT LPAREN RPAREN LCURLY LET IDENT EQ INT_LIT SEMICOLON RCURLY
```

we'll want to turn this into something more structured, essentially a tree:

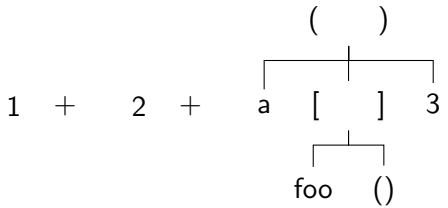


This is an abstract syntax tree (AST).

Token Trees

This isn't the only way to parse tokens into trees though!

```
1 + 2 + (a[foo()] + 3)
```



Rust macros will primarily operate on token trees, but we also need to be aware of the AST.

Table of Contents

Types of Macros

Broadly speaking there are 3 (or 4) different syntaxes for using macros in Rust:

Types of Macros

Broadly speaking there are 3 (or 4) different syntaxes for using macros in Rust:

- Function-like, e.g., `vec![1, 2, 3]`.

Types of Macros

Broadly speaking there are 3 (or 4) different syntaxes for using macros in Rust:

- Function-like, e.g., `vec![1, 2, 3]`.
- Outer attributes, e.g., `#[derive(Clone)]`, `#[cfg(test)]`.

Types of Macros

Broadly speaking there are 3 (or 4) different syntaxes for using macros in Rust:

- Function-like, e.g., `vec![1, 2, 3]`.
- Outer attributes, e.g., `#[derive(Clone)]`, `#[cfg(test)]`.
- Inner attributes, e.g., `#![warn(rust_2018_idioms)]`.

Types of Macros

Broadly speaking there are 3 (or 4) different syntaxes for using macros in Rust:

- Function-like, e.g., `vec![1, 2, 3]`.
- Outer attributes, e.g., `#[derive(Clone)]`, `#[cfg(test)]`.
- Inner attributes, e.g., `#![warn(rust_2018_idioms)]`.

Why (or 4)? There's a special internal syntax-extension syntax we'll see later, but it's currently only used by some compiler built-ins (`macro_rules!`).

Rules for Macros

Like we said before, these operate on token trees. How does the compiler know what token tree something like `println!` is going to operate on?

Rules for Macros

Like we said before, these operate on token trees. How does the compiler know what token tree something like `println!` is going to operate on?

Function-like macros always operate on the immediately following token tree, which is required to be a non-leaf. This means all of these are accepted:

```
print!("This is a test");
print![" of the emergency"];
print!{" broadcast system.\n"};
```

Rules for Macros

When expanding a macro, it expands to fill the corresponing place in the token tree.
Suppose we have a macro `two!` which expands to `1 + 1`.
Then if we have something like:

```
println!("Two times two is {}", 2 * two!());
```

in Rust this will always become

Rules for Macros

When expanding a macro, it expands to fill the corresponing place in the token tree.
Suppose we have a macro `two!` which expands to `1 + 1`.
Then if we have something like:

```
println!("Two times two is {}", 2 * two!());
```

in Rust this will always become

```
println!("Two times two is {}", 2 * (1 + 1));
```

Note that we had to add parens, because it expands to fill the location in the token tree, replacing the current AST node, instead of just spewing out tokens.

Why Use a Macro?

For the most part, there's less use for macros in Rust than for instance, C or C++; they're also more restricted.
What might some uses be?

Why Use a Macro?

For the most part, there's less use for macros in Rust than for instance, C or C++; they're also more restricted.
What might some uses be?

- variadic operations

Why Use a Macro?

For the most part, there's less use for macros in Rust than for instance, C or C++; they're also more restricted.
What might some uses be?

- variadic operations
- domain-specific languages

Why Use a Macro?

For the most part, there's less use for macros in Rust than for instance, C or C++; they're also more restricted.
What might some uses be?

- variadic operations
- domain-specific languages
- boilerplate code generation

Why Use a Macro? Example: `assert_eq!`

Let's take a look at `assert_eq!`. We've seen this before, with writing test cases.

```
#[test]
fn my_test_fn() {
    assert_eq!(fib(6), 8);
}
```

Why might this be a macro instead of a function?

Why Use a Macro? Example: `assert_eq!`

Let's take a look at `assert_eq!`. We've seen this before, with writing test cases.

```
#[test]
fn my_test_fn() {
    assert_eq!(fib(6), 8);
}
```

Why might this be a macro instead of a function?

We can also use it like so:

```
#[test]
fn my_test_fn() {
    assert_eq!(fib(6), 8, "Testing the 6th fibbonacci number");
}
```

Defining a Declarative Macro

The first way to define macros, which we'll call declarative macros, uses `macro_rules!`

```
macro_rules! assert_eq {
    ($left : expr, $right : expr $(,) ?) => { ... };
    ($left : expr, $right : expr, $($arg : tt) +) => { ... };
}
```

Some things to note:

Defining a Declarative Macro

The first way to define macros, which we'll call declarative macros, uses `macro_rules!`

```
macro_rules! assert_eq {
    ($left : expr, $right : expr $(,) ?) => { ... };
    ($left : expr, $right : expr, $($arg : tt) +) => { ... };
}
```

Some things to note:

- The macro name

Defining a Declarative Macro

The first way to define macros, which we'll call declarative macros, uses `macro_rules!`

```
macro_rules! assert_eq {
    ($left : expr, $right : expr $(,) ?) => { ... };
    ($left : expr, $right : expr, $($arg : tt) +) => { ... };
}
```

Some things to note:

- The macro name
- We introduce arguments with `$name`

Defining a Declarative Macro

The first way to define macros, which we'll call declarative macros, uses `macro_rules!`

```
macro_rules! assert_eq {
    ($left : expr, $right : expr $(,) ?) => { ... };
    ($left : expr, $right : expr, $($arg : tt) +) => { ... };
}
```

Some things to note:

- The macro name
- We introduce arguments with `$name`
- and list the "types" of them

Defining a Declarative Macro

The first way to define macros, which we'll call declarative macros, uses `macro_rules!`

```
macro_rules! assert_eq {
    ($left : expr, $right : expr $(,) ?) => { ... };
    ($left : expr, $right : expr, $($arg : tt) +) => { ... };
}
```

Some things to note:

- The macro name
- We introduce arguments with `$name`
- and list the "types" of them
- We also have some control over repeated constructs

“Types” of Token Trees

- 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`, ...)
- literal: a literal (e.g. `"Hello World!"`, `3.14`, `'X'`, ...)
- 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>`, ...)
- stmt: a statement
- tt: a single token tree
- ty: a type
- vis: a possible empty visibility qualifier (e.g. `pub`, `pub(in crate)`, ...)

Repetition Constructs

We can use the following sequence

```
$ ( ... ) sep rep
```

where `( ... )` is the group being repeated; `sep` is some token which separates the groups (think something like `,`); and `rep` is one of:

- `?`—at most one (no `sep`)
- `+`—at least one
- `*`—any amount

You then would use this same syntax when dealing with the repeated group.

Declarative Macro Example

Let's define a simpler version of the `vec!` macro, which supports the syntax `vec![1, 2, 3, 4]` (and allows a trailing comma).

Declarative Macro Example

Let's define a simpler version of the `vec!` macro, which supports the syntax `vec![1, 2, 3, 4]` (and allows a trailing comma).

```
macro_rules! vec {
    ( $( $x:expr ) , * $ (,) ?) => {
        {
            let mut v = Vec::new();
            $(
                v.push($x);
            )*

            v
        }
    };

}
```

While we can only defined a function-like macro with declarative macros, proc macros allow us to also defined attributes.

Here, instead of writing something like a `match` expression, we'll write a function which operates on token trees.

However, proc macros are a little bit more cumbersome.

For instance,

```
[lib]
proc-macro = true
```

is required in your `Cargo.toml`, and they can't be used in the same crate, because the compiler needs to compile them.

Defining a Procedular Macro

There are a couple different ways we define proc macros, depending on their use. For function-like macros, it'll take one argument, and have `#[proc_macro]`:

```
#[proc_macro]
pub fn my_proc_macro(input: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

For general attribute macros, it'll take two arguments, one for attribute args and another for the item itself, with `#[proc_macro_attribute]`:

```
#[proc_macro_attribute]
pub fn my_attribute(input: TokenStream, annotated_item: TokenStream)
    -> TokenStream {
    TokenStream::new()
}
```

For derive macros, it'll take two arguments, one for attribute args and another for the item itself, with `#[proc_macro_derive]`:

```
#[proc_macro_derive(MyDerive)]
pub fn my_derive(annotated_item: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

Proc Macros: Function-like Example

The simplest thing we could write for a function-like proc macro would just be the identity macro:

```
#[proc_macro]
pub fn ident(x: TokenStream) -> TokenStream {
    x
}
```

So if we used `ident!(foo)` it would just expand to `foo`.
We could implement much the same thing as our `vec!` macro, but we'd have to go through the `TokenStream` and manually validate all of it—that's a lot of work! In general, if we can use a declarative macro we'll want to, and if we can avoid a macro, we'll perfer that to both.

Proc Macros: Derive

A more motiviating example is that of derive macros. We can imagine lots of cases where we have trait we want to make easy to implement for clients of a library we're implementing, but the implementation is often rote, and just invovles recursing on the fields of a struct.

Let's imagine we have a trait `Hello`, defined like so:

```
trait Hello {
    fn hello() -> String;
}
```

```rust
use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(Hello)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}
```

```rust
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!",
                    stringify!(#name));
            }
        }
    };
    gen.into()
}
```

So, we've already had to pull in two dependencies `quote` and `syn`, in order to implement this, and that's for a relative simple example.

This is somewhat constant overhead, but less than ideal.

So, we've already had to pull in two dependencies `quote` and `syn`, in order to implement this, and that's for a relative simple example.

This is somewhat constant overhead, but less than ideal.

Another factor is compile time—proc macros can lead to greatly expanded compilation times, especially with `quote` and `syn`.