Lecture 9: Macros

04.01.2024

*Warning

This lecture may be difficult to follow if you have not had the following corequisites:

- CIS 2400
- CIS 2620
- CIS 3410

I will try to explain some of the relevant concepts, but please stop me if you have any questions!

Three Types of Macros

`println!()`

Functions

Expand into function calls

`#[derive(..)]`

Derives

Automatic implementations for structs, enums, or union types

`#[test]`

Attributes

Annotations that modify code behavior

TL;DR: Macros are code that generate code

Expanded into source code at compile time

This expansion happens early in compilation, before any static checking.

Macros take longer to compile, but faster to execute

It also allows for variadic arguments, which functions don't currently have support for.

Example Use Case

```
pub struct ParallelExecutorImpl<const NUM_THREADS: usize> {}
…
macro_rules! parallel_executor {
    ($($num_threads:literal),*) => {
        $(
            paste::item! {
                #[allow(unused)]
                pub type [<ParallelExecutor $num_threads>] =
ParallelExecutorImpl<$num_threads>;
            }
        )*
    };
}

parallel_executor!(
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
);
```

Compiler Detour

CIS 3410 Speedrun

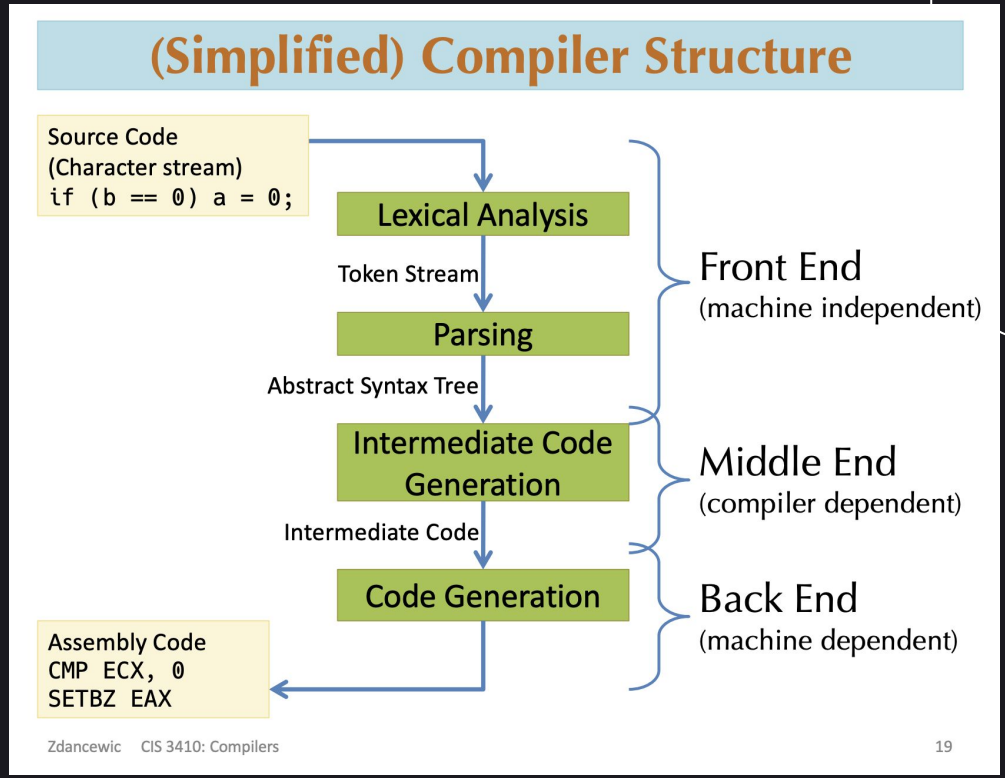
Rust Compiler (rustc) Oversimplified

The Rust compiler (rustc) reads the .rs files and parses the source code into an Abstract Syntax Tree (AST).

- ***Macro Expansion***
- Type checking

Intermediate Representation (IR) Generation: The compiler converts the AST into an IR that's easy to optimize

Code Generation: The compiler translates the optimized IR into machine code specific to the target platform (e.g., x86_64, ARM). This step is often handled by LLVM in Rust



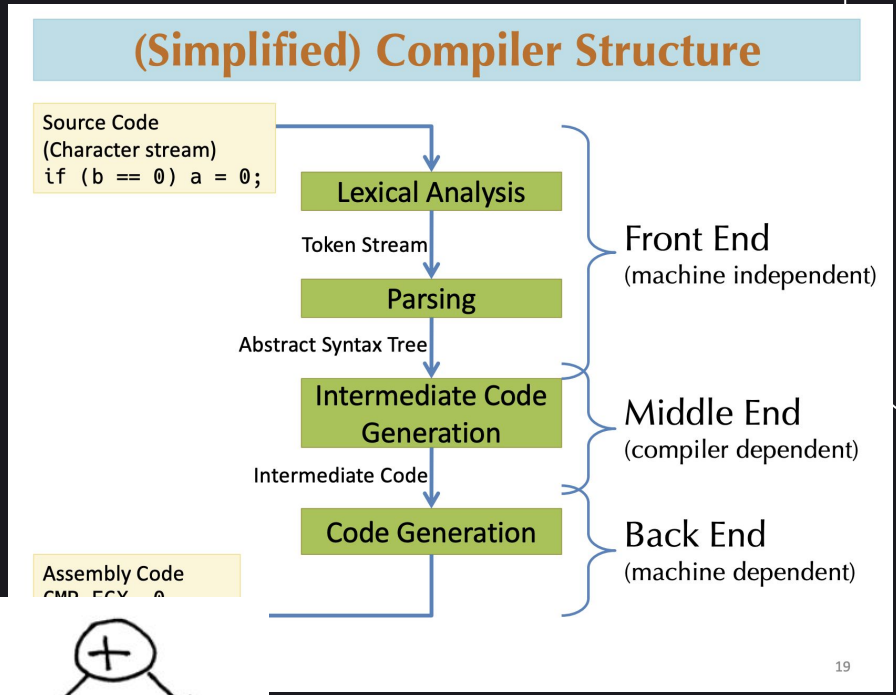
Key Vocabulary

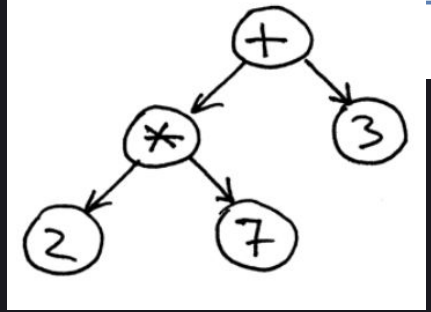
TokenStream

- Stream of lexical tokens such as `Punct` (.), `Ident` (foo), `Literal` (1.0)

Abstract Syntax Tree (AST)

- Tree representation of a program





Breaking down macro_rules implementation

`vec![]` macro

```
let x: Vec<u32> = vec![1, 2, 3];
```

Initialize an empty vector
For each number:
    push it to an empty vec
Return resulting vector expression

`vec![]` macro

```
let x: Vec<u32> = vec![1, 2, 3];
```

Could translate to...

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
```

vec![] macro

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
fn main() {
    assert_eq!(vec![1,2,3], [1, 2, 3]);
}
```

vec![] macro

```
macro_rules! vec { // defining a macro named vec
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
fn main() {
    assert_eq!(vec![1,2,3], [1, 2, 3]);
}
```

vec![] macro

```
macro_rules! vec { // defining a macro named vec
    ( $( $x:expr ),* ) => { // pattern matching
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
fn main() {
    assert_eq!(vec![1,2,3], [1, 2, 3]);
}
```

vec![] macro

```
macro_rules! vec { // defining a macro named vec
    ( $( $x:expr ),* ) => { // pattern matching
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )* // use matched result
            temp_vec
        }
    };
}
fn main() {
    assert_eq!(vec![1,2,3], [1, 2, 3]);
}
```

vec![] macro

```
macro_rules! vec { // defining a macro named vec
    ( $( $x:expr ),* ) => { // pattern matching
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec // return value
        }
    };
}
fn main() {
    assert_eq!(vec![1,2,3], [1, 2, 3]);
}
```

Macro Pattern Matching Syntax

Similar to `match` statements

```
enum Language {
        Expressions,
        Statements,
        Items,
        Patterns
}
match my_code {
        Expressions(_) => _,
        ...
}
```

```
( $( $x:expr ),* ) =>
```

$name: designator

- `item`: an item
- `block`: a block
- `stmt`: a statement
- `pat`: a pattern
- `expr`: an expression
- `ty`: a type
- `ident`: an identifier
- `path`: a path
- `tt`: a token tree (a single token by matching `()`, `[]`, or `{}`)
- `meta`: the contents of an attribute

```
( $( $x:expr ),* ) =>
```

Macro Pattern Matching Syntax

Similar to `match` statements, you can specify different patterns that a macro can be invoked with:

- Expressions
- Statements
- Items
- Patterns

```
( $( $x:expr ),* ) =>
```

For example, optional args.The macro expander looks up macro invocations by name, and tries each macro rule in turn. It transcribes the first successful match.

Macro By Example

```
macro_rules! vec { // defining a macro named vec
    ( $( $x:expr ),* ) => { // pattern matching
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
fn main() {
    assert_eq!(vec![1,2,3], [1, 2,
}
```

Tries to pattern match for an “expression” type and assign it to name “x”

Kleene Star Recap

The Kleene star operator consists of $ and parentheses, optionally followed by a separator token, followed by * or +.

- * means zero or more repetitions
- + means at least one repetition.

($($num:expr),*) ($($num:expr),+)

macro!(1, 2, 3) macro!(1.0, 2.0) macro!() macro!(1.0, 2)

Kleene Star Recap

The Kleene star operator consists of $ and parentheses, optionally followed by a separator token, followed by * or +

- **Which of the following expressions matches against each of the two patterns?**

($($num:expr),*)
macro!(1, 2, 3) macro!(1.0, 2.0) macro!()

($($num:expr),+)
macro!(1.0, 2) macro!(1.0, 2)

Kleene Star Recap

The Kleene star operator consists of $ and parentheses, optionally followed by a separator token, followed by * or +.

- * means zero or more repetitions
- + means at least one repetition.

($($num:expr),*)
macro!(1, 2, 3) macro!()

($($num:expr),+)
macro!(1.0, 2.0) ~~macro!(1.0, 2)~~

vec![] macro

```
macro_rules! vec { // defining a macro named vec
    ( $( $x:expr ),* ) => { // pattern matching
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x); // uses x in expr
            )*
            temp_vec
        }
    };
}
fn main() {
    assert_eq!(vec![1,2,3], [1, 2, 3]);
}
```

vec![] macro

```
macro_rules! vec { // defining a macro named vec
    ( $( $x:expr ),* ) => { // pattern matching
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x); // uses x in expr
            )*
            temp_vec
        }
    };
}
fn main() {
    assert_eq!(vec![1,2,3], [1, 2, 3]);
}
```

Because ($(),*) is used, for each x, it pushes it to temp_vec

Repetition Rules

When Macro By Example encounters a repetition, it examines all of the $ name s that occur in its body. At the "current layer", they all must repeat the same number of times,

```
$( $i:ident ),* ; $( $j:ident ),* ) =>
    ( $( ($i,$j) ),* ) // zip-like
```

is valid if given the argument (a,b,c ; d,e,f), but not (a,b,c ; d,e). The repetition walks through the choices at that layer in lockstep, so the former input transcribes to (a,d), (b,e), (c,f).

Common Macros

Benchmarking `vec!` macro

See live code

Benchmarking vec! macro

Why is this faster?

1. **Pre-allocation of Memory**: knows the number of elements in advance.
2. **Optimized Implementation**: the actual vec macro is optimized by the compiler (see code)

```
     Running benches/bench_macro.rs (target/release/deps/bench_macro-aeacaf319f5ffadc)
Gnuplot not found, using plotters backend
vec_macro               time:   [82.886 ns 89.728 ns 98.274 ns]
Found 8 outliers among 100 measurements (8.00%)
  4 (4.00%) high mild
  4 (4.00%) high severe

vec_push                time:   [261.21 ns 264.43 ns 268.38 ns]
Found 5 outliers among 100 measurements (5.00%)
  3 (3.00%) high mild
  2 (2.00%) high severe
```

assert! macro

```
macro_rules! assert {
    ($cond:expr) => (
        if !$cond {
            panic!(concat!("assertion failed: ",
stringify!($cond)))
        }
    );
    ($cond:expr, $($arg:tt)+) => (
        if !$cond {
            panic!($($arg)+)
        }
    );
}
```

println! macro

```
macro_rules! println {
    () => {
        $crate::print!("\n")
    };
    ($($arg:tt)*) => {{

$crate::io::_print($crate::format_args_nl!($($arg)*));
    }};
}
```

Proc Macros

Declarative vs Procedural Macros

Declarative: macro rules
Procedural: must be inner crate

Procedural macros operate over token streams instead of AST nodes:

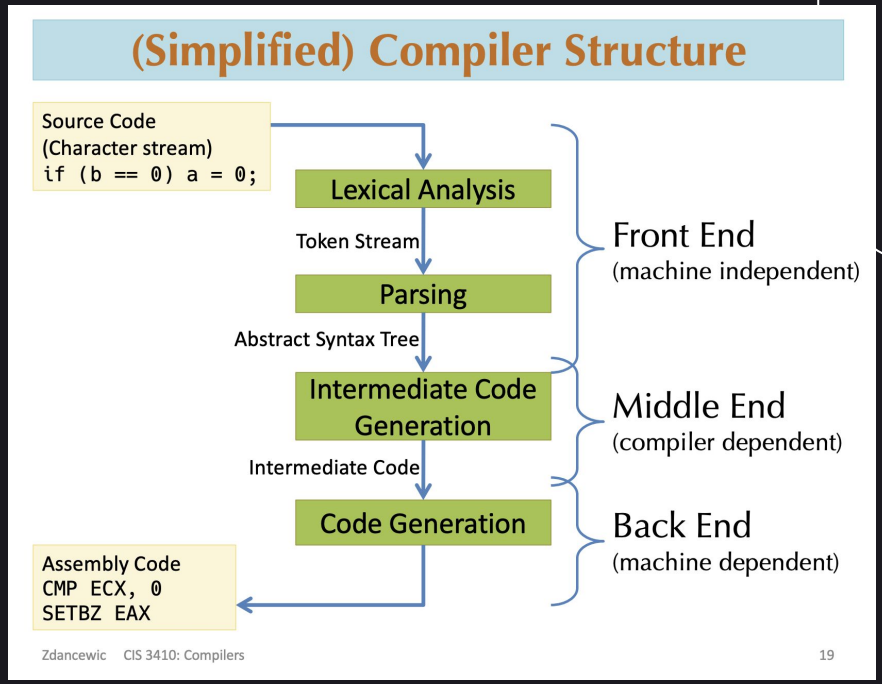
- A stream of token trees, where each token tree is a lexical token (e.g. Ident, Punct, Literal)

Declarative vs Procedural Macros

Declarative: macro rules
Procedural: must be inner crate

Procedural macros operate over token streams instead of AST nodes:

- A stream of token trees, where each token tree is a lexical token (e.g. Ident, Punct, Literal)



Different types of Proc Macros

- Function-like macros - `custom!(...)`
- Derive macros - `#[derive(CustomDerive)]`
- Attribute macros - `#[CustomAttribute]`

Run code at compile time that operates over Rust syntax, both consuming and producing Rust syntax.

Example Proc Macro

```
extern crate proc_macro_examples;
use proc_macro_examples::make_answer;

make_answer!();

fn main() {
    println!("{}", answer());
}
```

How proc_macros work

As functions, they must either return **syntax**, **panic**, or **loop endlessly**.

1. Returned syntax either replaces or adds the syntax depending on the kind of procedural macro.

2. Panics are caught and are turned into a compiler error.

3. Endless loops will hang the compiler.

How proc_macros work

As functions, they must either return **syntax**, **panic**, or **loop endlessly**.

1. Returned syntax either replaces or adds the syntax depending on the kind of procedural macro.

Tokio's #[main] macro takes an `async fn main` and turns it into a normal main function that spins up a runtime to run the Future.

2. Panics are caught and are turned into a compiler error.

3. Endless loops will hang the compiler.

How proc_macros work

As functions, they must either return **syntax**, **panic**, or **loop endlessly**.

1. Returned syntax either replaces or adds the syntax depending on the kind of procedural macro.

Tokio's #[main] macro takes an `async fn main` and turns it into a normal main function that spins up a runtime to run the Future.

2. Panics are caught and are turned into a compiler error.

sqlx's macros try to connect to a local db to queries at compile time.

```
pub fn query_rows(input: TokenStream) -> TokenStream {
    panic!("the column you named wasn't in the db!")
}
```

3. Endless loops will hang the compiler.

How proc_macros work

As functions, they must either return **syntax**, **panic**, or **loop endlessly**.

1. Returned syntax either replaces or adds the syntax depending on the kind of procedural macro.

Tokio's #[main] macro takes an `async fn main` and turns it into a normal main function that spins up a runtime to run the Future.

2. Panics are caught and are turned into a compiler error.

sqlx's macros try to connect to a local db to queries at compile time.

```
pub fn query_rows(input: TokenStream) -> TokenStream {
    panic!("the column you named wasn't in the db!")
}
```

3. Endless loops will hang the compiler.

```
pub fn inf(input: TokenStream) -> TokenStream { loop {} }
```

Proc Macro Errors

1. Panic

```
pub fn query_rows(input: TokenStream) -> TokenStream {
    panic!("the column you named wasn't in the db!")
}
```

2. emit a compile_error macro invocation.

```
#[proc_macro_attribute]
pub fn only_structs(input: TokenStream) -> TokenStream {
    // ... find out if it's a struct

    let gen = quote! {
        compile_error!("The only_structs macro only goes on
structs")
    };
    gen.into()
}
```

Proc Macro Good Practice

Proc Macros are unhygienic, which means because the generated code is written inline, it can be affected by external items and imports:

- Use absolute paths when importing
- Ensure generated function names do not clash with common names

Macros Summary