Macros!

CIS 198 Lecture 13

What Are Macros?

- In C, a macro looks like this:

```c
#define FOO 10  // untyped integral constant
#define SUB(x, y) ((x) - (y))  // parentheses are important!
#define BAZ a  // relies on there being an `a` in context!

int a = FOO;
short b = FOO;
int c = -SUB(2, 3 + 4);
int d = BAZ;
```

- The preprocessor runs before the compiler, producing:

```c
int a = 10;                 // = 10
short b = 10;               // = 10
int c = -((2) - (3 + 4)); // = -(2 - 7) = 5
int d = a;                  // = 10
```

Why C Macros Suck[1]

- C does a direct *token-level* substitution.
  - The preprocessor has no idea what variables, types, operators, numbers, or anything else actually *mean*.
- Say we had defined `SUB` like this:

```c
#define SUB(x, y) x - y

int c = -SUB(2, 3 + 4);
```

- This would break terribly! After preprocessing:

```c
int c = -2 - 3 + 4;  // = -1, not 5.
```

[1] GCC Docs: Macro Pitfalls

Why C Macros Suck

- Further suppose we decided to rename `a`:

```
#define FOO 10
#define BAZ a  // relies on there being an `a` in context!

int a_smith = FOO;
int d = BAZ;
```

- Now, the preprocessor produces garbage!

```
int a_smith = 10;  // = 10
int d = a;         // error: `a` is undeclared
```

Why C Macros Suck

- Since tokens are substituted directly, results can be surprising:

```c
#define SWAP(x, y) do { \
    (x) = (x) ^ (y);    \
    (y) = (x) ^ (y);    \
    (x) = (x) ^ (y);    \
} while (0)  // Also, creating multiple statements is weird.

int x = 10;
SWAP(x, x);  // `x` is now 0 instead of 10
```

- And arguments can be executed multiple times:

```c
#define DOUBLE(x) ((x) + (x))

int x = DOUBLE(foo());  // `foo` gets called twice
```

Why C Macros Suck

- C macros also can't be recursive:

```c
#define foo (4 + foo)

int x = foo;
```

- This expands to

```c
int x = 4 + foo;
```

  - (This particular example is silly, but recursion *is* useful.)

Why C Macros Suck

- In C, macros are also used to include headers (to use code from other files):

```c
#include <stdio.h>
```

  - Since this just dumps `stdio.h` into this file, each file now gets bigger and bigger with additional includes.
  - This is a major contributor to long build times in C/C++ (especially in older compilers).

Rust Macros from the Bottom Up

- Almost all material stolen from Daniel Keep's *excellent* book:
  - The Little Book of Rust Macros (TLBORM).
  - This section from Chapter 2.

Rust Syntax Extensions

- Rust has a generalized system called *syntax extensions*. Anytime you see one of these, it means a syntax extension is in use:
  - `#[foo]` and `#![foo]`
    - These are used for attributes.
  - `foo! arg`
    - Always `foo!(...)`, `foo![...]`, or `foo!{...}`
    - *Sometimes* means `foo` is a macro.
  - `foo! arg arg`
    - Used only by `macro_rules! name { definition }`
- The third form is the one used by macros, which are a special type of syntax extension - defined within a Rust program.
- These can also be implemented by *compiler plugins*, which have much more power than macros.

Rust Macros

- A Rust macro looks like this:

```
macro_rules! incr {  // define the macro
    // syntax pattern => replacement code
    ( $x:ident ) => { $x += 1; };
}

let mut x = 0;
incr!(x);  // invoke a syntax extension (or macro)
```

- So... this is totally foreign. The heck's going on?

Rust Syntax - Token Streams

- Before we dive in, we need to know a little about Rust's lexer and parser.
  - A *lexer* is a compiler stage which turns the original source (a string) into a stream of tokens.
  - A string of code like `((2 + 3) + 10)` will turn into a stream like
    - `'(' '(' '2' '+' '3' ')' '+' '10' ')'`.
- Tokens can be:
  - Identifiers: `foo`, `Bambous`, `self`, `we_can_dance`, ...
  - Integers: `42`, `72u32`, `0_______0`, ...
  - Keywords: `_`, `fn`, `self`, `match`, `yield`, `macro`, ...
  - Lifetimes: `'a`, `'b`, ...
  - Strings: `""`, `"Leicester"`, `r##"venezuelan beaver"##`, ...
  - Symbols: `[`, `:`, `::`, `->`, `@`, `<-`, ...
- In C, macros see the token stream as input.

Rust Syntax - Token Trees

- After lexing, a small amount of parsing is done to turn it into a *token tree*.
  - This *isn't* a full-fledged AST (abstract syntax tree).
  - For the token stream `'(' '(' '2' '+' '3' ')' '+' '10' ')'`, the token tree looks like this:
    ```
    Parens
    ├── Parens
    │   ├── '2'
    │   ├── '+'
    │   └── '3'
    ├── '+'
    └── 10
    ```
- In Rust, macros see *one* token tree as input.
  - When you do `println!("{}", (5+2))`, the `"{}", (5+2)` will get parsed into a token tree, but *not* fully parsed into an AST.

Rust Syntax - AST

- The AST (abstract syntax tree) is the fully-parsed tree.
    - All syntax extension (and macro) invocations are expanded, then parsed into sub-ASTs after the initial AST construction.
        - Syntax extensions must output valid, contextually-correct Rust.
- Syntax extension calls can appear in place of the following syntax kinds, by outputting a valid AST of that kind:
    - Patterns (e.g. in a `match` or `if let`).
    - Statements (e.g. `let x = 4;`).
    - Expressions (e.g. `x * (y + z)`).
    - Items (e.g. `fn`, `struct`, `impl`, `use`).
- They *cannot* appear in place of:
    - Identifiers, match arms, struct fields, or types.

Macro Expansion

- Let's parse this Rust code into an AST:

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

```
Let { name: eight
      init: BinOp {  op: Mul
                    lhs: LitInt { val:  2 }
    /* macro */     rhs: Macro  { name: four
    /* input */                   body: () } } }
```

- If `four!()` is defined to expand to `1 + 3`, this expands to:

```
Let { name: eight
      init: BinOp {  op: Mul
                    lhs: LitInt { val: 2 }
    /* macro  */    rhs: BinOp  {  op: Add
    /* output */                  lhs: LitInt { val: 1 }
    /*        */                  rhs: LitInt { val: 3 } } } }
```

```rust
let eight = 2 * (1 + 3);
```

Macro Rules

- Put simply, a macro is just a compile-time pattern match:

```
macro_rules! mymacro {
    ($pattern1) => {$expansion1};
    ($pattern2) => {$expansion2};
    // ...
}
```

- The `four!` macro is simple:

```
macro_rules! four {
    // For empty input, produce `1 + 3` as output.
    () => {1 + 3};
}
```

Macro Rules

- Any valid Rust tokens can appear in the match:

```
macro_rules! imaginary {
    (twentington) => {"20ton"};
    (F00 & nee) => {"f0e"};
}

imaginary!(twentington);
imaginary!(F00&nee);
imaginary!(schinty six); // won't compile; is a real number
```

Macro Rules - Captures

- Portions of the input token tree can be *captured*:

```
macro_rules! sub {
    ($e1:expr, $e2:expr) => { ... };
}
```

- Captures are always written as `$name:kind`.
  - Possible kinds are:
    - `item`: an item, like a function, struct, module, etc.
    - `block`: a block (i.e. `{ some; stuff; here }`)
    - `stmt`: a statement
    - `pat`: a pattern
    - `expr`: an expression
    - `ty`: a type
    - `ident`: an identifier
    - `path`: a path (e.g. `foo`, `::std::mem::replace`, ...)
    - `meta`: a meta item; the things that go inside `#[...]`
    - `tt`: a single token tree

Macro Rules - Captures

- Captures can be substituted back into the expanded tree

```
macro_rules! sub {
    ( $e1:expr , $e2:expr ) => { $e1 - $e2 };
}
```

- A capture will always be inserted as a **single** AST node.
  - For example, `expr` will always mean a valid Rust expression.
  - This means we're no longer vulnerable to C's substitution problem (the invalid order of operations).
  - Multiple expansions will still cause multiple evaluations:

```
macro_rules! twice {
    ( $e:expr ) => { { $e; $e } }
}

fn foo() { println!("foo"); }

twice!(foo());  // expands to { foo(); foo() }: prints twice
```

Macro Rules - Repetitions

- If we want to match a list, a variable number of arguments, etc., we can't do this with the rules we've seen so far.
  - *Repetitions* allow us to define repeating subpatterns.
  - These have the form `$ ( ... ) sep rep`.
    - `$` is a literal dollar token.
    - `( ... )` is the paren-grouped pattern being repeated.
    - `sep` is an *optional* separator token.
      - Usually, this will be `,` or `;`.
    - `rep` is the *required* repeat control. This can be either:
      - `*` zero or more repeats.
      - `+` one or more repeats.
  - The same pattern is used in the output arm.
    - The separator doesn't have to be the same.

Macro Rules - Repetitions

- We can use these to reimplement our own `vec!` macro:

```
macro_rules! myvec {
    (   $(                  // Start a repetition
            $elem:expr  // Each repetition matches one expr
        ) ,             // Separated by commas
          *             // Zero or more repetitions
    ) => {
        { // Braces so we output only one AST (block kind)
            let mut v = Vec::new();

            $(                      // Expand a repetition
                v.push($elem); // Expands once for each input rep
             ) *                    // No sep; zero or more reps

            v                       // Return v from the block.
        }
    }
}
println!("{:?}", myvec![3, 4]);
```

Macro Rules - Repetitions

- Condensed:

```
macro_rules! myvec {
    ( $( $elem:expr ),* ) => {
        {
            let mut v = Vec::new();
            $( v.push($elem); )*
            v
        }
    }
}
println!("{:?}", myvec![3, 4]);
```

Macro Rules - Matching

- Macro rules are matched in order.
- The parser can never backtrack. Say we have:

```
macro_rules! dead_rule {
    ($e:expr) => { ... };
    ($i:ident +) => { ... };
}
```

- If we call it as `dead_rule(x +);`, it will actually fail.
    - `x +` isn't a valid expression, so we might think it would fail on the first match and then try again on the second.
    - This doesn't happen!
    - Instead, since it *starts* out looking like an expression, it commits to that match case.
        - When it turns out not to work, it can't *backtrack* on what it's parsed already, to try again. Instead it just fails.

Macro Rules - Matching

- To solve this, we need to put more specific rules first:

```
macro_rules! dead_rule {
    ($i:ident +) => { ... };
    ($e:expr) => { ... };
}
```

- Now, when we call `dead_rule!(x +);`, the first case will match.
- If we called `dead_rule!(x + 2);`, we can now fall through to the second case.
  - Why does this work?
  - Because if we've seen `$i:ident +`, the parser already knows that this looks like the beginning of an expression, so it can fall through to the second case.

Macro Expansion - Hygiene

- In C, we talked about how a macro can implicitly use (or conflict) with an identifier name in the calling context (`#define BAZ a`).
- Rust macros are *partially hygenic*.
  - Hygenic with regard to most identifiers.
    - These identifiers get a special context internal to the macro expansion.
  - NOT hygenic: generic types (`<T>`), lifetime parameters (`<'a>`).

```
macro_rules! using_a {
    ($e:expr) => {     { let a = 42;  $e }  }
} // Note extra braces ^                 ^

let four = using_a!(a / 10); // this won't compile - nice!
```

  - We can imagine that this expands to something like:

```
let four = { let using_a_1232424_a = 42; a / 10 };
```

Macro Expansion - Hygiene

- But if we *want* to bind a new variable, it's possible.
  - If a token comes in as an input to the function, then it is part of the caller's context, not the macro's context.

```
macro_rules! using_a {
    ($a:ident, $e:expr) => {  { let $a = 42;  $e }  }
}        // Note extra braces ^                  ^

let four = using_a!(a, a / 10); // compiles!
```

  - This expands to:

```
let four = { let a = 42; a / 10 };
```

Macro Expansion - Hygiene

- It's also possible to create identifiers that will be visible outside of the macro call.
    - This won't work due to hygiene:

```
macro_rules! let_four {
    () => { let four = 4; }
}         // ^ No extra braces

let_four!();
println!("{}", four); // `four` not declared
```

    - But this will:

```
macro_rules! let_four {
    ($i:ident) => { let $i = 4; }
}                   // ^ No extra braces

let_four!(myfour);
println!("{}", myfour); // works!
```

Nested and Recursive Macros

- If a macro calls another macro (or itself), this is fine:

```
macro_rules! each_tt {
    () => {};
    ( $_tt:tt $($rest:tt)* ) => { each_tt!( $($rest)* ); };
}
```

  - The compiler will keep expanding macros until there are none left in the AST (or the recursion limit is hit).
  - The compiler's recursion limit can be changed with `#![recursion_limit="64"]` at the crate root.
    - 64 is the default.
    - This applies to all recursive compiler operations, including auto-dereferencing and macro expansion.

Macro Debugging

- Rust has an unstable feature for debugging macro expansion.
  - Especially recursive macro expansions.

```
#![feature(trace_macros)]
macro_rules! each_tt {
    () => {};
    ( $_tt:tt $($rest:tt)* ) => { each_tt!( $($rest)* ); };
}

trace_macros!(true);
each_tt!(spim wak plee whum);
trace_macros!(false);
```

  - This will cause the compiler to print:

```
each_tt! { spim wak plee whum }
each_tt! { wak plee whum }
each_tt! { plee whum }
each_tt! { whum }
each_tt! {  }
```

- More tips on macro debugging in TLBORM 2.3.4

Macro Scoping

- Macro scoping is unlike everything else in Rust.
  - Macros are immediately visible in submodules:

```
macro_rules! X { () => {}; }

mod a {  // Or `mod a` could be in `a.rs`.
    X!(); // valid
}
```

  - Macros are only defined *after* they appear in a module:

```
mod a { /* X! undefined here */ }

mod b {
    /* X! undefined here */
    macro_rules! X { () => {}; }
    X!(); // valid
}

mod c { /* X! undefined */ } // They don't leak between mods.
```

Macro Scoping

- Macros can be exported from modules:

```rust
#[macro_use]  // outside of the module definition
mod b {
    macro_rules! X { () => {}; }
}

mod c {
    X!(); // valid
}
```

  - Or from crates, using `#[macro_export]` in the crate.
- There are a few other weirdnesses of macro scoping.
  - See TLBORM 2.3.5 for more.
- In general, to avoid too much scope weirdness:
  - Put your crate-wide macros at the top of your root module (`lib.rs` or `main.rs`).

Rust Macros Design Patterns

- This section from TLBORM Chapter 4.
  - I won't cover most of the chapter.

Macro Callbacks

- Because of the way macros are expanded, "obviously correct" macro invocations like this won't actually work:

```
macro_rules! expand_to_larch {
    () => { larch };
}

macro_rules! recognise_tree {
    (larch) => { println!("larch") };
    (redwood) => { println!("redwood") };
    ($($other:tt)*) => { println!("dunno??") };
}

recognise_tree!(expand_to_larch!());
```

  - This will be expanded like so:

```
-> recognize_tree!{ expand_to_larch ! ( ) };
-> println!("dunno??");
```

  - Which will match the third pattern, not the first.

Macro Callbacks

- This can make it hard to split a macro into several parts.
  - This isn't always a problem - `expand_to_larch ! ( )` won't match an `ident`, but it *will* match an `expr`.
- The problem can be worked around by using a *callback* pattern:

```
macro_rules! call_with_larch {
    ($callback:ident) => { $callback!(larch) };
}

call_with_larch!(recognize_tree);
```

  - This expands like this:

```
-> call_with_larch! { recognise_tree }
-> recognise_tree! { larch }
-> println!("larch");
```

Macro TT Munchers

- This is one of the most powerful and useful macro design patterns. It allows for parsing fairly complex grammars.
- A *tt muncher* is a macro which matches a bit at the beginning of its input, then recurses on the remainder of the input.
  - `( $some_stuff:expr $( $tail:tt )* ) =>`
  - Usually needed for any kind of actual language grammar.
  - Can only match against literals and grammar constructs which can be captured by `macro_rules!`.
  - Cannot match unbalanced groups.

Macro TT Munchers

```
macro_rules! mixed_rules {
    () => {}; // Base case
    (trace $name:ident ; $( $tail:tt )*) => {
        {
            println!(concat!(stringify!($name), " = {:?}"), $name);
            mixed_rules!($($tail)*);  // Recurse on the tail of the i
        }
    };
    (trace $name:ident = $init:expr ; $( $tail:tt )*) => {
        {
            let $name = $init;
            println!(concat!(stringify!($name), " = {:?}"), $name);
            mixed_rules!($($tail)*);  // Recurse on the tail of the i
        }
    };
}
```

Macros Rule! Mostly!

- Macros are pretty great - but not perfect.
  - Macro hygiene isn't perfect.
  - The scope of where you can use a macro is weird.
  - Handling crates inside of exported macros is weird.
  - It's impossible to construct entirely new identifiers (e.g. by concatenating two other identifiers).
  - ...
- A new, incompatible macro system may appear in future Rust.
  - This would be a new syntax for writing syntax extensions.