Macros

Rust, In Theory and in Practice

CAS CS 392 M1

High Level

Macros are used for **metaprogramming**, i.e., the generation of code at compile-time, e.g.,

» `#[derive(...)]`

» `println!("{}, {}", 1, 2)`

» `vec![1, 2, 3]`

Benefits and Drawbacks

```
macro_rules! vec {
    () => (
        $crate::vec::Vec::new()
    );
    ($elem:expr; $n:expr) => (
        $crate::vec::from_elem($elem, $n)
    ...
```

Benefit: More control to the programmer, can write DSLs, variadic functions, etc.

Downside: More control to the programmer, macros can be hard to read and debug, they can make code less clear

Declarative Macros

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Also called "macros by example"

You specify a small grammar that you want the rule to parse, and what you want the data in the input to expand to

Fragment Specifiers

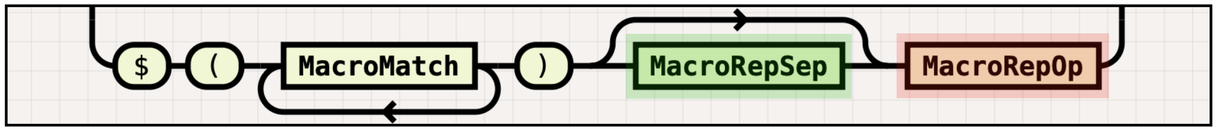
```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {...}
```

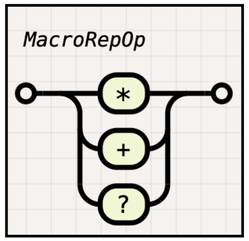
Fragment specifiers describe the kind of data that can be parsed:

block, expr, ident, item, literal, pat, path, stmt, tt (token tree), ty (type), vis (visibility qualifier)

Repetitions

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {...}
```





Patterns can include repetitions and separators:

- (*) zero or more
- (+) one or more
- (?) zero or one

Use Cases

```rust
use regex_macro::regex;

for item in my_iter {
    // this is still only compiled once!
    if regex!("[0-9a-f]+").is_match(item) {
        // frobnicate
    }
}
```

Declarative Macros are useful for snippets of repeated patterns

Simple examples: regex_macro and lazy_static

demo

(rforth, simplified)

Procedural Macros

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

The same as declarative macros in principle

But you're working with code as *a piece of data* and you have the full power of Rust

The sky's the limit

syn and quote

syn is a crate that can parse TokenSteams into Rust ADTs

quote is a crate that will convert user-written program (with embedded values) into a TokenStream

Derivable Traits

```
#[derive(Clone, Debug)]
struct Matrix<T> {
    shape: (usize, usize),
    data: Vec<T>,
}
```

They can also be used to implement derivable traits

This requires the `#[proc_macro_derive(...)]` attribute

demo

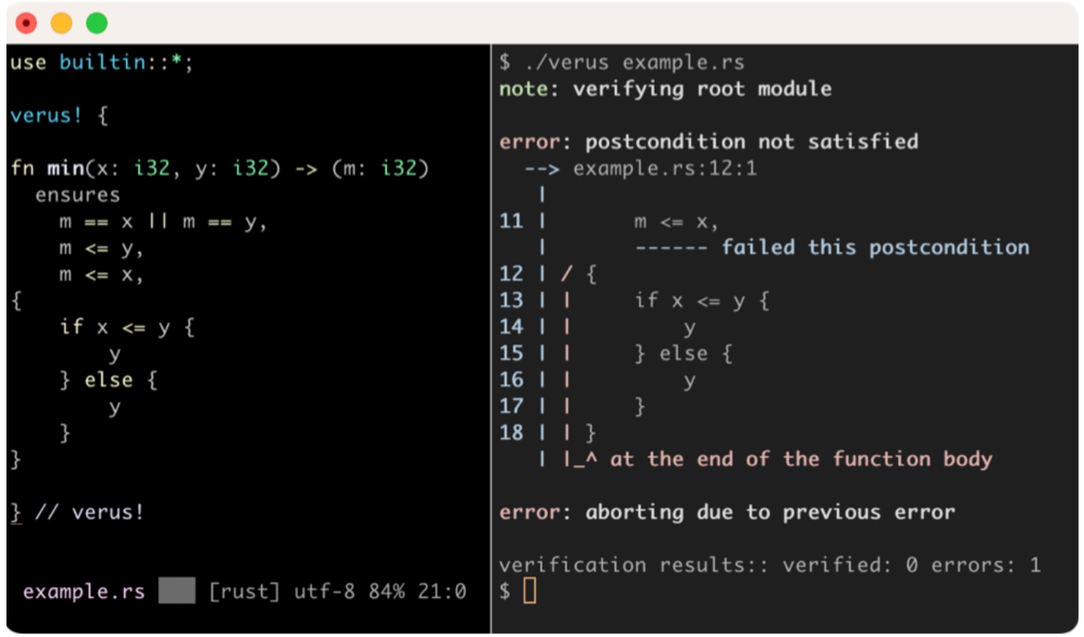
(functors, simplified)

Use Cases

```
let alice_cal: HashMap<String, bool> = /* { "Monday" -> true, ... } */
let bob_cal:   HashMap<String, bool> = /* { "Monday" -> false, ...} */
let mut count = 0;
for (day, available) in alice_cal {
    if available && *bob_cal.get(&day).unwrap() {
        count += 1;
    }
}
println!("Overlapping days: {}", count);
```

Fig. 1. Rust code that computes the overlapping number of days in two calendars.

```
let alice_cal: HashMap<String, Secret<lat::A,bool>> = /* { "Monday" -> Secret(true), ... } */
let bob_cal:   HashMap<String, Secret<lat::B,bool>> = /* { "Monday" -> Secret(false), ...} */
let mut count = secret_block!(lat::AB { wrap_secret(0) });
for (day, available) in alice_cal {
    secret_block!(lat::AB {
        if unwrap_secret(available) &&
           *unwrap_secret_ref(::std::option::Option::unwrap(
               ::std::collections::HashMap::get(&bob_cal, &day))) {
            *unwrap_secret_mut_ref(&mut count) += 1;
        }
    });
}
println!("Overlapping days: {}", count.declassify());
```



Procedural macros as useful for larger-scale generation of code

Two interesting examples: **Cocoon** and **Verus**

Workshop Task

```
ocaml_adt! {
  type Foo =
    | Bar of i32 * i32
    | Baz
    | Buzz of i32
}
```

Write a declarative macro that allows you to write a (non-recursive) enumeration using OCaml syntax

Challenge: Write a procedural macro, and handle recursion