

# Macros

**Rust, In Theory and in Practice**

# 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

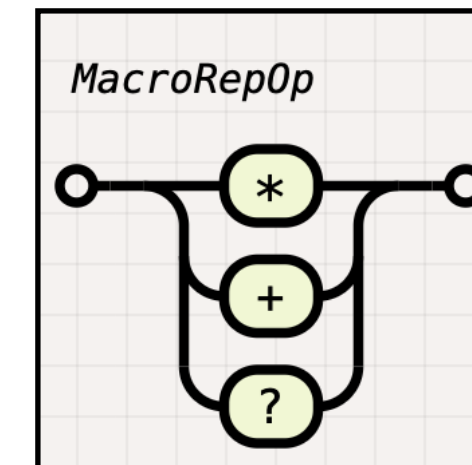
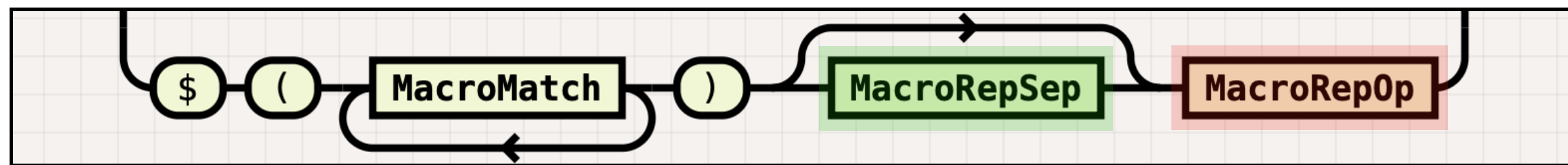
```
#[macro_export]  
macro_rules! vec {  
    ( $( $x:expr ),* ) => { ... }
```

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

```
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 `TokenStreams` 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

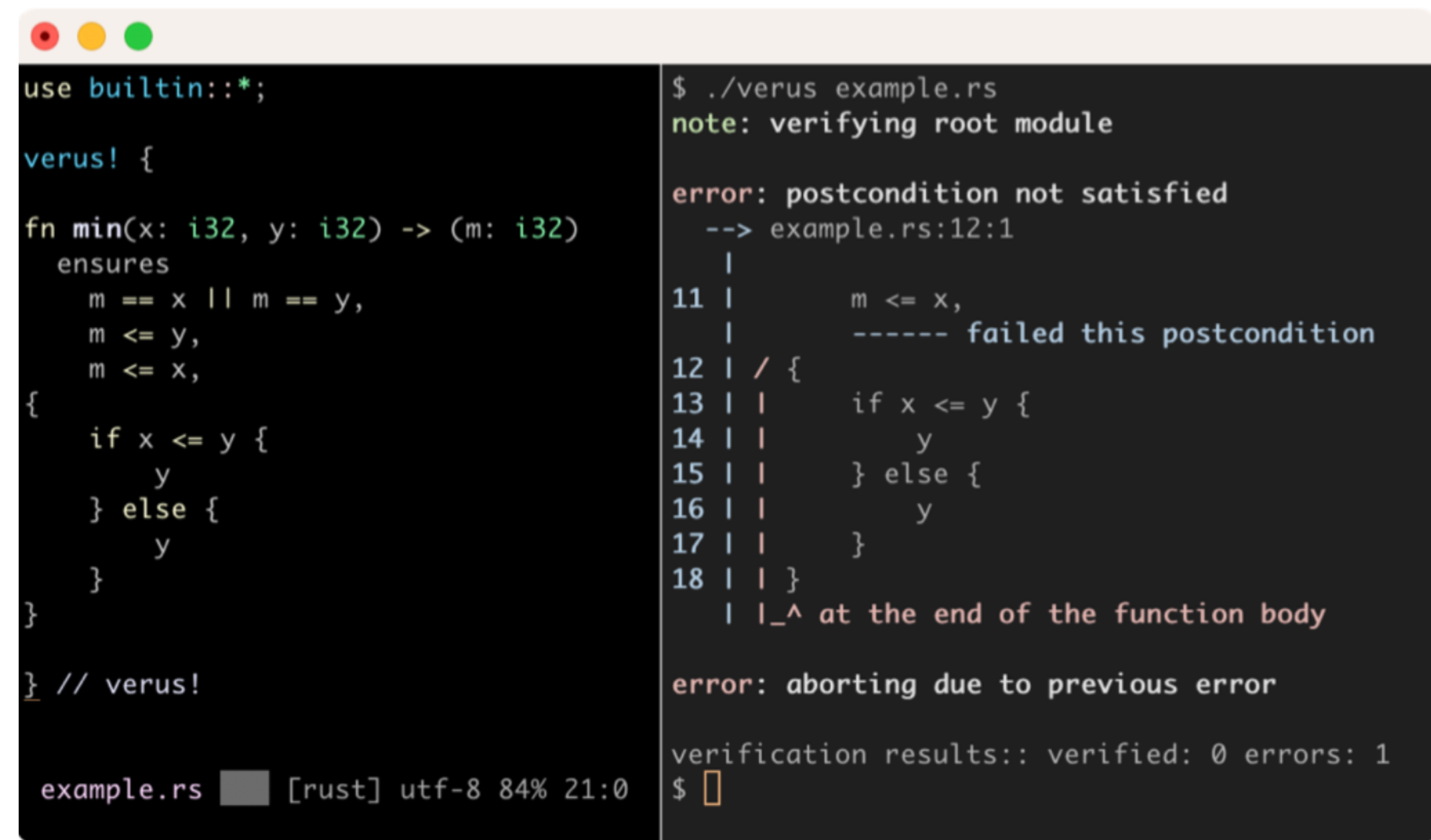
(functors, simplified)

# 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());
```



The screenshot shows a code editor with a dark theme. On the left, there is Rust code for a function `min` that takes two integers `x` and `y` and returns the minimum. The code includes an `ensures` clause with a postcondition. On the right, the output of the Verus verifier is shown. It indicates that the postcondition was not satisfied at line 12, column 1, and provides a detailed trace of the execution. The error message states: "error: postcondition not satisfied --> example.rs:12:1". The trace shows the execution of the `min` function, with the postcondition failing at line 12. The final output shows "verification results:: verified: 0 errors: 1".

```
use builtin::*;

verus! {

fn min(x: i32, y: i32) -> (m: i32)
  ensures
    m == x || m == y,
    m <= y,
    m <= x,
  {
    if x <= y {
      y
    } else {
      y
    }
  }
} // verus!

example.rs [rust] utf-8 84% 21:0

$ ./verus example.rs
note: verifying root module

error: postcondition not satisfied
--> example.rs:12:1
|
11 |         m <= x,
|         ----- failed this postcondition
12 | / {
13 | |     if x <= y {
14 | |         y
15 | |     } else {
16 | |         y
17 | |     }
18 | | }
| |_^ at the end of the function body

error: aborting due to previous error

verification results:: verified: 0 errors: 1
$
```

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