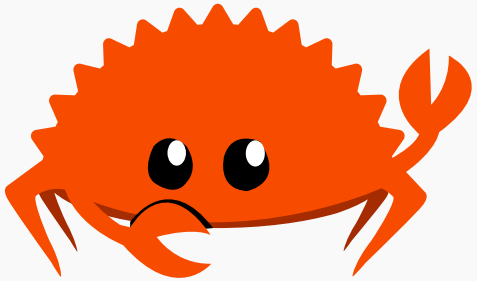
Rust

A boring and expressive language

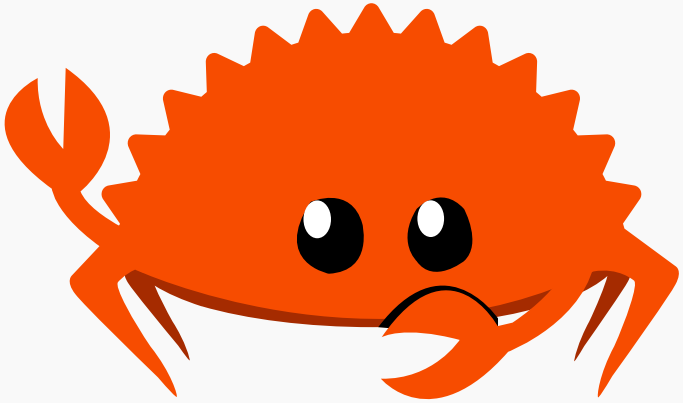
Victor Diez Ruiz



```
1  fn main() {
2    println!("Hello 🦀");
3  }
```

Why Rust rocks

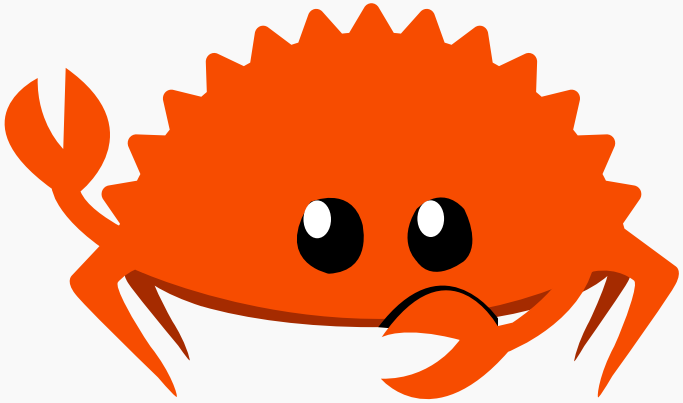
1. Lifetimes & Ownership
2. Inmutability by default
3. Algebraic Data Types
4. Error handling
5. Pattern Matching
6. Traits
7. Macros
8. Ecosystem



Lifetimes & Ownership

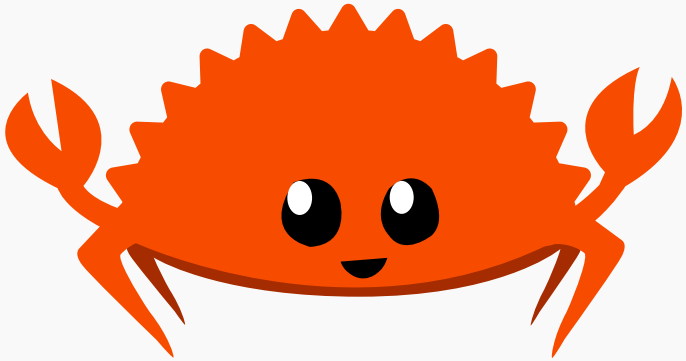
Save the environment

```
1  fn main() {
2      let a = 2;
3      let b = 3;
4      println!("{}", a + b);
5  }
```

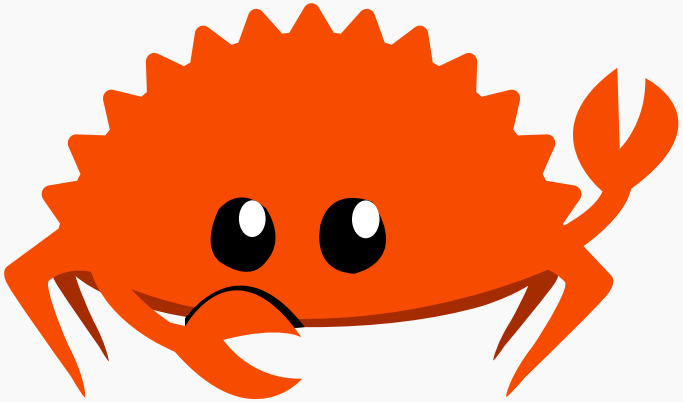


Save the environment

```
1  fn main() { <scope>
2      let a = 2;
3      let b = 3;
4      println!("{}", a + b);
5  } </scope>
```

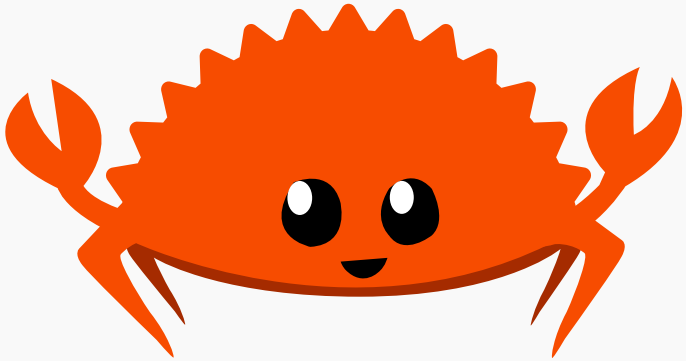


Everything eventually dies



```
1  fn main() {
2    let a = 2;
3    {
4      let b = 3;
5    }
6    println!("{}", a + b);
7  }
```

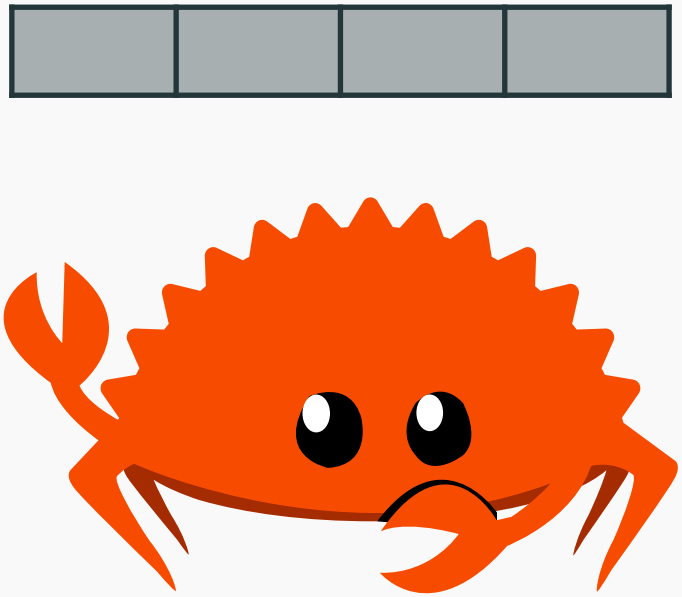
Everything eventually dies



```
1  fn main() { <'a>
2      let a = 2;
3      { <'b>
4          let b = 3;
5      } </'b>
6      println!("{}", a + b);
7  } </'a>
```

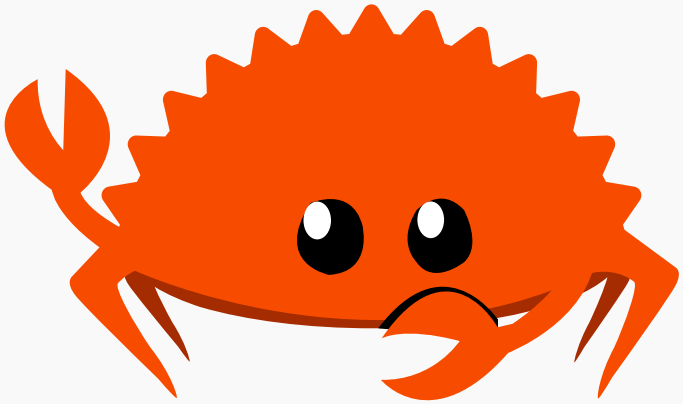
Sir that’s mine

```
1  fn main() {
2    let a: 'a = 2;
3    {
4      let b: 'b = 3;
5    }
6    println!("{}", a + b);
7  }
```



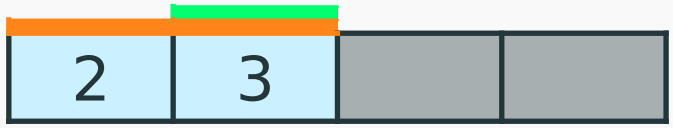
Sir that’s mine

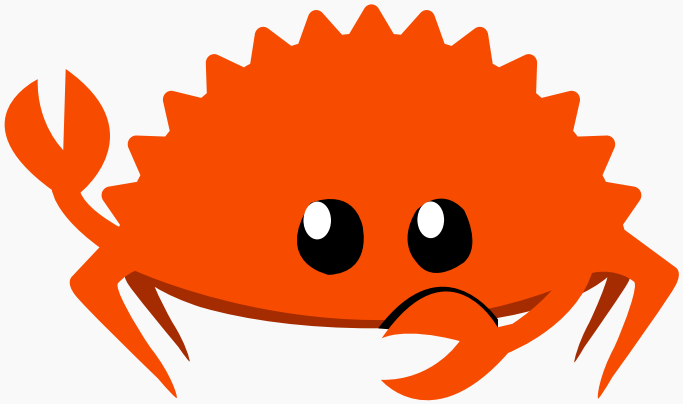
```
1  fn main() { <'a>
2      let a: 'a = 2;
3      {
4          let b: 'b = 3;
5      }
6      println!("{}", a + b);
7  }
```



Sir that’s mine

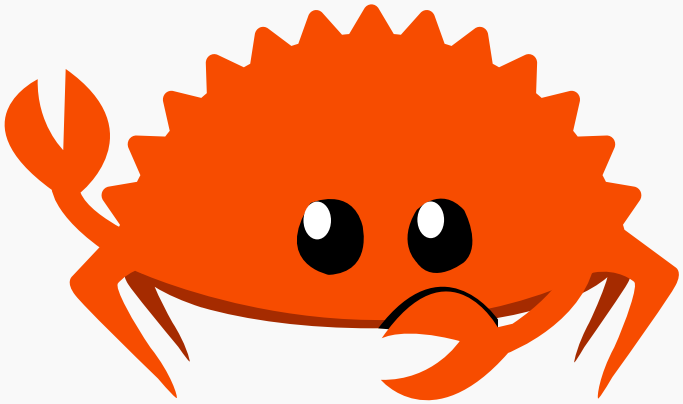
```
1  fn main() { <'a>
2      let a: 'a = 2;
3      { <'b>
4          let b: 'b = 3;
5      }
6      println!("{}", a + b);
7  }
```





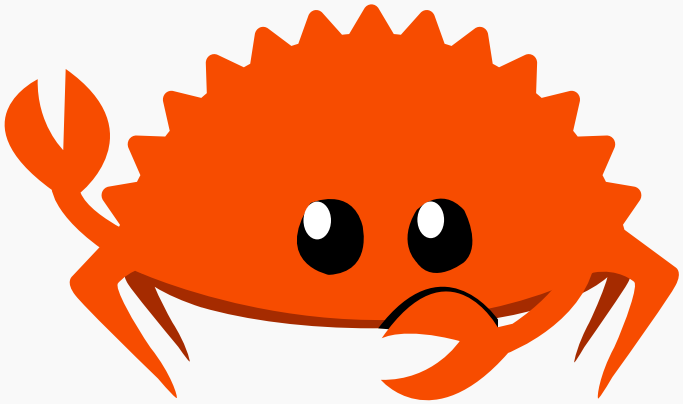
Sir that’s mine

```
1  fn main() { <'a>
2      let a: 'a = 2;
3      { <'b>
4          let b: 'b = 3;
5      } </'b>
6      println!("{}", a + b);
7  }
```



Sir that’s mine

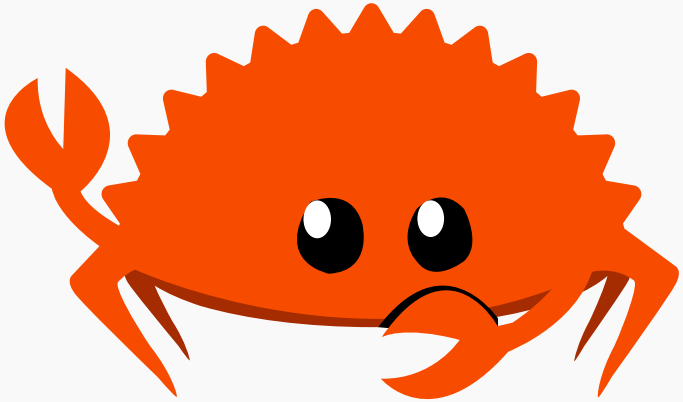
```
1  fn main() { <'a>
2      let a: 'a = 2;
3      { <'b>
4          let b: 'b = 3;
5      } </'b>
6      println!("{}", a + b);
7  } </'a>
```



Inmutability by default

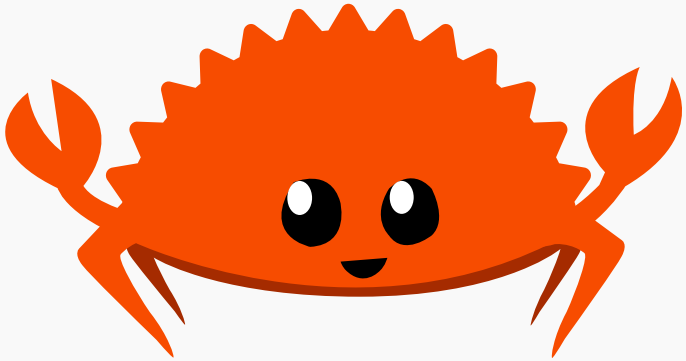
Can’t touch this

```
1  fn main() {
2    let a = 2;
3    let mut b = 3;
4
5    a = 3;
6    b = 2;
7  }
```



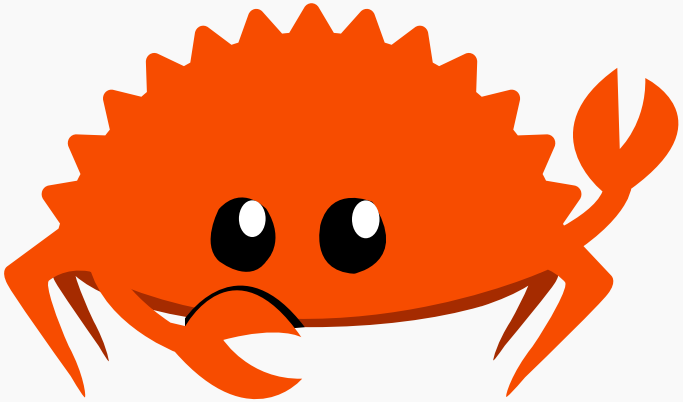
Can’t touch this

```
1  fn main() {
2      let a = 2;
3      let mut b = 3;
4
5      a = 3;
6      b = 2;
7  }
```



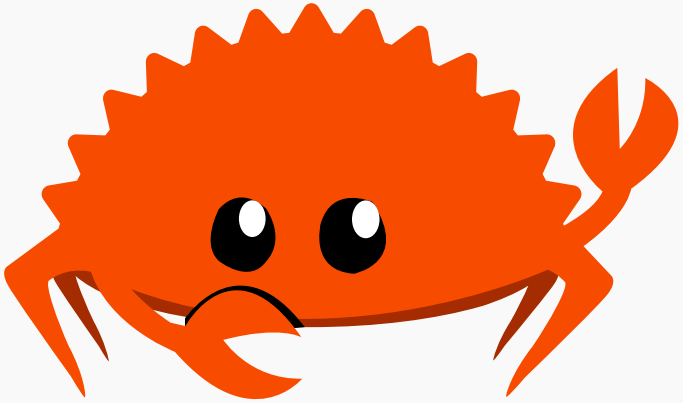
Algebraic Data Types

Types as numbers ?!?!?!



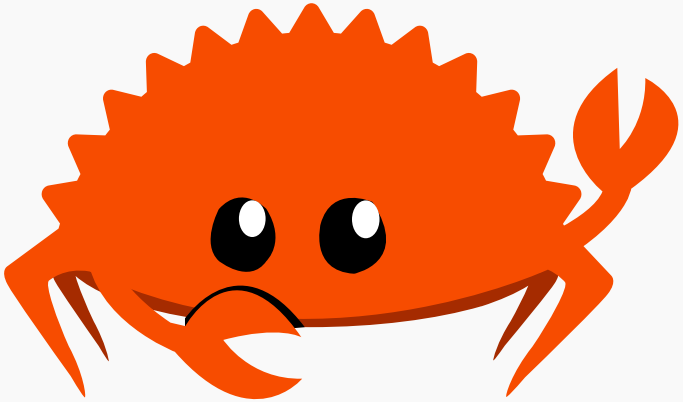
```
bool : { true, false } = 2
```

Types as numbers ?!?!?!



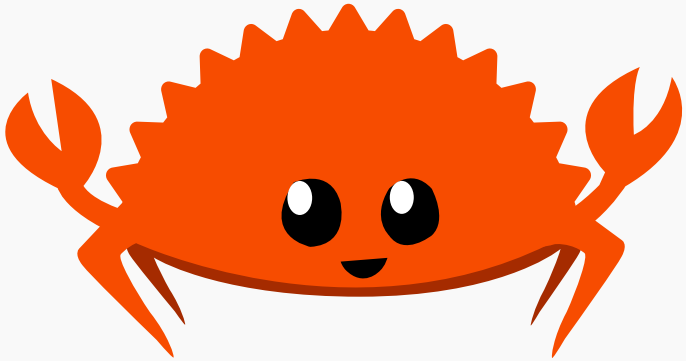
```
bool : { true, false } = 2
u8   : { 0, ..., 255 } = 256
```

Types as numbers ?!?!?!



```
unit : { ()           } = 1
bool : { true, false } = 2
u8   : { 0, ..., 255 } = 256
```

Types as numbers ?!?!?!

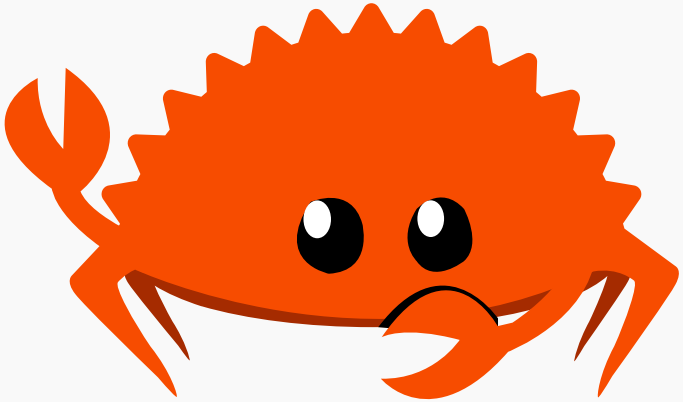


```
!    : {              } = 0
unit : { ()           } = 1
bool : { true, false  } = 2
u8   : { 0, ..., 255  } = 256
```

Math with types ?!?!

Addition

```
bool{2} + unit{1} = {3}
```

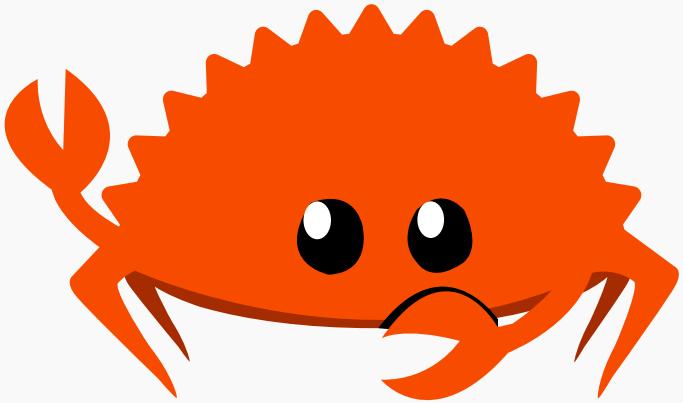


Math with types ?!?!

Addition

`bool{2} + unit{1} = {3}`

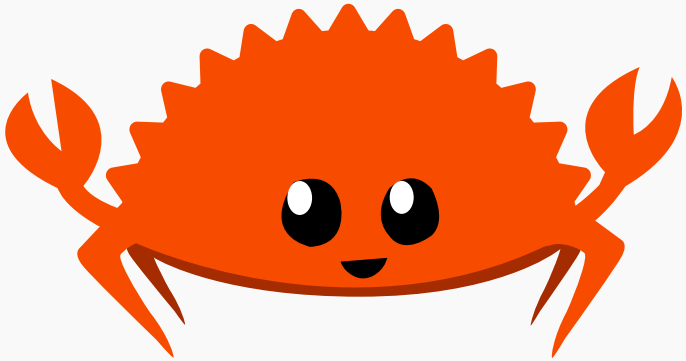
```
1  enum MaybeBool {
2    Some(bool),
3    None
4  }
```



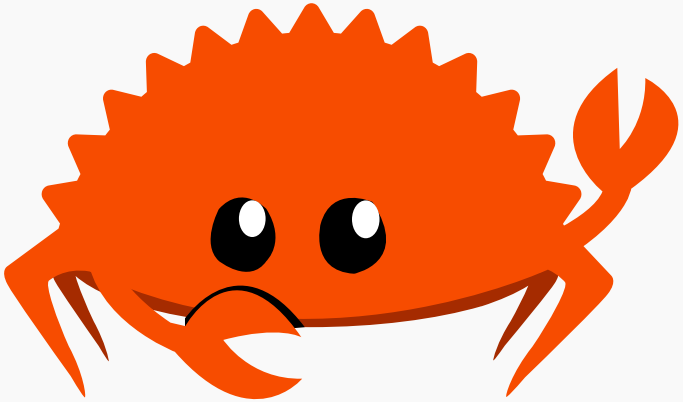
Math with types ?!?!

```
unit{1} * 4 = {4}
```

```
1 enum Directions {
2   North,
3   East,
4   West,
5   South,
6 }
```



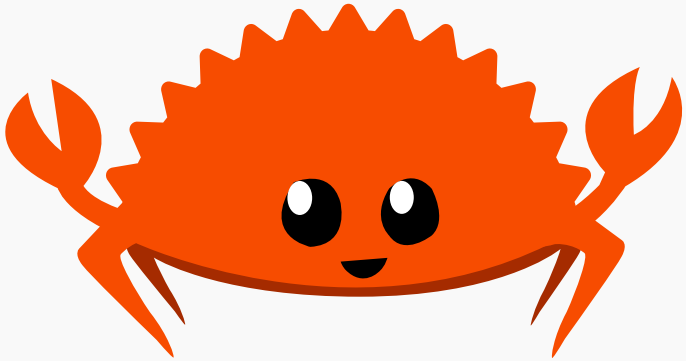
More math with types ?!



Multiplication

```
bool{2} * Direction{4} = {8}
```

More math with types ?!



Multiplication

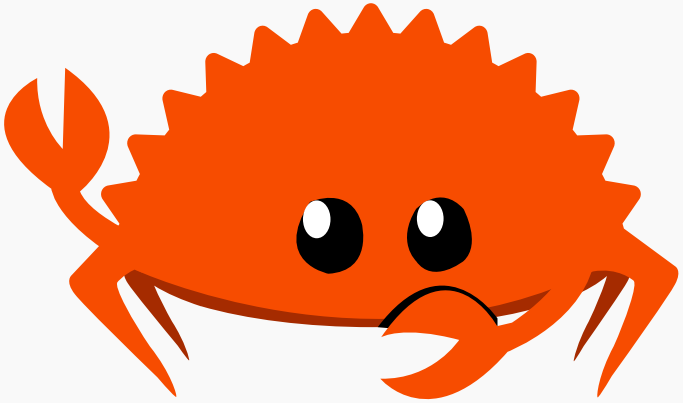
`bool{2} * Direction{4} = {8}`

```
1  struct Robot {
2    lastDir: Direction,
3    enabled: bool
4  }
```

What the **** is $\text{bool}^{\text{bool}}$?!

Exponentiation

```
bool{2} ^ bool{2} = {4}
```

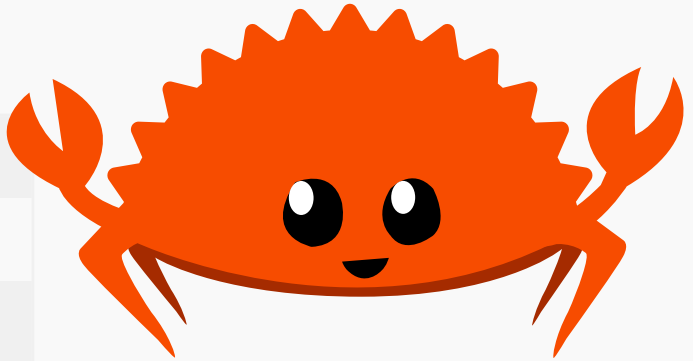


What the **** is $\text{bool}^{\text{bool}}$?!

Exponentiation

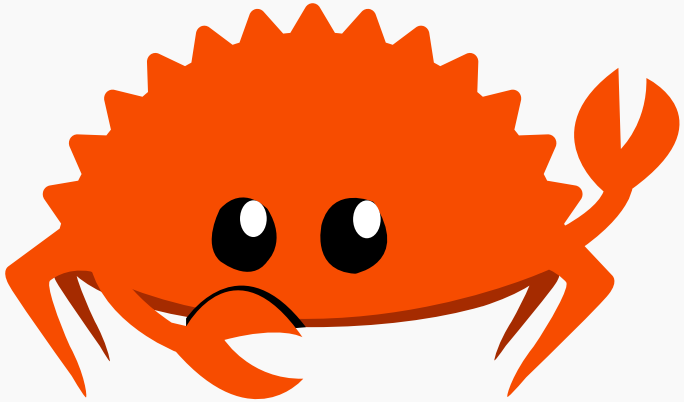
```
bool{2} ^ bool{2} = {4}
```

```
1  fn id   (v: bool) → bool { v     }
2  fn not  (v: bool) → bool { !v    }
3  fn true (_: bool) → bool { true  }
4  fn false(_: bool) → bool { false }
```



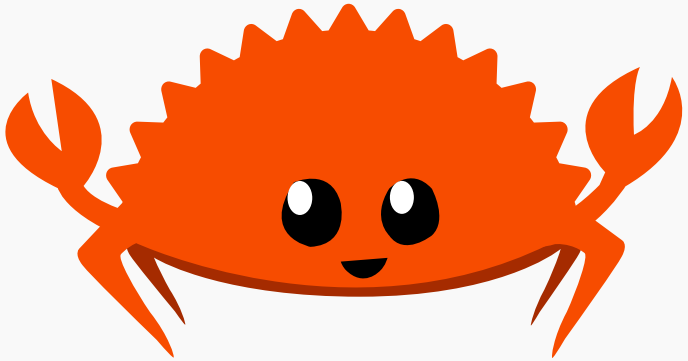
Error handling

I don’t like exceptions



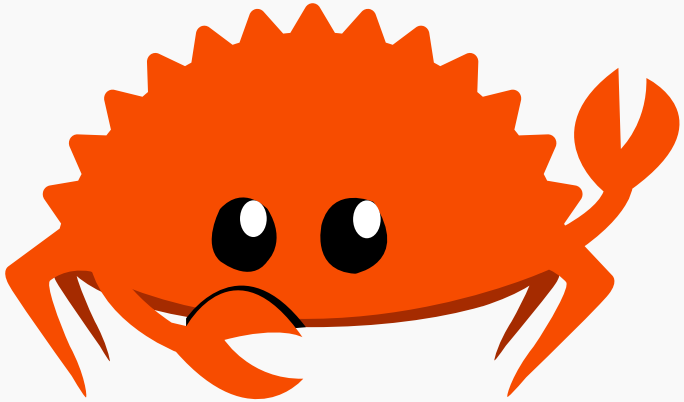
```
1  fn try_parse(input: String)
2     → Option<Phone>;
```

I don’t like exceptions



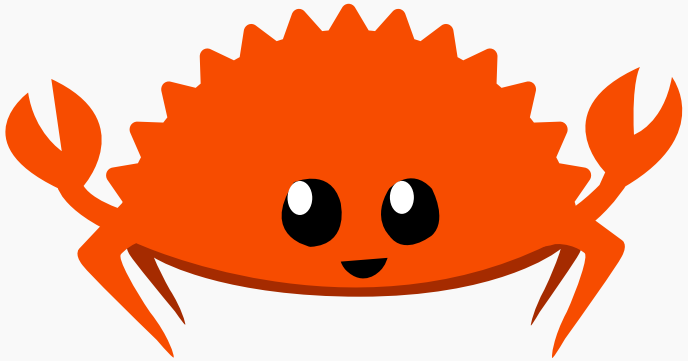
```
1  enum Option<T> {
2    Some(T),
3    None
4  }
```

Without exception



```
1  fn try_parse(input: String)
2     → Result<Phone, ParseError>;
```

Without exception

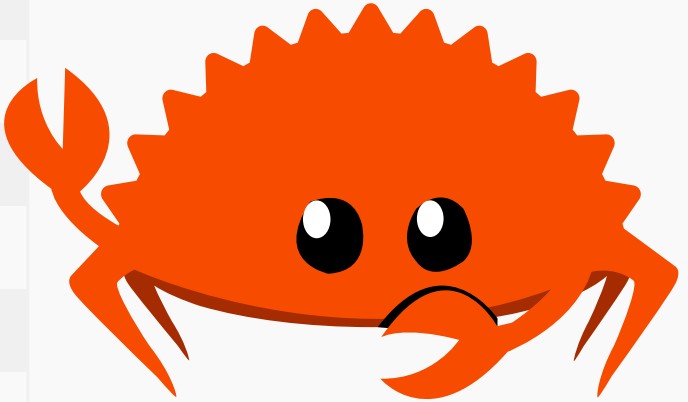


```
enum Result<T, E> {
  Ok(T),
  Err(E)
}
```

Pattern Matching

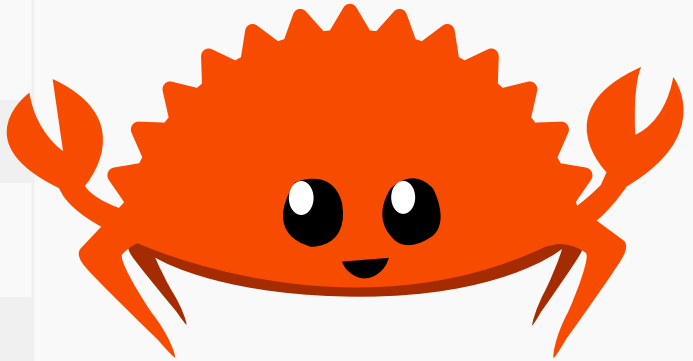
Not like tinder

```
1 match result {
2   Ok(phone) →
3     println!("tlf: {}", phone),
4   Err(cause) →
5     println!("error: {}", cause)
6 }
```

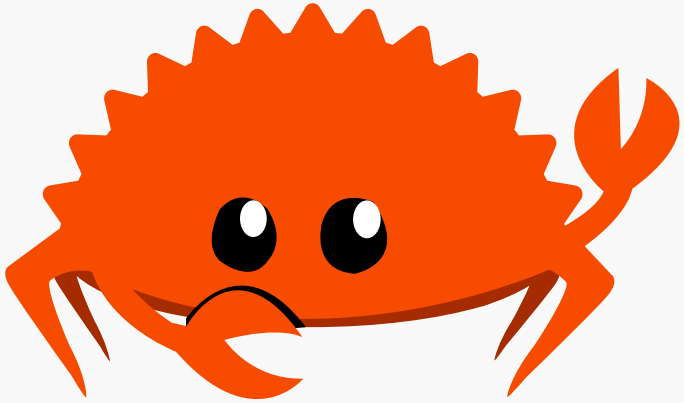


Not like tinder

```
1 match result {
2   Ok(phone) →
3     println!("tlf: {}", phone),
4   Err(cause) →
5     println!("error: {}", cause)
6 }
```

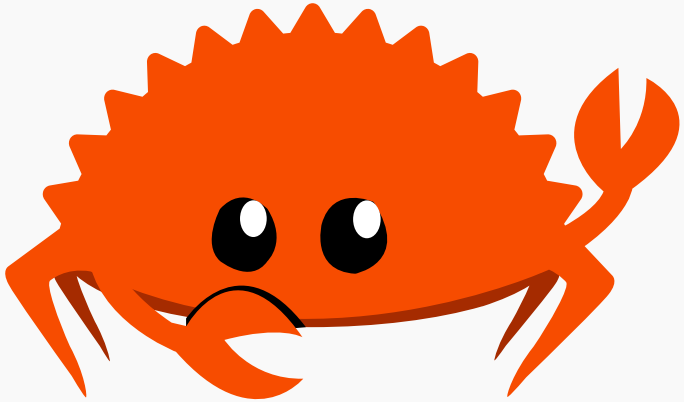


Why stop there?



```
let Ok(theme) = get_theme()
  else { return CannotGetTheme; };
```

Why stop there?

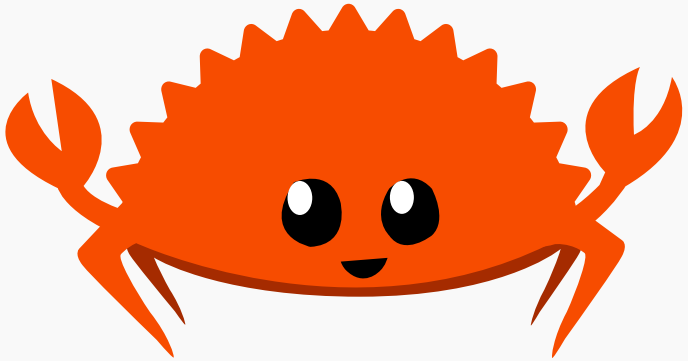


```
let Ok(theme) = get_theme()
  else { return CannotGetTheme; };
```

↓

```
let theme = get_theme()?;
```

Why stop there?

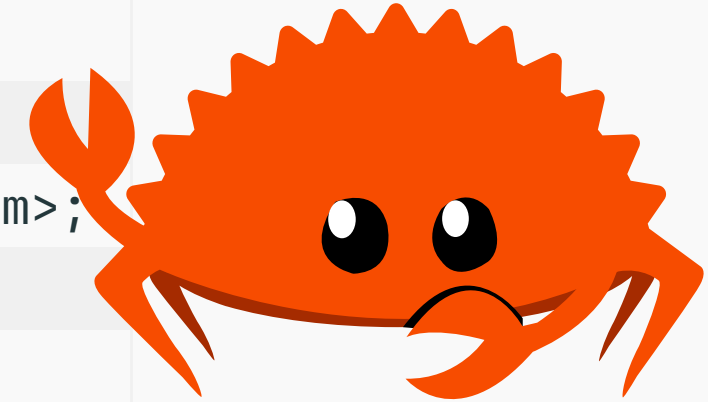


```
while let Some(e) = iter.next()
{
    ...
}
```

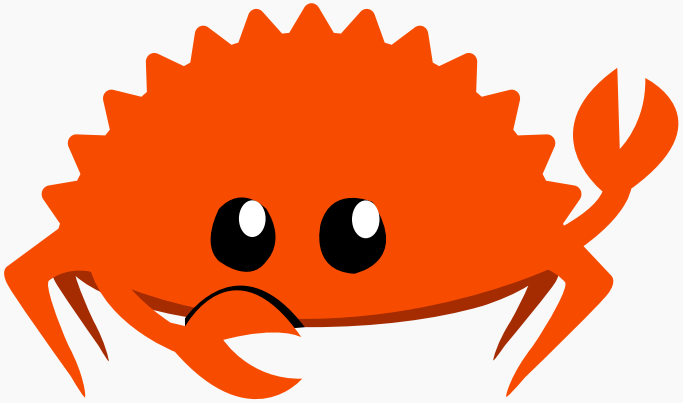
Traits

A good way to *interface* with other code

```
1  trait Iterator {
2    type Item;
3
4    fn next(&mut self) → Option<Self::Item>;
5    fn count(self) → usize
6      where Self: Sized { ... }
7  }
```

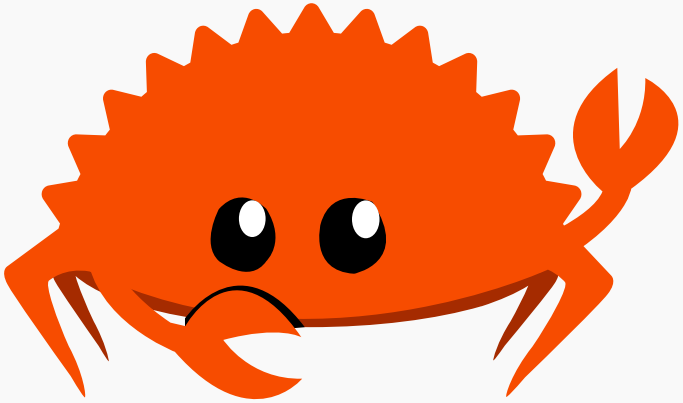


Usefulness is also a trait



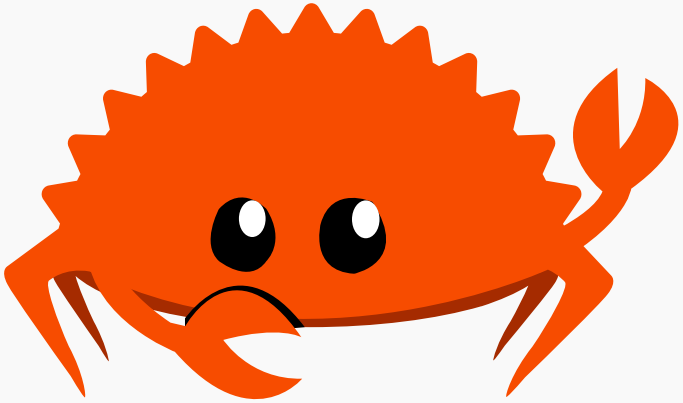
Debug

Usefulness is also a trait



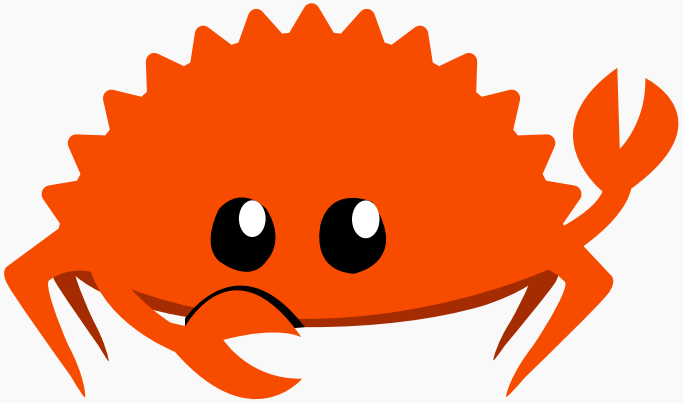
Debug
Default

Usefulness is also a trait



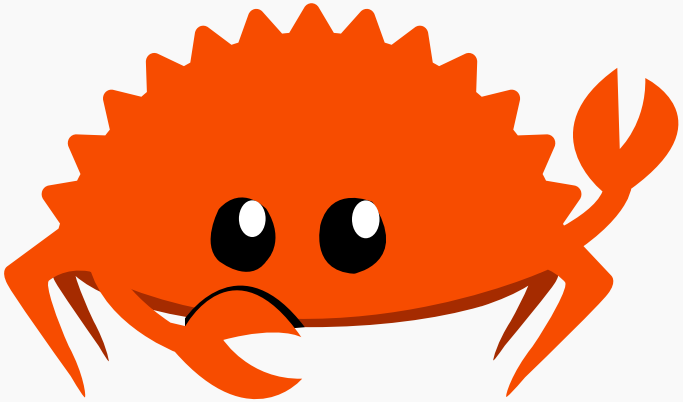
Debug
Default
Clone, Copy

Usefulness is also a trait



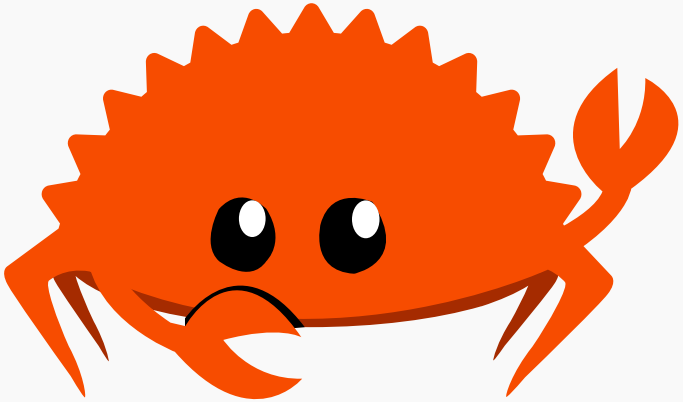
Debug
Default
Clone, Copy
std::ops::*

Usefulness is also a trait



Debug
Default
Clone, Copy
std::ops::*
Iterator

Usefulness is also a trait

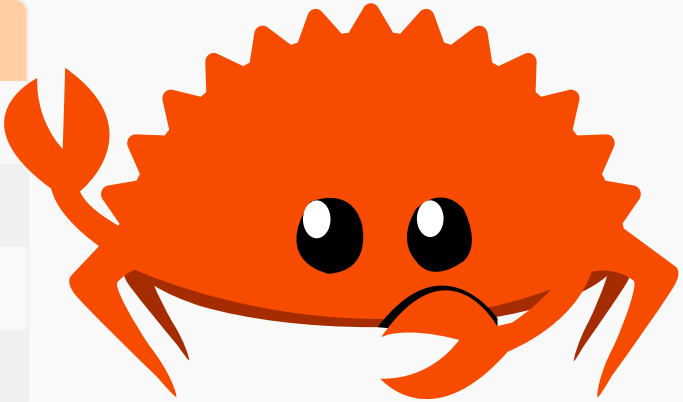


Debug
Default
Clone, Copy
std::ops::*
Iterator
Drop

Macros

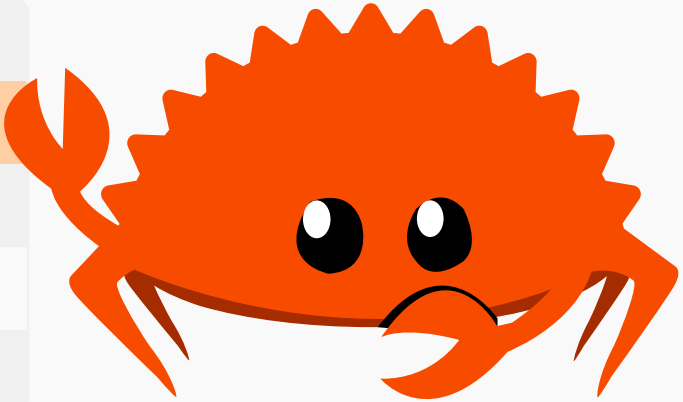
Python in Rust !?!?

```
println!("Im a macro!");
let pos = vec![1,2,3];
assert!(true, "Me too");
panic!("BOOM!");
todo!("Too lazy to finish");
```



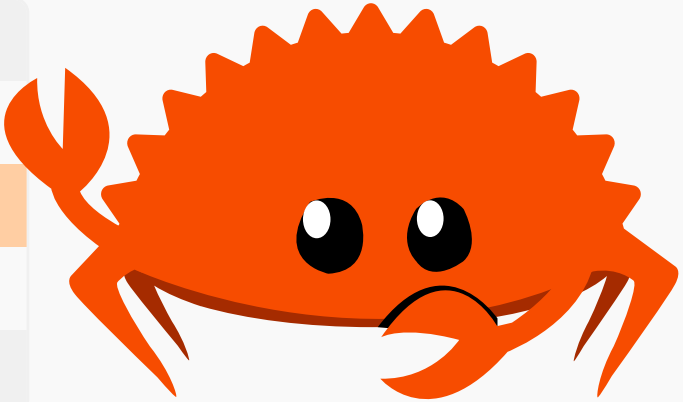
Python in Rust !?!?

```
println!("Im a macro!");
let pos = vec![1,2,3];
assert!(true, "Me too");
panic!("BOOM!");
todo!("Too lazy to finish");
```



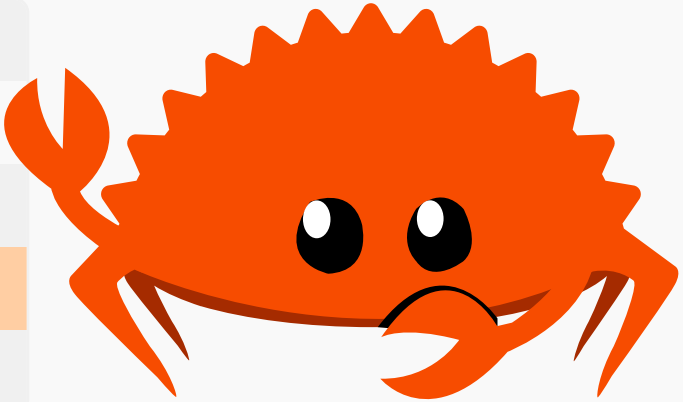
Python in Rust !?!?

```rust
println!("Im a macro!");
let pos = vec![1,2,3];
assert!(true, "Me too");
panic!("BOOM!");
todo!("Too lazy to finish");
```



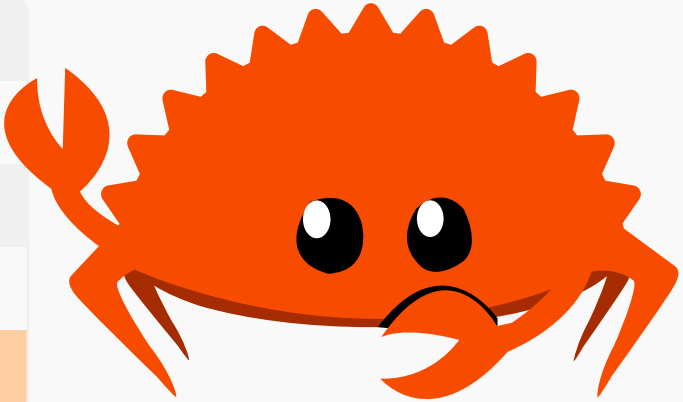
Python in Rust !?!?

```rust
println!("Im a macro!");
let pos = vec![1,2,3];
assert!(true, "Me too");
panic!("BOOM!");
todo!("Too lazy to finish");
```

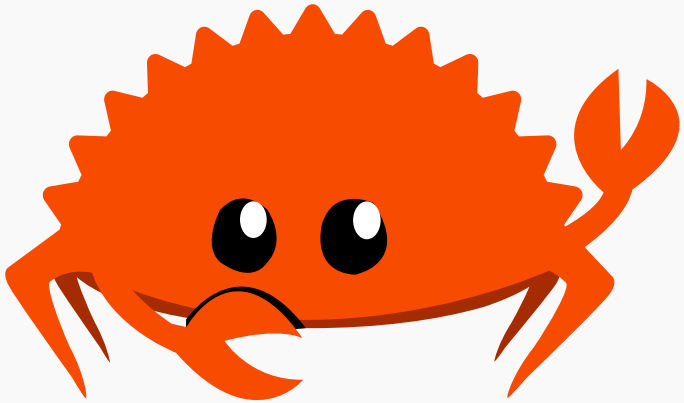


Python in Rust !?!?

```
println!("Im a macro!");
let pos = vec![1,2,3];
assert!(true, "Me too");
panic!("BOOM!");
todo!("Too lazy to finish");
```



Write macros with macros!



```
1  macro_rules! debug_print {
2    ($expression: expr) ⇒ {
3      println!("{} = {:?}",
4        stringify!($expression),
5        $expression
6      );
7    }
8  }
```

```
1  debug_print!(3 + 3);
```

As easy as `#[derive(Debug)]`

Ecosystem

Bateries mostly included!

Barman bring me my wine

You don’t need docs if you can read code

Plant a tree, have a son, write a book

Something very important