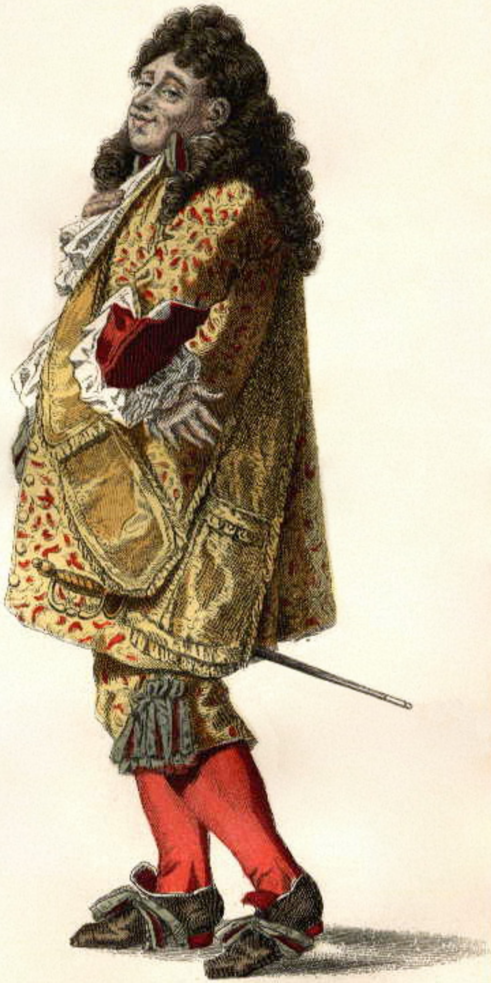
FUNCTIONAL GUERILLA IN THE LAND OF RUST

Tomasz Barański
@JustTomo



LE BOURGEOIS GENTILHOMME.

M. JOURDAIN.

Suivez-moi, que j'aille un peu montrer mon habit par la ville,
Acte III. sc. 1.

RUST?

Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.

https://www.rust-lang.org/en-US/

> *Rust is a* ***systems programming*** *language that runs blazingly fast, prevents segfaults, and guarantees thread safety.*

https://www.rust-lang.org/en-US/

```
fn main() {
    for n in 1..101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", n);
        }
    }
}
```

Where is the

FUNCTIONAL PROGRAMMING

in that?

FUNCTIONS

FIRST-CLASS

```
let add5 = |x| x + 5;

let result: i32 = add5(10);
```

```
fn adder(x: i32) -> Box<Fn(i32) -> i32>
{
    Box::new(move |y| {x + y})
}
```

```rust
fn adder(x: i32) -> impl Fn(i32) -> i32 {
    move |y| x + y
}
```

```rust
fn apply_to(x: i32, fun: &Fn(i32) -> i32) -> i32 {
    fun(x)
}

let x = apply_to(10, &adder(5));
let y = apply_to(10, &|x| x * 2);
```

```
fn apply_to(x: T, fun: &Fn(T) -> S) -> S {
    fun(x)
}

let x: i32 = apply_to(10, &|x| x + 10);
let y: i32 = applyt_to("Abc", &|x| x.len());
let z: &str = apply_to("Abc\ndef",
                       &|x| x.lines().nth(1).unwrap());
```

PURE

```
fn double_me(x: i64) -> i64 {
    x + x
}
```

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn move_by(&self, dx: i32, dy: i32) -> Point {
        /* ... */
    }
}
```

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn move_by_mut(&mut self, dx: i32, dy: i32) {
        /* ... */
    }
}
```

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn move_by(&self, dx: i32, dy: i32) -> Point {
        Point { x: self.x + dx, y: self.y + dy }
    }
}
```

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn move_by_mut(&mut self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }
}
```

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn move_by_mut(&self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }
}
```

```
Compiling fp-in-rust-code v0.1.0
error: cannot assign to immutable field `self.x`
  --> src/main.rs:27:9
   |
27 |         self.x += dx;
   |         ^^^^^^^^^^^^
```

```rust
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    fn move_by(&self, dx: i32, dy: i32) -> Point {
        println!("Changing the world, one println! at a time");
        Point { x: self.x + dx, y: self.y + dy }
    }
}
```

```
struct Point {
    x: i32,
    y: i32,
}

static mut COUNTER: i32 = 0;

impl Point {
    fn move_by(&self, dx: i32, dy: i32) -> Point {
        COUNTER += 1;
        Point { x: self.x + dx, y: self.y + dy }
    }
}
```

```
   Compiling fp-in-rust-code v0.1.0
error[E0133]: use of mutable static requires unsafe function or b
  --> src/main.rs:23:9
   |
23 |         COUNTER += 1;
   |         ^^^^^^^ use of mutable static
```

```
struct Point {
    x: i32,
    y: i32,
}

static mut COUNTER: i32 = 0;

impl Point {
    fn move_by(&self, dx: i32, dy: i32) -> Point {
        unsafe { COUNTER += 1; }
        Point { x: self.x + dx, y: self.y + dy }
    }
}
```

ALGEBRAIC DATA TYPES

```
enum Bool {
    False,
    True
}

let is_it: Bool = Bool::True;
```

```
enum Shape {
    Circle(f32, f32, f32),
    Rectangle(f32, f32, f32, f32)
}

let circle = Shape::Circle(1.0, 2.0, 3.0);
```

```
fn surface(shape: &Shape) -> f32 {
    use Shape::*;

    match *shape {
        Circle(_, _, r) =>
            std::f32::consts::PI * r * r,

        Rectangle(x1, y1, x2, y2) =>
            (x2 - x1).abs() * (y2 - y1).abs()
    }
}

let x = surface(&circle);
```

```
data Maybe a = Nothing | Just a
```

```
enum Option<T> {
    None,
    Some(T),
}
```

```
let this = Some("This");
let that: Option<i32> = Some(10);
let other = None;
```

```
#[derive(Debug)]
struct Car<T, S, V> {
    company: T,
    model: S,
    year: V
}

let car = Car {company: "Ford", model: "Mustang", year: 1967};
println!("The car is {:?}", car);
```

```
The car is Car { company: "Ford", model: "Mustang", year: 1967 }
```

```
impl<T, S, V> Car<T, S, V> {
    fn show(&self) -> String
    {
        format!("This {} {} was made in {}.",
                self.company, self.model, self.year)
    }
}
```

```rust
use std::fmt::Display;

impl<T, S, V> Car<T, S, V> {
    fn show(&self) -> String
        where T: Display,
              S: Display,
              V: Display
    {
        format!("This {} {} was made in {}.",
                self.company, self.model, self.year)
    }
}

println!("{}", car.show());
```

TRAITS

AKA TYPECLASSES

```
data Color = Red | Green | Blue

let c1 = Red
    c2 = Green
in c1 == c2
```

```haskell
class  Eq a  where
    (==), (/=)           :: a -> a -> Bool

    x /= y               = not (x == y)
    x == y               = not (x /= y)
```

```rust
enum Color {
    Red,
    Green,
    Blue
}

let c1 = Color::Blue;
let c2 = Color::Red;

if c1 == c2 { /*...*/ }
```

```rust
trait Eq<R = Self> {
    fn eq(&self, other: &R) -> bool { !self.ne(other) }
    fn ne(&self, other: &R) -> bool { !self.eq(other) }
}
```

```
impl Eq for Color {
    fn eq(&self, other: &Color) -> bool {
        use Color::*;

        match (self, other) {
            (&Red, &Red)     => true,
            (&Green, &Green) => true,
            (&Blue, &Blue)   => true,
            _                => false
        }
    }
}
```

```rust
#[derive(Eq)]
enum Color {
    Red,
    Green,
    Blue
}
```

PartialEq, Eq

PartialOrd, Ord

Clone, Copy

Add, AddAssign, Sub, SubAssign, etc

Iterator

Fn, FnMut, FnOnce

Index

Default

Drop

Send, Sync

```rust
#[macro_use]
extern crate hello_world_derive;

trait HelloWorld {
    fn hello_world();
}

#[derive(HelloWorld)]
struct FrenchToast;

fn main() {
    FrenchToast::hello_world();
}
```

MACROS

vec!

try!

println!

panic!

FUNCTION COMPOSITION

WITH MACROS

```rust
let l1: Vec<i32> = vec![5,-3,-6,7,-3,2,-19,24];

let l2: Vec<i32> = l1.iter()
    .map(|&x| x.abs().neg())
    .collect();
```

```
let l1: Vec<i32> = vec![5,-3,-6,7,-3,2,-19,24];

let l2: Vec<i32> = l1.iter()
    .map(|&x| x.abs().neg())
    .collect();
```

```
macro_rules! cm {
    ($f:ident . $g:ident) => (|x| x.$g().$f())
}
```

```
let l3: Vec<i32> = l1.iter()
    .map(cm!(neg . abs))
    .collect();
```

```
let l3: Vec<i32> = l1.iter().map(|x| x.abs().neg()).collect();
```

```
fn even(x: i32) -> bool {
    x % 2 == 0
}
```

```
macro_rules! c {
    ($f:expr , $g:expr) => (|&x| $f($g(x)));
}
```

```
let l4: Vec<bool> = l1.iter()
    .map(c!(even, i32::abs))
    .collect();
```

```
let l4: Vec<bool> = l1.iter()
    .map(|&x| even(i32::abs(x)))
    .collect();
```

```
let l3: Vec<i32> = l1.iter()
    .map(c!(Neg::neg, i32::abs))
    .collect();
```

```
macro_rules! c {
    ($f:expr , $g:expr) => (|&x| $f($g(x)));

    ($f:expr , $g:expr , $h:expr) => (|&x| $f($g($h(x))));
}
```

```rust
macro_rules! c {
    ($f:expr , $g:expr) => (|&x| $f($g(x)));

    ($f:expr , $g:expr , $($h:expr),+) => (
        |x| $f( c!($g, $( $h ),*) (x) )
    );
}
```

```rust
let l6: Vec<bool> = l1.iter()
    .map(c!(even, Neg::neg, i32::abs))
    .collect();
```

```
let l6: Vec<bool> = l1.iter()
    .map(c!(even, Neg::neg, i32::abs))
    .collect();
```

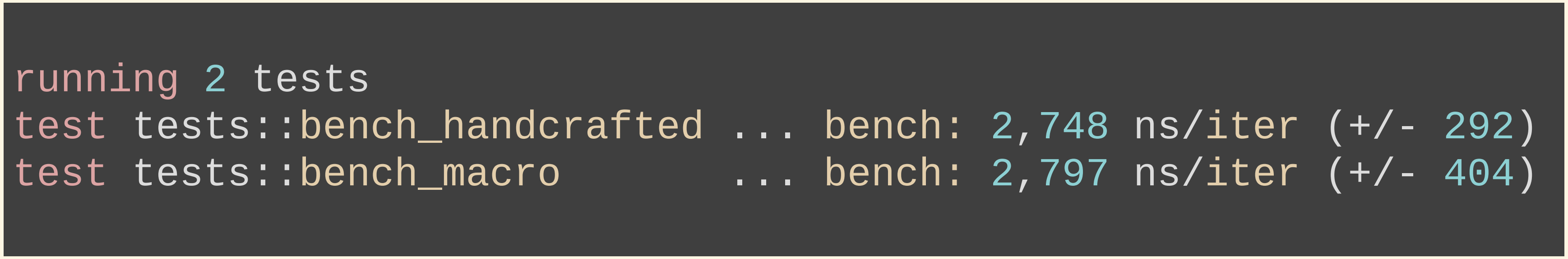
```
let l6: Vec<bool> = l1.iter()
    .map(|x| even( (|&x| Neg::neg(i32::abs(x))) (x) ) )
    .collect();
```

```
let l7: Vec<&i32> = l1.iter()
.filter(c!(even, |x| x as i32, i32::count_ones, Neg::neg, i32::ab
.collect();
```

```
let l7: Vec<&i32> =
  l1.iter().filter(|x|
    even((|x|
      (|x|
        x as
          i32)((|x|
            i32::count_ones((|&x|
              Neg::neg(i32::abs(x)))(x)))(x)))(x))).collect();
```

```
#[bench]
fn bench_macro(b: &mut Bencher) {
  b.iter(|| {
    let l1: Vec<i32> = (1..1_000).collect();
    let l7: Vec<&i32> = l1.iter()
      .filter(c!(even, |x| x as i32, i32::count_ones, Neg::neg, i
      .collect();
  });
}
```

```
#[bench]
fn bench_handcrafted(b: &mut Bencher) {
  b.iter(|| {
    let l1: Vec<i32> = (1..1_000).collect();
    let l7: Vec<&i32> = l1.iter()
      .filter(|x| even(x.abs().neg().count_ones() as i32))
      .collect();
  })
}
```



```
running 2 tests
test tests::bench_handcrafted ... bench: 2,748 ns/iter (+/- 292)
test tests::bench_macro       ... bench: 2,797 ns/iter (+/- 404)
```

SUMMARY

RUST

First-class functions

Pure enough

Algebraic Data Types

Traits

Macros

Excelent performance