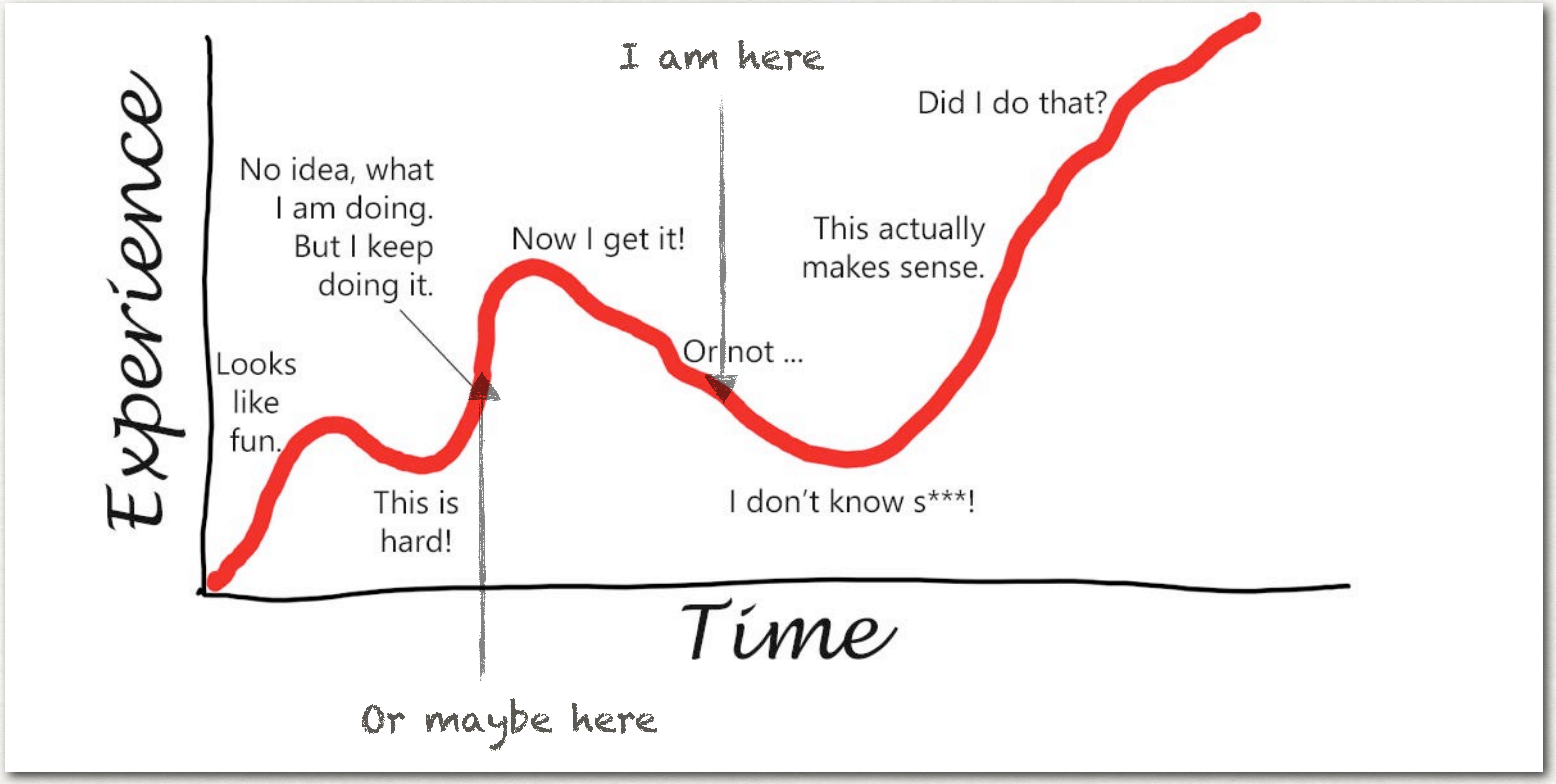
RUST, FOR CLJ DEVS

GOALS

- Talk about personal experience of using rust
- Highlight the rewarding and challenging parts
- What's similar, what's very different

CONTEXT

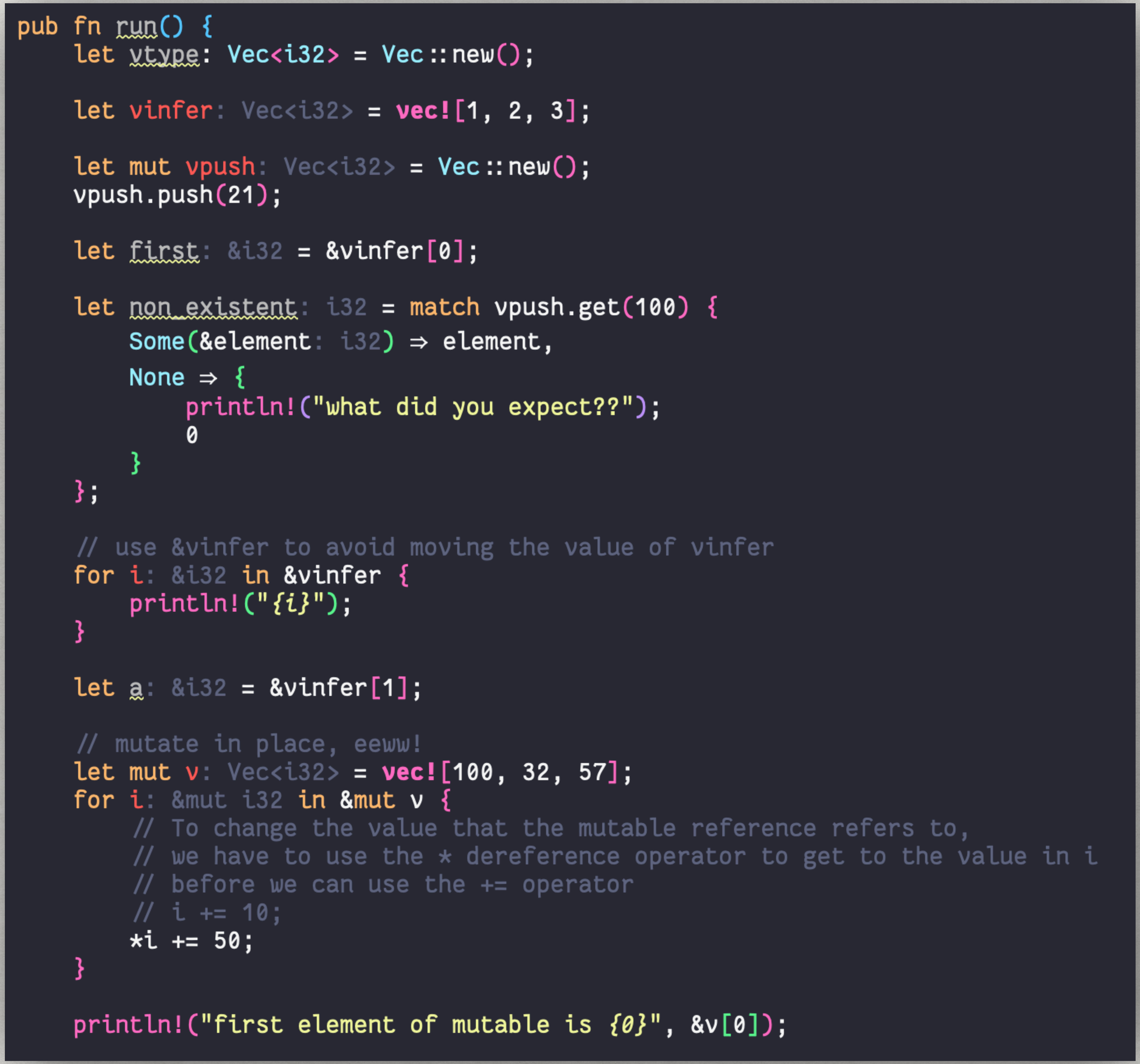

I am here
Experience
No idea, what I am doing. But I keep doing it.
Now I get it!
Did I do that?
This actually makes sense.
Looks like fun.
Or not ...
This is hard!
I don't know s***!
Time
Or maybe here

THE REWARDING BITS

- Dev experience
- Type system
- Enums
- Pattern matching
- Mutability
- Error handling
- Borrow checking

DEV LOOP

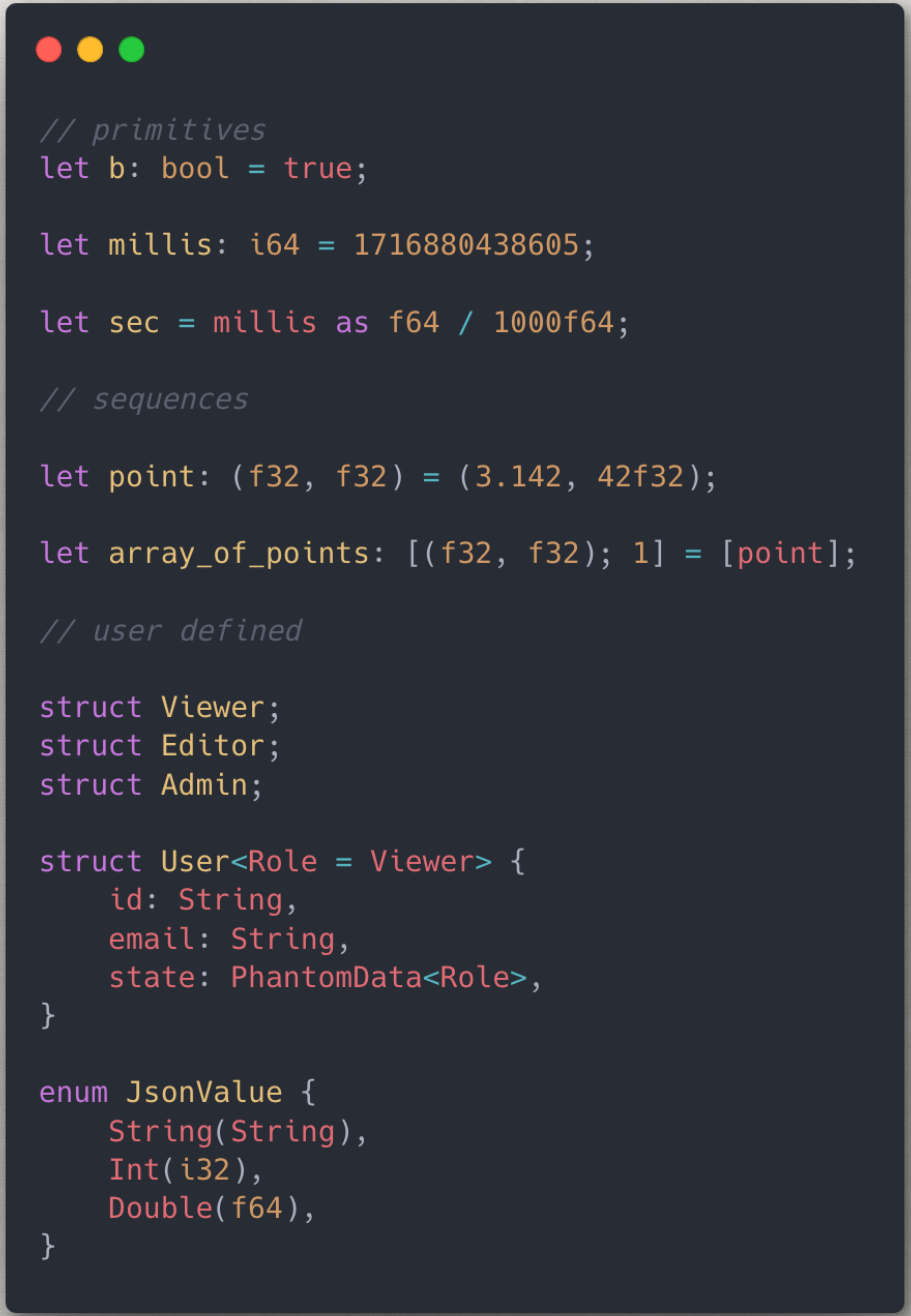
- Emacs
  - rust-mode
  - rust-analyser
  - flycheck-rust
- VSCode
  - rust-analyzer
  - Debugging support



```rust
pub fn run() {
    let vtype: Vec<i32> = Vec::new();

    let vinfer: Vec<i32> = vec![1, 2, 3];

    let mut vpush: Vec<i32> = Vec::new();
    vpush.push(21);

    let first: &i32 = &vinfer[0];

    let non_existent: i32 = match vpush.get(100) {
        Some(&element: i32) => element,
        None => {
            println!("what did you expect??");
            0
        }
    };

    // use &vinfer to avoid moving the value of vinfer
    for i: &i32 in &vinfer {
        println!("{i}");
    }

    let a: &i32 = &vinfer[1];

    // mutate in place, eeww!
    let mut v: Vec<i32> = vec![100, 32, 57];
    for i: &mut i32 in &mut v {
        // To change the value that the mutable reference refers to,
        // we have to use the * dereference operator to get to the value in i
        // before we can use the += operator
        // i += 10;
        *i += 50;
    }

    println!("first element of mutable is {0}", &v[0]);
```

TYPE SYSTEM

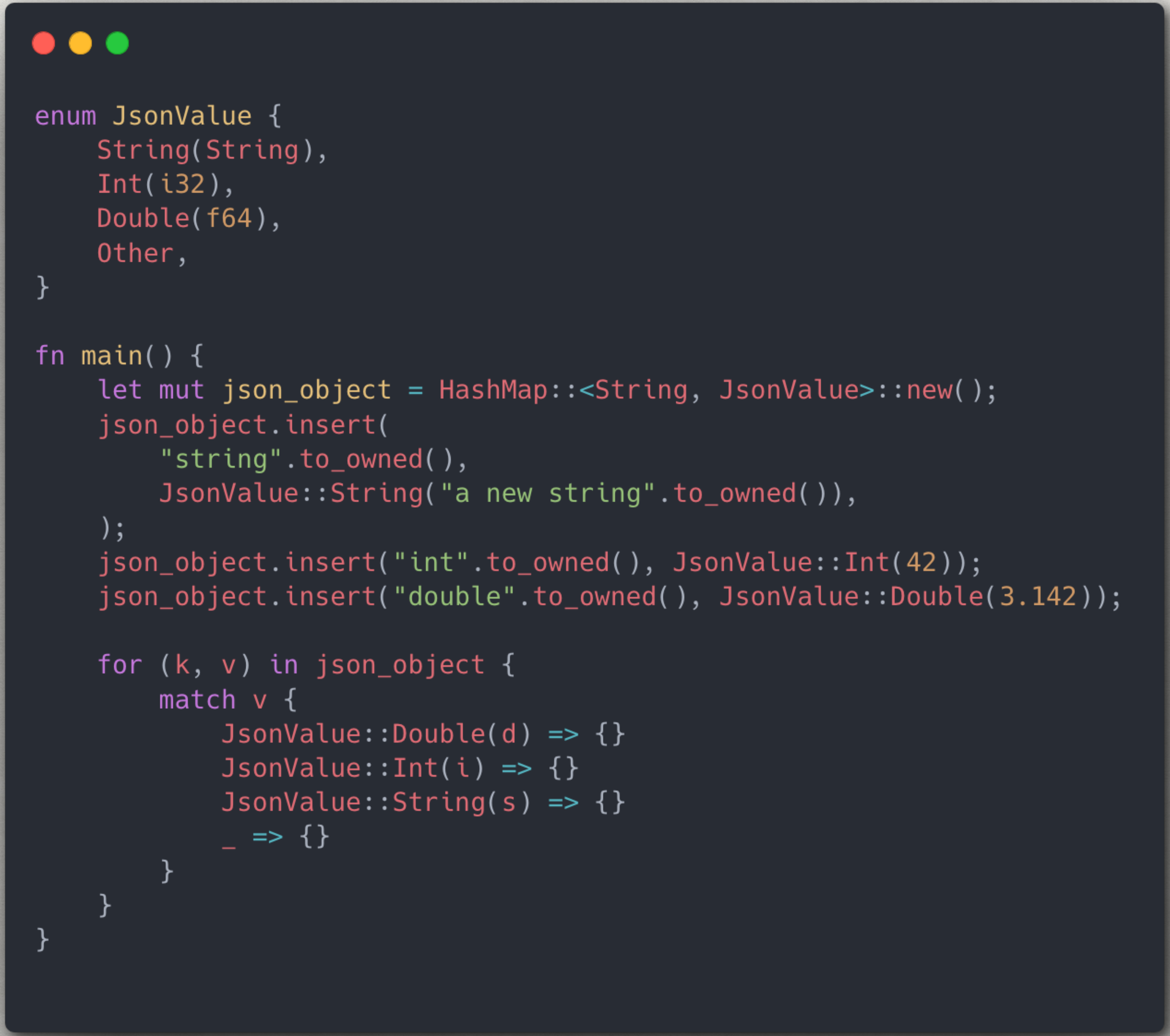
- Primitive types: bool, i32, f64,
- Sequence types: tuples, arrays, slices
- User defined: structs, enums
- Trait types
- Function types



```rust
// primitives
let b: bool = true;

let millis: i64 = 1716880438605;

let sec = millis as f64 / 1000f64;

// sequences

let point: (f32, f32) = (3.142, 42f32);

let array_of_points: [(f32, f32); 1] = [point];

// user defined

struct Viewer;
struct Editor;
struct Admin;

struct User<Role = Viewer> {
    id: String,
    email: String,
    state: PhantomData<Role>,
}

enum JsonValue {
    String(String),
    Int(i32),
    Double(f64),
}
```

ENUMS

- Sum type
- Associated data
- Pattern matching
- Heterogenous collections

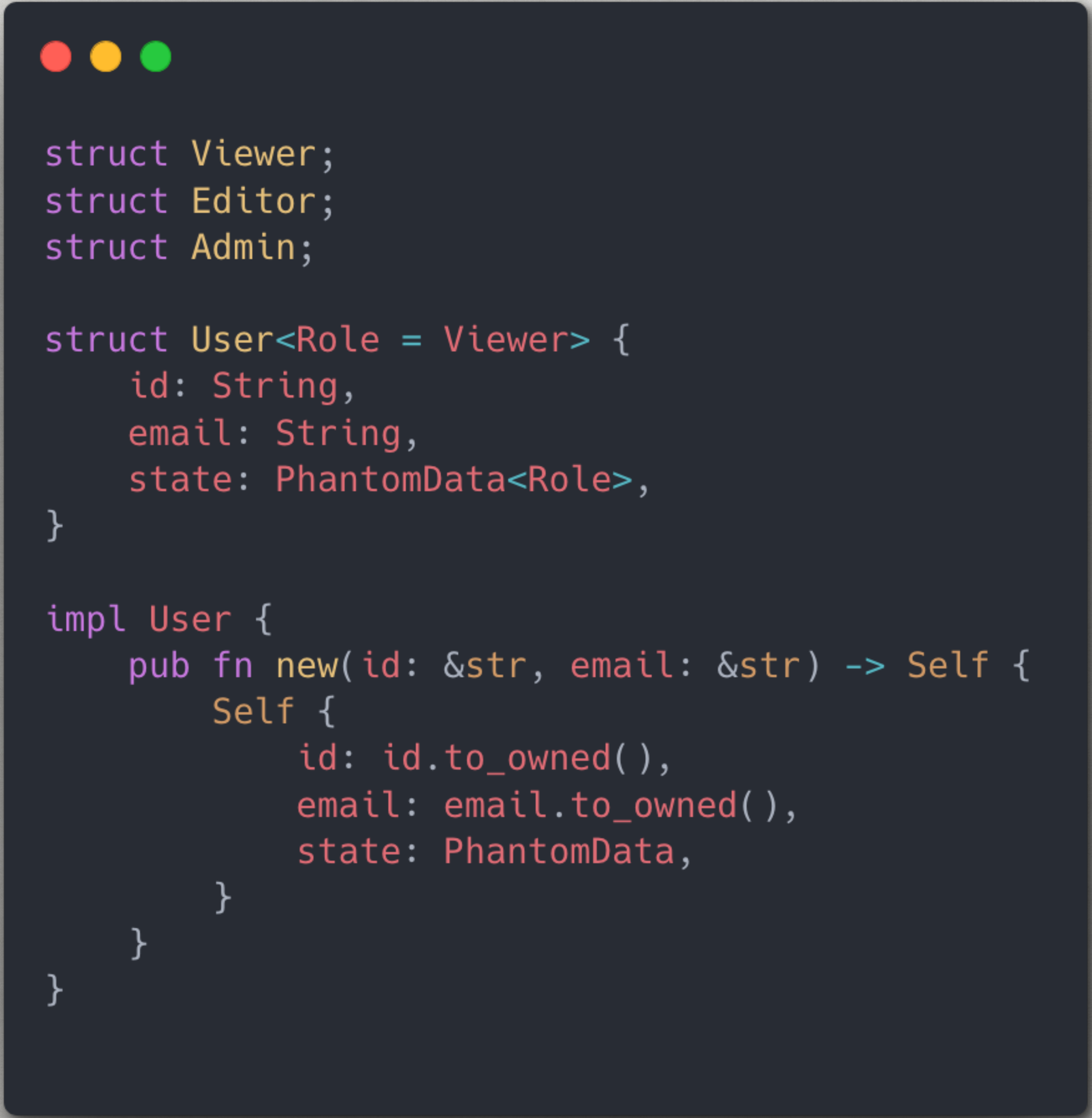


```rust
enum JsonValue {
    String(String),
    Int(i32),
    Double(f64),
    Other,
}

fn main() {
    let mut json_object = HashMap::<String, JsonValue>::new();
    json_object.insert(
        "string".to_owned(),
        JsonValue::String("a new string".to_owned()),
    );
    json_object.insert("int".to_owned(), JsonValue::Int(42));
    json_object.insert("double".to_owned(), JsonValue::Double(3.142));

    for (k, v) in json_object {
        match v {
            JsonValue::Double(d) => {}
            JsonValue::Int(i) => {}
            JsonValue::String(s) => {}
            _ => {}
        }
    }
}
```

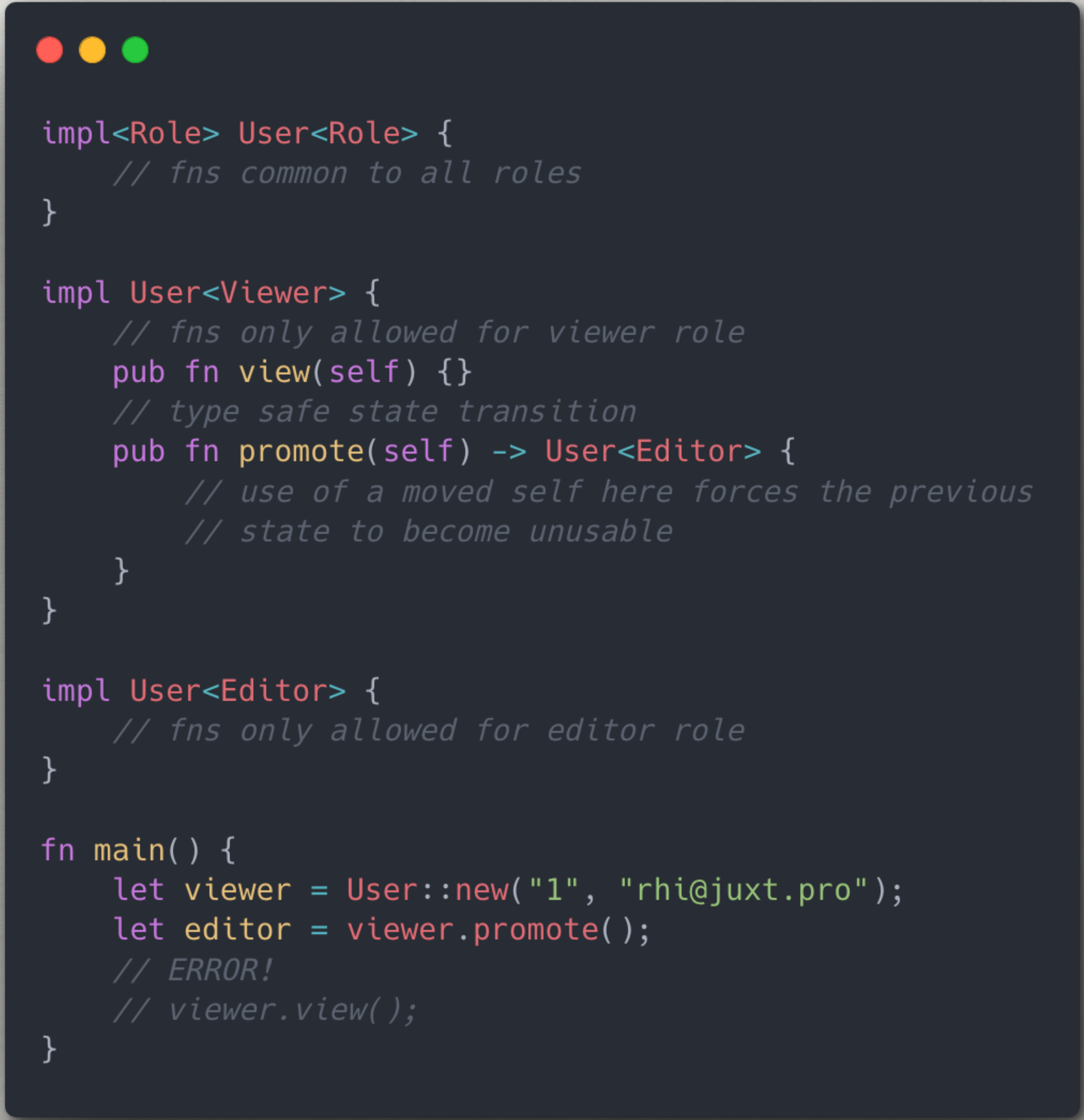
TYPE STATE PATTERN

- Define possible states for struct
- Define struct to be generic over state
- Define state specific functions
- Define transition functions

TYPE STATE PATTERN



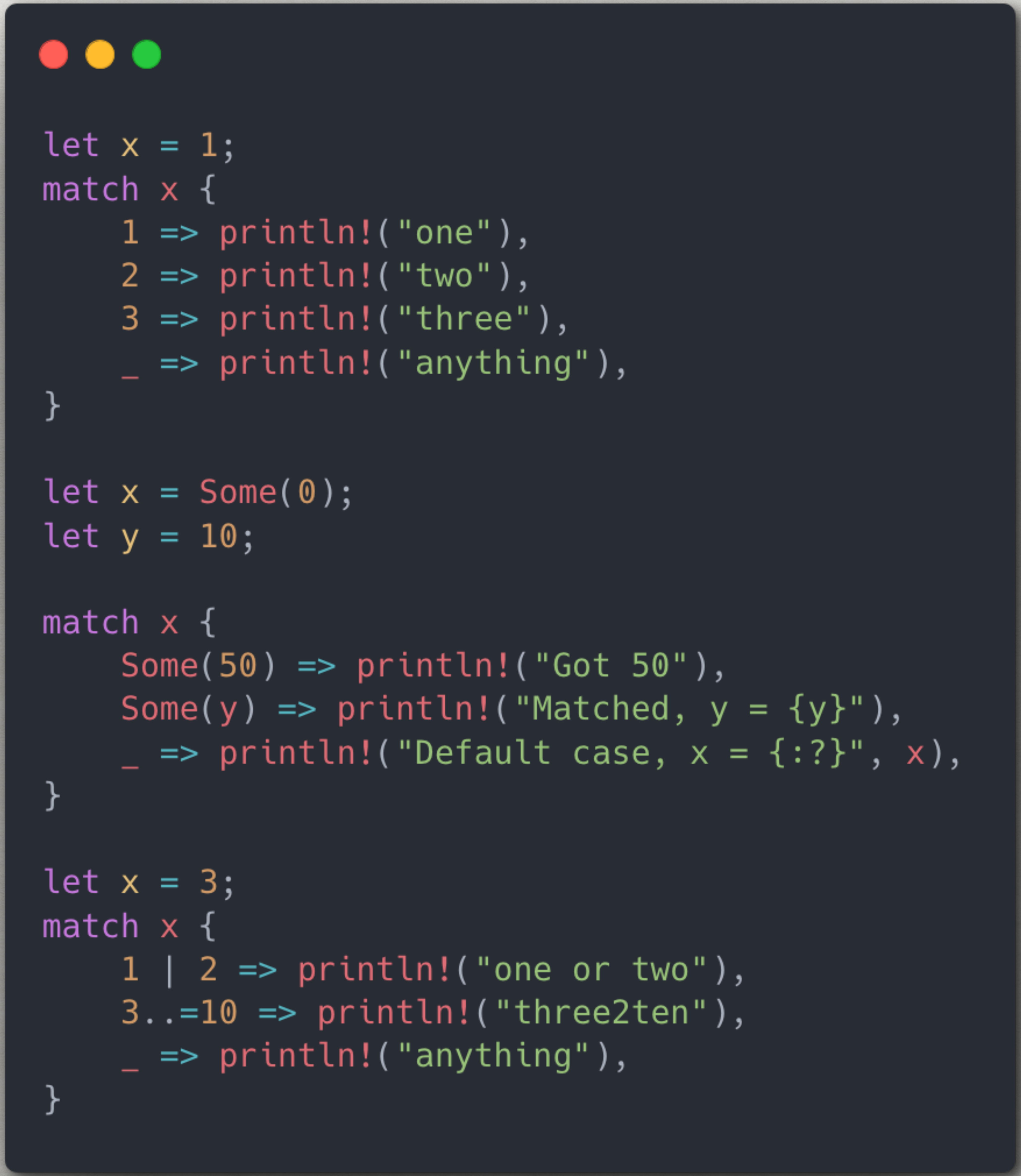
```rust
struct Viewer;
struct Editor;
struct Admin;

struct User<Role = Viewer> {
    id: String,
    email: String,
    state: PhantomData<Role>,
}

impl User {
    pub fn new(id: &str, email: &str) -> Self {
        Self {
            id: id.to_owned(),
            email: email.to_owned(),
            state: PhantomData,
        }
    }
}
```



```rust
impl<Role> User<Role> {
    // fns common to all roles
}

impl User<Viewer> {
    // fns only allowed for viewer role
    pub fn view(self) {}
    // type safe state transition
    pub fn promote(self) -> User<Editor> {
        // use of a moved self here forces the previous
        // state to become unusable
    }
}

impl User<Editor> {
    // fns only allowed for editor role
}

fn main() {
    let viewer = User::new("1", "rhi@juxt.pro");
    let editor = viewer.promote();
    // ERROR!
    // viewer.view();
}
```

PATTERN MATCHING

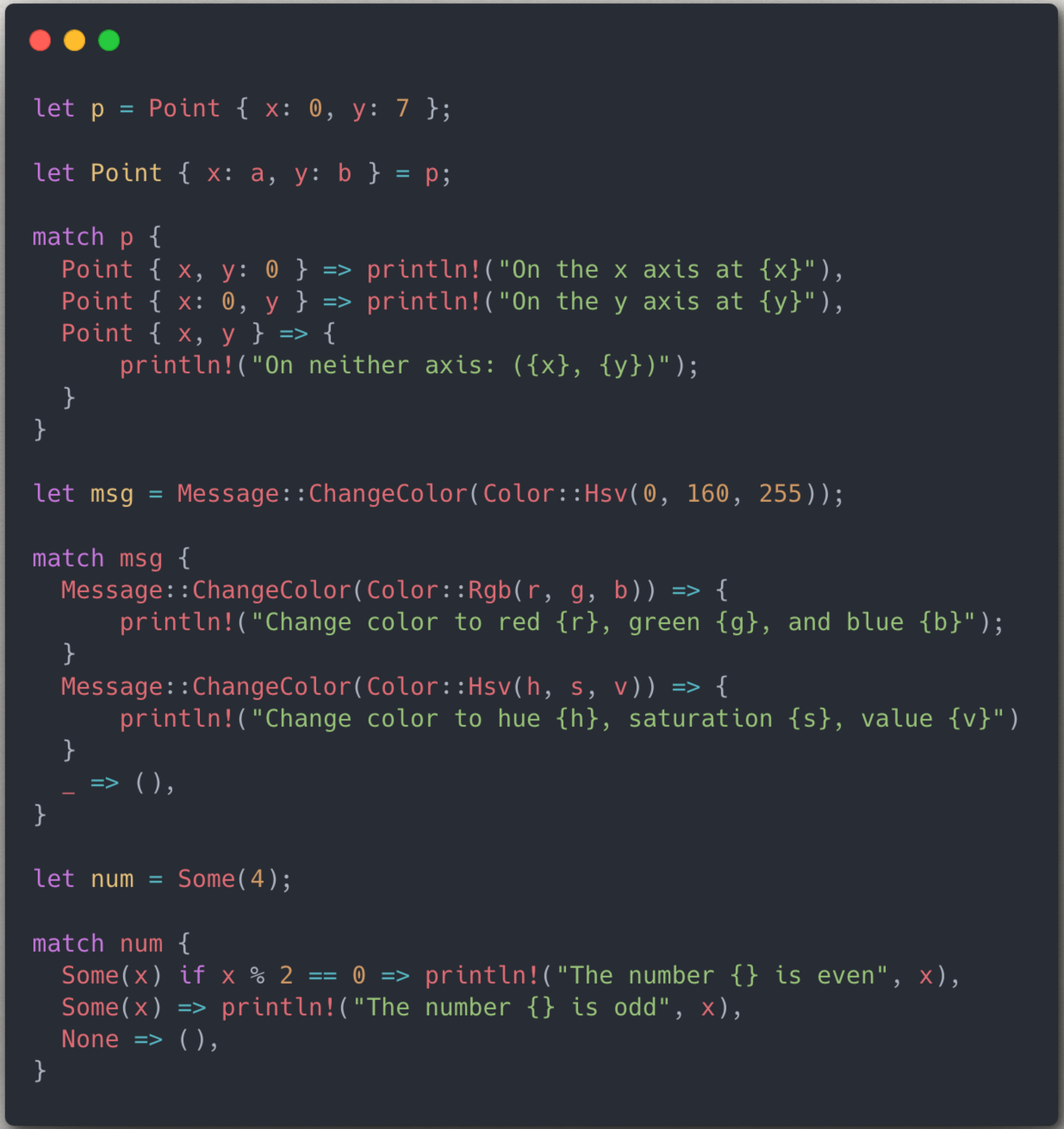
- match the structure of a value
- bind variables to its parts
- Expression, can return values
- Literals, named variables, ranges
- Enums



```rust
let x = 1;
match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}

let x = Some(0);
let y = 10;

match x {
    Some(50) => println!("Got 50"),
    Some(y) => println!("Matched, y = {y}"),
    _ => println!("Default case, x = {:?}", x),
}

let x = 3;
match x {
    1 | 2 => println!("one or two"),
    3..=10 => println!("three2ten"),
    _ => println!("anything"),
}
```

PATTERN MATCHING

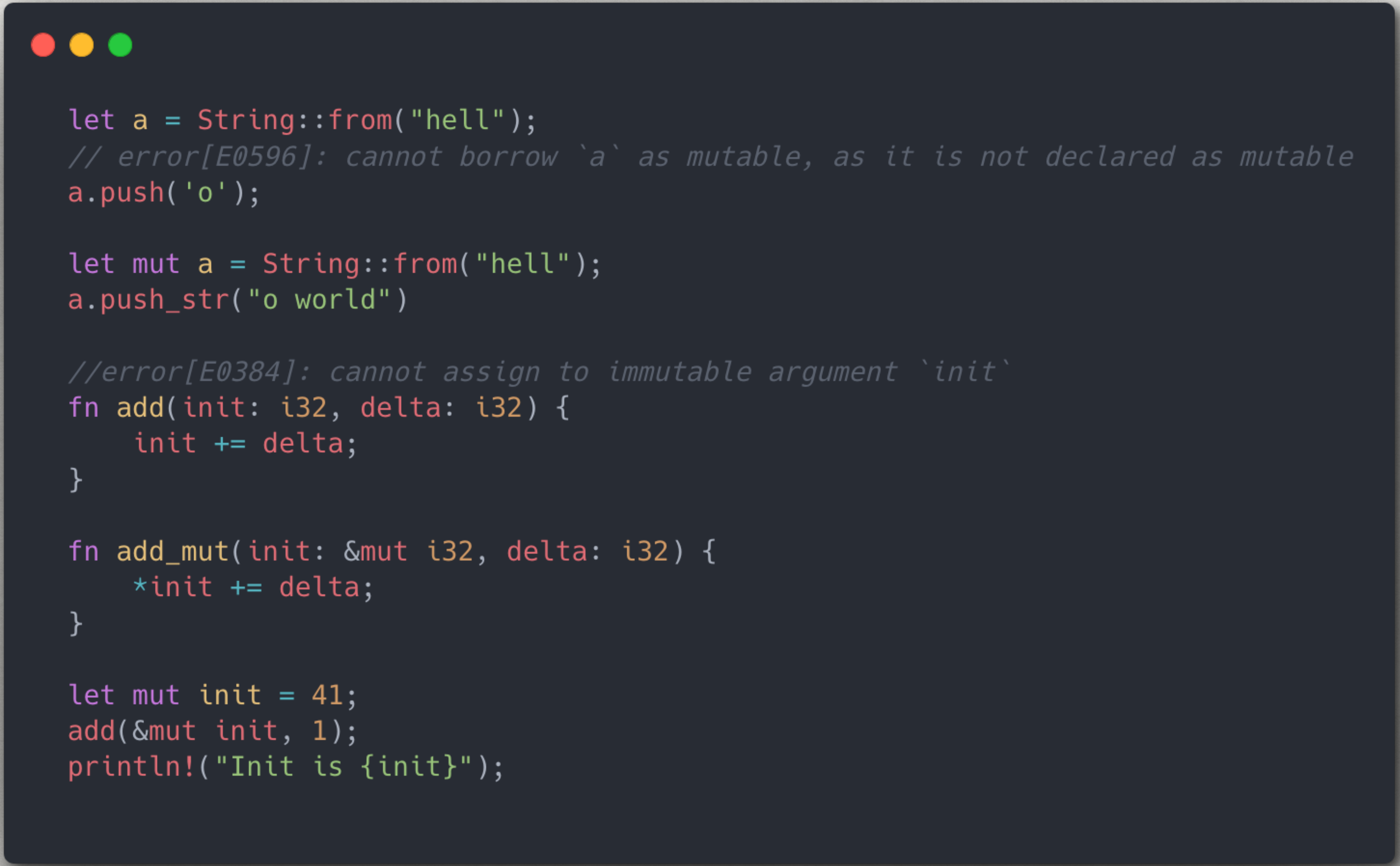
- Destructuring
- match nested structs, enums
- Conditionals



```rust
let p = Point { x: 0, y: 7 };

let Point { x: a, y: b } = p;

match p {
  Point { x, y: 0 } => println!("On the x axis at {x}"),
  Point { x: 0, y } => println!("On the y axis at {y}"),
  Point { x, y } => {
      println!("On neither axis: ({x}, {y})");
  }
}

let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

match msg {
  Message::ChangeColor(Color::Rgb(r, g, b)) => {
      println!("Change color to red {r}, green {g}, and blue {b}");
  }
  Message::ChangeColor(Color::Hsv(h, s, v)) => {
      println!("Change color to hue {h}, saturation {s}, value {v}")
  }
  _ => (),
}

let num = Some(4);

match num {
  Some(x) if x % 2 == 0 => println!("The number {} is even", x),
  Some(x) => println!("The number {} is odd", x),
  None => (),
}
```

MUTABILITY

- Immutable by default
- Borrow checker controls mutation

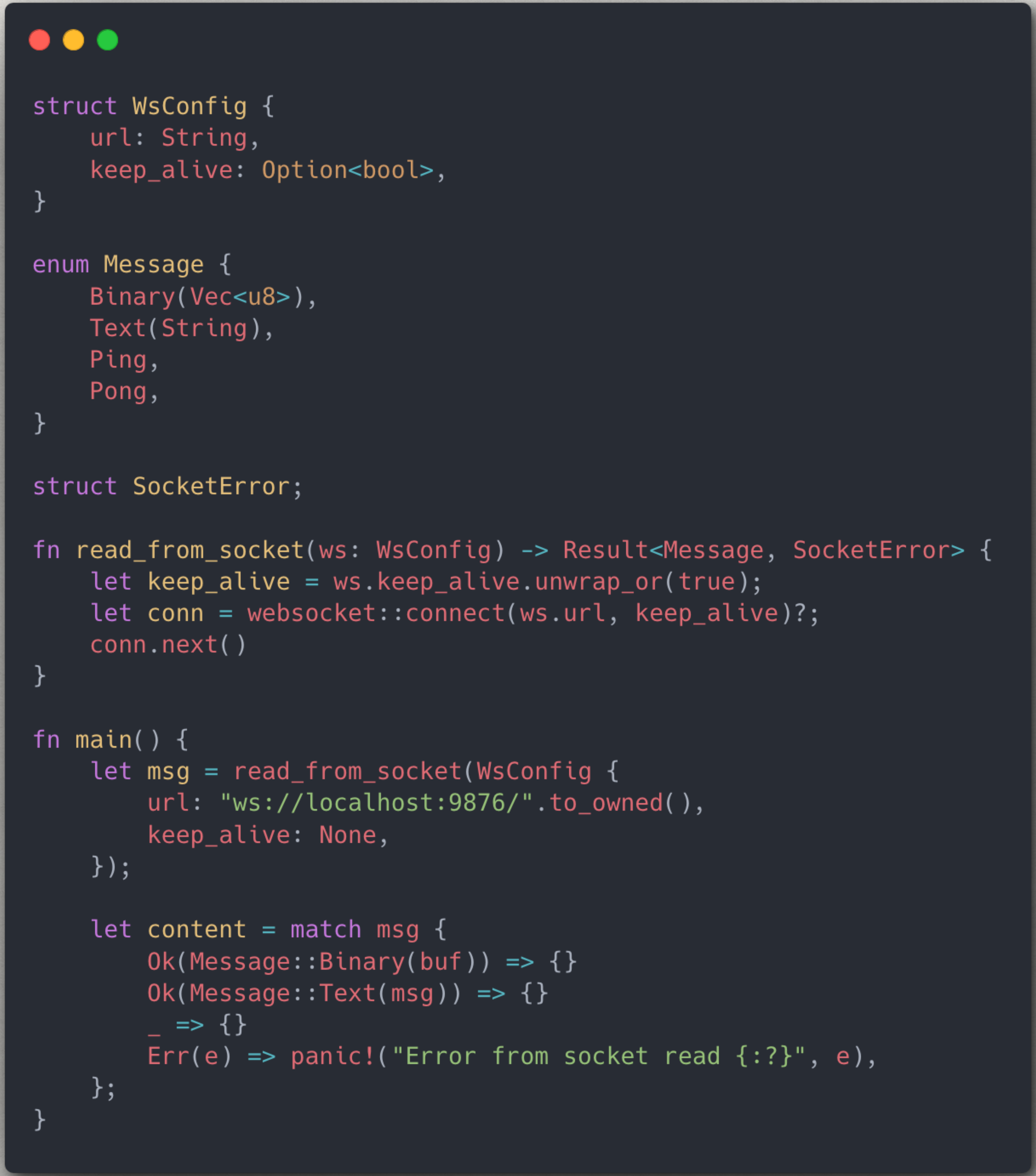


```rust
let a = String::from("hell");
// error[E0596]: cannot borrow `a` as mutable, as it is not declared as mutable
a.push('o');

let mut a = String::from("hell");
a.push_str("o world")

//error[E0384]: cannot assign to immutable argument `init`
fn add(init: i32, delta: i32) {
    init += delta;
}

fn add_mut(init: &mut i32, delta: i32) {
    *init += delta;
}

let mut init = 41;
add(&mut init, 1);
println!("Init is {init}");
```

ERROR HANDLING

OPTION & RESULT

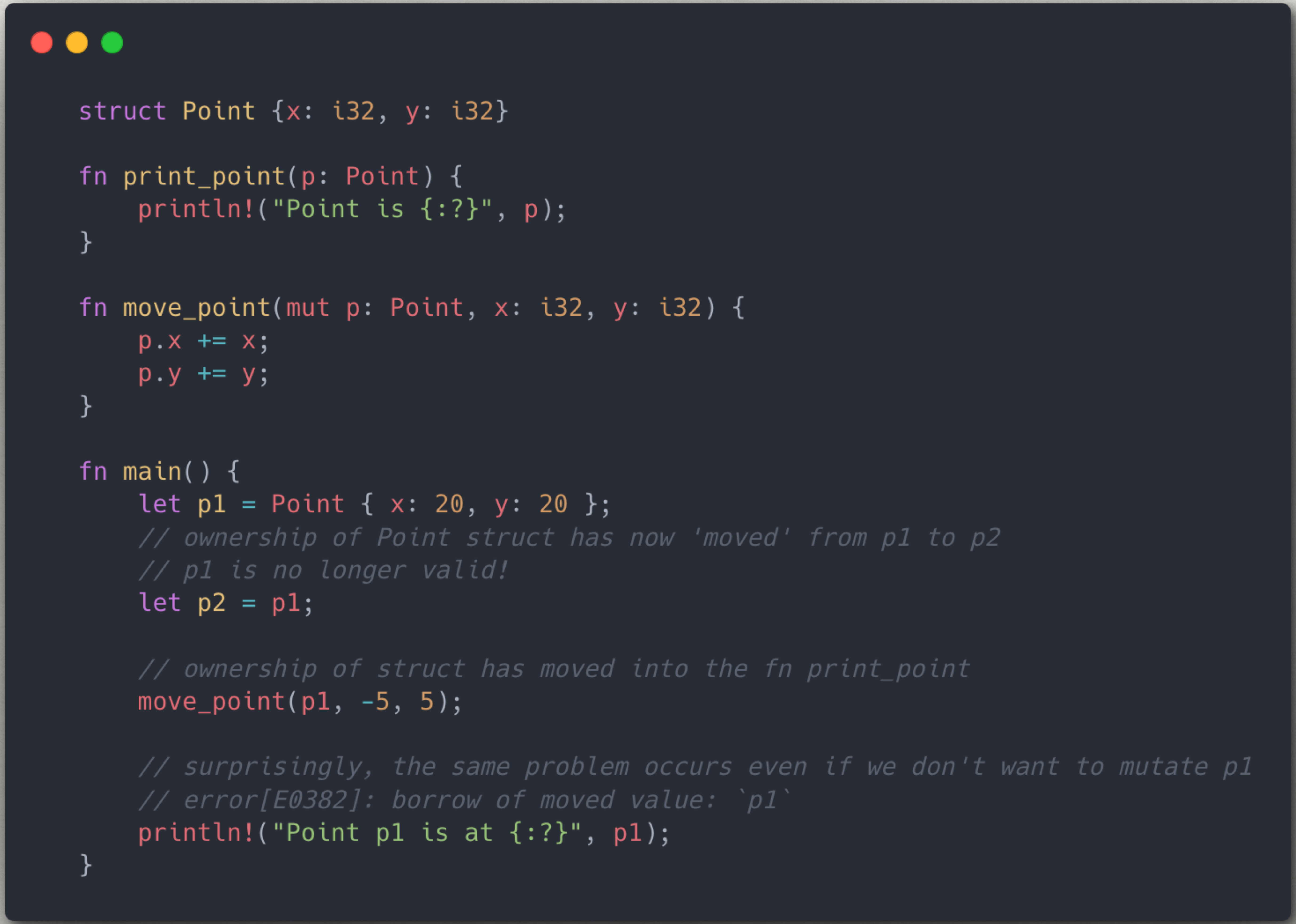
- Option
  - when a value can be absent
- Result
  - when a value can be either valid or error
- ? Operator for early exits



```rust
struct WsConfig {
    url: String,
    keep_alive: Option<bool>,
}

enum Message {
    Binary(Vec<u8>),
    Text(String),
    Ping,
    Pong,
}

struct SocketError;

fn read_from_socket(ws: WsConfig) -> Result<Message, SocketError> {
    let keep_alive = ws.keep_alive.unwrap_or(true);
    let conn = websocket::connect(ws.url, keep_alive)?;
    conn.next()
}

fn main() {
    let msg = read_from_socket(WsConfig {
        url: "ws://localhost:9876/".to_owned(),
        keep_alive: None,
    });

    let content = match msg {
        Ok(Message::Binary(buf)) => {}
        Ok(Message::Text(msg)) => {}
        _ => {}
        Err(e) => panic!("Error from socket read {:?}", e),
    };
}
```

BORROW CHECKER

- Ownership rules
  - Each value in Rust has an owner.
  - There can only be one owner at a time.
  - When the owner goes out of scope, the value will be dropped.

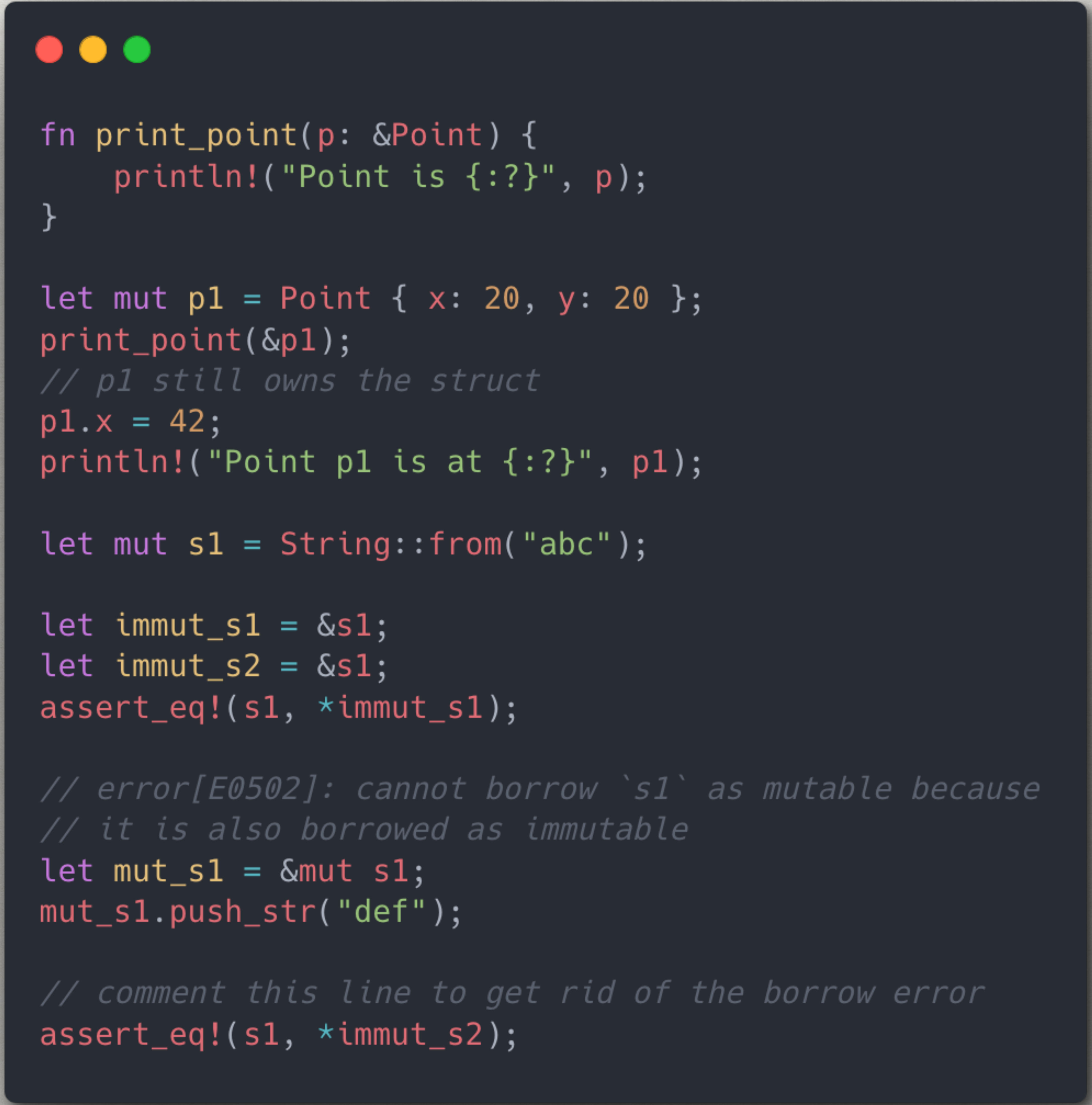


```rust
struct Point {x: i32, y: i32}

fn print_point(p: Point) {
    println!("Point is {:?}", p);
}

fn move_point(mut p: Point, x: i32, y: i32) {
    p.x += x;
    p.y += y;
}

fn main() {
    let p1 = Point { x: 20, y: 20 };
    // ownership of Point struct has now 'moved' from p1 to p2
    // p1 is no longer valid!
    let p2 = p1;

    // ownership of struct has moved into the fn print_point
    move_point(p1, -5, 5);

    // surprisingly, the same problem occurs even if we don't want to mutate p1
    // error[E0382]: borrow of moved value: `p1`
    println!("Point p1 is at {:?}", p1);
}
```

BORROW CHECKER

REFERENCES

- Each value in Rust has an owner.
- There can only be one owner at a time.
- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.
- Like, compile time read-write locking

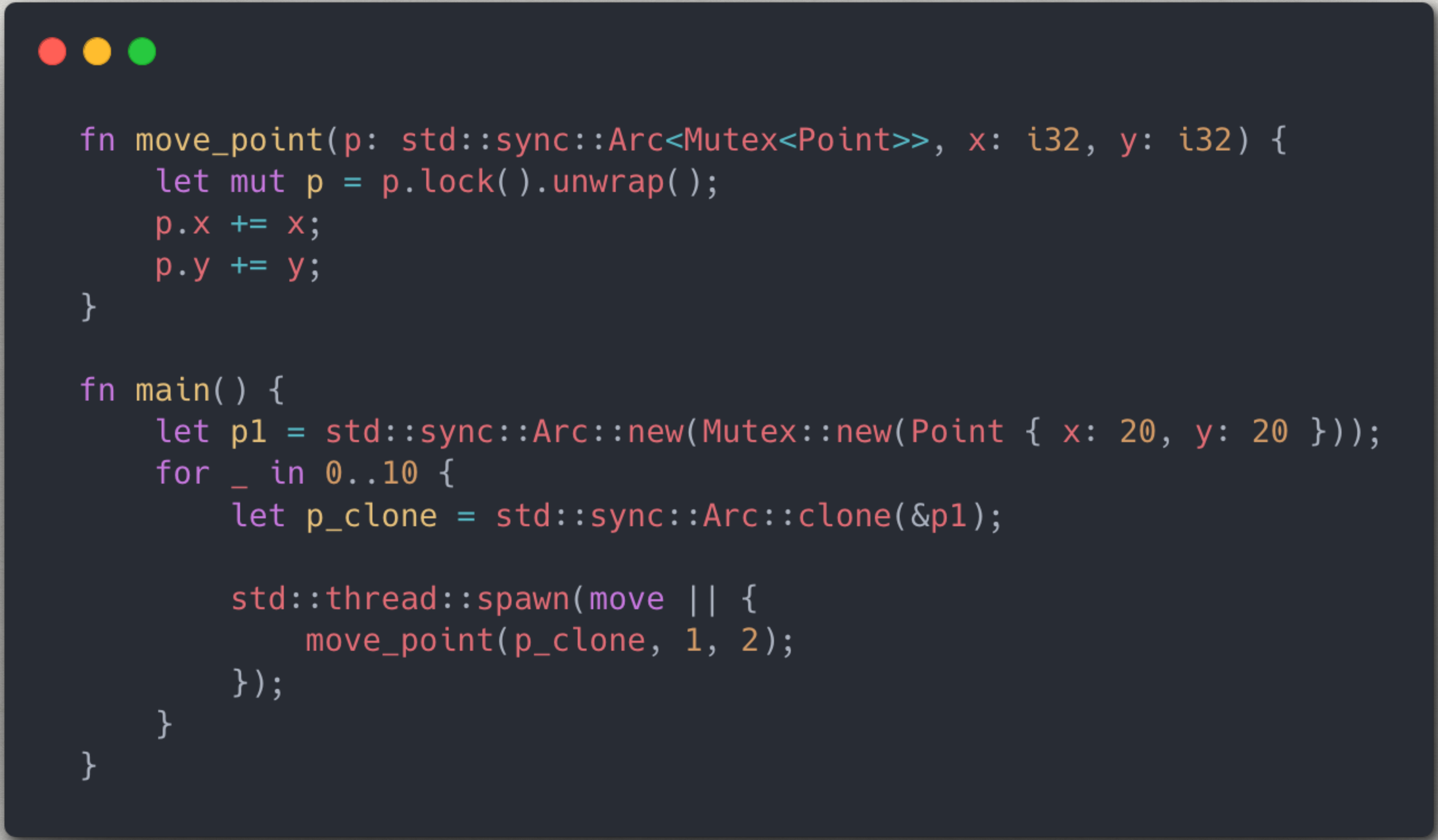


```rust
fn print_point(p: &Point) {
    println!("Point is {:?}", p);
}

let mut p1 = Point { x: 20, y: 20 };
print_point(&p1);
// p1 still owns the struct
p1.x = 42;
println!("Point p1 is at {:?}", p1);

let mut s1 = String::from("abc");

let immut_s1 = &s1;
let immut_s2 = &s1;
assert_eq!(s1, *immut_s1);

// error[E0502]: cannot borrow `s1` as mutable because
// it is also borrowed as immutable
let mut_s1 = &mut s1;
mut_s1.push_str("def");

// comment this line to get rid of the borrow error
assert_eq!(s1, *immut_s2);
```

THE CHALLENGING BITS

- Shared-state
- Collections
- Functions and Closures
- Concurrency

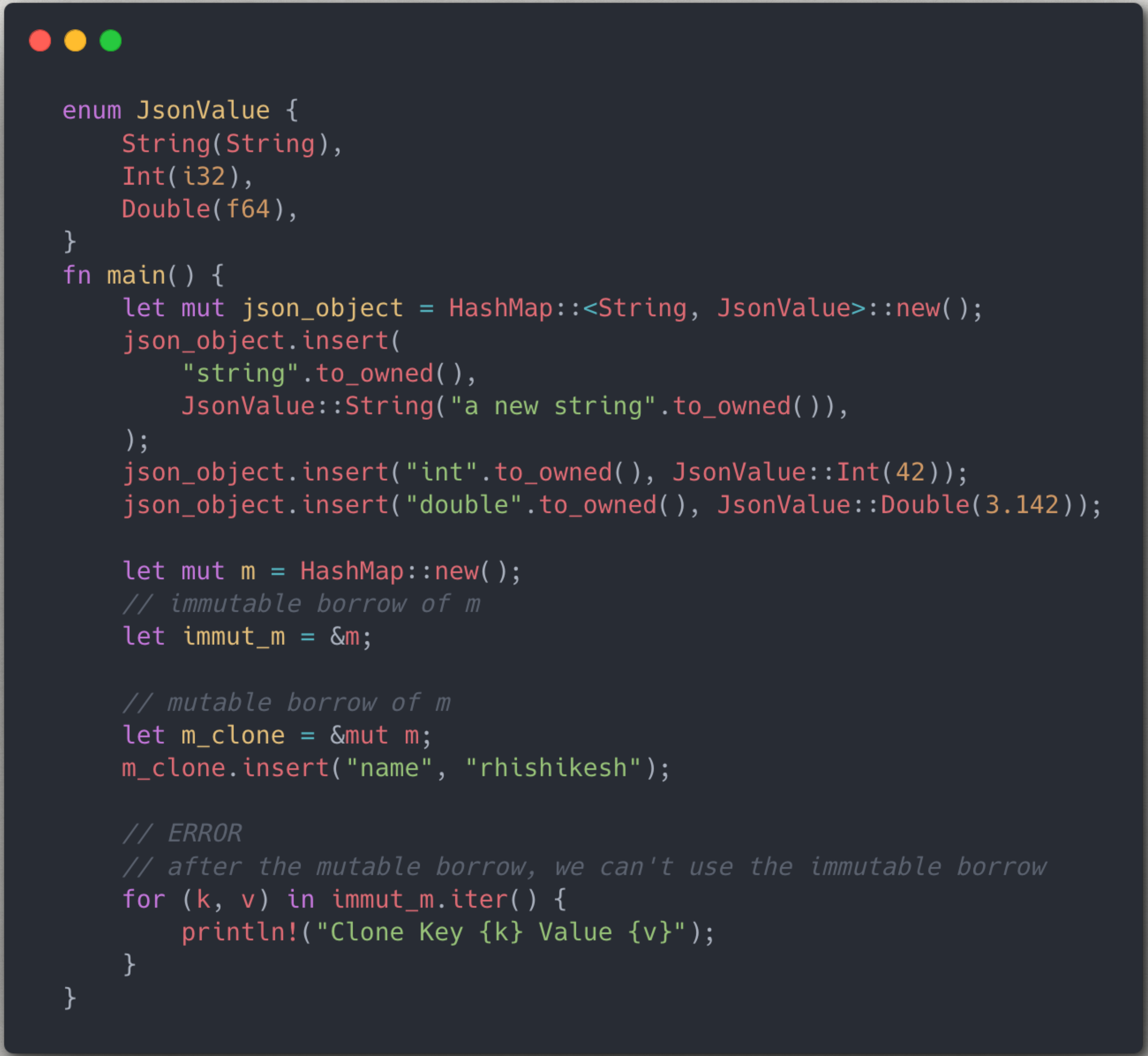
SHARED STATE



```
fn move_point(p: std::sync::Arc<Mutex<Point>>, x: i32, y: i32) {
    let mut p = p.lock().unwrap();
    p.x += x;
    p.y += y;
}

fn main() {
    let p1 = std::sync::Arc::new(Mutex::new(Point { x: 20, y: 20 }));
    for _ in 0..10 {
        let p_clone = std::sync::Arc::clone(&p1);

        std::thread::spawn(move || {
            move_point(p_clone, 1, 2);
        });
    }
}
```

COLLECTIONS

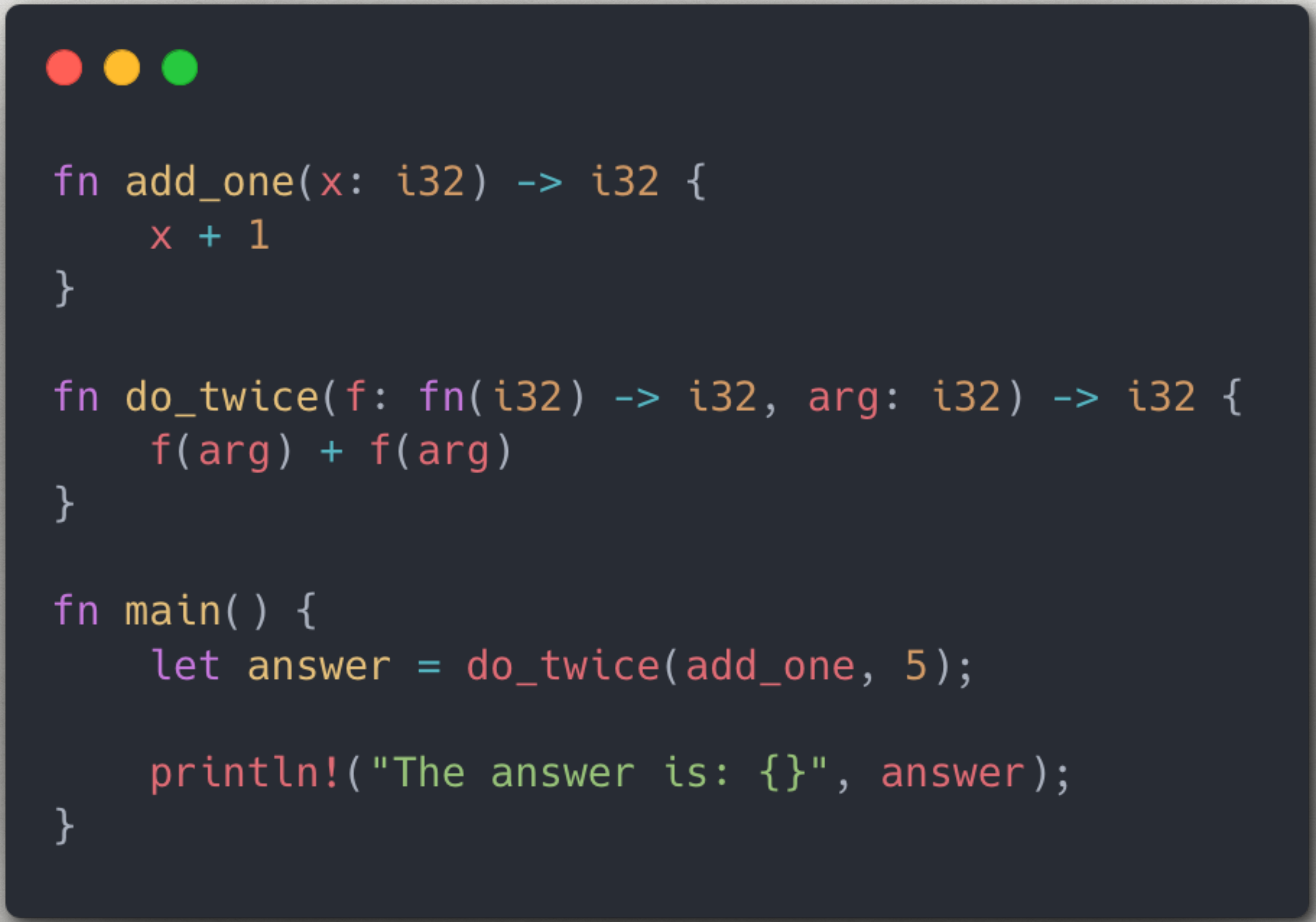
- Vec, HashMap, HashSet
- map, filter, fold, flatten etc.
- Collections are typed and homogeneous
- In-place updates



```rust
enum JsonValue {
    String(String),
    Int(i32),
    Double(f64),
}
fn main() {
    let mut json_object = HashMap::<String, JsonValue>::new();
    json_object.insert(
        "string".to_owned(),
        JsonValue::String("a new string".to_owned()),
    );
    json_object.insert("int".to_owned(), JsonValue::Int(42));
    json_object.insert("double".to_owned(), JsonValue::Double(3.142));

    let mut m = HashMap::new();
    // immutable borrow of m
    let immut_m = &m;

    // mutable borrow of m
    let m_clone = &mut m;
    m_clone.insert("name", "rhishikesh");

    // ERROR
    // after the mutable borrow, we can't use the immutable borrow
    for (k, v) in immut_m.iter() {
        println!("Clone Key {k} Value {v}");
    }
}
```

FUNCTIONS AND CLOSURES

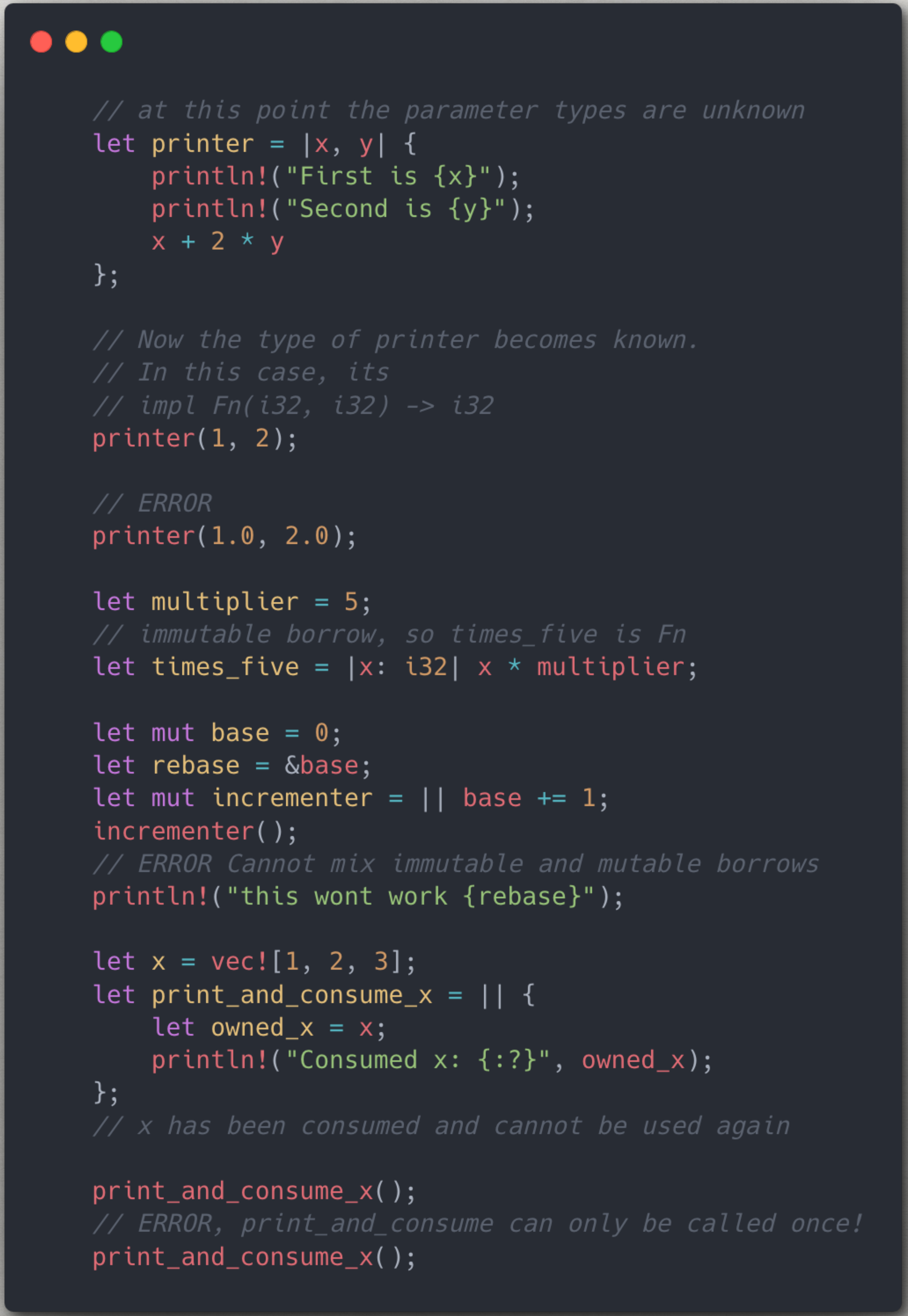
- fn is a type
  - Functions coerce to fn
  - No capture
- Closures are traits
  - Anonymous functions that capture env
  - Fn, FnMut, FnOnce



```rust
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

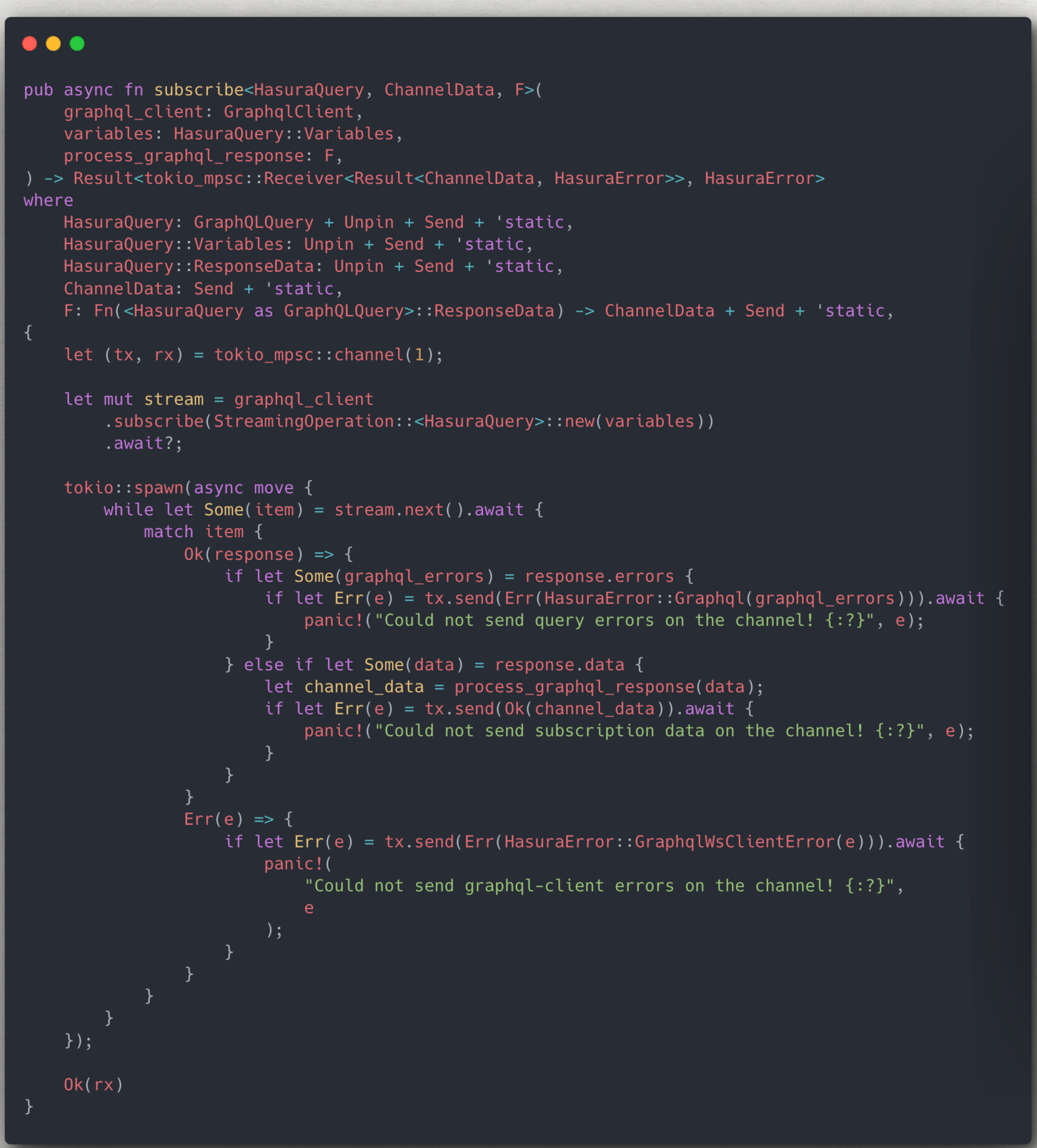
CLOSURES

- Borrow rules apply to closure captures
  - Compiler decides
  - Immutable borrow, Fn
  - Mutable borrow, FnMut
  - Owned, FnOnce
  - *move* to be explicit

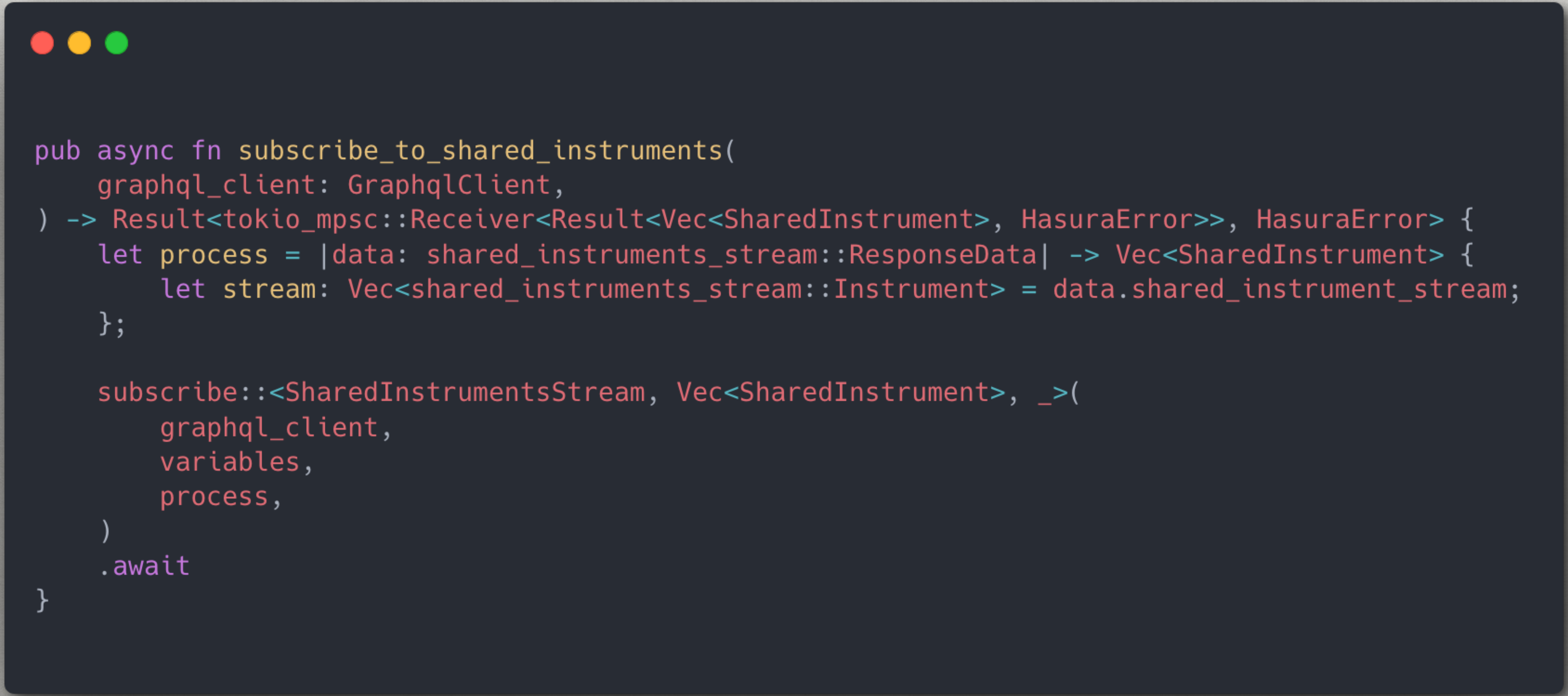


```rust
// at this point the parameter types are unknown
let printer = |x, y| {
    println!("First is {x}");
    println!("Second is {y}");
    x + 2 * y
};

// Now the type of printer becomes known.
// In this case, its
// impl Fn(i32, i32) -> i32
printer(1, 2);

// ERROR
printer(1.0, 2.0);

let multiplier = 5;
// immutable borrow, so times_five is Fn
let times_five = |x: i32| x * multiplier;

let mut base = 0;
let rebase = &base;
let mut incrementer = || base += 1;
incrementer();
// ERROR Cannot mix immutable and mutable borrows
println!("this wont work {rebase}");

let x = vec![1, 2, 3];
let print_and_consume_x = || {
    let owned_x = x;
    println!("Consumed x: {:?}", owned_x);
};
// x has been consumed and cannot be used again

print_and_consume_x();
// ERROR, print_and_consume can only be called once!
print_and_consume_x();
```

CLOSURES GET MESSY



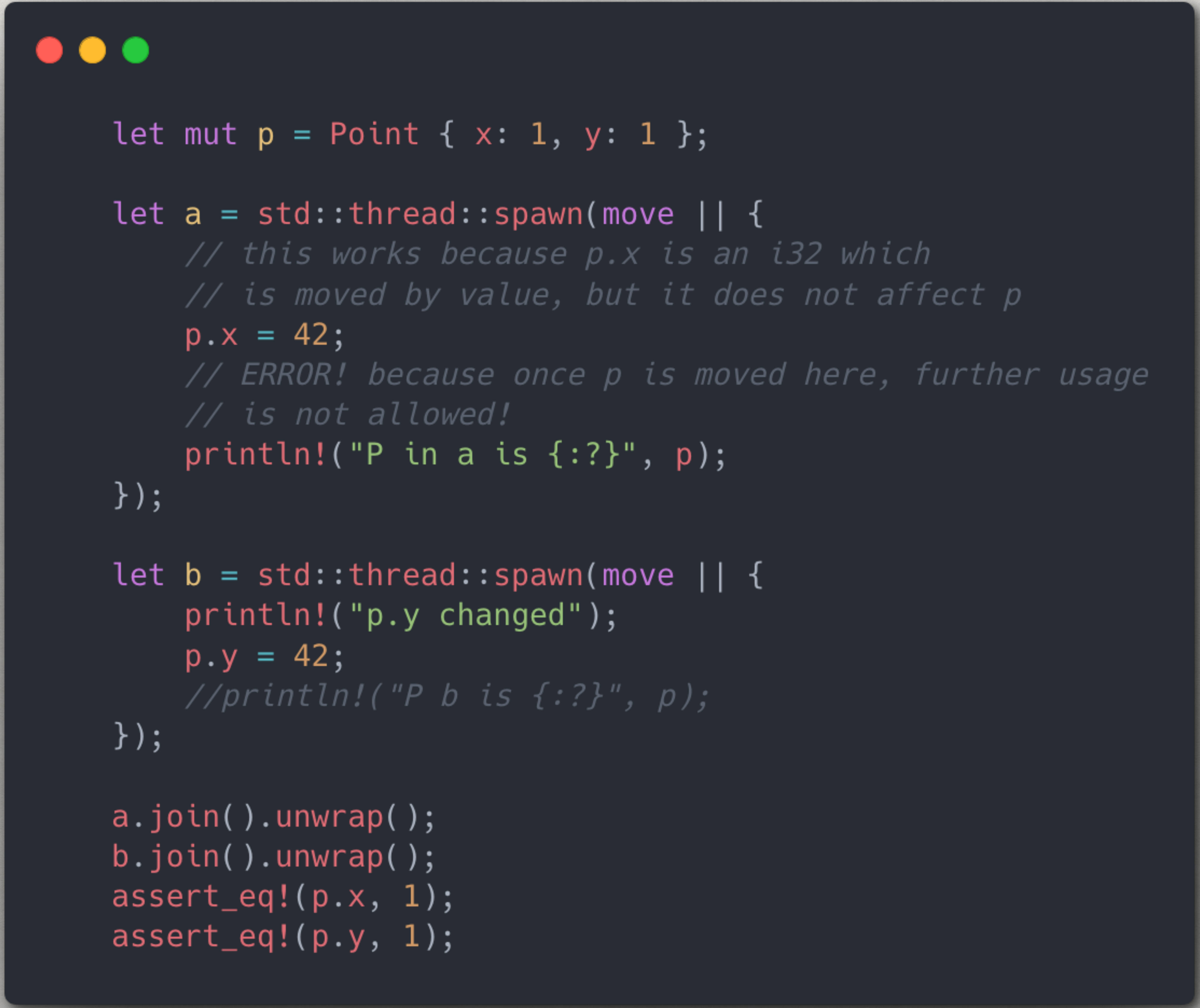
```rust
pub async fn subscribe<HasuraQuery, ChannelData, F>(
    graphql_client: GraphqlClient,
    variables: HasuraQuery::Variables,
    process_graphql_response: F,
) -> Result<tokio_mpsc::Receiver<Result<ChannelData, HasuraError>>, HasuraError>
where
    HasuraQuery: GraphQLQuery + Unpin + Send + 'static,
    HasuraQuery::Variables: Unpin + Send + 'static,
    HasuraQuery::ResponseData: Unpin + Send + 'static,
    ChannelData: Send + 'static,
    F: Fn(<HasuraQuery as GraphQLQuery>::ResponseData) -> ChannelData + Send + 'static,
{
    let (tx, rx) = tokio_mpsc::channel(1);

    let mut stream = graphql_client
        .subscribe(StreamingOperation::<HasuraQuery>::new(variables))
        .await?;

    tokio::spawn(async move {
        while let Some(item) = stream.next().await {
            match item {
                Ok(response) => {
                    if let Some(graphql_errors) = response.errors {
                        if let Err(e) = tx.send(Err(HasuraError::Graphql(graphql_errors))).await {
                            panic!("Could not send query errors on the channel! {:?}", e);
                        }
                    } else if let Some(data) = response.data {
                        let channel_data = process_graphql_response(data);
                        if let Err(e) = tx.send(Ok(channel_data)).await {
                            panic!("Could not send subscription data on the channel! {:?}", e);
                        }
                    }
                }
                Err(e) => {
                    if let Err(e) = tx.send(Err(HasuraError::GraphqlWsClientError(e))).await {
                        panic!(
                            "Could not send graphql-client errors on the channel! {:?}",
                            e
                        );
                    }
                }
            }
        }
    });

    Ok(rx)
}
```



```rust
pub async fn subscribe_to_shared_instruments(
    graphql_client: GraphqlClient,
) -> Result<tokio_mpsc::Receiver<Result<Vec<SharedInstrument>, HasuraError>>, HasuraError> {
    let process = |data: shared_instruments_stream::ResponseData| -> Vec<SharedInstrument> {
        let stream: Vec<shared_instruments_stream::Instrument> = data.shared_instrument_stream;
    };

    subscribe::<SharedInstrumentsStream, Vec<SharedInstrument>, _>(
        graphql_client,
        variables,
        process,
    )
    .await
}
```

CONCURRENCY

- Fearless concurrency
  - Only if you don't fear 20+ line compiler errors!
- async/await
- 'static + Send



```rust
let mut p = Point { x: 1, y: 1 };

let a = std::thread::spawn(move || {
    // this works because p.x is an i32 which
    // is moved by value, but it does not affect p
    p.x = 42;
    // ERROR! because once p is moved here, further usage
    // is not allowed!
    println!("P in a is {:?}", p);
});

let b = std::thread::spawn(move || {
    println!("p.y changed");
    p.y = 42;
    //println!("P b is {:?}", p);
});

a.join().unwrap();
b.join().unwrap();
assert_eq!(p.x, 1);
assert_eq!(p.y, 1);
```

WHAT I MISS FROM CLOJURE

- REPL
- Collections, specially heterogeneous data
- Atoms
- Ease of exploration
- *Macros*

WHAT I MIGHT MISS FROM RUST

- The rust compiler's amazing error messages!
- Ease and confidence of refactoring code
- Enums
- Error handling with Option and Result types

CONCLUSION

- In my experience, rust and clojure are quite complimentary to each other. Areas where rust fades have quite a bit of overlap with Clojure's strengths and vice-versa.
- Learning rust over the past 5 months has been a great roller-coaster ride and hopefully it will help me improve my Clojure skills too
- References
  - [[https://gist.github.com/oakes/4af1023b6c5162c6f8f0][rust for clojurists]]
  - https://doc.rust-lang.org/stable/rust-by-example/
  - https://doc.rust-lang.org/stable/book/