FP Block

www.fpblock.com

When to be weird

Michael Snoyman
VP Software, FP Block
Functional Conference 2025

weird 1 of 2 adjective

wird

Synonyms of *weird* >

1 : of strange extraordinary character: **ODD**, **Fantastic**

2 : of, relating to, or caused by **witchcraft** or the supernatural

weirdness noun

weird 2 of 2 noun

1 : **FATE, DESTINY**
especially : ill fortune

2 : **SOOTHSAYER**

What's weird?

I probably should have checked the dictionary before naming the talk

Let's focus on the first definition: strange, extraordinary

Though witchcraft may be relevant too :)

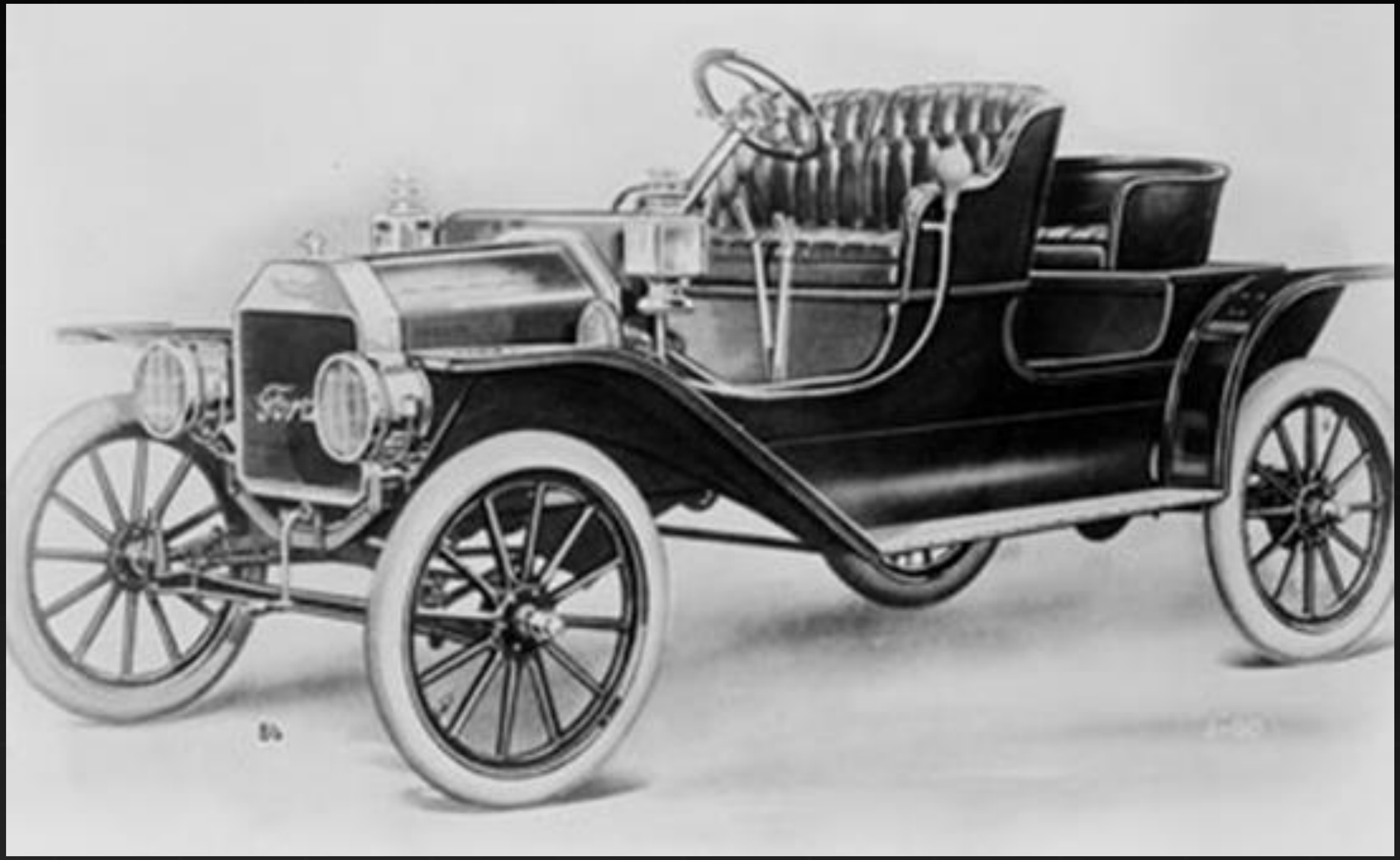
Weird can be good

FP Block

www.fpblock.com

THE FLAT EARTH SOCIETY HAS MEMBERS ALL AROUND THE GLOBE

But, sometimes follow the crowd



Disruption can be weird

If I’d listen to customers, I’d given them a faster horse

- Henry Ford

Weird may just be weird

Weird investments can be great

Early Adopter

Get in before the big boom

Get Rich

Or, it can suck

Everyone else left the market

You stuck around waiting for the ‘inevitable’ pump

But it just didn’t happen

FP Block

www.fpblock.com

Why should we care?

We want technology and developer experience to improve

The world regularly makes terrible tech decisions – why?

We'd like to be able to predict the direction of our industry

And more: how can we effectively shape the future?

Weird things that won

Structured code (vs GOTO and jumps)

COBOL and “human language” programming languages (and thankfully later lost)

Map/reduce vs for-loops-galore

XMLHttpRequest, aka AJAX, aka DHTML, aka The Modern Web (for better or worse)

Object oriented coding

Encapsulation

Competition of Ideas

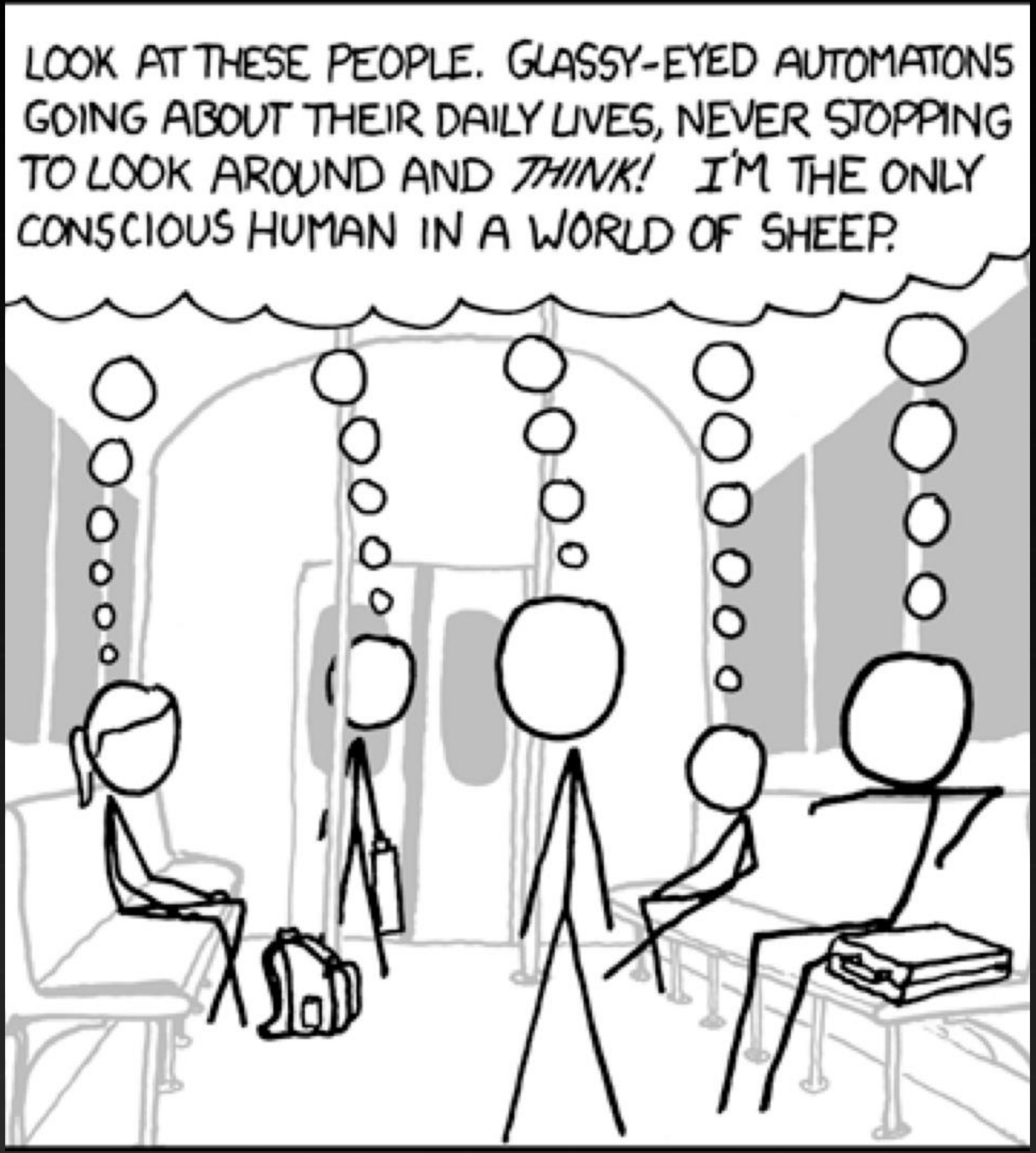
	LOW COST	HIGH COST
LOW BENEFIT	Easier to slip into sprints	Run away
HIGH BENEFIT	Do this first	Difficult to get approved

Yes, I am talking economics again

Economics assumption: perfect knowledge and rational actors

Neither is true! We won't always make perfect decisions

Let's discuss what should win and what has won


LOOK AT THESE PEOPLE. GLASSY-EYED AUTOMATONS GOING ABOUT THEIR DAILY LIVES, NEVER STOPPING TO LOOK AROUND AND *THINK!* I'M THE ONLY CONSCIOUS HUMAN IN A WORLD OF SHEEP.

Is functional programming good weird?

- Overall: yes
- But: it's a bad question!
- FP is ill defined
- Even with a definition, too many things under the umbrella
- Not a binary, some things are better, some worse
- Need to analyze things individually
- Part of why FP is dead: we already won; many of our ideas are already mainstream
- But not all of them

Patterns to successful weirdness

What succeeds?

- No need to rewrite the world
- Large benefits
- Small costs
- “Sexiness”
- Easier to work with (worse is better)

What doesn’t succeed?

- Correctness (much to our chagrin)
- “Reason about code”
- Provides non-tangible benefits

Ultimately: software has to be judged in the free market.
We need weird things that the market will value.

Let’s talk some tech

- That was lots of words without any code! Boo!!!
- Time to talk about concrete examples
- Warning in advance: I’m going to be opinionated
  - Everyone will disagree with something I’m about to say
  - My point isn’t to convince people I’m right
  - Goal is to share my thought process

Victory Lap

Where we've already won

FP Block

www.fpblock.com

Typed Programming

- People used to think types were a chore with no benefit
    - Old school C, C++, and Java certainly promoted this concept!
- Many of us fought a valiant fight for strong type systems
- We're not winning. We already won.
    - Python: type annotations galore
    - JavaScript: eww, everyone uses TypeScript!
    - JVM and .NET embracing ideas from Scala and F#
    - Rust is very strongly typed and gaining ground
    - Even Go had to add generics (arguably a nod to typed programming)

Why typed programming won?

- Type inference!
  - No one wanted to write int x = 5 all the time
- Weak type systems (like C) hid the potential values of types
  - FP languages demonstrated you can achieve lots of bug prevention
- Maybe more importantly: developer productivity
  - LSP/IntelliSense/Autocomplete is better with types
  - Avoids the need for lots of boilerplate tests (but you still need to test your code!!!)

Immutability By Default

- I'd argue: one of the core tenets of functional programming
- Considered best practice in many programming languages
  - Though some older languages struggle to do it well!
- Extends beyond just code
  - Immutable Infrastructure/Terraform/Infrastructure as Code
  - Arguably: blockchain, Git, content-addressable storage

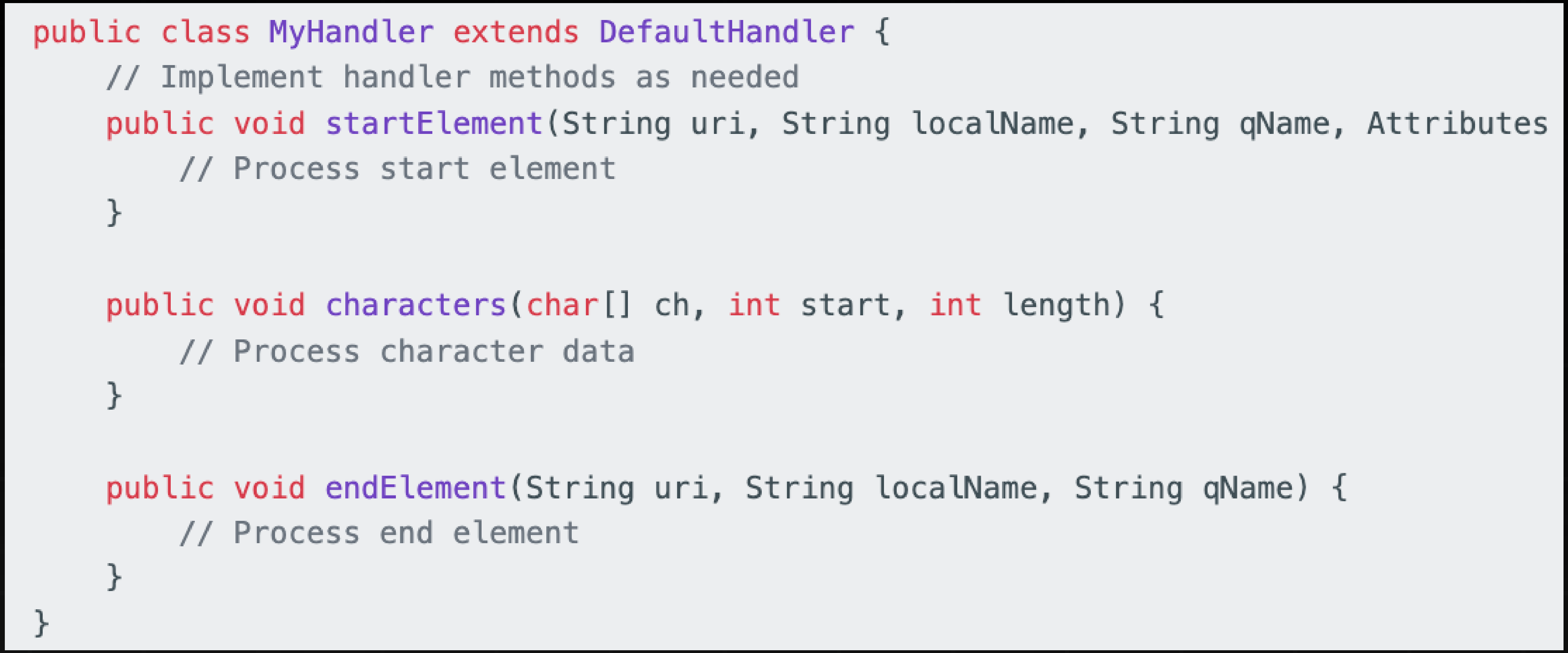
Winning
But not quite there yet

Sum Types

- The #1 feature, bar none, that carries the best power-to-weight ratio
- Core to ML and Haskell, adopted by Rust and TypeScript extensively
- Destroys the need for the Visitor Pattern
- Easy to explain to people
- Demonstrable benefits
    - Compiler-driven development
    - Reduced bugs
    - Simpler to implement than Visitor pattern or Church-encoding

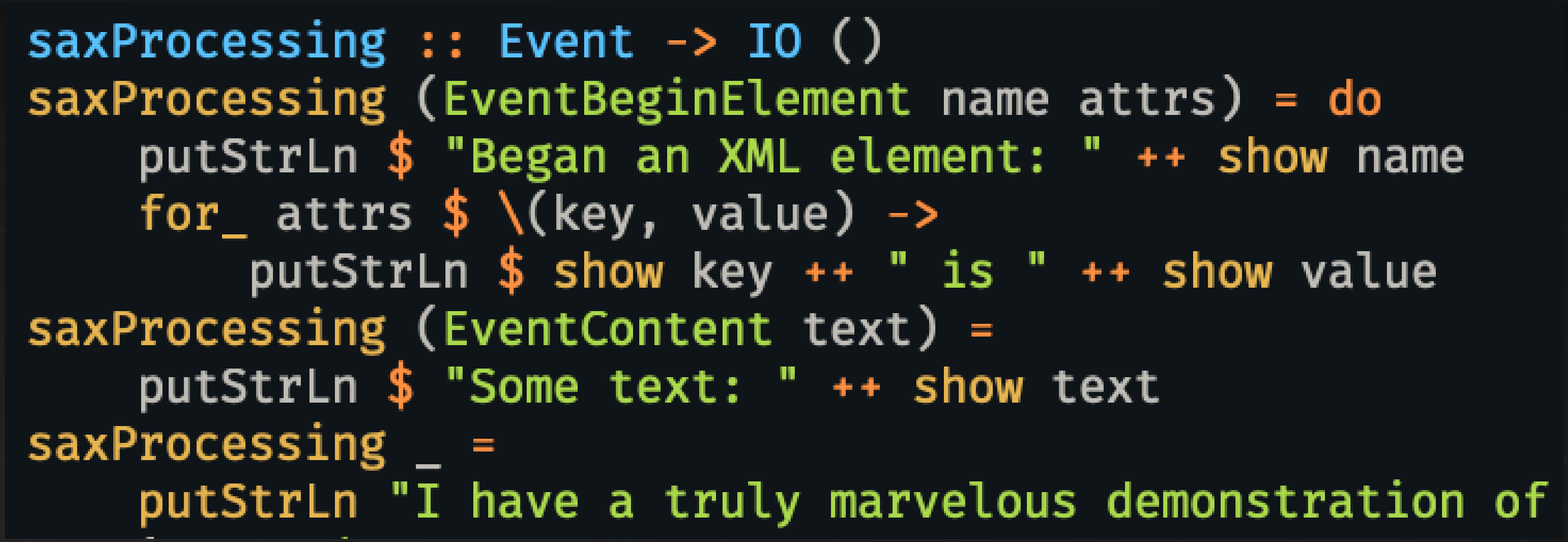
XML Processing Example

Visitor pattern (Java/SAX)



```java
public class MyHandler extends DefaultHandler {
    // Implement handler methods as needed
    public void startElement(String uri, String localName, String qName, Attributes
        // Process start element
    }

    public void characters(char[] ch, int start, int length) {
        // Process character data
    }

    public void endElement(String uri, String localName, String qName) {
        // Process end element
    }
}
```

Sum Types



```haskell
saxProcessing :: Event -> IO ()
saxProcessing (EventBeginElement name attrs) = do
    putStrLn $ "Began an XML element: " ++ show name
    for_ attrs $ \(key, value) ->
        putStrLn $ show key ++ " is " ++ show value
saxProcessing (EventContent text) =
    putStrLn $ "Some text: " ++ show text
saxProcessing _ =
    putStrLn "I have a truly marvelous demonstration of
```

FP Block

www.fpblock.com

Why haven't sum types fully won yet?

Not available in many popular languages (Java, C#, Go)

Many programmers still unaware of sum types

Name is arguably scary: sounds too mathematical

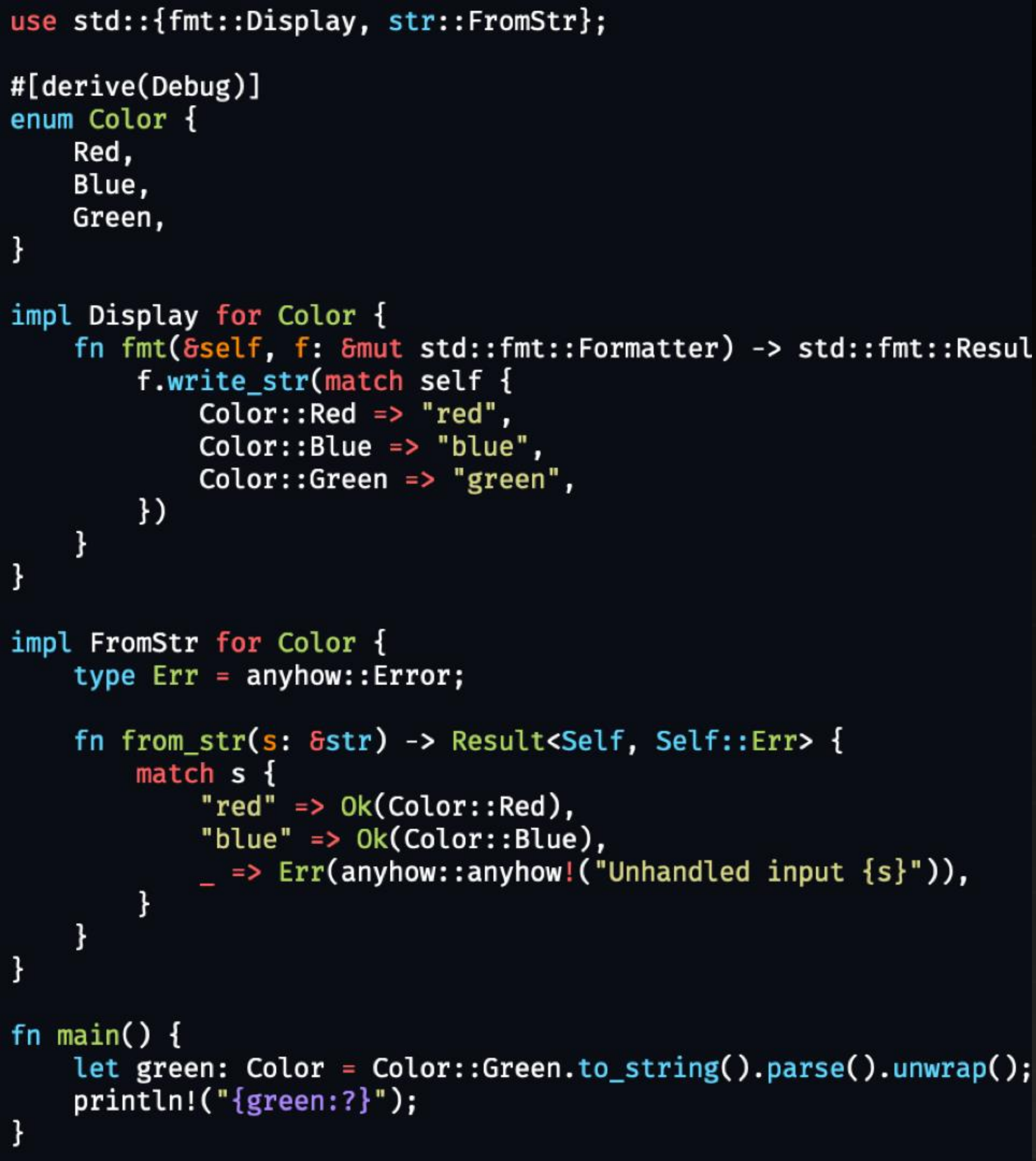
Some people fail to see the advantage of sum types (though that IME is rare)

Pattern matching

- Often tied into sum types, but not always
- Already understood by most programmers (switch statements)
- Easy to explain “switch on steroids”

Auto-Deriving

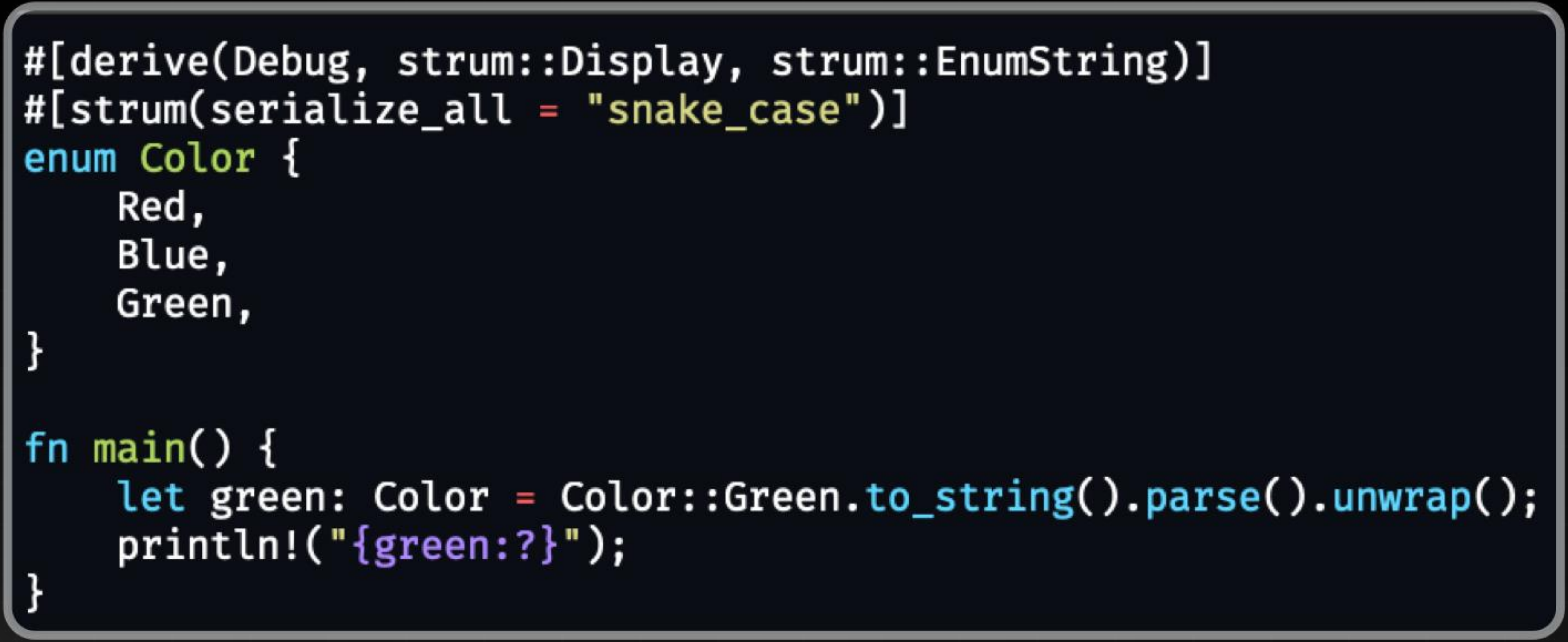
- `Haskell's deriving, Rust's #[derive(…)]`
- Haskell: extend with generics, TH, others
- Rust: proc macros
    - Removes boilerplate
    - Easier to write code
    - Easier to maintain code
    - Avoid bugs (next slide)



```rust
use std::{fmt::Display, str::FromStr};

#[derive(Debug)]
enum Color {
    Red,
    Blue,
    Green,
}

impl Display for Color {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Resul
        f.write_str(match self {
            Color::Red => "red",
            Color::Blue => "blue",
            Color::Green => "green",
        })
    }
}

impl FromStr for Color {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s {
            "red" => Ok(Color::Red),
            "blue" => Ok(Color::Blue),
            _ => Err(anyhow::anyhow!("Unhandled input {s}")),
        }
    }
}

fn main() {
    let green: Color = Color::Green.to_string().parse().unwrap();
    println!("{green:?}");
}
```

Spot the bug

www.fpblock.com



```rust
#[derive(Debug, strum::Display, strum::EnumString)]
#[strum(serialize_all = "snake_case")]
enum Color {
    Red,
    Blue,
    Green,
}

fn main() {
    let green: Color = Color::Green.to_string().parse().unwrap();
    println!("{green:?}");
}
```

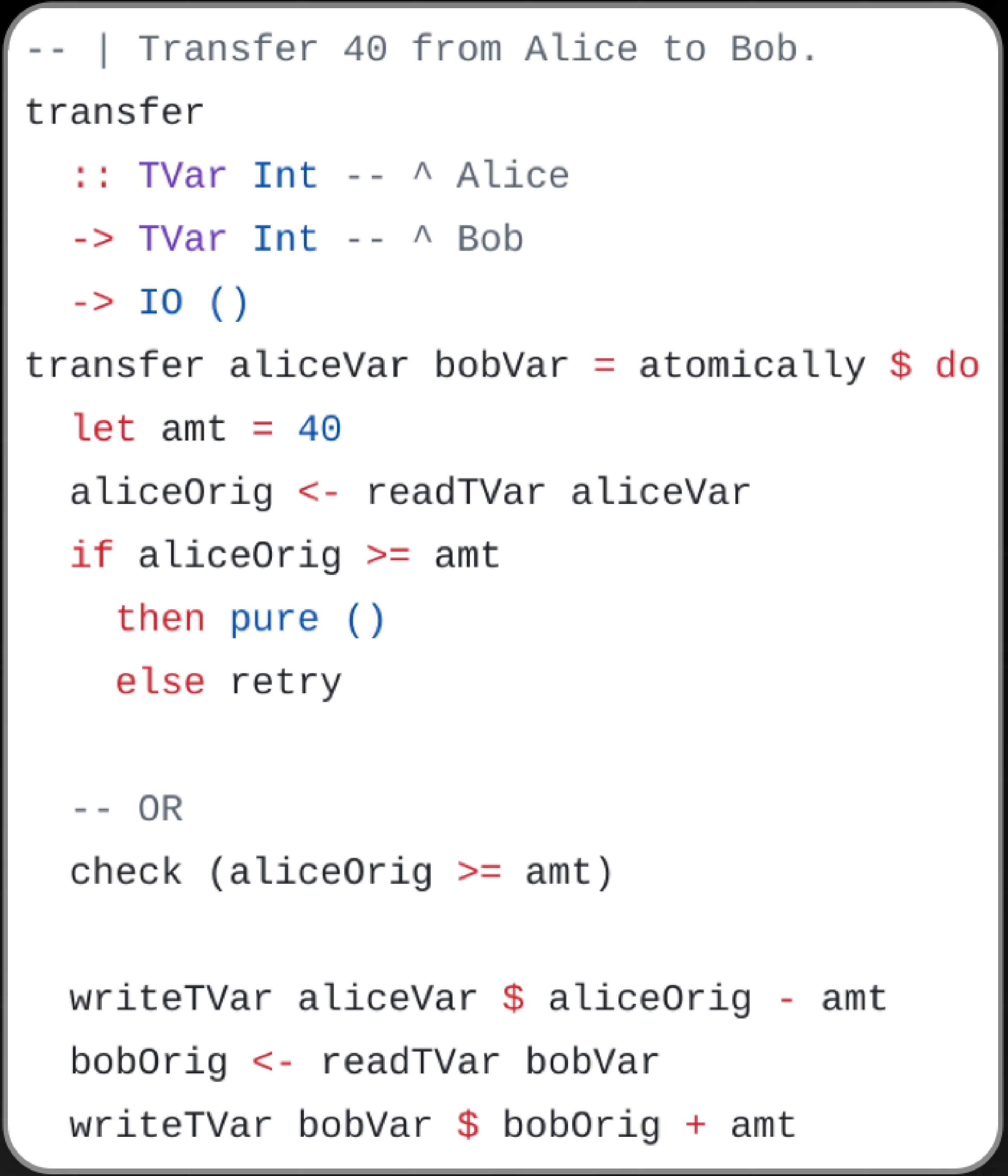
Better, Shorter, Safer

Great But Niche

There's no free lunch

Software Transactional Memory

- The absolute best way to do shared-memory concurrency
- Easy to learn
- Solves a real, well understood problem (data races)
- Performance is Good Enough, sometimes better than alternatives
- But the big catch: it only works well in a pure programming language



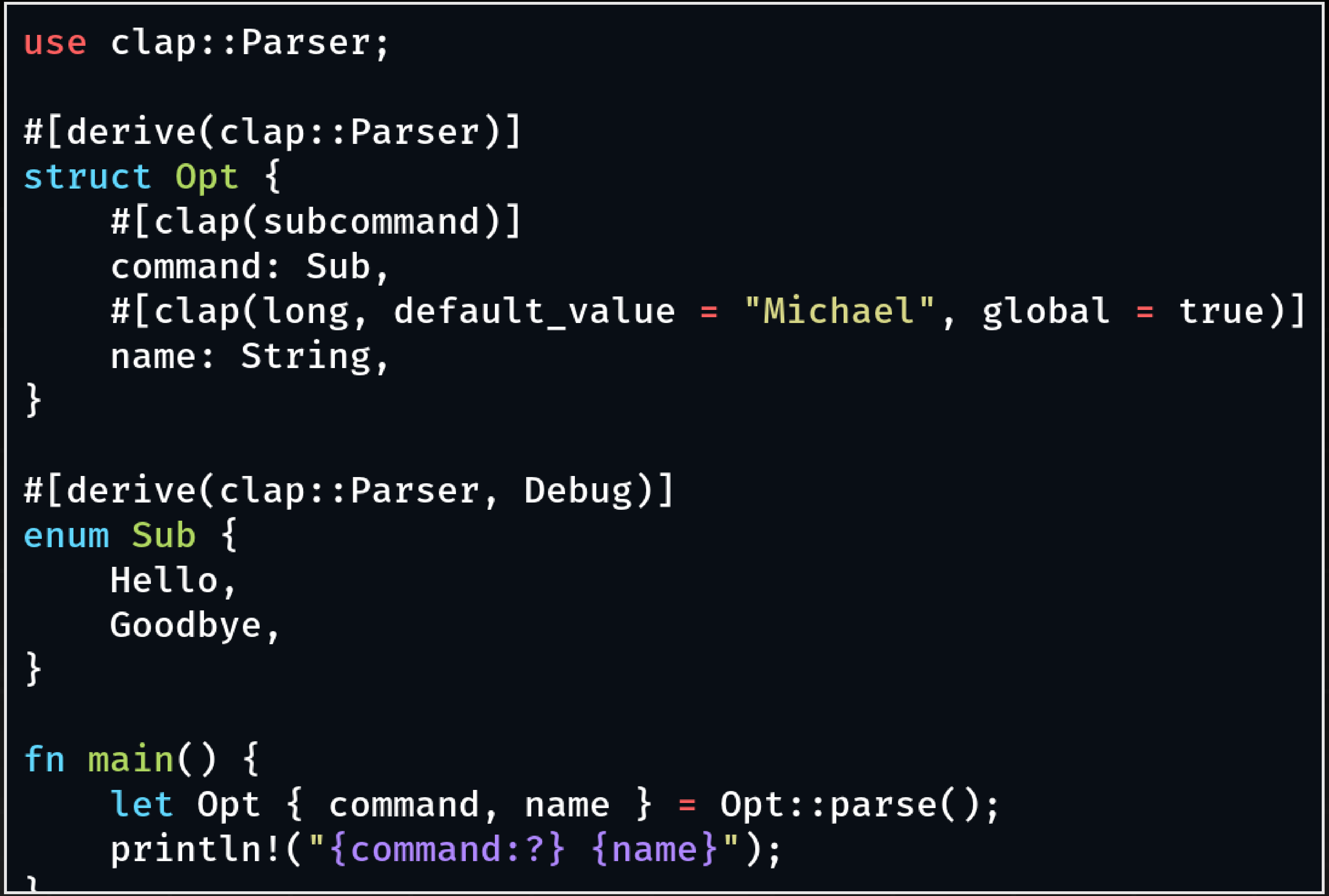
```haskell
-- | Transfer 40 from Alice to Bob.
transfer
  :: TVar Int -- ^ Alice
  -> TVar Int -- ^ Bob
  -> IO ()
transfer aliceVar bobVar = atomically $ do
  let amt = 40
  aliceOrig <- readTVar aliceVar
  if aliceOrig >= amt
    then pure ()
    else retry

  -- OR
  check (aliceOrig >= amt)

  writeTVar aliceVar $ aliceOrig - amt
  bobOrig <- readTVar bobVar
  writeTVar bobVar $ bobOrig + amt
```

Green Threads

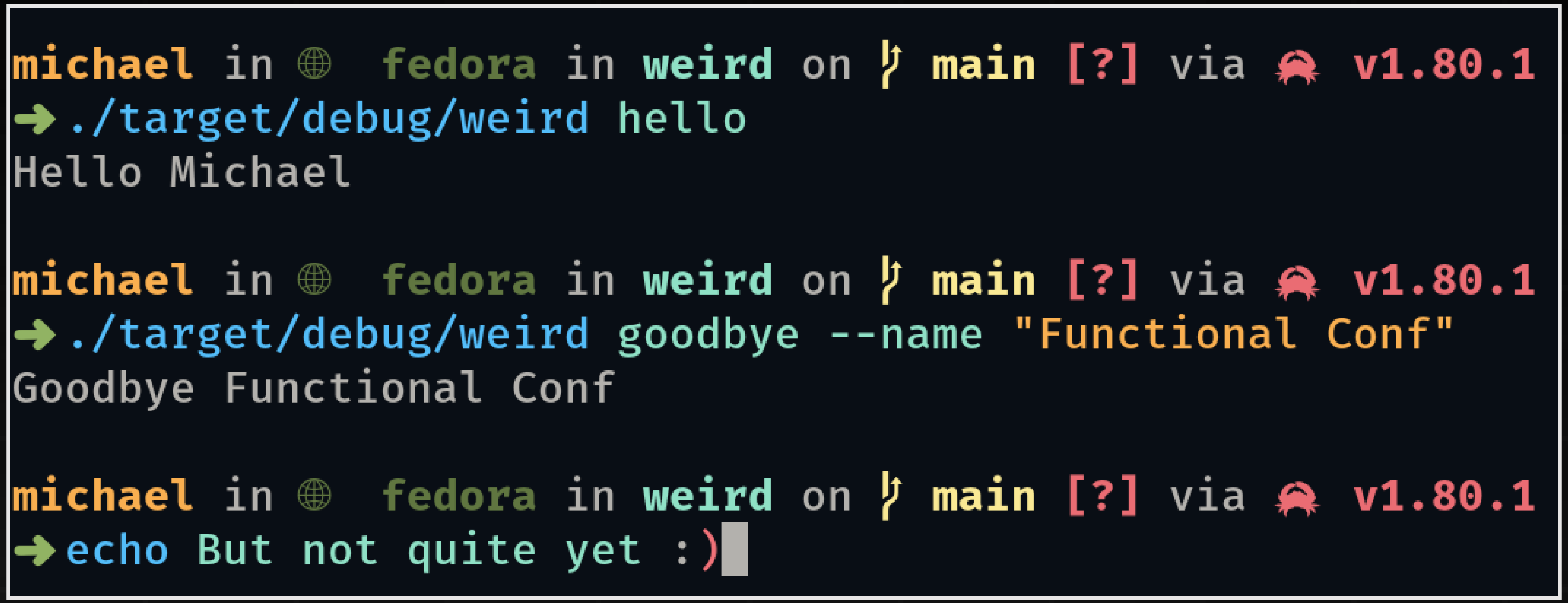
- Simplest way to write asynchronous I/O code
- Avoids the "function coloring" problem entirely
- Composes with existing error handling
- No need to litter code with async/await noise
- The catch
  - Plenty of languages don't support it!
  - Some languages (like Rust) intentionally avoid it because it conflicts with other goals

Macros/Metaprogramming/Codegen

- Auto-deriving is one example
- Serde and Clap are great Rust libraries leaning into this
- Again: shorter, safer code
- Downsides
  - Longer compile times
  - Less transparency about your code
  - Difficult to make modifications
- Code on next two slides

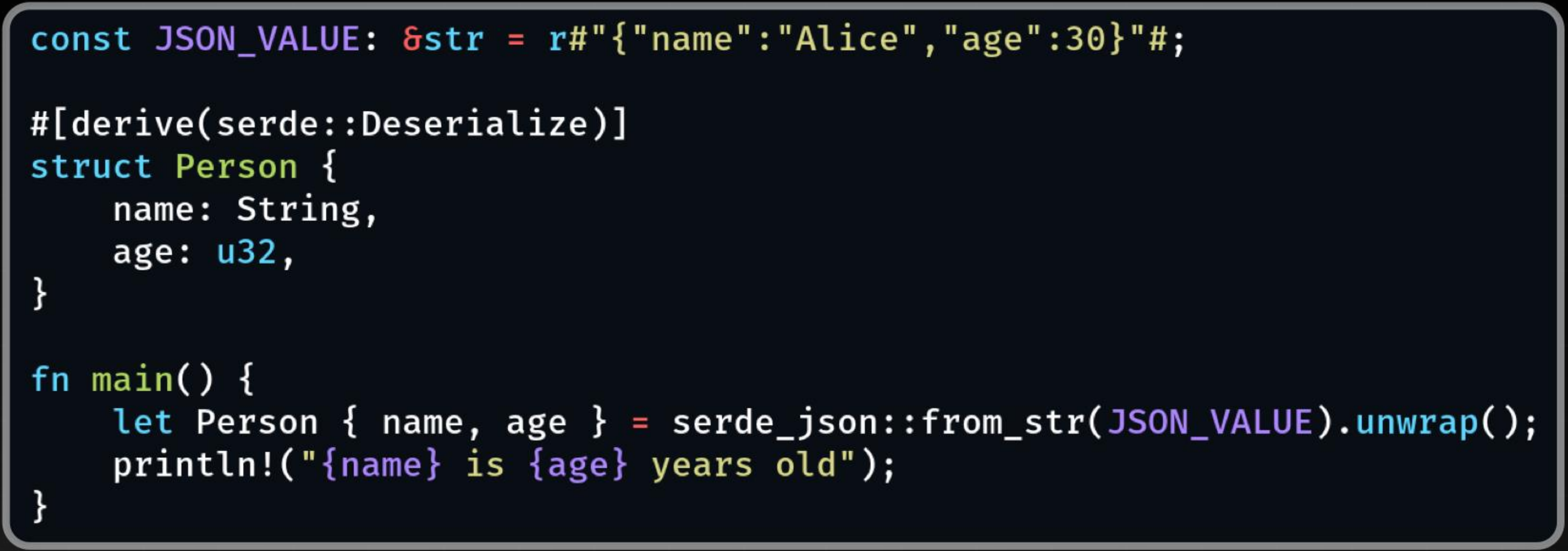


```rust
use clap::Parser;

#[derive(clap::Parser)]
struct Opt {
    #[clap(subcommand)]
    command: Sub,
    #[clap(long, default_value = "Michael", global = true)]
    name: String,
}

#[derive(clap::Parser, Debug)]
enum Sub {
    Hello,
    Goodbye,
}

fn main() {
    let Opt { command, name } = Opt::parse();
    println!("{command:?} {name}");
}
```



```
michael in 🌐  fedora in weird on  main [?] via 🦀 v1.80.1
➜ ./target/debug/weird hello
Hello Michael

michael in 🌐  fedora in weird on  main [?] via 🦀 v1.80.1
➜ ./target/debug/weird goodbye --name "Functional Conf"
Goodbye Functional Conf

michael in 🌐  fedora in weird on  main [?] via 🦀 v1.80.1
➜ echo But not quite yet :)
```

JSON Parsing with Serde



```rust
const JSON_VALUE: &str = r#"{"name":"Alice","age":30}"#;

#[derive(serde::Deserialize)]
struct Person {
    name: String,
    age: u32,
}

fn main() {
    let Person { name, age } = serde_json::from_str(JSON_VALUE).unwrap();
    println!("{name} is {age} years old");
}
```

JSON Parsing with Serde

Monads

- Good general-purpose way to think about coding
- Identifies a recurring pattern, recommends better APIs
- Solves a surprisingly large class of different problems (I/O, error handling, async)
- I've even had JavaScript developers describe Rust as "monadic" :)
- But
    - Deservedly or not, considered very complicated (scares people away)
    - Difficult to explain a value proposition – it doesn't directly solve a problem people know they have

Typeclasses / Traits

- Great feature, full stop
- Easy to understand (ignoring advanced features)
- Solves a real problem people understand (type safe code generalization)
- Alternatives exist (interfaces, modules, dictionary passing), IMO all inferior
- Downside
    - Requires language support, some programmers simply can't use them
    - Many programmers in other languages don’t realize what they’re missing
    - Could arguably be moved to the “already won” section

The Controversial Parts

This is how flame wars start

Docker vs Nix vs Unikernels

- Three solutions to similar problems
- Docker is the clear winner by marketshare
- Docker won because:
  - Delivered 80% of the benefits (isolate my code from system-level libraries/files/etc)
  - With a fraction of the effort (use existing libraries, tools, don't rewrite the build system, etc.)
- I love the idea of unikernels, and want to play with them, but it's never been worth the effort
- I've only had negative interactions with Nix (yes, you can all flame me)

Millions of Syntactic Extensions

- Lots of people love 'em some code golf
- Two minor examples: Haskell's LambdaCase, Rust's let-else
- I’m absolutely in the ”get off my lawn" category
- I'm not a fan of adding lots of new ways to do the same thing
  - Benefit: code gets marginally shorter
  - Cost: more difficult to read code, decision point of how to write things constantly
- Almost everyone I've worked with probably disagrees with my stance

Laziness By Default, Referential Transparency

- Yes, they are two totally different concepts!
- In both cases:
  - There are strong reasons to like them
  - The choice is very different from other programming languages
  - The benefits are difficult to explain to someone in under 5 minutes
- Didn't I say STM was amazing because of purity? Yes.
  - Benefit(STM) > Cost(STM)
  - Benefit(STM) < Cost(STM) + Cost(Purity)

Honorable Mentions: The Positives

- REPL-based development (though I'm personally not a huge fan)
- Hot code loading
- Type driven development (typed holes, the undefined trick, etc.)
- Associated types
- Doctests
- Return type polymorphism (e.g. mempty :: Monoid m => m)
- Refinement types

Why? They add useful functionality, don't add a lot of cognitive overhead, compose well with other features.

Honorable Mentions : Don’t Be Weird

- Content-addressed code
- Linear typing
- Dependent types
- Uniqueness types
- GADTs
- Algebraic effects

Why? Benefits difficult to explain, require more cognitive overhead to learn and use.

They could be great in the right package (e.g. Rust's affine types for ownership)

Closing Thoughts

Put down the pitchforks please

The Novelty Budget

- Credit to Mark Wotton https://x.com/mwotton
- Similar to the 80/20 rule
- You can only use so much novel tech before your project will break
- Aka don't be too weird
- Weirdness introduces potential incompatibilities and other unknowns
- Don't be greedy! Choose 1-3 novel things, use standard technology elsewhere
- When your weird choices succeed, they'll become standard, and you can start adding new weird things
- I've personally seen way too many projects killed by being too weird

Conclusion

- Don't be a lemming, but don't be a revolutionary
- Find the highest value things you can change and focus on those
- We don't need 100% perfection in our tech stack
- Take the big wins, outcompete other players, demonstrate that your ideas work
- Soon enough you'll change the world