Introduction to Rust (Day 2)

Security concepts in Rust

Security concepts in Rust

Ownership

Ownership

n. The act, state, or right of possessing something.

- Ownership rules:
    - Each value in Rust has a variable that’s **called its *owner*.**
    - There can **only** be **one owner** at a time.
    - When the owner goes out of scope, the **value will be dropped.**

- One of the most important concepts in Rust.
    - Provide **memory and thread safety** guarantees at compile time
        - no runtime garbage collector needed!
    - **Prevent sharing an already deallocated variable**
        - prevent use after free, double moves, etc..

- Further readings:
    - The Rust book / What Is Ownership? - link
    - Rust doc 1.8.0 / Ownership - link
    - Rust By Example / Ownership and moves - link
    - Learning Rust / Ownership - link
    - 28 Days of Rust — Part 1: Ownership and the Borrow Checker - link
    - Ownership in Rust - part#1, part#2

RUST IS OBSESSED WITH OWNERSHIP

every VALUE BELONGS TO A VARIABLE

values are like pet hamsters, they need an owner

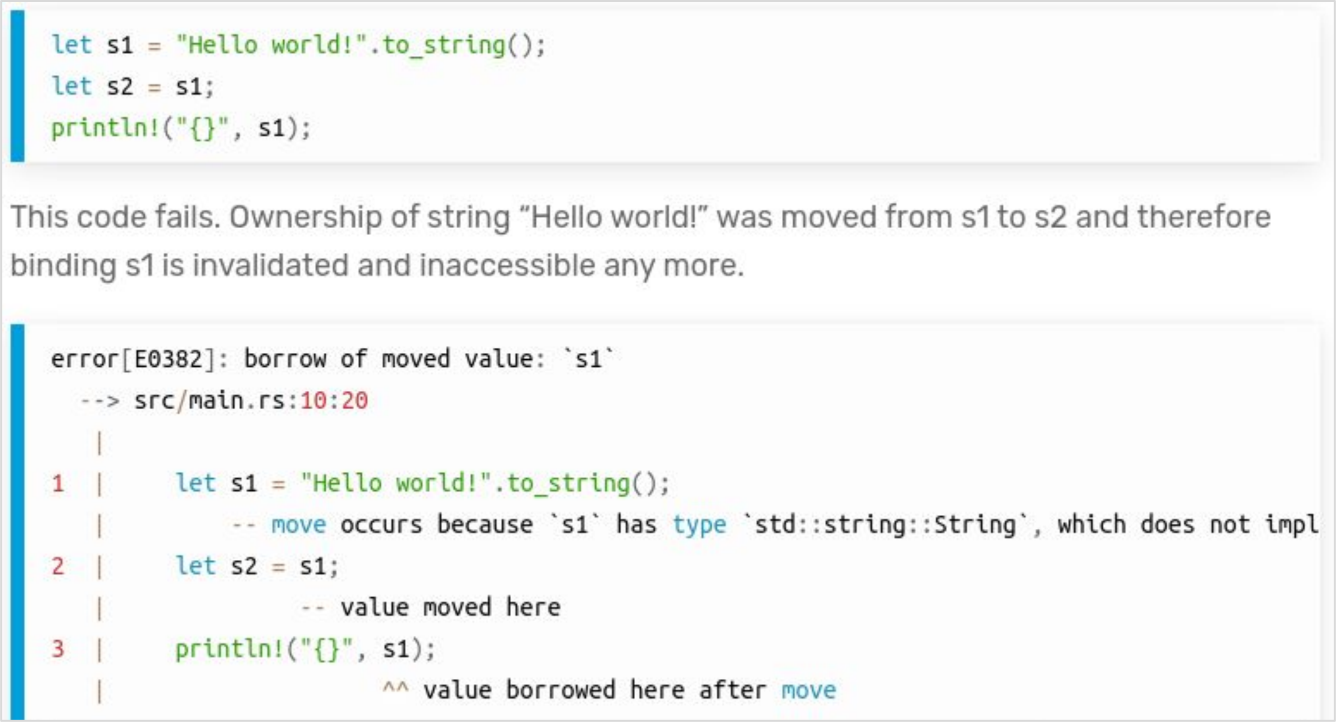
IF LeBron owns this hamster, then we assign LeBron to Shaq, OWNERSHIP MOVES to Shaq. LeBron no longer owns the hamster.

```
let leBron = 'hamster'
let shaq = leBron
```

* VALUES CAN ONLY HAVE ONE OWNER AT A TIME

VARIABLE
value
leBron
shaq

Ownership & Scope



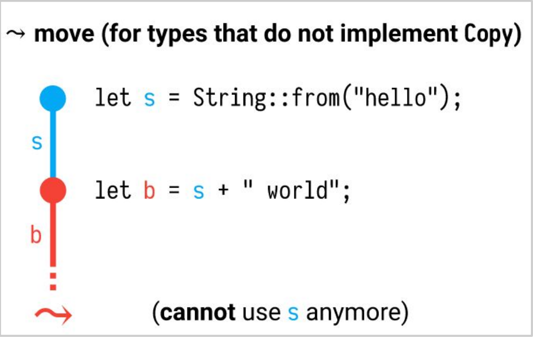
```rust
let s1 = "Hello world!".to_string();
let s2 = s1;
println!("{}", s1);
```

This code fails. Ownership of string "Hello world!" was moved from s1 to s2 and therefore binding s1 is invalidated and inaccessible any more.

```
error[E0382]: borrow of moved value: `s1`
  --> src/main.rs:10:20
   |
1  |     let s1 = "Hello world!".to_string();
   |         -- move occurs because `s1` has type `std::string::String`, which does not impl
2  |     let s2 = s1;
   |              -- value moved here
3  |     println!("{}", s1);
   |                    ^^ value borrowed here after move
   |
```

Move, Clone and Copy semantics

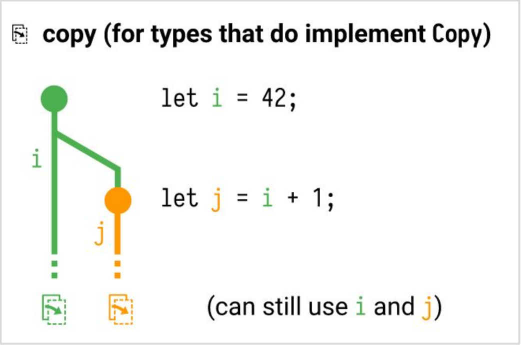
- `Move` trait*
  - When we transfer ownership, we’ve ‘moved’ the thing we refer to.
  - It’s the **default behavior for heap variable (like String, Vector, etc.)**
- `Clone` trait*
  - When a data is cloned, it **creates an exactly identical copy** of the data.
  - This variable is **independent** of the original data.
- `Copy` trait*
  - Basic variable types **stored in the stack implement Copy by default**
    - like int, float, bool, char, tuples (containing copy types)
  - All types that implements the Copy trait
    - will be copied bit-by-bit at the surface level, when the data is duplicated.
- Further readings:
  - Moving, Cloning, and Copying Coloring Books in Rust - link
  - Rust Move vs Copy - link
  - **Wrapper Types in Rust: Choosing Your Guarantees - link**



⤳ move (for types that do not implement Copy)

```
let s = String::from("hello");

let b = s + " world";
```

(**cannot** use s anymore)



⎘ copy (for types that do implement Copy)

```
let i = 42;

let j = i + 1;
```

(can still use i and j)

*trait is a language feature that tells the Rust compiler about functionality a type must provide.

Security concepts in Rust

Borrowing

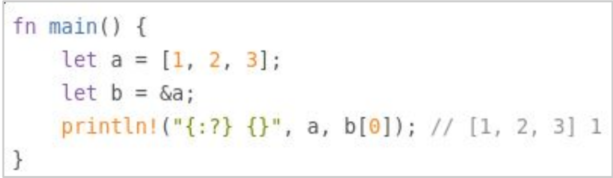
Borrowing

v. To receive something with the promise of returning it.

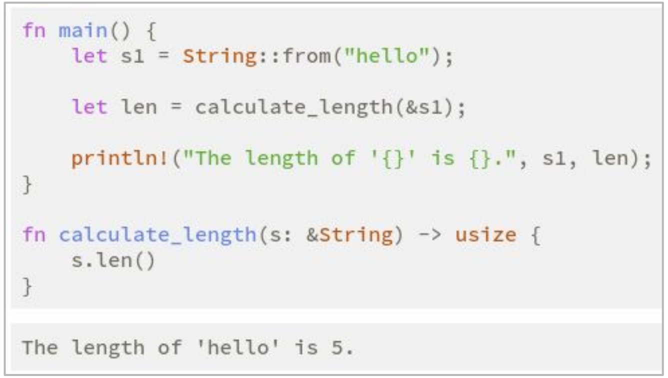
- Borrowing mechanism
  - **Two types of Borrowing, shared and mutable.**
  - Access to data without taking ownership over it.
  - Instead of passing objects by value (T)
    - objects can be **passed by reference (&T).**

- **Shared Borrowing (`&T`)**
  - A piece of data can be borrowed by a single or multiple users
    - but data should not be altered **(immutable)**

- Further readings:
  - The Rust book / References and Borrowing - link
  - Rust doc 1.8.0 / References and Borrowing - link
  - Rust by Example / Borrowing - link
  - Learning Rust / Borrowing - link
  - Fear not the Rust Borrow Checker - link



```
fn main() {
    let a = [1, 2, 3];
    let b = &a;
    println!("{:?} {}", a, b[0]); // [1, 2, 3] 1
}
```

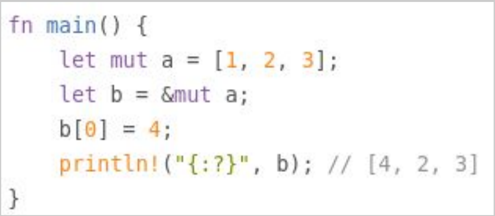


```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

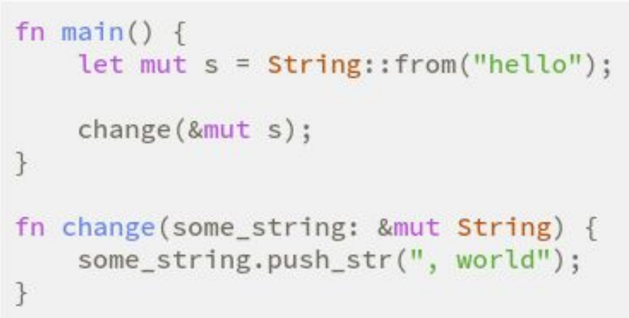
```
The length of 'hello' is 5.
```

Mutable borrowing

- **Mutable Borrowing (`&mut T`)**
  - A piece of data can be borrowed and altered by a single user **(mutable)**
    - but the data should not be accessible for any other users at that time.



```
fn main() {
    let mut a = [1, 2, 3];
    let b = &mut a;
    b[0] = 4;
    println!("{:?}", b); // [4, 2, 3]
}
```



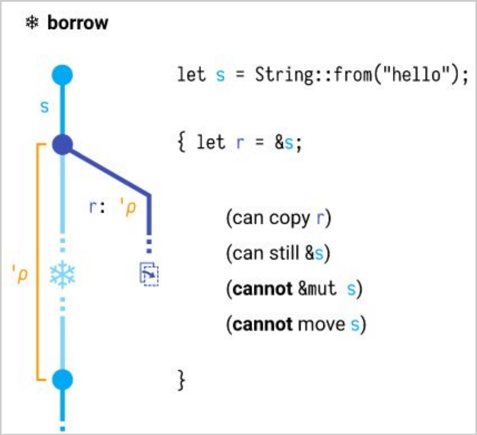
```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

- The rules of borrowing/references:
  - You cannot borrow a mutable reference from an immutable object
  - You **cannot borrow more than one mutable reference**
    - You can borrow multiple immutable references
  - **A mutable and an immutable reference cannot exist simultaneously**
  - References must always be valid.
  - The lifetime of a borrowed reference must end before the lifetime from the owner object

Shared vs Mutable references

- Shared references - `&T`
  - **nonexclusive** control (can be Copy)
  - **immutable**, like bindings.
  - cannot Move referent
  - must end before referent’s lifetime

- Mutable references - `&mut T`
  - **exclusive** (can be Move)
  - **mutable**, like bindings.
  - cannot Move referent
  - must end before referent’s lifetime



❄ **borrow**

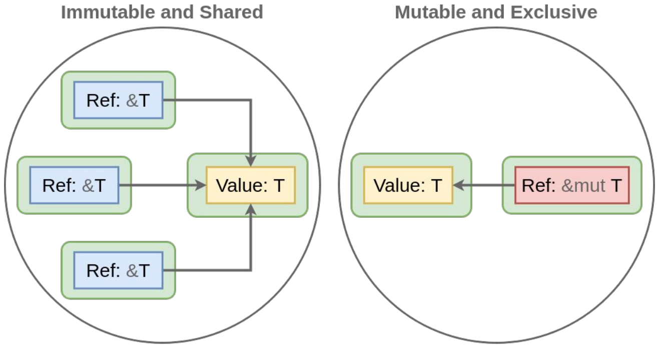
```
let s = String::from("hello");

{ let r = &s;

}
```

(can copy r)

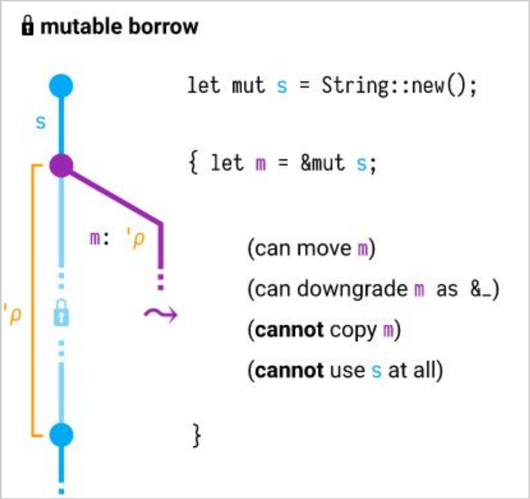
(can still &s)

(**cannot** &mut s)

(**cannot** move s)



(Exclusivity guaranteed at compile time)



🔒 **mutable borrow**

```
let mut s = String::new();

{ let m = &mut s;

}
```

(can move m)

(can downgrade m as &_)

(**cannot** copy m)

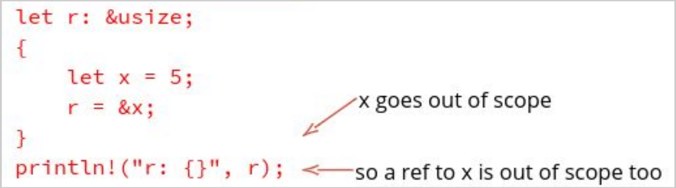
(**cannot** use s at all)

Security concepts in Rust

Lifetimes

Lifetimes and scopes

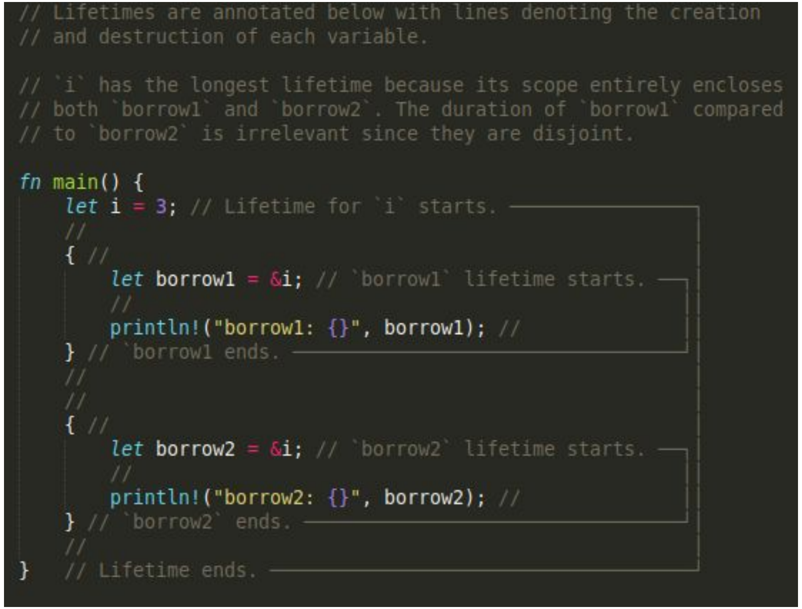
- Lifetimes is the mechanism that **tags a piece of code with a life scope.**
  - This information is used by the compiler to set how long referenced resources should be alive.
  - The Lifetime is the **length of time a variable is usable**



```rust
let r: &usize;
{
    let x = 5;
    r = &x;
}
println!("r: {}", r);
```

x goes out of scope

so a ref to x is out of scope too



```rust
// Lifetimes are annotated below with lines denoting the creation
// and destruction of each variable.

// `i` has the longest lifetime because its scope entirely encloses
// both `borrow1` and `borrow2`. The duration of `borrow1` compared
// to `borrow2` is irrelevant since they are disjoint.

fn main() {
    let i = 3; // Lifetime for `i` starts. ────────────────┐
    //                                                      │
    { //                                                    │
        let borrow1 = &i; // `borrow1` lifetime starts. ──┐│
        //                                                ││
        println!("borrow1: {}", borrow1); //              ││
    } // `borrow1 ends. ──────────────────────────────────┘│
    //                                                      │
    //                                                      │
    { //                                                    │
        let borrow2 = &i; // `borrow2` lifetime starts. ──┐│
        //                                                ││
        println!("borrow2: {}", borrow2); //              ││
    } // `borrow2` ends. ─────────────────────────────────┘│
    //                                                      │
}   // Lifetime ends. ─────────────────────────────────────┘
```

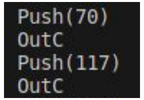
- Further readings:
  - Rust book / Validating References with Lifetimes - link
  - Rust doc 1.8.0 / Lifetimes - link
  - Rust By Example / Lifetimes - link
  - Learning Rust / Lifetimes - link
  - The Rustonomicon / Lifetimes - link
  - Understanding Rust Lifetimes - link
  - Lifetime Reference - link
  - Reading Lifetimes - link

Today’s project

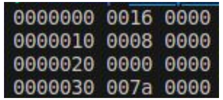
EXERCISE: Create a simple Stack-based Virtual Machine

- Goal:
  - Write a fully functional Stack Machine

- Features:
  - It should be able to **compile** small programs from *.svm* to *.bin*

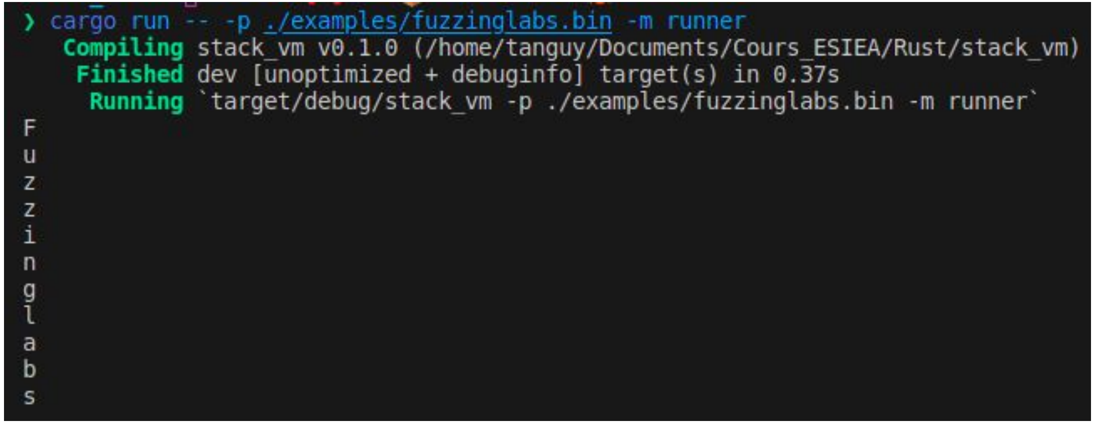


```
Push(70)
OutC
Push(117)
OutC
```



```
0000000 0016 0000
0000010 0008 0000
0000020 0000 0000
0000030 007a 0000
```

  - It should be able to **run/interpret** those programs
  - Programs should be able to take user **inputs** from **stdin**
  - Programs should be able to print **numbers** and **characters**



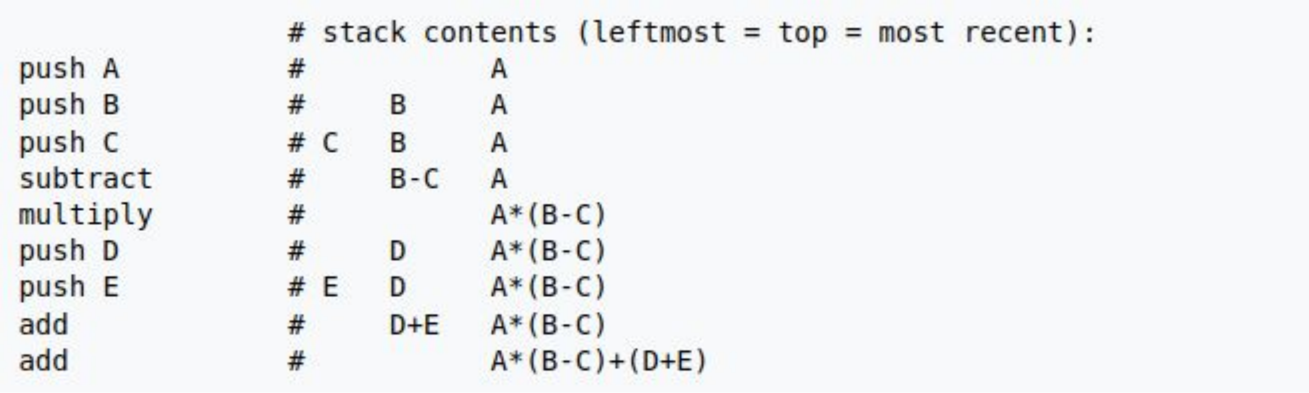
```
❯ cargo run -- -p ./examples/fuzzinglabs.bin -m runner
   Compiling stack_vm v0.1.0 (/home/tanguy/Documents/Cours_ESIEA/Rust/stack_vm)
    Finished dev [unoptimized + debuginfo] target(s) in 0.37s
     Running `target/debug/stack_vm -p ./examples/fuzzinglabs.bin -m runner`
F
u
z
z
i
n
g
l
a
b
s
```

EXERCISE #1

Compiler & Disassembler

REMINDER: Stack-based Virtual Machine

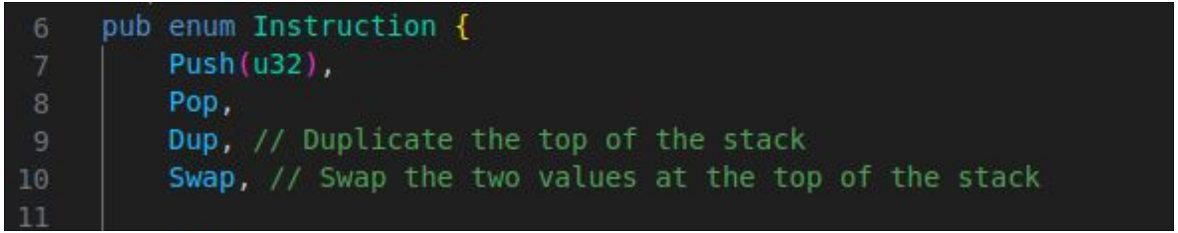
- Can be seen as a very simple processor emulation
- Virtual Machine that uses a **stack** as **memory** to store **data**
- In our case it uses a small number of instructions mostly to do math
- Similar to the REPL calculator you did in Java
- Wiki: https://en.wikipedia.org/wiki/Stack_machine



```
                # stack contents (leftmost = top = most recent):
push A          #           A
push B          #     B     A
push C          # C   B     A
subtract        #     B-C   A
multiply        #           A*(B-C)
push D          #     D     A*(B-C)
push E          # E   D     A*(B-C)
add             #     D+E   A*(B-C)
add             #           A*(B-C)+(D+E)
```

STEP 1-1: Instructions for the StackVm

- Steps:
  - Create a new project using *cargo*
  - Create a new file called *instruction.rs* in the *src* directory
  - Create an **enum** with **instructions** the vm will understand

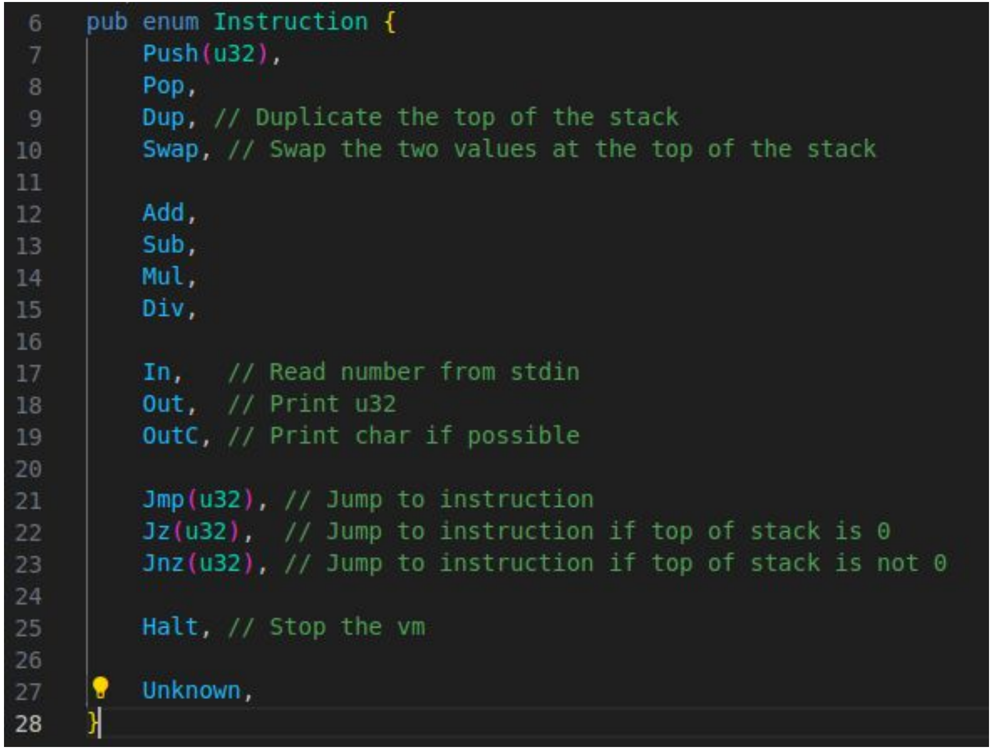


```
pub enum Instruction {
    Push(u32),
    Pop,
    Dup, // Duplicate the top of the stack
    Swap, // Swap the two values at the top of the stack

```

- Add instructions
  - Add, mul, div, jump, etc.

- Solution on the next slide

SOLUTION: Instructions for the StackVm



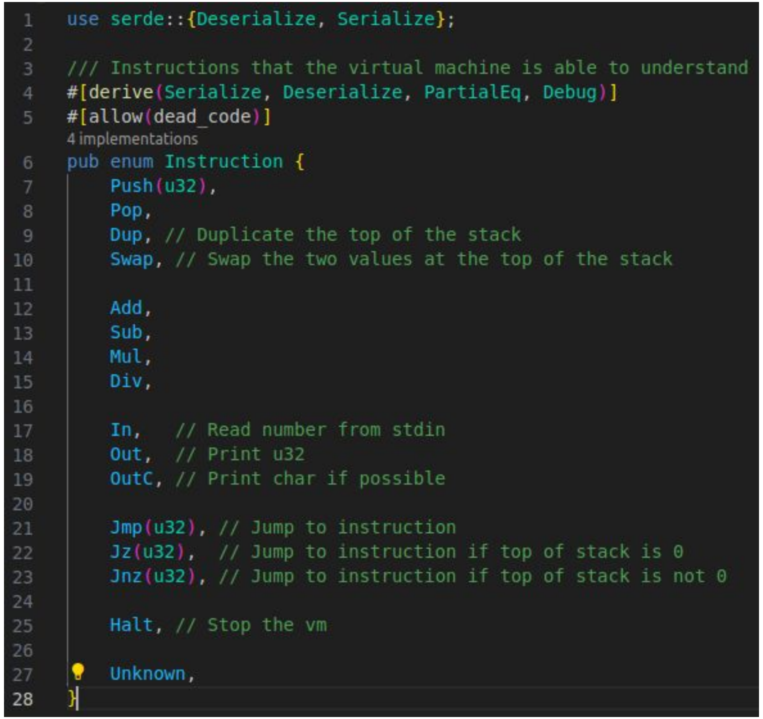
```rust
pub enum Instruction {
    Push(u32),
    Pop,
    Dup, // Duplicate the top of the stack
    Swap, // Swap the two values at the top of the stack

    Add,
    Sub,
    Mul,
    Div,

    In,   // Read number from stdin
    Out,  // Print u32
    OutC, // Print char if possible

    Jmp(u32), // Jump to instruction
    Jz(u32),  // Jump to instruction if top of stack is 0
    Jnz(u32), // Jump to instruction if top of stack is not 0

    Halt, // Stop the vm

    Unknown,
}
```

STEP 1-2: Serde for serialization and deserialization

- Serde
  - Framework for **serializing** and **deserializing** Rust data structures
  - Doc: https://serde.rs/derive.html

- Steps:
  - Read the documentation to find how to serialize and serialize the **Instruction** enum
  - Modify the code according to the documentation

- The Rust Playground
  - Allows you to experiment with Rust without installing anything or creating new projects
    - Run and share code snippets in your browser

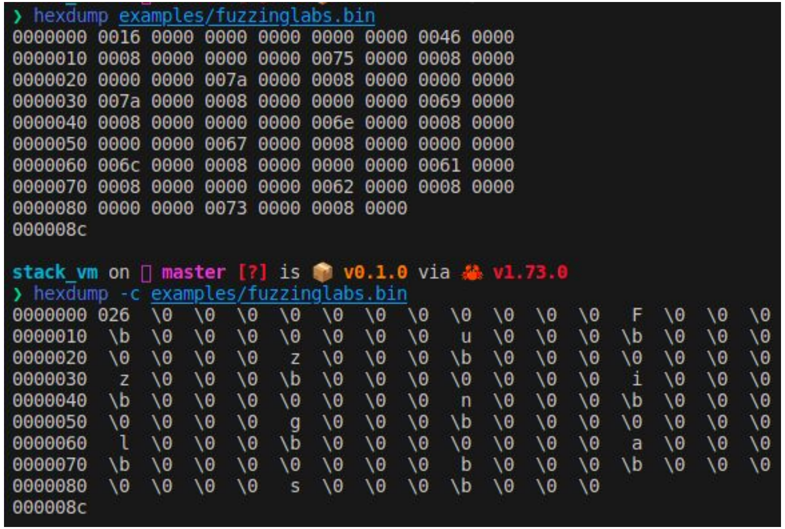


```rust
use serde::{Deserialize, Serialize};

/// Instructions that the virtual machine is able to understand
#[derive(Serialize, Deserialize, PartialEq, Debug)]
#[allow(dead_code)]
4 implementations
pub enum Instruction {
    Push(u32),
    Pop,
    Dup, // Duplicate the top of the stack
    Swap, // Swap the two values at the top of the stack

    Add,
    Sub,
    Mul,
    Div,

    In,   // Read number from stdin
    Out,  // Print u32
    OutC, // Print char if possible

    Jmp(u32), // Jump to instruction
    Jz(u32),  // Jump to instruction if top of stack is 0
    Jnz(u32), // Jump to instruction if top of stack is not 0

    Halt, // Stop the vm

    Unknown,
}
```

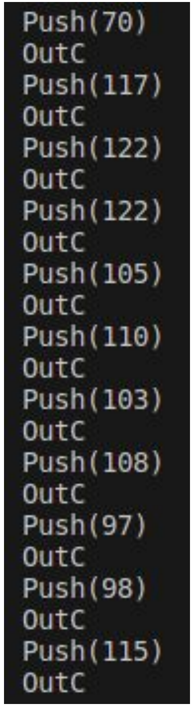
STEP 2: Bin file loader & printer

- Write a function to **read** a binary file and **deserialize**, call it **disassemble**
- Steps:
  - Create a new *disassembler.rs* file
  - Create the **disassemble** function
  - **Read** the file
  - **Deserialize** it
  - Return the **vector** of **instructions**
- Write a function to **pretty print** the **instructions**

- Solution on the next slide

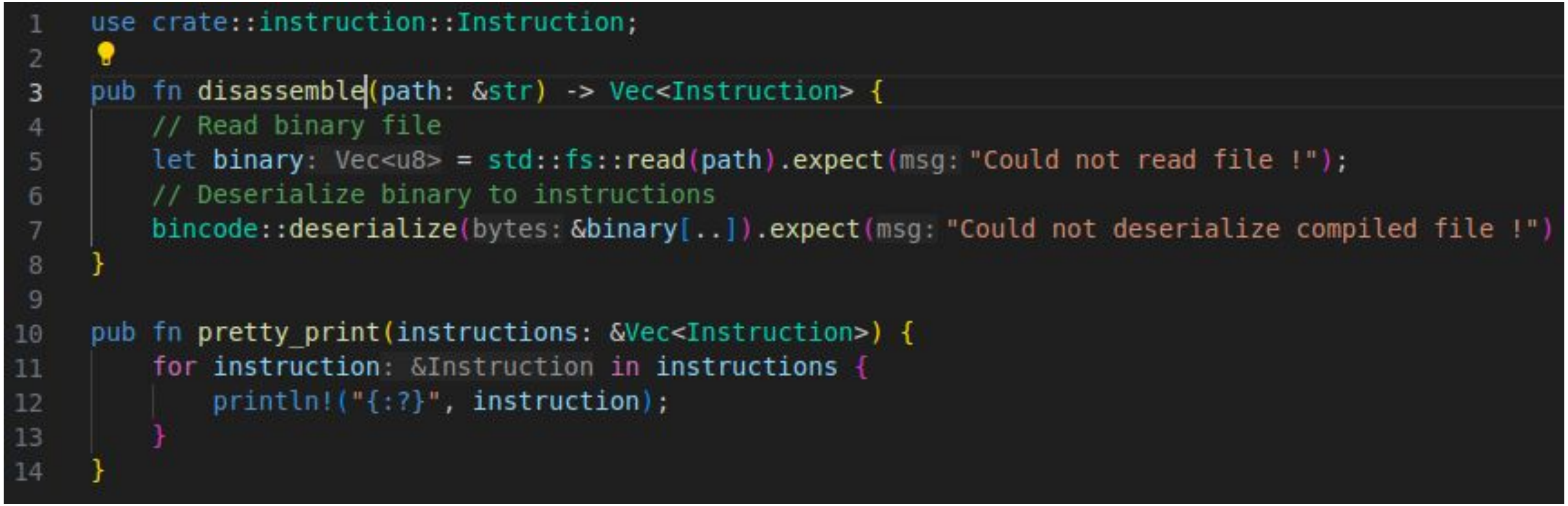


```
❯ hexdump examples/fuzzinglabs.bin
0000000 0016 0000 0000 0000 0000 0000 0046 0000
0000010 0008 0000 0000 0000 0075 0000 0008 0000
0000020 0000 0000 007a 0000 0008 0000 0000 0000
0000030 007a 0000 0008 0000 0000 0000 0069 0000
0000040 0008 0000 0000 0000 006e 0000 0008 0000
0000050 0000 0000 0067 0000 0008 0000 0000 0000
0000060 006c 0000 0008 0000 0000 0000 0061 0000
0000070 0008 0000 0000 0000 0062 0000 0008 0000
0000080 0000 0000 0073 0000 0008 0000
000008c

stack_vm on  master [?] is 📦 v0.1.0 via 🦀 v1.73.0
❯ hexdump -c examples/fuzzinglabs.bin
0000000 026  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0   F  \0  \0  \0
0000010  \b  \0  \0  \0  \0  \0  \0  \0   u  \0  \0  \0  \b  \0  \0  \0
0000020  \0  \0  \0  \0   z  \0  \0  \0  \b  \0  \0  \0  \0  \0  \0  \0
0000030   z  \0  \0  \0  \b  \0  \0  \0  \0  \0  \0  \0   i  \0  \0  \0
0000040  \b  \0  \0  \0  \0  \0  \0  \0   n  \0  \0  \0  \b  \0  \0  \0
0000050  \0  \0  \0  \0   g  \0  \0  \0  \b  \0  \0  \0  \0  \0  \0  \0
0000060   l  \0  \0  \0  \b  \0  \0  \0  \0  \0  \0  \0   a  \0  \0  \0
0000070  \b  \0  \0  \0  \0  \0  \0  \0   b  \0  \0  \0  \b  \0  \0  \0
0000080  \0  \0  \0  \0   s  \0  \0  \0  \b  \0  \0  \0
000008c
```



```
Push(70)
OutC
Push(117)
OutC
Push(122)
OutC
Push(122)
OutC
Push(105)
OutC
Push(110)
OutC
Push(103)
OutC
Push(108)
OutC
Push(97)
OutC
Push(98)
OutC
Push(115)
OutC
```

SOLUTION: Bin file loader & printer



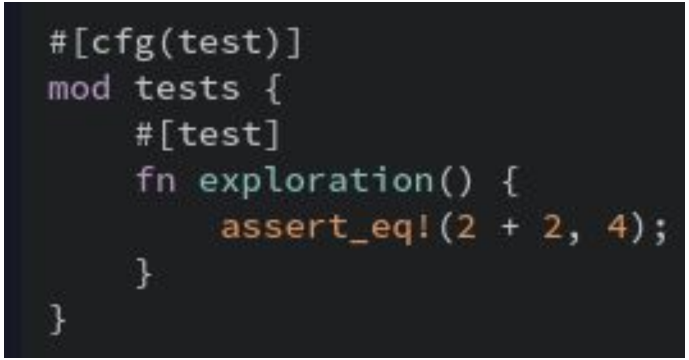
```rust
use crate::instruction::Instruction;

pub fn disassemble(path: &str) -> Vec<Instruction> {
    // Read binary file
    let binary: Vec<u8> = std::fs::read(path).expect(msg: "Could not read file !");
    // Deserialize binary to instructions
    bincode::deserialize(bytes: &binary[..]).expect(msg: "Could not deserialize compiled file !")
}

pub fn pretty_print(instructions: &Vec<Instruction>) {
    for instruction: &Instruction in instructions {
        println!("{:?}", instruction);
    }
}
```

STEP 3: Basic Instructions unit-tests

- Rust unit-test
  - https://doc.rust-lang.org/book/ch11-01-writing-tests.html

- Steps:
  - One to test **serialization**
    - `Vec<Instruction>` → `Vec<u8>`
  - One to test **deserialization**
    - `test.bin` → `Vec<Instruction>`

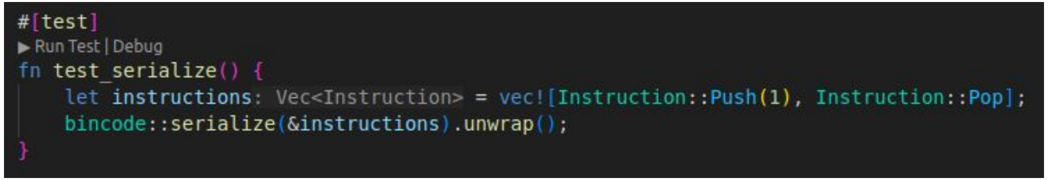
- Solution on the next slide



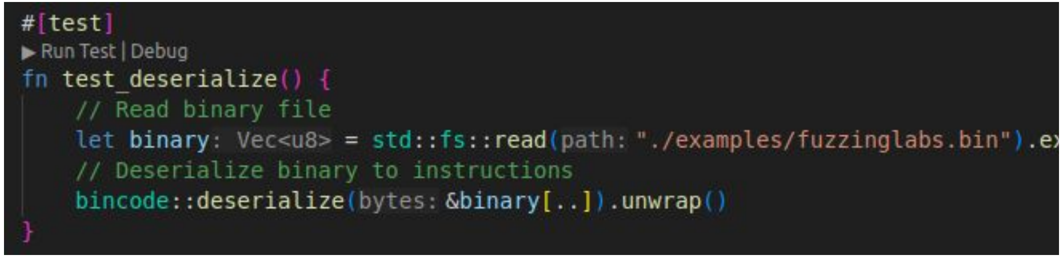
```
#[cfg(test)]
mod tests {
    #[test]
    fn exploration() {
        assert_eq!(2 + 2, 4);
    }
}
```

SOLUTION: Basic Instructions unit-tests

- One to test **serialization**
  - `Vec<Instruction>` → `Vec<u8>`
- One to test **deserialization**
  - `test.bin` → `Vec<Instruction>`



```
#[test]
▶ Run Test | Debug
fn test_serialize() {
    let instructions: Vec<Instruction> = vec![Instruction::Push(1), Instruction::Pop];
    bincode::serialize(&instructions).unwrap();
}
```

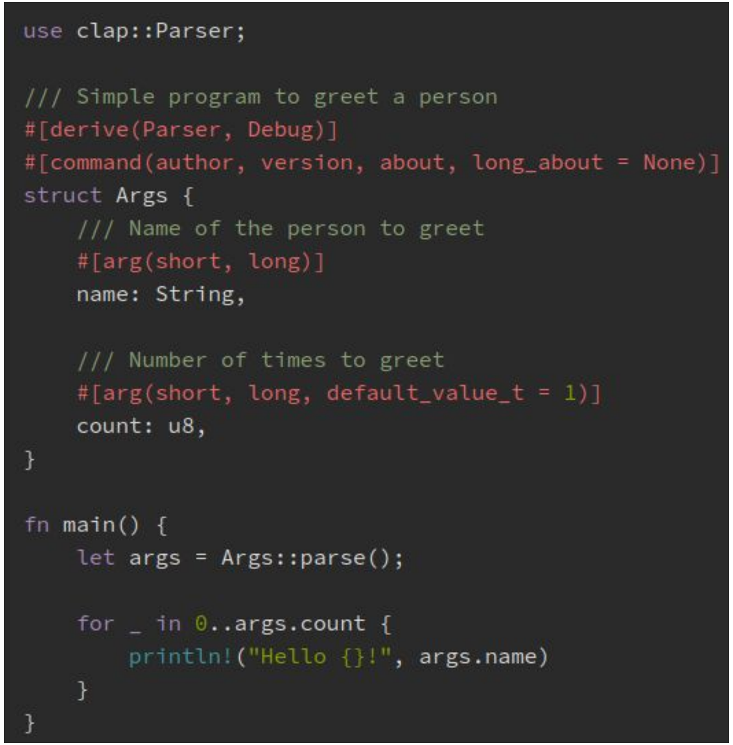


```
#[test]
▶ Run Test | Debug
fn test_deserialize() {
    // Read binary file
    let binary: Vec<u8> = std::fs::read(path: "./examples/fuzzinglabs.bin").ex
    // Deserialize binary to instructions
    bincode::deserialize(bytes: &binary[..]).unwrap()
}
```

STEP 4: Handling arguments

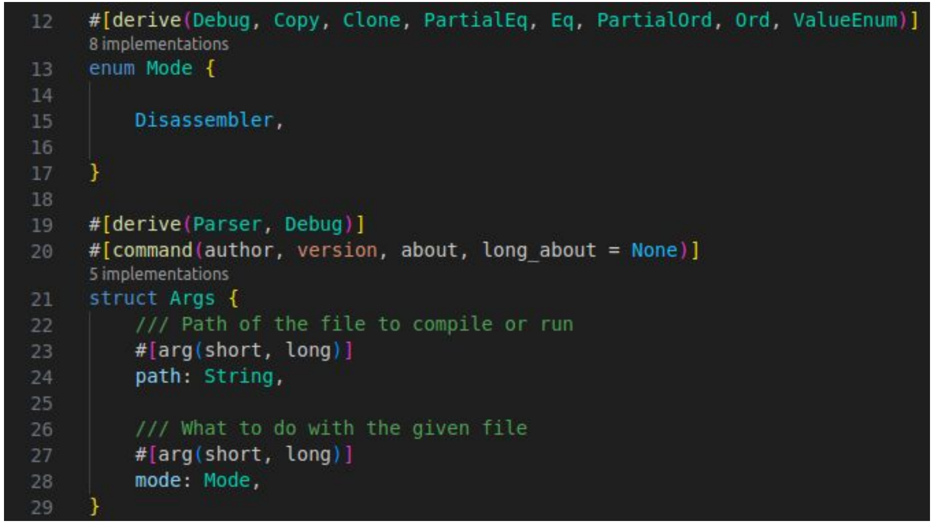
- clap
  - Command Line Argument Parser for Rust
  - Doc: https://docs.rs/clap/latest/clap/
- Steps:
  - Add the **clap** crate to your project
  - Handle the following **arguments**:
    - `./vm -p test.bin -m disassembler`
      - `-p Path`
      - `-m Mode`
  - Disassembler mode should call the **disassemble** function from the step-2

- Solution on the next slide

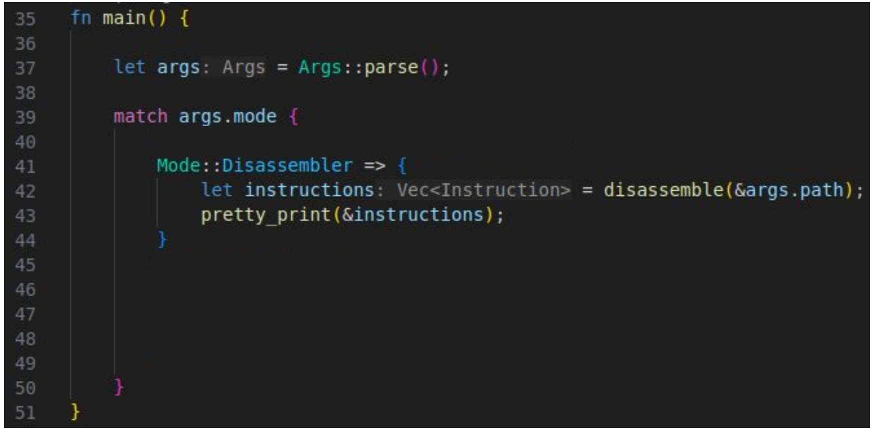


```rust
use clap::Parser;

/// Simple program to greet a person
#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
struct Args {
    /// Name of the person to greet
    #[arg(short, long)]
    name: String,

    /// Number of times to greet
    #[arg(short, long, default_value_t = 1)]
    count: u8,
}

fn main() {
    let args = Args::parse();

    for _ in 0..args.count {
        println!("Hello {}!", args.name)
    }
}
```

SOLUTION: Handling arguments



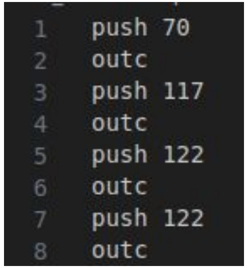
```rust
#[derive(Debug, Copy, Clone, PartialEq, Eq, PartialOrd, Ord, ValueEnum)]
8 implementations
enum Mode {

    Disassembler,

}

#[derive(Parser, Debug)]
#[command(author, version, about, long_about = None)]
5 implementations
struct Args {
    /// Path of the file to compile or run
    #[arg(short, long)]
    path: String,

    /// What to do with the given file
    #[arg(short, long)]
    mode: Mode,
}
```



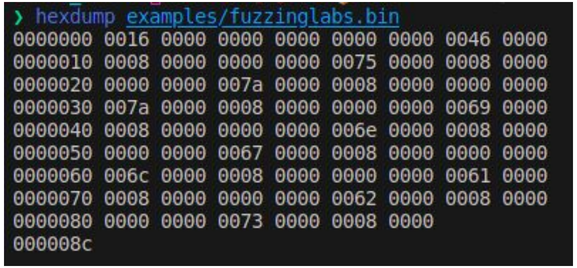
```rust
fn main() {

    let args: Args = Args::parse();

    match args.mode {

        Mode::Disassembler => {
            let instructions: Vec<Instruction> = disassemble(&args.path);
            pretty_print(&instructions);
        }





    }
}
```

STEP 5-1: Compiler, Basic instructions

- Steps:
  - Create another file called *compiler.rs*
  - Write a **compile** function that reads a svm file line by line and converts the content to a vector of **Instruction**
    - Split each line into tokens
    - Use a match to get the right instruction
  - Just write the code for the **push, pop, dup, swap, out** and **outc** instructions for now
  - **Serialize** the vector of instructions
  - **Write** the vector of instructions to the file by adding *.bin* to the path



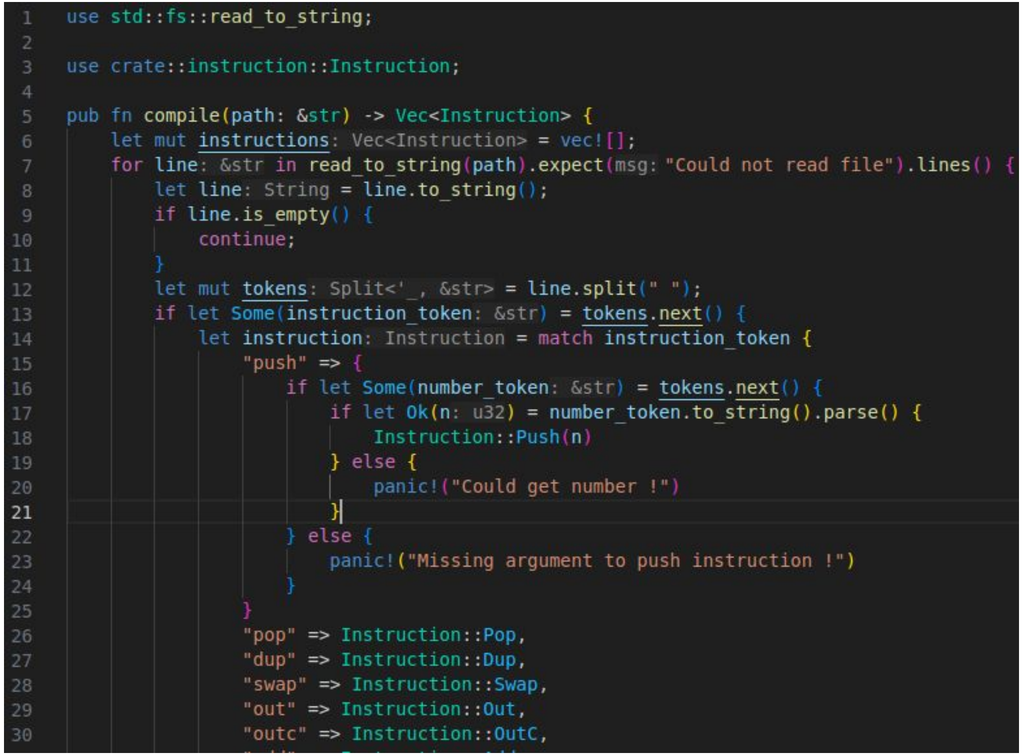
```
push 70
outc
push 117
outc
push 122
outc
push 122
outc
```



```
❯ hexdump examples/fuzzinglabs.bin
0000000 0016 0000 0000 0000 0000 0000 0046 0000
0000010 0008 0000 0000 0000 0075 0000 0008 0000
0000020 0000 0000 007a 0000 0008 0000 0000 0000
0000030 007a 0000 0008 0000 0000 0000 0069 0000
0000040 0008 0000 0000 0000 006e 0000 0008 0000
0000050 0000 0000 0067 0000 0008 0000 0000 0000
0000060 006c 0000 0008 0000 0000 0000 0061 0000
0000070 0008 0000 0000 0000 0062 0000 0008 0000
0000080 0000 0000 0073 0000 0008 0000
000008c
```

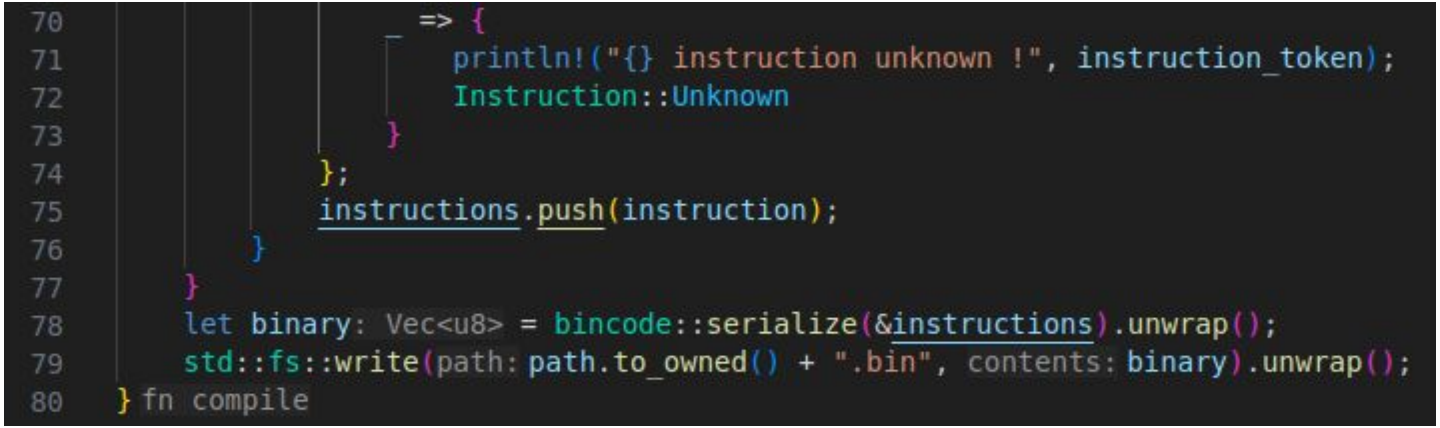
- Solution on the next slide

SOLUTION: Compiler, Basic instructions



```rust
use std::fs::read_to_string;

use crate::instruction::Instruction;

pub fn compile(path: &str) -> Vec<Instruction> {
    let mut instructions: Vec<Instruction> = vec![];
    for line: &str in read_to_string(path).expect(msg: "Could not read file").lines() {
        let line: String = line.to_string();
        if line.is_empty() {
            continue;
        }
        let mut tokens: Split<'_, &str> = line.split(" ");
        if let Some(instruction_token: &str) = tokens.next() {
            let instruction: Instruction = match instruction_token {
                "push" => {
                    if let Some(number_token: &str) = tokens.next() {
                        if let Ok(n: u32) = number_token.to_string().parse() {
                            Instruction::Push(n)
                        } else {
                            panic!("Could get number !")
                        }
                    } else {
                        panic!("Missing argument to push instruction !")
                    }
                }
                "pop" => Instruction::Pop,
                "dup" => Instruction::Dup,
                "swap" => Instruction::Swap,
                "out" => Instruction::Out,
                "outc" => Instruction::OutC,
```

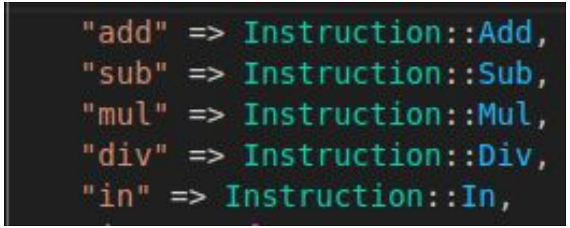
SOLUTION: Compiler, Basic instructions



```
                _ => {
                    println!("{} instruction unknown !", instruction_token);
                    Instruction::Unknown
                }
            };
            instructions.push(instruction);
        }
    }
    let binary: Vec<u8> = bincode::serialize(&instructions).unwrap();
    std::fs::write(path: path.to_owned() + ".bin", contents: binary).unwrap();
} fn compile
```

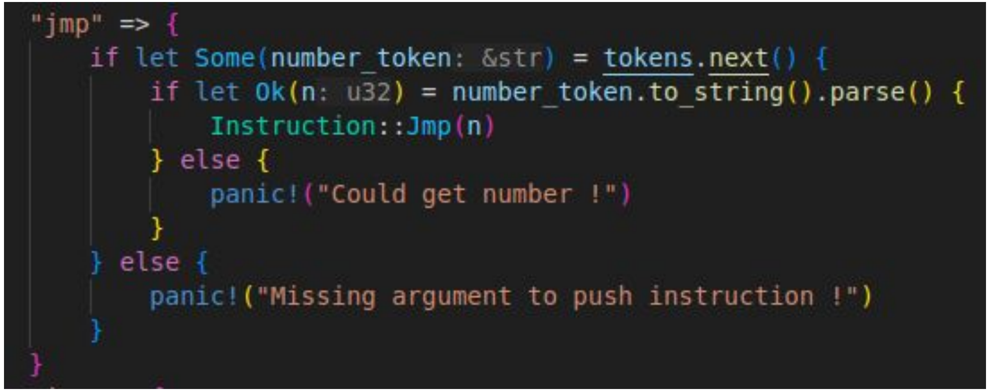
STEP 5-2: Compiler, Math instructions

- Steps:
  - Add the following instructions to the compiler
    - **Add, Sub, Mul, Div, In**



```
"add" => Instruction::Add,
"sub" => Instruction::Sub,
"mul" => Instruction::Mul,
"div" => Instruction::Div,
"in" => Instruction::In,
```

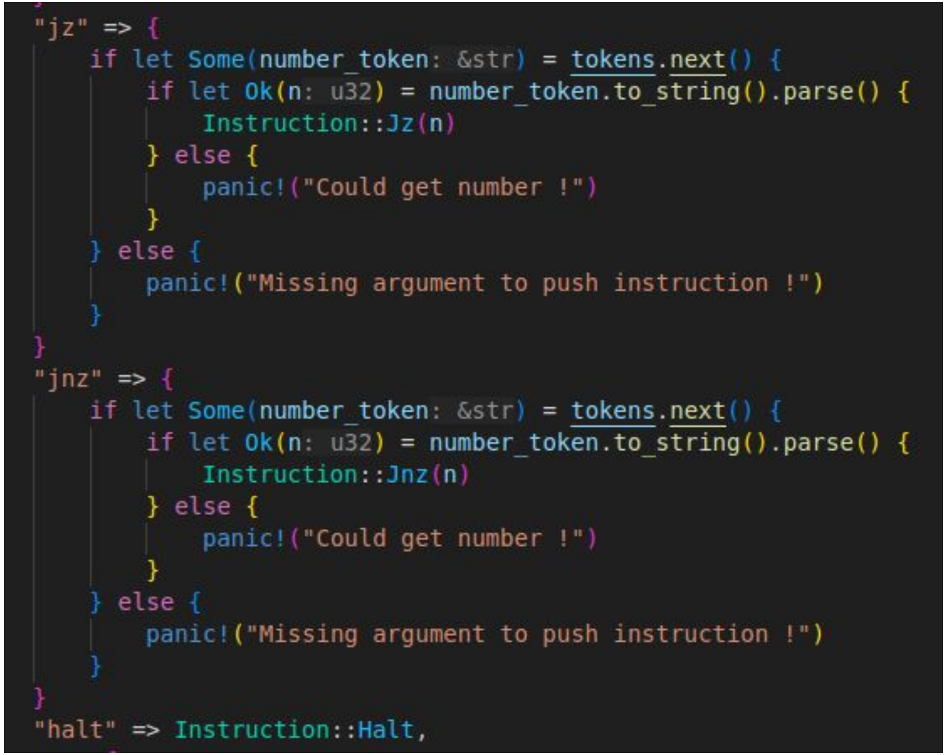
- Steps:
  - Add the following instructions to the compiler
    - **Jmp, Jz, Jnz, Halt**



```
"jmp" => {
    if let Some(number_token: &str) = tokens.next() {
        if let Ok(n: u32) = number_token.to_string().parse() {
            Instruction::Jmp(n)
        } else {
            panic!("Could get number !")
        }
    } else {
        panic!("Missing argument to push instruction !")
    }
}
```

- Solution on the next slide

SOLUTION: Compiler, Jump instructions



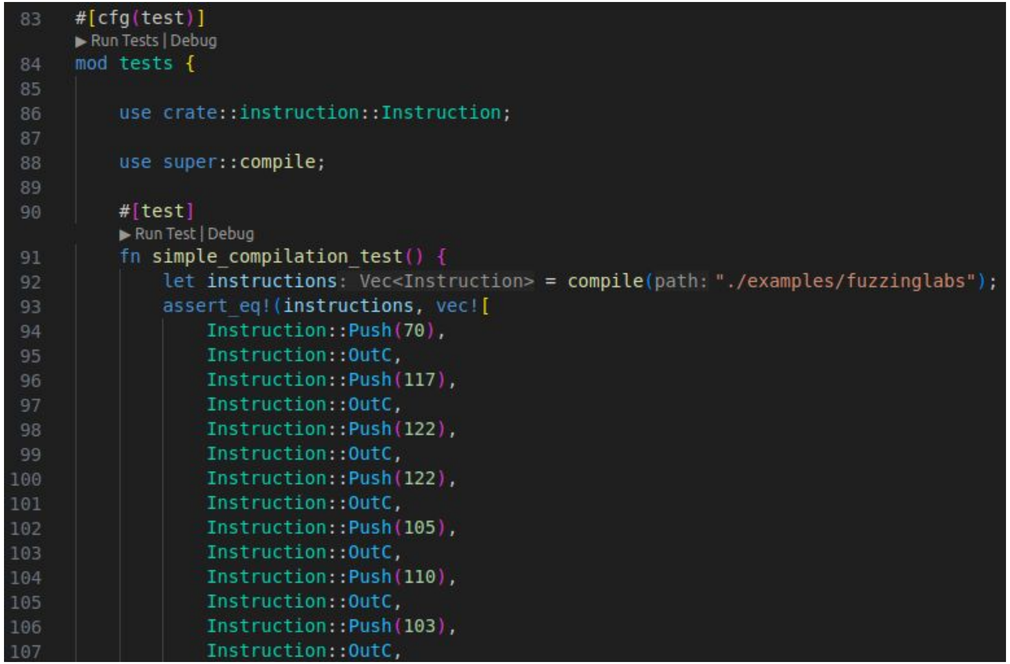
```
"jz" => {
    if let Some(number_token: &str) = tokens.next() {
        if let Ok(n: u32) = number_token.to_string().parse() {
            Instruction::Jz(n)
        } else {
            panic!("Could get number !")
        }
    } else {
        panic!("Missing argument to push instruction !")
    }
}
"jnz" => {
    if let Some(number_token: &str) = tokens.next() {
        if let Ok(n: u32) = number_token.to_string().parse() {
            Instruction::Jnz(n)
        } else {
            panic!("Could get number !")
        }
    } else {
        panic!("Missing argument to push instruction !")
    }
}
"halt" => Instruction::Halt,
```

STEP 5-3: Write tests

- `simple_compilation_test`
  - Write a **test** to check the **compilation** process

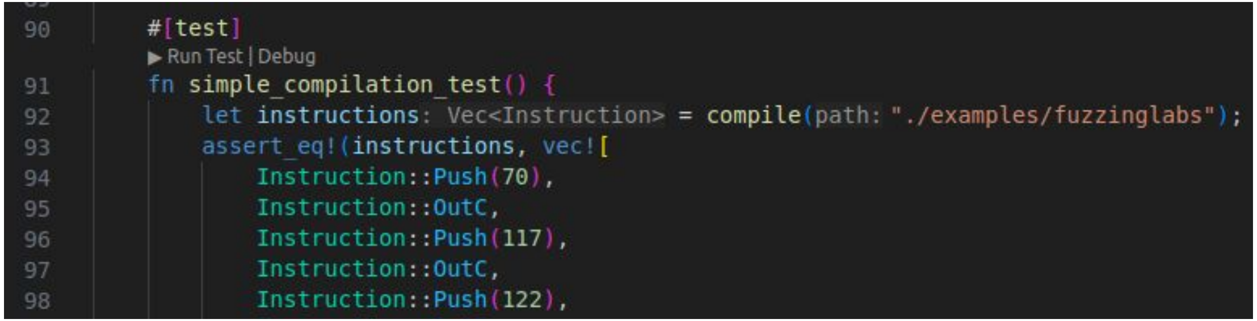
- `diff_test`
  - Write a test to check if:
    - **compile** file**.svm**
    - **disassemble** file**.bin**
    - does it gave the same result?

- Solution on the next slide

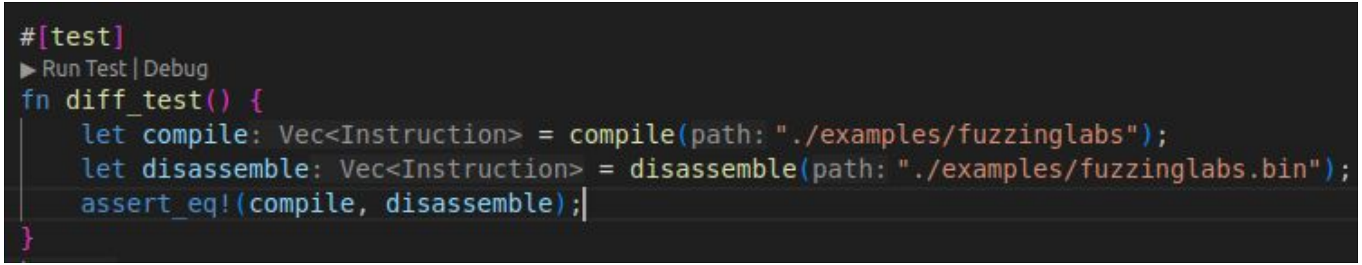


```rust
#[cfg(test)]
mod tests {

    use crate::instruction::Instruction;

    use super::compile;

    #[test]
    fn simple_compilation_test() {
        let instructions: Vec<Instruction> = compile(path: "./examples/fuzzinglabs");
        assert_eq!(instructions, vec![
            Instruction::Push(70),
            Instruction::OutC,
            Instruction::Push(117),
            Instruction::OutC,
            Instruction::Push(122),
            Instruction::OutC,
            Instruction::Push(122),
            Instruction::OutC,
            Instruction::Push(105),
            Instruction::OutC,
            Instruction::Push(110),
            Instruction::OutC,
            Instruction::Push(103),
            Instruction::OutC,
```

SOLUTION: Write tests



```
#[test]
fn simple_compilation_test() {
    let instructions: Vec<Instruction> = compile(path: "./examples/fuzzinglabs");
    assert_eq!(instructions, vec![
        Instruction::Push(70),
        Instruction::OutC,
        Instruction::Push(117),
        Instruction::OutC,
        Instruction::Push(122),
```



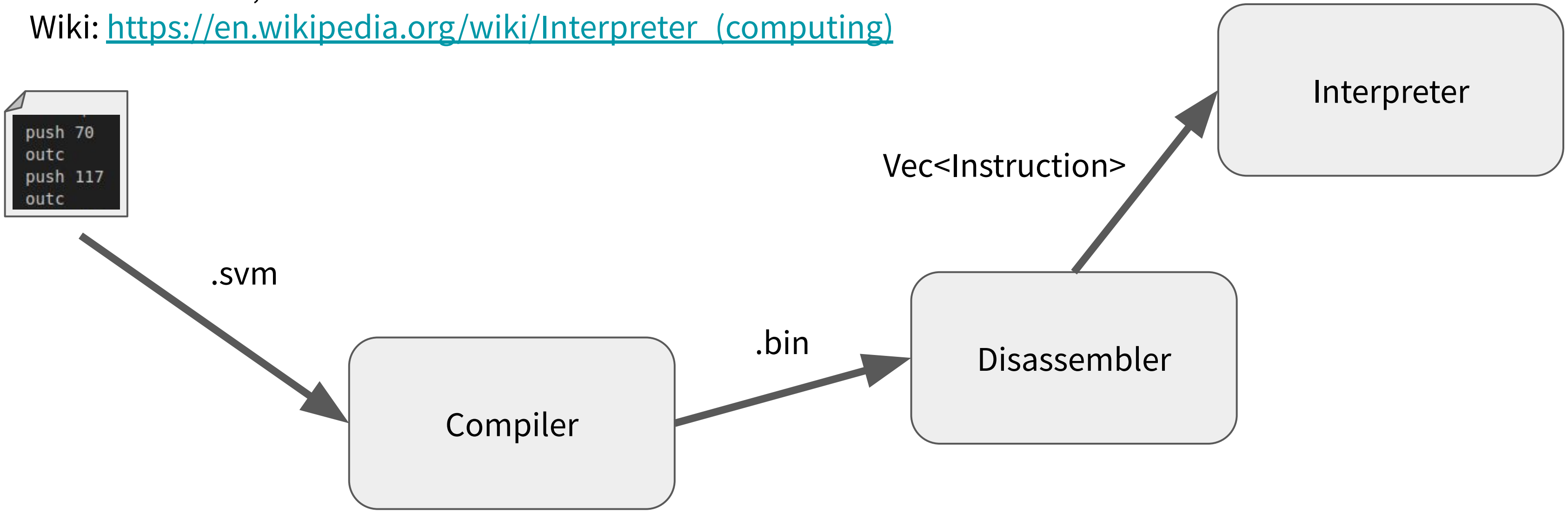
```
#[test]
fn diff_test() {
    let compile: Vec<Instruction> = compile(path: "./examples/fuzzinglabs");
    let disassemble: Vec<Instruction> = disassemble(path: "./examples/fuzzinglabs.bin");
    assert_eq!(compile, disassemble);
}
```

EXERCISE #2

Interpreter

What is an Interpreter?

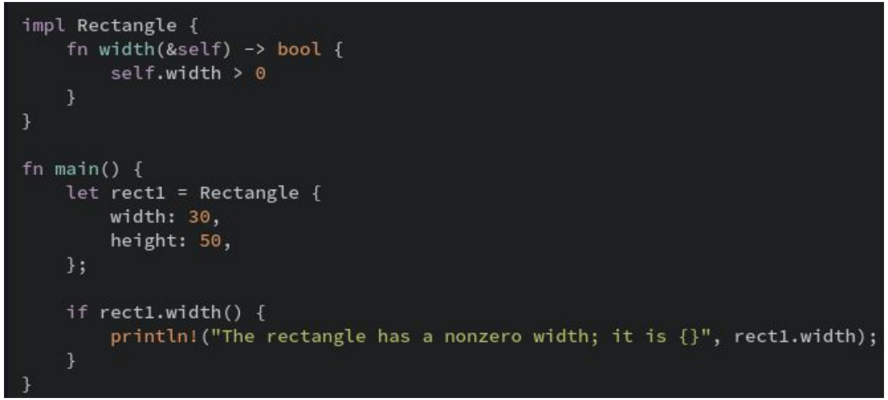
- Interpreter
  - Directly **execute** the code, without converting it into **machine code**
  - In our case, it will **execute** the vector of **Instruction**
- Wiki: https://en.wikipedia.org/wiki/Interpreter_(computing)



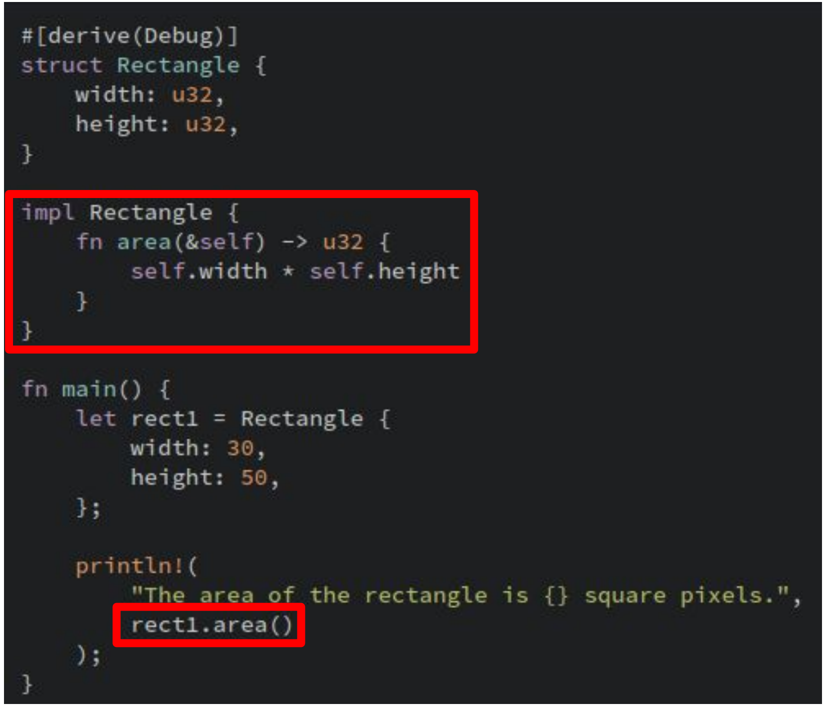
```
push 70
outc
push 117
outc
```

Methods for Structures in Rust

- Similar to function but defined in the **context** of a structure
- Use the keyword **impl** to create them
- They can have the same name as a **field**



```rust
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```



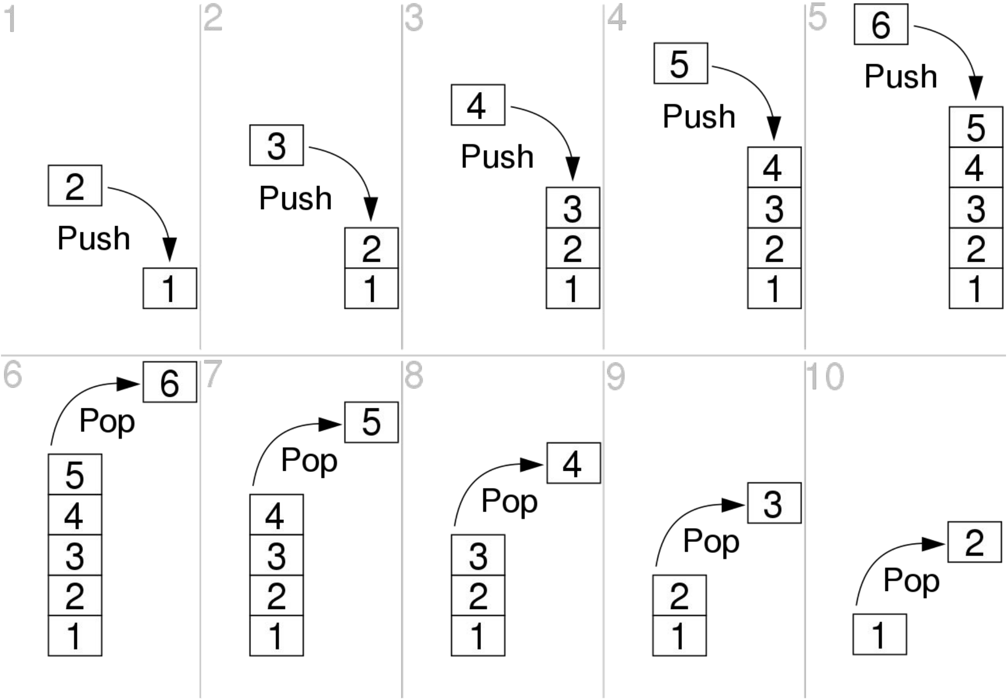
```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

STEP 6: Creating the stack structure

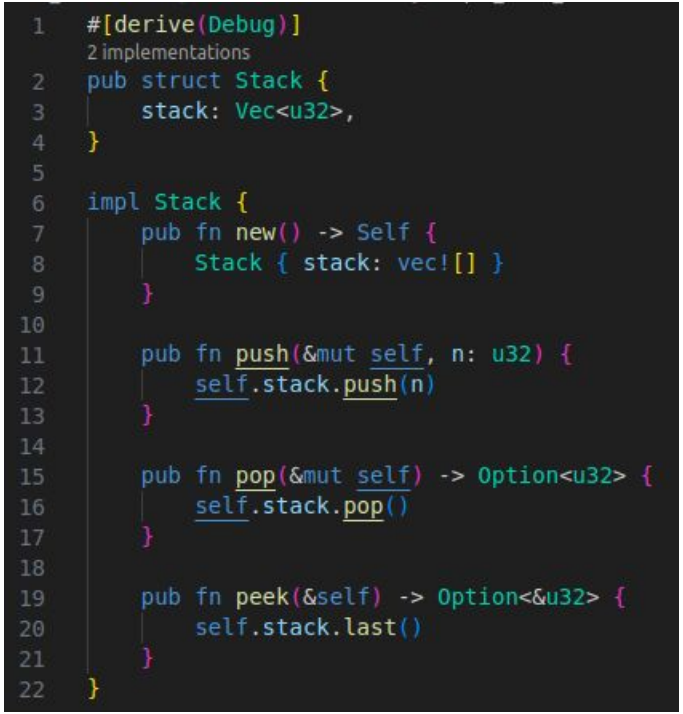
- Steps:
  - Create a new file called *stack.rs* in the *src* directory
  - Create a new structure called **Stack** with the necessary fields
  - Implement methods **new**, **push**, **pop** and **peek** for the structure

- Test:
  - Create a test to check if the stack works properly

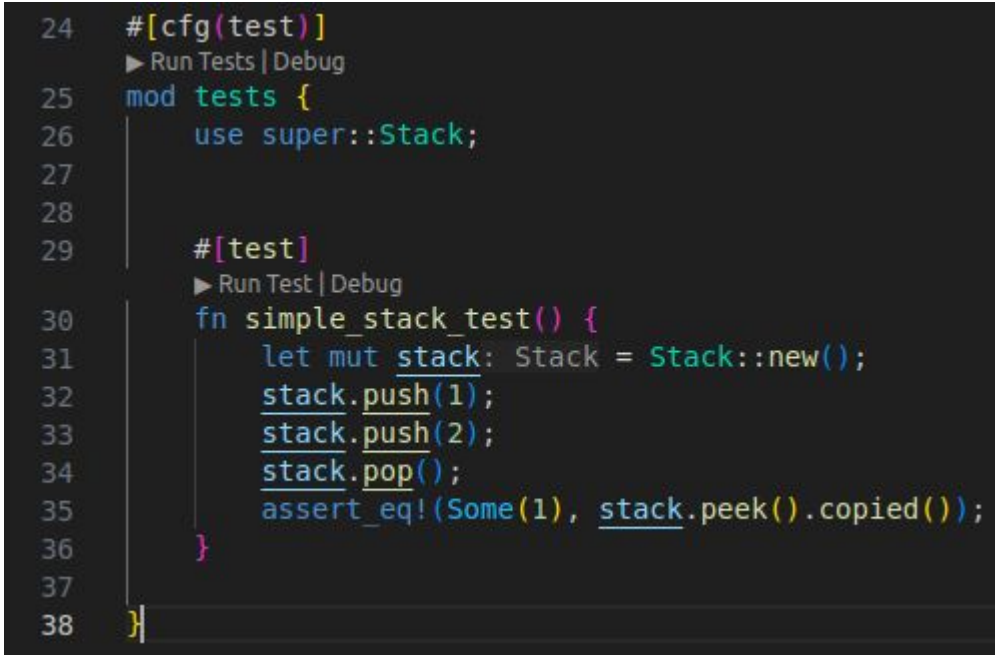
- Solution on the next slide



SOLUTION: Creating the Stack structure



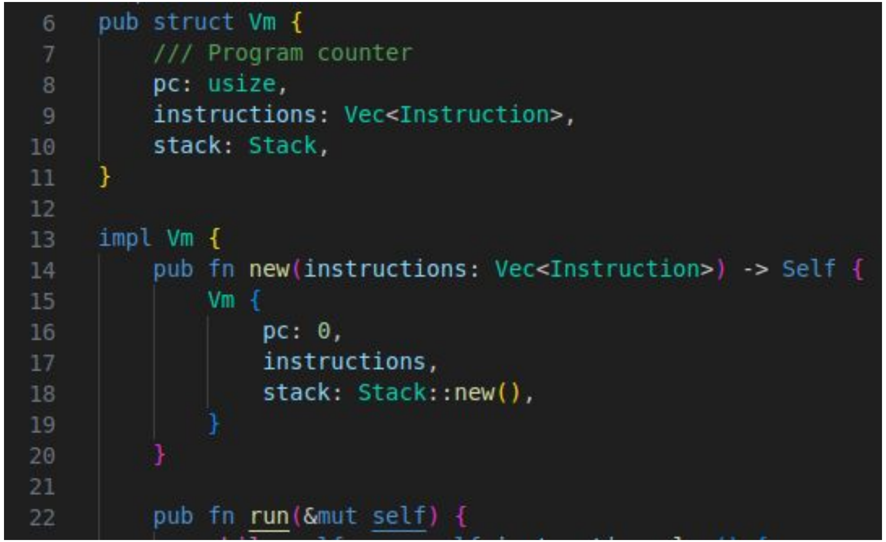
```rust
#[derive(Debug)]
pub struct Stack {
    stack: Vec<u32>,
}

impl Stack {
    pub fn new() -> Self {
        Stack { stack: vec![] }
    }

    pub fn push(&mut self, n: u32) {
        self.stack.push(n)
    }

    pub fn pop(&mut self) -> Option<u32> {
        self.stack.pop()
    }

    pub fn peek(&self) -> Option<&u32> {
        self.stack.last()
    }
}
```



```rust
#[cfg(test)]
mod tests {
    use super::Stack;


    #[test]
    fn simple_stack_test() {
        let mut stack: Stack = Stack::new();
        stack.push(1);
        stack.push(2);
        stack.pop();
        assert_eq!(Some(1), stack.peek().copied());
    }

}
```

STEP 7-1: Implement VM struct

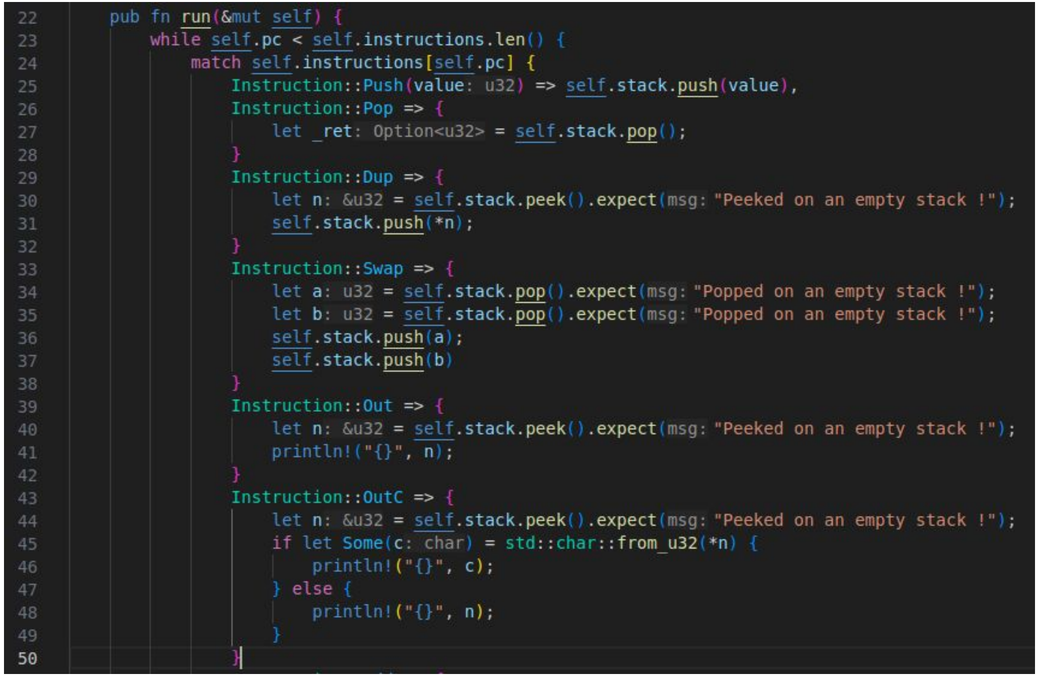
- Steps:
  - Create a new file *vm.rs* in the *src* directory
  - Create a new structure called **Vm,** it should contain the following
    - The **instructions** Vector
    - The **stack**
    - A program counter (to keep track of the instruction to execute)
  - Create a **new method** to initialize the structure
  - Create an empty **run** method



```rust
pub struct Vm {
    /// Program counter
    pc: usize,
    instructions: Vec<Instruction>,
    stack: Stack,
}

impl Vm {
    pub fn new(instructions: Vec<Instruction>) -> Self {
        Vm {
            pc: 0,
            instructions,
            stack: Stack::new(),
        }
    }

    pub fn run(&mut self) {
```

STEP 7-2: The run function for basic instructions

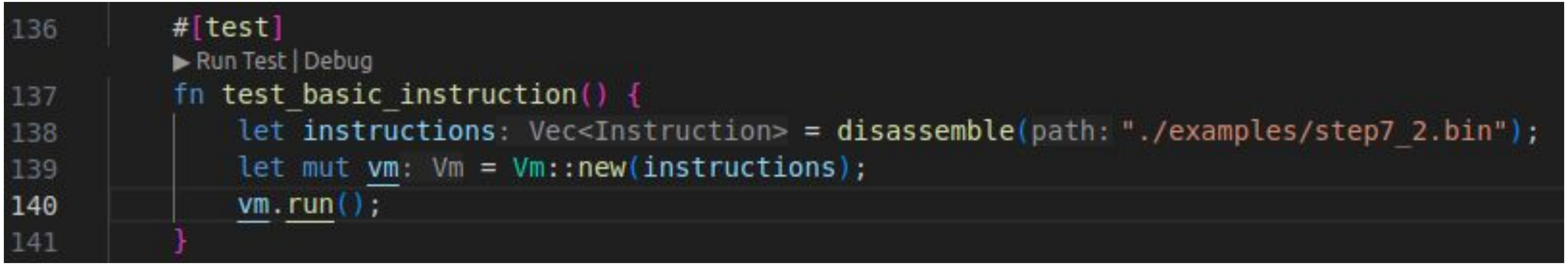
- Steps:
  - **Loop** through the **instructions** vector using the **program counter**
  - Use a match to execute the right instruction, do it for:
    - push, pop, dup, swap, out, outc
  - Write a test to see if your VM can run **step7_2.bin**



```rust
    pub fn run(&mut self) {
        while self.pc < self.instructions.len() {
            match self.instructions[self.pc] {
                Instruction::Push(value: u32) => self.stack.push(value),
                Instruction::Pop => {
                    let _ret: Option<u32> = self.stack.pop();
                }
                Instruction::Dup => {
                    let n: &u32 = self.stack.peek().expect(msg: "Peeked on an empty stack !");
                    self.stack.push(*n);
                }
                Instruction::Swap => {
                    let a: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    let b: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    self.stack.push(a);
                    self.stack.push(b)
                }
                Instruction::Out => {
                    let n: &u32 = self.stack.peek().expect(msg: "Peeked on an empty stack !");
                    println!("{}", n);
                }
                Instruction::OutC => {
                    let n: &u32 = self.stack.peek().expect(msg: "Peeked on an empty stack !");
                    if let Some(c: char) = std::char::from_u32(*n) {
                        println!("{}", c);
                    } else {
                        println!("{}", n);
                    }
                }
```

- Solution on the next slide

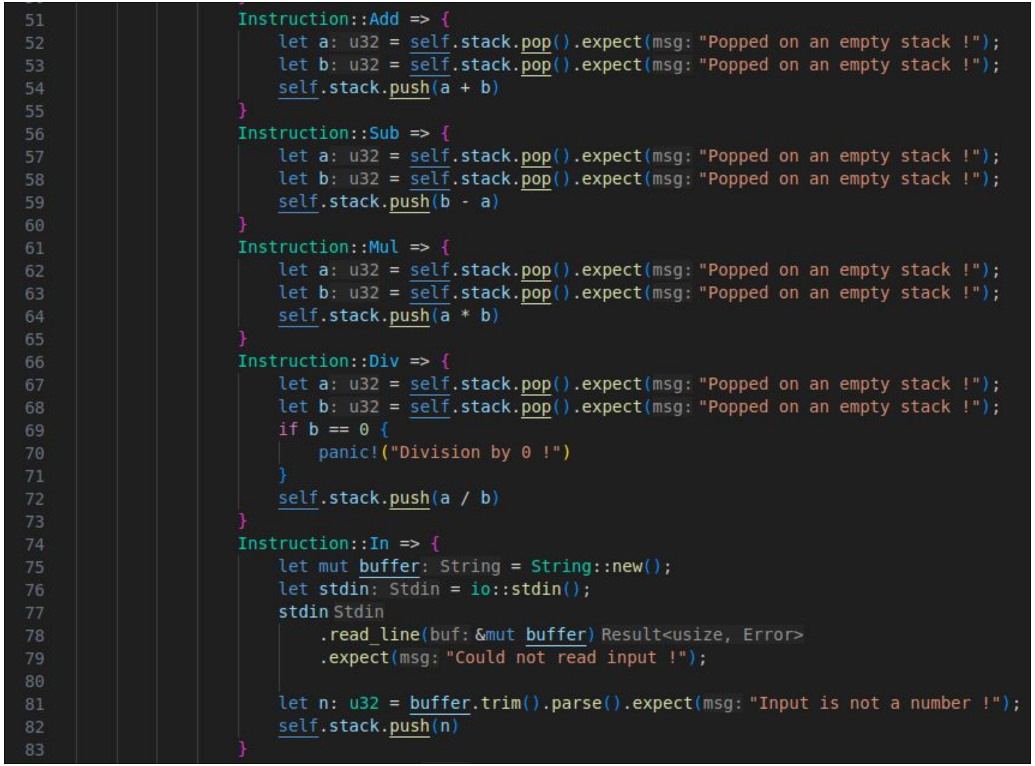
SOLUTION: Test for basic instructions



```
#[test]
fn test_basic_instruction() {
    let instructions: Vec<Instruction> = disassemble(path: "./examples/step7_2.bin");
    let mut vm: Vm = Vm::new(instructions);
    vm.run();
}
```

STEP 7-3: Math and In instructions

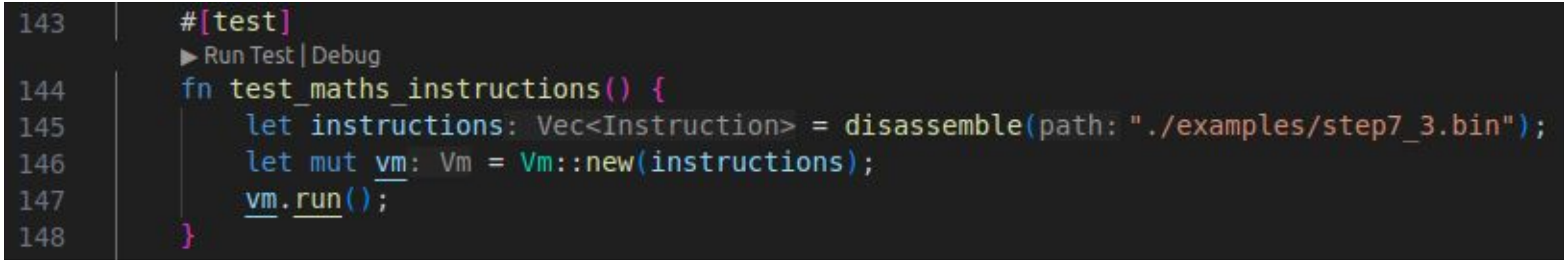
- Steps:
  - Use a match to execute the right instruction, do it for:
    - add, sub, mul, div, in
  - Write a test to see if your vm can run **step7_3.bin**



```
                Instruction::Add => {
                    let a: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    let b: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    self.stack.push(a + b)
                }
                Instruction::Sub => {
                    let a: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    let b: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    self.stack.push(b - a)
                }
                Instruction::Mul => {
                    let a: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    let b: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    self.stack.push(a * b)
                }
                Instruction::Div => {
                    let a: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    let b: u32 = self.stack.pop().expect(msg: "Popped on an empty stack !");
                    if b == 0 {
                        panic!("Division by 0 !")
                    }
                    self.stack.push(a / b)
                }
                Instruction::In => {
                    let mut buffer: String = String::new();
                    let stdin: Stdin = io::stdin();
                    stdin Stdin
                        .read_line(buf: &mut buffer) Result<usize, Error>
                        .expect(msg: "Could not read input !");

                    let n: u32 = buffer.trim().parse().expect(msg: "Input is not a number !");
                    self.stack.push(n)
                }
```

- Solution on the next slide

SOLUTION: Test for math instructions

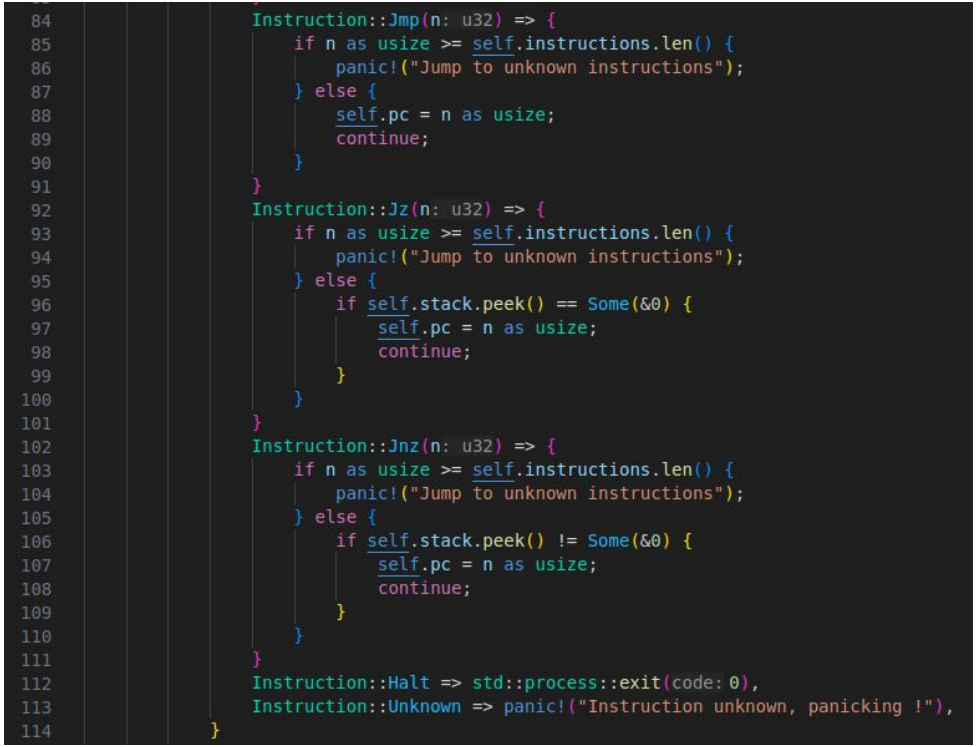


```
#[test]
fn test_maths_instructions() {
    let instructions: Vec<Instruction> = disassemble(path: "./examples/step7_3.bin");
    let mut vm: Vm = Vm::new(instructions);
    vm.run();
}
```

STEP 7-4: Jump and halt instructions

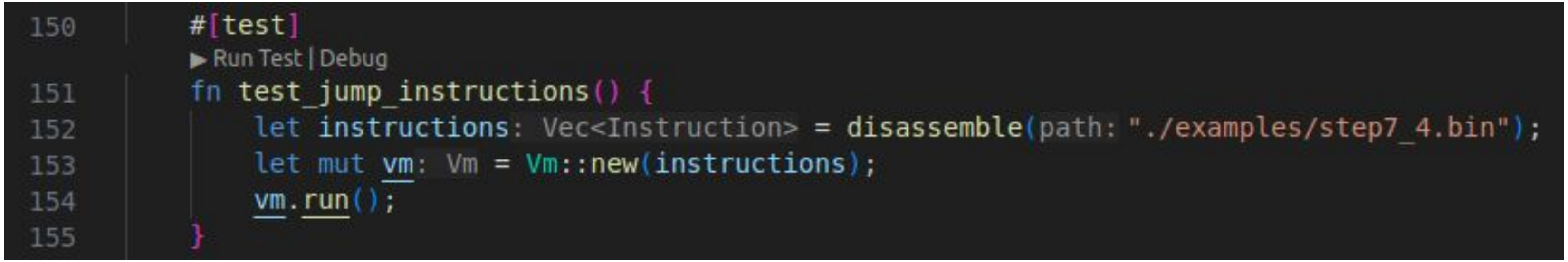
- Steps:
    - Use a match to execute the right instruction, do it for:
        - jmp, jz, jnz, halt
    - Write a test to see if your vm can run **step7_4.bin**

- Solution on the next slide



```
                    Instruction::Jmp(n: u32) => {
                        if n as usize >= self.instructions.len() {
                            panic!("Jump to unknown instructions");
                        } else {
                            self.pc = n as usize;
                            continue;
                        }
                    }
                    Instruction::Jz(n: u32) => {
                        if n as usize >= self.instructions.len() {
                            panic!("Jump to unknown instructions");
                        } else {
                            if self.stack.peek() == Some(&0) {
                                self.pc = n as usize;
                                continue;
                            }
                        }
                    }
                    Instruction::Jnz(n: u32) => {
                        if n as usize >= self.instructions.len() {
                            panic!("Jump to unknown instructions");
                        } else {
                            if self.stack.peek() != Some(&0) {
                                self.pc = n as usize;
                                continue;
                            }
                        }
                    }
                    Instruction::Halt => std::process::exit(code: 0),
                    Instruction::Unknown => panic!("Instruction unknown, panicking !"),
                }
```

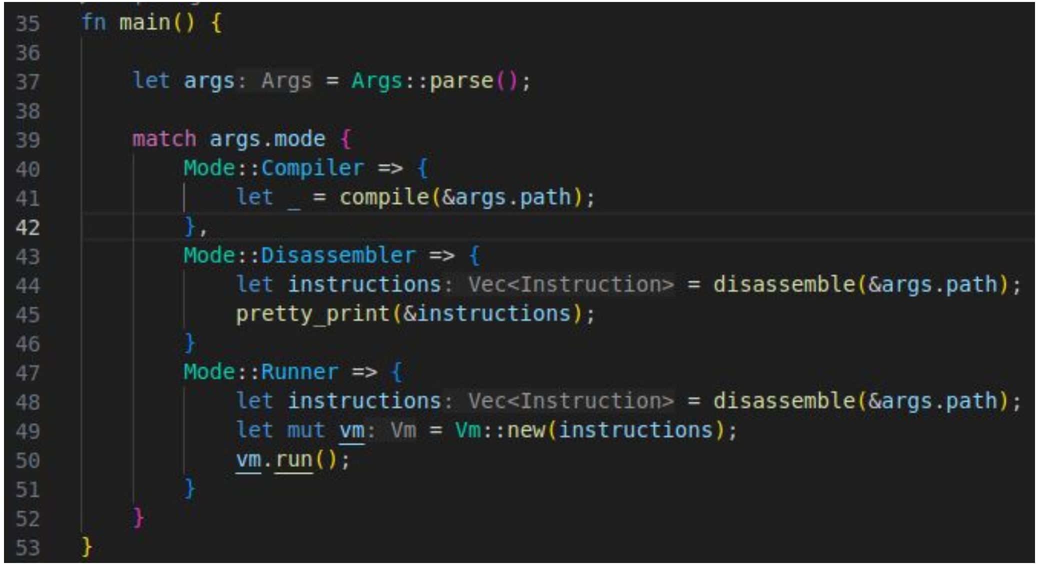
SOLUTION: Test for jump instructions



```
#[test]
fn test_jump_instructions() {
    let instructions: Vec<Instruction> = disassemble(path: "./examples/step7_4.bin");
    let mut vm: Vm = Vm::new(instructions);
    vm.run();
}
```

STEP 8: Add runner mode

- Steps:
  - Add new modes **runner** and **compiler** that calls the **run** method of the Vm structure and the **compile** function:
    - ./vm -p -m runner|compiler|disassembler
      - -p Path
      - -m Mode
  - Test your VM with the **game.svm** program !



```
fn main() {

    let args: Args = Args::parse();

    match args.mode {
        Mode::Compiler => {
            let _ = compile(&args.path);
        },
        Mode::Disassembler => {
            let instructions: Vec<Instruction> = disassemble(&args.path);
            pretty_print(&instructions);
        }
        Mode::Runner => {
            let instructions: Vec<Instruction> = disassemble(&args.path);
            let mut vm: Vm = Vm::new(instructions);
            vm.run();
        }
    }
}
```

EXTRA EXERCISE: Going deeper

Create new programs

- Improve game.svm
  - to loop until answer is found without quitting after each input verification
- Create a fibonacci program
- Create a factorial program
- Etc.

VM improvement

- Add new **Instructions**
  - jump if less than, cmp, etc.
- Add a **Memory** Instructions to the VM
  - like **store** and **load** instructions
- Add **Labels** and **relative jumps** Instructions

Questions?