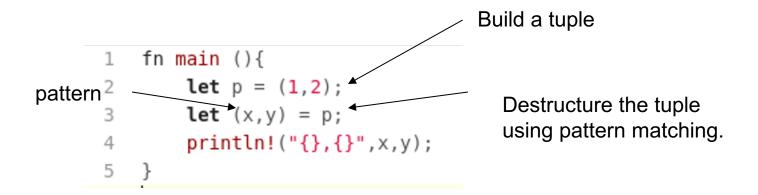
Pattern Matching

- In computer science, *pattern matching* is the act of checking a given sequence of tokens or a given tree structure for the *presence of the constituents of some pattern.*
- A pattern looks like the input sequence or the input tree with the exception that it may contain variables.
- A simple program in Rust that does pattern matching:



Note: the pattern looks just like the structure except that it has variables in it which will get instantiated with values from the structure.

Pattern Matching

- Pattern matching was developed in the 1970's for functional programming languages.
- In functional programming languages virtually all data structures have structure that can be pattern matched – abstract data types.
- Pattern matching is a way to compute on those data structures.
- Virtually all modern programming languages support one form or another of pattern matching
 - E.g. Python, Rust, Swift, Scala, Haskell as well as Prolog (being the oldest language on this list)
- Code using pattern matching tends to be much more succinct and readable.

Rust Pattern Matching

- In Rust, similar to functional programming languages, all data structures have structure that can be explicitly matched using pattern matching.
- This includes enums, struct, and primitive data types

Rust Pattern Matching

```
enum Coin {
                       Here we use 'match'
    Penny,
                       to pattern match the value
    Nickel,
    Dime,
                       in parameter coin.
    Quarter,
                                                                                  Generate 1 through 4
fn value(in_cents(coin: Coin) -> u8 {
                                                                                  sequence.
    match coin {
                                                              fn main() {
        Coin::Penny => 1,
        Coin::Nickel => 5,
                                                                  let x = 2;
        Coin::Dime => 10.
        Coin::Quarter => 25,
                                                                  match x
                                                                      1..=4 => println!("one through four"),
                   struct Point {
                                                                      5 => println!("five"),
                      x: i32,
                                                                      => println!("something else"),
                      y: i32,
                  fn main() {
                       let p = Point { x: 0, y: 7 };
                       match p {
                           Point \{x, y: 0\} = println!("Point is on the x axis at <math>\{\}\}", x),
                           Point { x: 0, y } => println!("Point is on the y axis at {}", y),
                           Point \{x, y\} \Rightarrow println!("Point is on neither axis: (\{\}, \{\})", x, y),
```

Rust Pattern Matchine

```
fn main () {
        // define a simple database
        let db = [
            // Format:
            // (Name, Profession, Marital Status, Age, Height, Weight)
 6
            ("Eddie", "Cook", "Married", 25, 75, 180),
            ("Betty", "Nurse", "Married", 28, 65, 125),
 8
            ("Joe", "Bus Driver", "Single", 35, 70, 192),
 9
            ("Zoey", "CEO", "Married", 45, 72, 120)
10
        ];
11
12
13
        // Compute the average age, height, and weight in the db
14
        let mut age sum = 0;
        let mut height sum = 0;
15
        let mut weight sum = 0;
16
        for ( , , ,age,height,weight) in db.iter() {
17
18
            age sum += age;
            height sum += height;
19
            weight sum += weight;
20
21
22
        println!("age avg={}, height avg={}, weight avg={}",
23
            age sum/db.len(),
            height sum/db.len(),
24
25
            weight sum/db.len()
26
          Screenshot
27
```

Rust Pattern Matching

- Rust allows pattern matching in many different places
- We have seen pattern matching in the 'let', 'for' and 'match' statements
- Beyond that we can pattern match in virtually all control structures and in functions.

Tree Traversal with Pattern Matching

- The idea is to construct a tree structure in Rust representing some computation
- and then walk this tree and generate an infix representation of the tree.
- Here we take advantage of Rust's smart pointers – the Box type – in order to create a tree structure.

Tree Traversal with Pattern Matching

```
fn main () {
                                                  17
                                                           // construct tree using the Node type
                                                  18
                                                  19
                                                           // (+, (*,2,x),4)
                                                           let t = Box::new(Node::Binop("+".to string(),
                                                  20
                                                                          Box::new(Node::Binop("*".to string(),
                                                  21
                                                                                Box::new(Node::Int(2)),
    #[derive(Debug)]
                                                  22
    enum Node {
                                                  23
                                                                                Box::new(Node::Var("x".to string())))),
        Binop(String,Box<Node>,Box<Node>),
                                                  24
                                                                          Box::new(Node::Int(4)));
3
                                                           println!("Tree Dump: {:?}",t);
       Int(i64),
                                                  25
       Var(String)
                                                           println!("Infix Notation: {}",compute infix(t));
                                                  26
6
                                                  27
7
    fn compute infix(node: Box<Node>) -> String {
        match *node {
9
            Node::Binop(op, l, r) =>
10
                    format!("({}{}{})",compute infix(l),op,compute infix(r)),
11
            Node:: Int(val) => val.to string(),
12
```

```
ubuntu$ ./pattern2
Tree Dump: Binop("+", Binop("*", Int(2), Var("x")), Int(4))
Infix Notation: ((2*x)+4)
ubuntu$ ■
```

Node::Var(name) => name

13 14

15 }

}

Assignments

 Read: doc.rust-lang.org/book/ch18-00patterns.html