

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:

```
1  fn main (){  
2      let p = (1,2);  
3      let (x,y) = p;  
4      println!("{}",x,y);  
5  }
```

Build a tuple

pattern

Destructure the tuple using 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 {  
    Penny,  
    Nickel,  
    Dime,  
    Quarter,  
}
```

Here we use 'match' to pattern match the value in parameter coin.

```
fn value_in_cents(coin: Coin) -> u8 {  
    match coin {  
        Coin::Penny => 1,  
        Coin::Nickel => 5,  
        Coin::Dime => 10,  
        Coin::Quarter => 25,  
    }  
}
```

```
1 struct Point {  
2     x: i32,  
3     y: i32,  
4 }  
5  
6 fn main() {  
7     let p = Point { x: 0, y: 7 };  
8  
9     match p {  
10         Point { x, y: 0 } => println!("Point is on the x axis at {}", x),  
11         Point { x: 0, y } => println!("Point is on the y axis at {}", y),  
12         Point { x, y } => println!("Point is on neither axis: ({}, {})", x, y),  
13     }  
14 }
```

Generate 1 through 4 sequence.

```
1 fn main() {  
2     let x = 2;  
3  
4     match x {  
5         1..=4 => println!("one through four"),  
6         5 => println!("five"),  
7         _ => println!("something else"),  
8     }  
9 }
```

Rust Pattern Matching

```
1
2 fn main () {
3     // define a simple database
4     let db = [
5         // Format:
6         // (Name, Profession, Marital Status, Age, Height, Weight)
7         ("Eddie", "Cook", "Married", 25, 75, 180),
8         ("Betty", "Nurse", "Married", 28, 65, 125),
9         ("Joe", "Bus Driver", "Single", 35, 70, 192),
10        ("Zoey", "CEO", "Married", 45, 72, 120)
11    ];
12
13    // Compute the average age, height, and weight in the db
14    let mut age_sum = 0;
15    let mut height_sum = 0;
16    let mut weight_sum = 0;
17    for (_,_,_,age,height,weight) in db.iter() {
18        age_sum += age;
19        height_sum += height;
20        weight_sum += weight;
21    }
22    println!("age avg={}, height avg={}, weight avg={}",
23        age_sum/db.len(),
24        height_sum/db.len(),
25        weight_sum/db.len()
26    );
27 }
```

Screenshot

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

```
1  #[derive(Debug)]
2  enum Node {
3      Binop(String, Box<Node>, Box<Node>),
4      Int(i64),
5      Var(String)
6  }
7
8  fn compute_infix(node: Box<Node>) -> String {
9      match *node {
10         Node::Binop(op, l, r) =>
11             format!("({}{}){}", op, compute_infix(l), compute_infix(r)),
12         Node::Int(val) => val.to_string(),
13         Node::Var(name) => name
14     }
15 }
```

```
17 fn main () {
18     // construct tree using the Node type
19     // (+, (*,2,x),4)
20     let t = Box::new(Node::Binop("+".to_string(),
21                                   Box::new(Node::Binop("*".to_string(),
22                                                         Box::new(Node::Int(2)),
23                                                         Box::new(Node::Var("x".to_string())))),
24                                   Box::new(Node::Int(4))));
25     println!("Tree Dump: {:?}",t);
26     println!("Infix Notation: {}",compute_infix(t));
27 }
```

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


Assignments

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