Practical Programming

Rust : Pattern Matching

David Bouchet

david.bouchet.epita@gmail.com

The `match` Keyword

The match keyword compares a value to a list of patterns.

When a first pattern matches the value, the code associated to this pattern is executed and the following patterns are ignored.

Simple Cases

```
fn test(i: u8)
{
    match i
    {
        0 => println!("zero"),
        1 | 2 => println!("one or two"),
        42 => println!("forty-two"),
        _ => println!("different from 0, 1, 2 and 42"),
    }
}
```

```
test(0);
test(1);
test(2);
test(42);
test(3);
test(255);
```

```
zero
one or two
one or two
forty-two
different from 0, 1, 2 and 42
different from 0, 1, 2 and 42
```

Matches Are Exhaustive

All cases must be tested.

```
match i
{
    0 => println!("zero"),
    1 | 2 => println!("one or two"),
    42 => println!("forty-two"),
}
```

```
error[E0004]: non-exhaustive patterns: `_` not covered
  --> number.rs:13:11
   |
13 |     match i
   |           ^ pattern `_` not covered
```

Order Matters

```
match i
{
    0 => println!("zero"),
    1 | 2 => println!("one or two"),
    _ => println!("different from 0, 1, 2 and 42"),
    42 => println!("forty-two"),
}
```

```
warning: unreachable pattern
  --> number.rs:18:9
   |
18 |         42 => println!("forty-two"),
   |         ^^
   |
```

The match Expression

```
fn test(i: u8)
{
    let s = match i
    {
        0 => "zero",
        1 | 2 => "one or two",
        42 => "forty-two",
        _ => "different from 0, 1, 2 and 42",
    };

    println!("{}", s);
}
```

```
test(0);
test(1);
test(2);
test(42);
test(3);
test(255);
```

```
zero
one or two
one or two
forty-two
different from 0, 1, 2 and 42
different from 0, 1, 2 and 42
```

Matching Ranges of Integers

```
fn test(i: u8)
{
    println!("{}", match i
    {
        0 ... 9 => "lower than ten",
        10 ... 99 => "lower than one hundred",
        100 => "one hundred",
        _ => "greater than one hundred",
    });
}
```

```
test(5);
test(42);
test(100);
test(255);
```

```
lower than ten
lower than one hundred
one hundred
greater than one hundred
```

Matching Ranges of Characters

```
fn test(c: char)
{
    println!("'{}' is {}.", c, match c
    {
        'A' ... 'M' => "between 'A' and 'M'",
        'N' ... 'Z' => "between 'N' and 'Z'",
        _ => "not a capital letter",
    });
}
```

```
test('C');
test('Z');
test('a');
```

```
'C' is between 'A' and 'M'.
'Z' is between 'N' and 'Z'.
'a' is not a capital letter.
```

Match Guards

```
fn test(i: u8, with_42: bool)
{
    match i
    {
        0 => println!("zero"),
        1 | 2 => println!("one or two"),
        42 if with_42 => println!("forty-two"),
        _ if with_42 => println!("different from 0, 1, 2 and 42"),
        _ => println!("different from 0, 1, 2"),
    }
}
```

```
test(42, false);
test(42, true);
test(50, false);
test(50, true);
```

```
different from 0, 1, 2
forty-two
different from 0, 1, 2
different from 0, 1, 2 and 42
```

Matching Tuples

```
fn test(a: u8, b: u8, c: u8)
{
    match (a, b, c)
    {
        (0, 0, i) => println!("a =  0, b =  0, i =  {}", i),
        (0, _, i) => println!("a =  0, b = ≠0, i =  {}", i),
        (1, _, _) => println!("a =  1, b =  ?, i =  ?"),
        (_, 7, _) => println!("a = ≠1, b =  7, i =  ?"),
        (i, _, j) => println!("a =  {}, b = ≠7, i =  {}", i, j),
    }
}
```

```
test(0, 0, 5);
test(0, 7, 5);
test(1, 7, 5);
test(2, 7, 5);
test(2, 6, 5);
```

```
a =  0, b =  0, i =  5
a =  0, b = ≠0, i =  5
a =  1, b =  ?, i =  ?
a = ≠1, b =  7, i =  ?
a =  2, b = ≠7, i =  5
```

Matching Enumerations (1)

```rust
enum Shape
{
    Point,
    Rectangle(u8, u8),
    Square(u8),
}
```

```rust
fn test(s: Shape)
{
    match s
    {
        Shape::Point =>
            println!("Point (no dimension)"),

        Shape::Rectangle(w, h) =>
            println!("Rectangle (w = {}, h = {})", w, h),

        Shape::Square(l) =>
            println!("Square (side length = {})", l),
    }
}
```

Matching Enumerations (2)

```
let p = Shape::Point;
let r = Shape::Rectangle(6, 8);
let s1 = Shape::Square(8);
let s2 = Shape::Square(2);

test(p);
test(r);
test(s1);
test(s2);
```

```
Point (no dimension)
Rectangle (w = 6, h = 8)
Square (side length = 8)
Square (side length = 2)
```

Example with `Option` (1)

```
fn print_min(slice: &[i32])
{
    let min = slice.iter().min();

    match min
    {
        None =>
            println!("The slice is empty."),

        Some(m) =>
            println!("The minimum of {:?} is {}.", slice, m),
    }
}
```

Example with `Option` (2)

```rust
let a1 = [10, 5, 13, 12];
let a2: Vec<i32> = vec![];

print_min(&a1);
print_min(&a2);
```

```
The minimum of [10, 5, 13, 12] is 5.
The slice is empty.
```

Matching with `if let` (1)

```rust
if let Some(x) = Some(2)
{
    dbg!(x);
}

let s = Some(2);

if let Some(x) = s
{
    dbg!(x);
}
```

```
x = 2
x = 2
```

Matching with `if let` (2)

```
fn test1(opt: Option<i32>)
{
    match opt
    {
        Some(i) => println!("{}", i),
        None => println!("None"),
    }
}
```

```
fn test2(opt: Option<i32>)
{
    if let Some(i) = opt {
        println!("{}", i);
    }

    else {
        println!("None");
    }
}
```

```
test1(Some(5));
test2(Some(5));
test1(None);
test2(None);
```

```
5
5
None
None
```

Matching with `while let`

```
let a = [10, 12, 51];
let mut iter = a.iter();

while let Some(x) = iter.next()
{
    dbg!(x);
}
```

```
x = 10
x = 12
x = 51
```

Error Handling with `Result` and `?` (1)

```
fn div(a: f64, b: f64) -> Result<f64, String>
{
    if b == 0.0
    {
        Err(String::from("Division by zero."))
    }

    else
    {
        Ok(a / b)
    }
}
```

Error Handling with `Result` and `?` (2)

```
fn print_div(a: f64, b: f64)
{
    print!("{} / {} = ", a, b);

    match div(a, b)
    {
        Err(msg) => println!("{}", msg),
        Ok(r) => println!("{}", r),
    }
}
```

```
print_div(10.0, 2.0);
print_div(3.0, 0.0);
print_div(-10.0, 2.0);
print_div(-10.0, -2.0);
```

```
10 / 2 = 5
3 / 0 = Division by zero.
-10 / 2 = -5
-10 / -2 = 5
```

Error Handling with `Result` and `?` (3)

```
fn sqrt(a: f64) -> Result<f64, String>
{
    if a.is_sign_negative()
    {
        Err(String::from("Square root of negative number."))
    }

    else
    {
        Ok(a.sqrt())
    }
}
```

Error Handling with `Result` and `?` (4)

```
fn print_sqrt_div(a: f64, b: f64)
{
    print!("sqrt_div({} / {}) = ", a, b);

    match sqrt_div(a, b)
    {
        Err(msg) => println!("{}", msg),
        Ok(r) => println!("{}", r),
    }
}
```

```
fn sqrt_div(a: f64, b: f64) -> Result<f64, String>
{
    // TODO: sqrt(div(a,b))
}
```

Error Handling with `Result` and `?` (5)

```
print_sqrt_div(10.0, 2.0);
print_sqrt_div(3.0, 0.0);
print_sqrt_div(-10.0, 2.0);
print_sqrt_div(-10.0, -2.0);
```

```
sqrt_div(10 / 2) = 2.23606797749979
sqrt_div(3 / 0) = Division by zero.
sqrt_div(-10 / 2) = Square root of negative number.
sqrt_div(-10 / -2) = 2.23606797749979
```

Error Handling with `Result` and `?` (6)

Version 1

```
fn sqrt_div(a: f64, b: f64) -> Result<f64, String>
{
    sqrt(div(a, b))
}
```

Error Handling with `Result` and `?` (7)

Version 1

```rust
fn sqrt_div(a: f64, b: f64) -> Result<f64, String>
{
    sqrt(div(a, b))
}
```

```
error[E0308]: mismatched types
  --> result.rs:65:10
   |
65 |     sqrt(div(a, b))
   |          ^^^^^^^^^ expected f64, found enum `std::result::Result`
   |
   = note: expected type `f64`
              found type `std::result::Result<f64, std::string::String>`
```

Error Handling with `Result` and `?` (8)

Version 2

```
fn sqrt_div(a: f64, b: f64) -> Result<f64, String>
{
    match div(a, b)
    {
        Err(msg) => Err(msg),
        Ok(q) => sqrt(q),
    }
}
```

Error Handling with `Result` and `?` (9)

Version 3

```
fn sqrt_div(a: f64, b: f64) -> Result<f64, String>
{
    let r = div(a, b);
    if let Ok(q) = r { sqrt(q) } else { r }
}
```

Error Handling with `Result` and `?` (10)

Version 4

```
fn sqrt_div(a: f64, b: f64) -> Result<f64, String>
{
    let r = div(a, b)?;
    sqrt(r)
}
```

Error Handling with `Result` and `?` (11)

Version 5

```
fn sqrt_div(a: f64, b: f64) -> Result<f64, String>
{
    sqrt(div(a, b)?)
}
```