part01.rs

Rust-101, Part 01: Expressions, Inherent methods

For Rust to compile this file, make sure to enable the corresponding line in `main.rs` before going on.

Even though our code from the first part works, we can still learn a lot by making it prettier. That's because Rust is an "expression-based" language, which means that most of the terms you write down are not just *statements* (executing code), but *expressions* (returning a value). This applies even to the body of entire functions!

Expression-based programming

For example, consider `sqr`:

```rust
fn sqr(i: i32) -> i32 { i * i }
```

Between the curly braces, we are giving the *expression* that computes the return value. So we can just write `i * i`, the expression that returns the square of `i`! This is very close to how mathematicians write down functions (but with more types).

Conditionals are also just expressions. This is comparable to the ternary `? :` operator from languages like C.

```rust
fn abs(i: i32) -> i32 { if i >= 0 { i } else { -i } }
```

And the same applies to case distinction with `match`: Every `arm` of the match gives the expression that is returned in the respective case. (We repeat the definition from the previous part here.)

```rust
enum NumberOrNothing {
    Number(i32),
    Nothing
}
use self::NumberOrNothing::{Number,Nothing};
fn number_or_default(n: NumberOrNothing, default: i32) -> i32 {
    match n {
        Nothing => default,
        Number(n) => n,
    }
}
```

It is even the case that blocks are expressions, evaluating to the last expression they contain.

```rust
fn compute_stuff(x: i32) -> i32 {
    let y = { let z = x*x; z + 14 };
    y*y
}
```

Let us now refactor `vec_min`.

```rust
fn vec_min(v: Vec<i32>) -> NumberOrNothing {
```

Remember that helper function `min_i32`? Rust allows us to define such helper functions *inside* other functions. This is just a matter of namespacing, the inner function has no access to the data of the outer one. Still, being able to nicely group functions can significantly increase readability.

```rust
    fn min_i32(a: i32, b: i32) -> i32 {
        if a < b { a } else { b }
    }

    let mut min = Nothing;
    for e in v {
```

Notice that all we do here is compute a new value for `min`, and that it will always end up being a `Number` rather than `Nothing`. In Rust, the structure of the code can express this uniformity.

```rust
        min = Number(match min {
            Nothing => e,
            Number(n) => min_i32(n, e)
        });
    }
```

The `return` keyword exists in Rust, but it is rarely used. Instead, we typically make use of the fact that the entire function body is an expression, so we can just write down the desired return value.

```rust
    min
}
```

Now that's already much shorter! Make sure you can go over the code above and actually understand every step of what's going on.

Inherent implementations

So much for `vec_min`. Let us now reconsider `print_number_or_nothing`. That function really belongs pretty close to the type `NumberOrNothing`. In C++ or Java, you would probably make it a method of the type. In Rust, we can achieve something very similar by providing an *inherent implementation*.

```rust
impl NumberOrNothing {
    fn print(self) {
        match self {
            Nothing => println!("The number is: <nothing>"),
            Number(n) => println!("The number is: {}", n),
        };
    }
}
```

So, what just happened? Rust separates code from data, so the definition of the methods on an `enum` (and also on `struct`, which we will learn about later) is independent of the definition of the type. `self` is like `this` in other languages, and its type is always implicit. So `print` is now a method that takes `NumberOrNothing` as the first argument, just like `print_number_or_nothing`.

Try making `number_or_default` from above an inherent method as well!

With our refactored functions and methods, `main` now looks as follows:

```rust
fn read_vec() -> Vec<i32> {
    vec![18,5,7,2,9,27]
}
pub fn main() {
    let vec = read_vec();
    let min = vec_min(vec);
    min.print();
}
```

You will have to replace `part00` by `part01` in the `main` function in `main.rs` to run this code.

Exercise 01.1: Write a function `vec_sum` that computes the sum of all values of a `Vec<i32>`.

Exercise 01.2: Write a function `vec_print` that takes a vector and prints all its elements.

index | previous | raw source | next