part00.rs

Rust-101, Part 00: Algebraic datatypes

As our first piece of Rust code, we want to write a function that computes the minimum of a list.

Getting started

Let us start by thinking about the *type* of our function. Rust forces us to give the types of all arguments, and the return type, before we even start writing the body. In the case of our minimum function, we may be inclined to say that it returns a number. But then we would be in trouble: What's the minimum of an empty list? The type of the function says we have to return *something*. We could just choose 0, but that would be kind of arbitrary. What we need is a type that is "a number, or nothing". Such a type (of multiple exclusive options) is called an "algebraic datatype", and Rust lets us define such types with the keyword `enum`. Coming from C(++), you can think of such a type as a `union`, together with a field that stores the variant of the union that's currently used.

An `enum` for "a number or nothing" could look as follows:

```
enum NumberOrNothing {
    Number(i32),
    Nothing
}
```

Notice that `i32` is the type of (signed, 32-bit) integers. To write down the type of the minimum function, we need just one more ingredient: `Vec<i32>` is the type of (growable) arrays of numbers, and we will use that as our list type.

Observe how in Rust, the return type comes *after* the arguments.

In the function, we first need some variable to store the minimum as computed so far. Since we start out with nothing computed, this will again be a "number or nothing":

```
fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
    let mut min = NumberOrNothing::Nothing;
```

We do not have to write a type next to `min`, Rust can figure that out automatically (a bit like `auto` in C++11). Also notice the `mut`: In Rust, variables are immutable per default, and you need to tell Rust if you want to change a variable later.

Now we want to *iterate* over the list. Rust has some nice syntax for iterators:

```
    for el in vec {
```

So `el` is an element of the list. We need to update `min` accordingly, but how do we get the current number in there? This is what pattern matching can do:

```
        match min {
```

In this case (*arm*) of the `match`, `min` is currently nothing, so let's just make it the number `el`.

```
            NumberOrNothing::Nothing => {
                min = NumberOrNothing::Number(el);
            },
```

In this arm, `min` is currently the number `n`, so let's compute the new minimum and store it. We will write the function `min_i32` just after we completed this one.

```
            NumberOrNothing::Number(n) => {
                let new_min = min_i32(n, el);
                min = NumberOrNothing::Number(new_min);
            }
        }
    }
```

Notice that Rust makes sure you did not forget to handle any case in your `match`. We say that the pattern matching has to be *exhaustive*.

Finally, we return the result of the computation.

```
    return min;
}
```

Now that we reduced the problem to computing the minimum of two integers, let's do that.

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

Phew. We wrote our first Rust function! But all this `NumberOrNothing::` is getting kind of ugly. Can't we do that nicer?

Indeed, we can: The following line tells Rust to take the constructors of `NumberOrNothing` into the local namespace. Try moving that above the function, and removing all the occurrences of `NumberOrNothing::`.

```
use self::NumberOrNothing::{Number,Nothing};
```

To call this function, we now just need a list. Of course, ultimately we want to ask the user for a list of numbers, but for now, let's just hard-code something.

`vec!` is a *macro* (as indicated by `!`) that constructs a constant `Vec<_>` with the given elements.

```
fn read_vec() -> Vec<i32> {
    vec![18,5,7,1,9,27]
}
```

Of course, we would also like to actually see the result of the computation, so we need to print the result. Of course Rust can print numbers, but after calling `vec_min`, we have a `NumberOrNothing`. So let's write a small helper function that prints such values.

`println!` is again a macro, where the first argument is a *format string*. For now, you just need to know that `{}` is the placeholder for a value, and that Rust will check at compile-time that you supplied the right number of arguments.

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

Putting it all together:

```
pub fn main() {
    let vec = read_vec();
    let min = vec_min(vec);
    print_number_or_nothing(min);
}
```

You can now use `cargo build` to compile your *crate*. That's Rust's name for a *compilation unit*, which in the case of Rust means an application or a library. Finally, try `cargo run` on the console to run it.

Yay, it said "1"! That's actually the right answer. Okay, we could have computed that ourselves, but that's beside the point. More importantly: You completed the first part of the course.