part09.rs

Rust-101, Part 09: Iterators

In the following, we will look into the iterator mechanism of Rust and make our `BigInt` compatible with the `for` loops. Of course, this is all about implementing certain traits again. In particular, an iterator is something that implements the `Iterator` trait. As you can see in the documentation, this trait mandates a single function `next` returning an `Option<Self::Item>`, where `Item` is an associated type chosen by the implementation. (There are many more methods provided for `Iterator`, but they all have default implementations, so we don't have to worry about them right now.)

For the case of `BigInt`, we want our iterator to iterate over the digits in normal, notational order: The most-significant digit comes first. So, we have to write down some type, and implement `Iterator` for it such that `next` returns the digits one-by-one. Clearly, the iterator must somehow be able to access the number it iterates over, and it must store its current location. However, it cannot *own* the `BigInt`, because then the number would be gone after iteration! That'd certainly be bad. The only alternative is for the iterator to *borrow* the number, so it takes a reference.

In writing this down, we again have to be explicit about the lifetime of the reference: We can't just have an `Iter`, we must have an `Iter<'a>` that borrows the number for lifetime `'a`. This is our first example of a data-type that's polymorphic in a lifetime, as opposed to a type.
`usize` here is the type of unsigned, pointer-sized numbers. It is typically the type of "lengths of things", in particular, it is the type of the length of a `Vec` and hence the right type to store an offset into the vector of digits.

Now we are equipped to implement `Iterator` for `Iter`.

We choose the type of things that we iterate over to be the type of digits, i.e., `u64`.

First, check whether there's any more digits to return.

We already returned all the digits, nothing to do.

Otherwise: Decrement, and return next digit.

All we need now is a function that creates such an iterator for a given `BigInt`.

Notice that when we write the type of `iter`, we don't actually have to give the lifetime parameter of `Iter`. Just as it is the case with functions returning references, you can elide the lifetime. The rules for adding the lifetimes are exactly the same. (See the last section of part 06.)

We are finally ready to iterate! Remember to edit `main.rs` to run this function.

Of course, we don't have to use `for` to apply the iterator. We can also explicitly call `next`.

`loop` is the keyword for a loop without a condition: It runs endlessly, or until you break out of it with `break` or `return`.

Each time we go through the loop, we analyze the next element presented by the iterator - until it stops.

Now, it turns out that this combination of doing a loop and a pattern matching is fairly common, and Rust provides some convenient syntactic sugar for it.

`while let` performs the given pattern matching on every round of the loop, and cancels the loop if the pattern doesn't match. There's also `if let`, which works similar, but of course without the loopy part.

Exercise 09.1: Write a testcase for the iterator, making sure it yields the corrects numbers.

Exercise 09.2: Write a function `iter_ldf` that iterates over the digits with the least-significant digits coming first. Write a testcase for it.

Iterator invalidation and lifetimes

You may have been surprised that we had to explicitly annotate a lifetime when we wrote `Iter`. Of course, with lifetimes being present at every reference in Rust, this is only consistent. But do we at least gain something from this extra annotation burden? (Thankfully, this burden only occurs when we define *types*, and not when we define functions - which is typically much more common.)

It turns out that the answer to this question is yes! This particular aspect of the concept of lifetimes helps Rust to eliminate the issue of *iterator invalidation*. Consider the following piece of code.

If you enable the bad line, Rust will reject the code. Why? The problem is that we are modifying the number while iterating over it. In other languages, this can have all sorts of effects from inconsistent data or throwing an exception (Java) to bad pointers being dereferenced (C++). Rust, however, is able to detect this situation. When you call `iter`, you have to borrow `b` for some lifetime `'a`, and you obtain `Iter<'a>`. This is an iterator that's only valid for lifetime `'a`. Gladly, we have this annotation available to make such a statement. Rust enforces that `'a` spans every call to `next`, which means it has to span the loop. Thus `b` is borrowed for the duration of the loop, and we cannot mutate it. This is yet another example for how the combination of mutation and aliasing leads to undesired effects (not necessarily crashes, think of Java), which Rust successfully prevents.

Iterator conversion trait

If you closely compare the `for` loop in `main` above, with the one in `part06::vec_min`, you will notice that we were able to write `for e in v` earlier, but now we have to call `iter`. Why is that? Well, the `for` sugar is not actually tied to `Iterator`. Instead, it demands an implementation of `IntoIterator`. That's a trait of types that provide a *conversion* function into some kind of iterator. These conversion traits are a frequent pattern in Rust: Rather than demanding that something is an iterator, or a string, or whatever; one demands that something can be converted to an iterator/string/whatever. This provides convenience similar to overloading of functions: The function can be called with lots of different types. By implementing such traits for your types, you can even make your own types work smoothly with existing library functions. As usually for Rust, this abstraction comes at zero cost: If your data is already of the right type, the conversion function will not do anything and trivially be optimized away.

If you have a look at the documentation of `IntoIterator`, you will notice that the function `into_iter` it provides actually consumes its argument. So we implement the trait for *references to* numbers, such that the number is not lost after the iteration.

With this in place, you can now replace `b.iter()` in `main` by `&b`. Go ahead and try it!
Wait, `&b`? Why that? Well, we implemented `IntoIterator` for `&BigInt`. If we are in a place where `b` is already borrowed, we can just do `for digit in b`. If however, we own `b`, we have to create a reference to it. Alternatively, we could implement `IntoIterator` for `BigInt` - which, as already mentioned, would mean that `b` is actually consumed by the iteration, and gone. This can easily happen, for example, with a `Vec`: Both `Vec` and `&Vec` (and `&mut Vec`) implement `IntoIterator`, so if you do `for e in v`, and `v` has type `Vec`, then you will obtain ownership of the elements during the iteration - and destroy the vector in the process. We actually did that in `part01::vec_min`, but we did not care. You can write `for e in &v` or `for e in v.iter()` to avoid this.


```
use part05::BigInt;
```

```
pub struct Iter<'a> {
    num: &'a BigInt,
    idx: usize, // the index of the last number that was returned
}
```

```
impl<'a> Iterator for Iter<'a> {

    type Item = u64;

    fn next(&mut self) -> Option<u64> {
        if self.idx == 0 {

            None
        } else {

            self.idx = self.idx - 1;
            Some(self.num.data[self.idx])
        }
    }
}
impl BigInt {

    fn iter(&self) -> Iter {
        Iter { num: self, idx: self.data.len() }
    }
}
```

```
pub fn main() {
    let b = BigInt::new(1 << 63) + BigInt::new(1 << 16) + BigInt::new(1 << 63);
    for digit in b.iter() {
        println!("{}", digit);
    }
}
fn print_digits_v1(b: &BigInt) {
    let mut iter = b.iter();

    loop {


        match iter.next() {
            None => break,
            Some(digit) => println!("{}", digit)
        }
    }
}
fn print_digits_v2(b: &BigInt) {
    let mut iter = b.iter();

    while let Some(digit) = iter.next() {
        println!("{}", digit)
    }
}
```

```
fn iter_invalidation_demo() {
    let mut b = BigInt::new(1 << 63) + BigInt::new(1 << 16) + BigInt::new(1 << 63);
    for digit in b.iter() {
        println!("{}", digit);
        /*b = b + BigInt::new(1);*/                                 /* BAD! */
    }
}
```

```
impl<'a> IntoIterator for &'a BigInt {
    type Item = u64;
    type IntoIter = Iter<'a>;
    fn into_iter(self) -> Iter<'a> {
        self.iter()
    }
}
```