Week 07 Weekly Exercises

Objectives

- Use and Understand the different Fn traits
- Practical usage of beginner and complex macros
- Theoretical understanding on the advantages of macros as a method of metaprogramming

---

Activities To Be Completed

The following is a list of all the markedactivities available to complete this week...

- **Fun types!**
- **Average Macro**
- **Repetition**
- **Currying**
- **Hooked - our small CPU**

The following practice activities are optional and are not **marked, or required** to be completed for the week.

> **None - all exercises this week are marked.**

Preparation

Before attempting the weekly exercises you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this week's exercises called `lab07`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab07
$ cd lab07
$ 6991 fetch lab 07
```

Or, if you're not working on CSE, you can download the provided code as a tar file.

EXERCISE:

Fun types!

NOTE:

Objectives

- Learn about the different Fn traits (Fn, FnMut, FnOnce)
- Use generic trait bounds to accept closures as arguments

In this exercise, you'll be implementing some standard library methods that accept closures as arguments. You'll be using the `Fn` traits to do this!

As a reminder, there are three `Fn` traits. To quote from the book

> 1. `FnOnce` applies to closures that can be called once. All closures implement at least this trait, because all closures can be called. A closure that moves captured values out of its body will only implement `FnOnce` and none of the other Fn traits, because it can only be called once.
> 2. `FnMut` applies to closures that don't move captured values out of their body, but that might mutate the captured values. These closures can be called more than once.
> 3. `Fn` applies to closures that don't move captured values out of their body and that don't mutate captured values, as well as closures that capture nothing from their environment. These closures can be called more than once without mutating their environment, which is important in cases such as calling a closure multiple times concurrently.

1. Implement `MyOption::map` You've been given a `MyOption` enum that is similar to the standard library's `Option` enum. Implement the map method on `MyOption` that takes a closure as an argument and returns a `MyOption` with the closure applied to the value inside the `Some` variant. If the `MyOption` is `None`, then return `None`.

You should not use the standard library's `Option::map` method in your implementation.

2. Implement `MyVec::map` You've been given a `MyVec` struct that is a wrapper over a `Vec`. Implement the `map` method on `MyVec` that takes a closure as an argument, and applies that closure to each element in the `Vec`. The `map` method should act "in place", and should not return anything.

You should not use the standard library's `Vec::map` method in your implementation.

3. Implement `MyVec::for_each` Implement the `for_each` method on `MyVec` that takes a closure as an argument, and runs that closure with each element in the `Vec` The `for_each` method should not return anything.

You should not use the standard library's `Vec::for_each` method in your implementation.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 8 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Average Macro

You **should** complete the first five exercises of MakroKata **before attempting this exericse**. This exercise will be much easier having done that, and the other exercises are each tiny (less than 10 lines of code). For marks in COMP6991, all you need to do is complete this exercise.

In this exercise, we will be writing a basic rust macro!

Your task is to create a macro called `avg` that takes in comma seperated expressions, and returns the average of those expressions.

We will make the assumption that the macro is always called with expressions that are all:

- numeric
- can be converted to a numeric type for addition and division

For example:

```
fn main() {
    let a = avg!(1, 2, 3, 4, 5);
    // might expand to

    /*
    let a = {
        let mut sum = 0;
        let mut len = 0;
        sum += 1;
        len += 1;
        sum += 2;
        len += 1;
        sum += 3;
        len += 1;
        sum += 4;
        len += 1;
        sum += 5;
        len += 1;
        sum / len
    };
    */
}
```

You cannot modify the main function.

> WARNING:
>
> Autotests do not exist for this exercise. We expect that you should be able to use the tool cargo-expand to test your code.

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 8 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Repetition

The following activity is taken verbatim from exercise 6 of MakroKata - a set of exercises which you can use as a resource to learn how to write macros in rust.

Consider the below macro:

```
macro_rules! listing_literals {
    (the $e1:literal) => {
        {
            let mut my_vec = Vec::new();
            my_vec.push($e1);
            my_vec
        }
    };
    (the $e1:literal and the $e2:literal) => {
        {
            let mut my_vec = Vec::new();
            my_vec.push($e1);
            my_vec.push($e2);
            my_vec
        }
    };
    (the $e1:literal and the $e2:literal and the $e3:literal) => {
        {
            let mut my_vec = Vec::new();
            my_vec.push($e1);
            my_vec.push($e2);
            my_vec.push($e3);
            my_vec
        }
    }
}

fn main() {
    let vec: Vec<&str> = listing_literals!(the "lion" and the "witch" and the "wardrobe");
    assert_eq!(vec, vec!["lion", "witch", "wardrobe"]);
    let vec: Vec<i32> = listing_literals!(the 9 and the 5);
    assert_eq!(vec, vec![9, 5]);
}
```

This is very clunky, and involves a large amount of repeated code. Imagine doing this for 10 arguments! What if we could say that we want a variable number of a particular patterns. That would let us say "give me any number of $e:expr tokens, and I'll tell you what to do with them'".

In this task, you will be creating an `if_any!` macro. If any of the first arguments are true, it should execute the block which will always be the last argument. You may not edit the `main` function; but once you have completed the exercise, your `if_any!` macro should expand to look like the following:

```
fn main() {
    // your macro call:
    //
    // if_any!(false, 0 == 1, true; {
    //      print_success();
    // })
    //
    // should expand to:
    if (false || 0 == 1 || true) {
        print_success();
    }
}
```

You can test this yourself, by running:

```
$ 6991 cargo expand main
fn main() {
    if (false || 0 == 1 || true) {
        print_success();
    }
}
```

> WARNING:
>
> Autotests do not exist for this exercise. We expect that you should be able to use the tool cargo-expand to test your code!

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 8 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

CHALLENGE EXERCISE:

Currying

> NOTE:
>
> The following activity is taken verbatim from exercise 11 of MakroKata - a set of exercises which you can use as a resource to learn how to write macros in rust.
>
> The original question text (q11) has more guidance which was not included in the exercise overview, and will serve as a good place to start if you would like hints!

This exercise is a sort of culmination of everything you've learned so far about macros. To complete it, you'll need to note one important fact -- macros can recurse into themselves.

```
enum LinkedList {
    Node(i32, Box<LinkedList>),
    Empty
}

macro_rules! linked_list {
    () => {
        LinkedList::Empty
    };
    ($expr:expr $(, $exprs:expr)*) => {
        LinkedList::Node($expr, Box::new(linked_list!($($exprs),*)))
    }
}

fn main() {
    let my_list = linked_list!(3, 4, 5);
}
```

The above example is very typical. The first rule is the "base case" -- an empty list of tokens implies an empty linked list. The second rule always matches one expression first (expr). This allows us to refer to it on its own, in this case to create the Node. The rest of the expressions (exprs) are stored in a repetition; and all we'll do with them is recurse into `linked_list!()`. If there's no expressions left, that call to linked_list!() will give back `Empty`; otherwise it'll repeat the same process. While Macro Recursion is incredibly powerful; it is also slow. As a result, there is a limit to the amount of recursion you are allowed to do. In rustc, the limit is 128; but you can configure by adding

`#![recursion_limit = "256"]` as a crate level attribute (i.e. - the first line of `main.rs`).

Before you complete the exercise, let's briefly discuss a concept called "currying". If you're already familiar with the concept, perhaps from your own experience of functional programming; you can skip until the exercise description.

In most imperative languages, the syntax to call a function with multiple arguments is `function(arg1, arg2, arg3)`. If you do not provide all the arguments, that is an error.

In many functional languages, however, the syntax for function calls is more akin to `function(arg1)(arg2)(arg3)`. The advantage of this notation is that if you specify less than the required number of arguments; it's not an error -- you get back a function that takes the rest of the arguments. A function that behaves this way is said to be "curried" (named after Haskell Curry, a famous mathematician).

A good example of this is a curried add function. In regular Rust, we'd say add is `move |a, b| a + b` is the add function.

If we curried that function, we'd have add is

`move |a| move |b| a + b`. What this means is that we can write

`let add_1 = add(1);` and we now have a function at `add_1` which, when called, will add 1 to anything.

In this exercise, you will build a macro which creates a curried function. The syntax for this function will be `curry!((a: i32) => (b: i32) => _, {a + b})`. Each pair of ident: `ty` is an argument; and the last `__` indicates that the compiler will infer the return type. The block provided last is, of course, the computation we want to do after receiving all the arguments. You can test your solution yourself, by running cargo expand, which should output:

```
$ 6991 cargo expand main
```

```rust
fn main() {
    let is_between = move |min: i32| move |max: i32| move |item: &i32| {
        min < *item && *item < max
    };
    let curry_filter_between = move |min: i32| move |max: i32| move |vec: &Vec<i32>| {
        let filter_between = is_between(min)(max);
        vec.iter()
            .filter_map(|i| if filter_between(i) { Some(*i) } else { None })
            .collect()
    };
    let between_3_7 = curry_filter_between(3)(7);
    let between_5_10 = curry_filter_between(5)(10);
    let my_vec = get_example_vec();
    let some_numbers: Vec<i32> = between_3_7(&my_vec);
    print_numbers(&some_numbers);
    let more_numbers: Vec<i32> = between_5_10(&my_vec);
    print_numbers(&more_numbers);
}
```

> WARNING:
>
> Autotests do not exist for this exercise. We expect that you should be able to use the tool cargo-expand to test your code.

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 8 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

CHALLENGE EXERCISE:

Hooked - our small CPU

In this exercise, you are given a basic CPU emulator, that reads in instructions, and executes them one by one.

You have been tasked to extend this CPU to support the new `Callback` instruction, which allows the CPU to execute a closure some amount of instructions in the future.

For example, if the CPU is currently at instruction 10, and the next instruction is

`Callback(Hook::new(2, |cpu| { println!("Hello World!"); }))`

Then the CPU should execute the next two instructions, and **then** execute the closure, printing "Hello World!".

The CPU should then continue executing instructions as normal.

You may modify whatever code you like, except for the main function, which should remain unchanged. A `Hook` wrapper type has been provided for you, which you should use to implement the `Callback` instruction.

The CPU should be able to execute the following program (test 3):

```
Instruction::Nop,
Instruction::PrintAccumulator,
Instruction::AddLiteral(1),
Instruction::Callback(Hook::new(2, |cpu: &mut Cpu| {
    cpu.accumulator += 6991;
})),
Instruction::Nop,
Instruction::Nop,
Instruction::JumpIfCondition(|cpu| cpu.accumulator <= 6991, 3.into()),
Instruction::SubLiteral(1),
Instruction::PrintAccumulator,
Instruction::Quit,
```

With the output:

```
        ...no-op
        ...print accumulator
Accumulator: 0
        ...adding 1
        ...callback instruction
        ...no-op
        ...no-op
        ...conditional jump
        ...subtracting 1
        ...print accumulator
Accumulator: 6991
```

You can run tests with the following command:

```
$ 6991 cargo run -- 1
  Compiling hooked v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.35s
     Running `target/debug/hooked 1`
        ...no-op
        ...print accumulator
Accumulator: 0
        ...adding 1
        ...print accumulator
Accumulator: 1
        ...conditional jump
```

> DANGER:
>
> This exercise will involve knowledge of Rust's ownership system, and understanding of the different Fn, FnMut, and FnOnce traits.
> It is a challenge exercise, and if you do not complete the exercise you will only lose a small fraction of course marks.
>
> You are welcome to work with peers on this exercise, but you must write your own code.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 8 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

Submission

When you are finished each exercise make sure you submit your work by running `give`.

You can run `give` multiple times.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

The due date for this week's exercises is **Week 8 Wednesday 21:00:00**.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

Automarking will be run continuously throughout the term, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run you can view your results here or by running this command on a CSE machine:

```
$ 6991 classrun -sturec
```
