Functional programming features

haskell-style

1: Lambdas

Functions inside functions

```
fn call_println(x: usize, y: usize) {
   println!("{x}, {y}");
}

fn add(a: usize, b: usize) -> usize {
   a + b
}

fn print_and_call(a: usize, b: usize) -> usize {
   call_println(a, b);
   add(a, b)
}
```

```
fn print_and_call(a: usize, b: usize) -> usize {
   fn call_println(x: usize, y: usize) {
       println!("{x}, {y}");
   }

   fn add(a: usize, b: usize) -> usize {
       a + b
   }

   call_println(a, b);
   add(a, b)
}
```

Functions inside functions - lambda

```
fn call_println(x: usize, y: usize) {
   println!("{x}, {y}");
}

fn add(a: usize, b: usize) -> usize {
   a + b
}

fn print_and_call(a: usize, b: usize) -> usize {
   call_println(a, b);
   add(a, b)
}
```

```
fn print_and_call(a: usize, b: usize) -> usize {
   let call_println = |x: usize, y: usize| {
       println!("{x}, {y}");
   };

   let add = |a: usize, b: usize| -> usize {
       a + b
   };

   call_println(a, b);
   add(a, b)
}
```

Functions inside functions - lambda

```
fn call_println(x: usize, y: usize) {
   println!("{x}, {y}");
}

fn add(a: usize, b: usize) -> usize {
   a + b
}

fn print_and_call(a: usize, b: usize) -> usize {
   call_println(a, b);
   add(a, b)
}
```

```
fn print_and_call(a: usize, b: usize) -> usize {
   let call_println = |x: usize, y: usize| {
       println!("{x}, {y}");
   };

   let add = |a: usize, b: usize| { a + b };

   call_println(a, b);
   add(a, b)
}
```

Functions inside functions - lambda

```
fn call_println(x: usize, y: usize) {
   println!("{x}, {y}");
}

fn add(a: usize, b: usize) -> usize {
   a + b
}

fn print_and_call(a: usize, b: usize) -> usize {
   call_println(a, b);
   add(a, b)
}
```

```
fn print_and_call(a: usize, b: usize) -> usize {
   let call_println = |x: usize, y: usize| {
       println!("{x}, {y}");
   };

   let add = |a: usize, b: usize| a + b;

   call_println(a, b);
   add(a, b)
}
```

So… Lamba are just sugar syntax for functions ?

No, lambdas can *capture* variables

```
fn print_and_call(a: usize, b: usize) -> usize {
    let call_println = |x: usize, y: usize| {
        println!("{x}, {y}");
    };

    let add = |a: usize, b: usize| a + b;

    call_println(a, b);
    add(a, b)
}
```

```
fn print_and_call(a: usize, b: usize) -> usize {
    let call_println = |x: usize, y: usize| {
        println!("{x}, {y}");
    };

    let another_variable = 4;
    let add = |a: usize, b: usize| a + b + another_variable;

    call_println(a, b);
    add(a, b)
}
```

So… Lamba are just sugar syntax for functions ?

No, lambdas that *capture* variables cannot be used as functions

```
fn call_function(f: fn(usize) -> usize, parameter: usize) -> usize {
   f(parameter)
}

fn add_one(x: usize) -> usize {
   x + 1
}

pub fn main() {









}
```

So… Lamba are just sugar syntax for functions ?

No, lambdas that *capture* variables cannot be used as functions

```
fn call_function(f: fn(usize) -> usize, parameter: usize) -> usize {
   f(parameter)
}

fn add_one(x: usize) -> usize {
   x + 1
}

pub fn main() {
   // OK (pass a function)
   println!("{}", call_function(add_one, 4));









}
```

So… Lamba are just sugar syntax for functions ?

No, lambdas that *capture* variables cannot be used as functions

```
fn call_function(f: fn(usize) -> usize, parameter: usize) -> usize {
    f(parameter)
}

fn add_one(x: usize) -> usize {
    x + 1
}

pub fn main() {
    // OK (pass a function)
    println!("{}", call_function(add_one, 4));

    // OK (pass a lambda without capture)
    let add_two = |x| x + 2;
    println!("{}", call_function(add_two, 4));




}
```

So… Lamba are just sugar syntax for functions ?

No, lambdas that *capture* variables cannot be used as functions

```
fn call_function(f: fn(usize) -> usize, parameter: usize) -> usize {
   f(parameter)
}

fn add_one(x: usize) -> usize {
   x + 1
}

pub fn main() {
   // OK (pass a function)
   println!("{}", call_function(add_one, 4));

   // OK (pass a lambda without capture)
   let add_two = |x| x + 2;
   println!("{}", call_function(add_two, 4));

   // KO (pass a lambda with a capture)
   let seven = 7;
   let add_seven = |x| x + seven;
   println!("{}", call_function(add_seven, 4));
}
```

So… Lamba are just sugar syntax for functions ?

You can return a lambda

```
pub fn main() {
    let add_seven = {
        let add_seven = |x| x + 7;
        add_seven
    };

    println!("{}", add_seven(4));
}
```

But not if it captures a variable by reference

```
pub fn main() {
    let add_seven = {
        let seven = 7;
        let add_seven = |x| x + seven; // does not compile
        add_seven
    };

    println!("{}", add_seven(4));
}
```

So… Lamba are just sugar syntax for functions ?

You can return a lambda

```
pub fn main() {
    let add_seven = {
        let add_seven = |x| x + 7;
        add_seven
    };

    println!("{}", add_seven(4));
}
```

But not if it captures a variable by reference

```
pub fn main() {
    let add_seven = {
        let seven = 7;
        let add_seven = move |x| x + seven;
        add_seven
    };

    println!("{}", add_seven(4));
}
```

Some lambdas can only be called once

```rust
pub fn main() {
    let print = {
        let string = String::from("hello");
        let print = move || {
            println!("{string}");
        };
        print
    };

    print();
    print();  // fine
}
```

```rust
pub fn main() {
    let print = {
        let string = String::from("hello");
        let print = move || {
            println!("{string}");
            drop(string);  // hum...
        };
        print
    };

    print();
    print();  // does not compile
}
```

Lambdas with capture are powerful, but tricky

- they are harder to understand
- they cannot replace all functions
- what they reference must be valid
- if they move a value, they may be call only once

Do not replace all your functions with lambda.

Lambda recap

	functions	lambda without capture	lambda with capture
can be defined inline	yes	yes	yes
multiple execution	yes	yes	yes/no
passed as function	yes	yes	no
IDE support	yes	no	no
use external variable	no	no	yes

2: Iterators

Yet again

The Iterator trait

https://doc.rust-lang.org/std/iter/trait.Iterator.html

You implement one method, you get a lot for free

(You may want to implement some manually)

You can decide if you want to move the object you iterate on by calling iter(), itert_mut() or into_iter(). These method return the appropriate iterator.

Required Methods

next

Provided Methods

advance_by
all
any
array_chunks
by_ref
chain
cloned
cmp
cmp_by
collect
collect_into
copied
count
cycle
enumerate
eq
eq_by
filter
filter_map
find

3: filter, map, reduce

Filter, map, reduce

A lot of programming tasks are in the form of

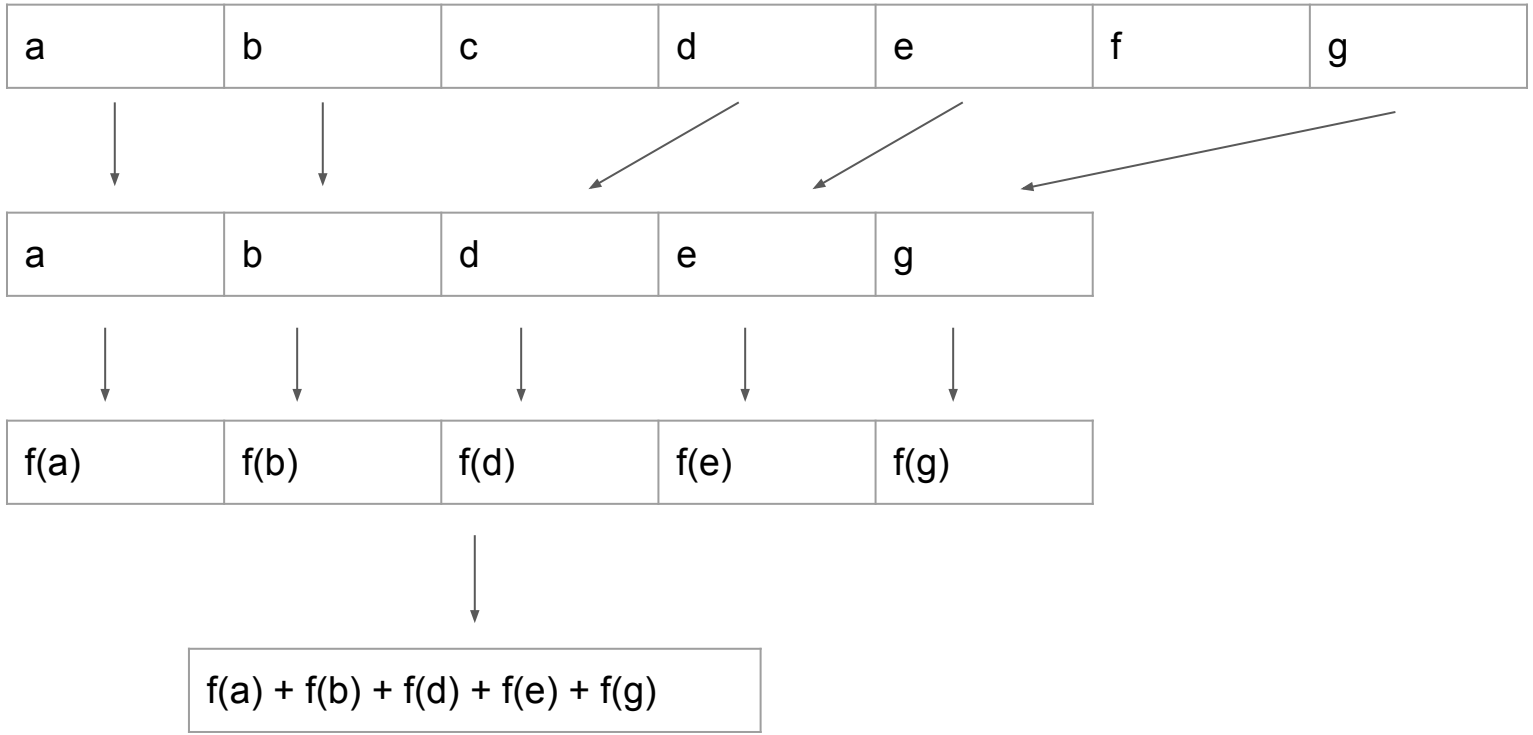
- iterate over elements
- take some based on a condition
- apply a function on the selected elements
- group them in a vector / sum them / count them

Filter, map, reduce

A lot of programming tasks are in the form of

- iterate over elements (iterator)
- take some based on a condition (filter)
- apply a function on the selected elements (map)
- group them in a vector / sum them / count them (reduce)

Filter, map, reduce



Example

```
#[derive(Debug)]
struct CreditCard {
   holder_name: String,
}

#[derive(Debug)]
struct Client {
   name: String,
   credit_card: CreditCard,
}
```

```
pub fn main() {
   let v = vec![
       Client {
           name: String::from( "John"),
           credit_card: CreditCard {
               holder_name: String::from( "John"),
           },
       },
       // etc
   ];

   let different_name = v
       .iter()
       .filter(|client| client.name != client.credit_card.holder_name);
   for client in different_name {
       println!("{:?}", client);
   }

   let different_name: Vec<&String> = v
       .iter()
       .filter(|client| client.name != client.credit_card.holder_name)
       .map(|client| &client.name)
       .collect();
   for client in &different_name {
       println!("{:?}", client);
   }
   println!("{:?}", different_name.capacity());
}
```

OK, but why?

It is functionally the same as a plain old for loop, but:

- it provides intention
- you remove unnecessary variables
- bugs are harder to write
- bound checks might be removed
- you enter an ecosystem of libraries

4: parallelisation

First step into parallel programming

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter() // <-- just change that!
          .map(|&i| i * i)
          .sum()
}
```

First step into parallel programming

```rust
fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter() // <-- just change that!
        .map(|&i| i * i)
        .sum()
}
```

```rust
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter() // <-- !!!
        .map(|&i| i * i)
        .sum()
}
```

First step into parallel programming

This does not compile… Why?

```
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    let mut sum: i32 = 0;
    input.par_iter().for_each(|&i| {
        // parallel code
        sum += i * i;
    });
    sum
}
```

First step into parallel programming

You need a mutex

```rust
use rayon::prelude::*;
use std::sync::Mutex;

fn sum_of_squares(input: &[i32]) -> i32 {
    let sum_mutex = Mutex::new(0);
    input.par_iter().for_each(|&i| {
        // parallal code
        let mut sum = sum_mutex.lock().unwrap();
        *sum += i * i;
    });
    let sum = sum_mutex.lock().unwrap();
    *sum
}
```

First step into parallel programming

```
fn sum_of_squares(input: &Vec<String>) -> Vec<String> {
    let mut vec = Vec::new();

    let vec_mutex = Mutex::new(&mut vec);
    input.par_iter().for_each(|string| {
        // parallal code
        let mut vec = vec_mutex.lock().unwrap();
        let string_upppercase = string.to_uppercase();
        vec.push(string_upppercase);
    });

    vec
}
```

First step into parallel programming

```
fn sum_of_squares(input: &Vec<String>) -> Vec<String> {
    let mut vec = Vec::new();

    let vec_mutex = Mutex::new(&mut vec);
    input.par_iter().for_each(|string| {
        // parallal code
        let string_upppercase = string.to_uppercase();
        // only keep the mutex for as few as possible
        let mut vec = vec_mutex.lock().unwrap();
        vec.push(string_upppercase);
    });

    vec
}
```

Remember to order your lock

- ordering your lock prevent deadlock
- deadlocks are hard to catch at compile time and during execution

You can also spawn thread manually

```
rayon::scope(|s| {
    for i in 0..n {
        s.spawn(move |_| {
            // parallel code
        });
    }
});
```

Remember that spawning threads has a cost

Threads have an overhead. Spawning them for a small workload usually lead to a slowdown in you program.

But you can batch your iterators!

```
let chunks = sequences.into_iter().chunks(BATCH_SIZE);
```