


# Discussion 5

## Discussion 10 - Thursday, July 17th

### Reminders

- Project 5 due **Friday, July 25th, 11:59PM**

### Lambda Calculus Review

Refer to [Cliff's Lambda Calculus Notes](https://bakalian.cs.umd.edu/assets/notes/lambdacalc.pdf)  (<https://bakalian.cs.umd.edu/assets/notes/lambdacalc.pdf>) for a complete resource.

### Bound vs. Free Variables

- A bound variable is one whose value is dependent on a parameter of a lambda function.
- A free variable is a variable that is not a bound variable: its value is **independent** of lambda function parameters.



```
(λx. x a ) b
  ↑ x is a bound variable

(λx. x a ) b
  ↑ a is a free variable

(λx. x a ) b
  ↑ b is a free variable
```

### Explicit Rules to remove ambiguity

- Expressions  $e$  are left associative

$a\ b\ c$  means  $(a\ b)\ c$

- The scope of a function goes until the **end of the entire expression** or until a (unmatched) parenthesis is reached

$\lambda x.\lambda y. a\ b$  means  $\lambda x. (\lambda y. (a\ b))$

### Alpha Conversion

- An alpha-conversion  $\alpha$  is the process of renaming all the variables that are bound together along with the bounded parameter to a different name to increase readability.

$(\lambda x. (\lambda x. (\lambda y. x\ y))\ x)\ x \rightarrow$   
 $(\lambda a. (\lambda b. (\lambda y. b\ y))\ a)\ x$       Alpha Conversion

## Beta Reduction

- The process of applying a function is called reducing. We call a function call a beta reduction  $\beta$ .

```

( $\lambda x. \lambda y. y x$ )  $a b \rightarrow$ 
(( $\lambda x. \lambda y. y x$ )  $a$ )  $b \rightarrow$ 
( $\lambda y. y a$ )  $b \rightarrow$ 
 $b a$ 

```

Left associativity  
Beta reduce the x function  
Beta reduce the y function

- When you cannot reduce any further, we say the expression is in beta normal form.

## Exercises

Make the parentheses explicit in the following expressions:

- $a b c$
- $\lambda a. \lambda b. c b$
- $\lambda a. a b \lambda a. a b$

► Solutions!

Identify the free variables in the following expressions:

- $\lambda a. a b a$
- $a (\lambda a. a) a$
- $\lambda a. (\lambda b. a b) a b$

► Solutions!

Apply alpha-conversions to the following:

- $\lambda a. \lambda a. a$
- $(\lambda a. a) a b$
- $(\lambda a. (\lambda a. (\lambda a. a) a) a)$

► Solutions!

Apply beta-reductions to the following:

- $(\lambda a. a b) x b$
- $(\lambda a. b) (\lambda a. \lambda b. \lambda c. a b c)$
- $(\lambda a. a a) (\lambda a. a a)$



## ► Solutions!

Consider the following subset of the church encodings:

image

Convert the following sentences to their equivalent lambda calc encodings:

1. `if true then false else true`
2. `if false then if false then false else true else false`

Next, reduce the lambda calc expressions to their simplest form. Verify that the resulting expression is equivalent to the english sentence when evaluated.

## ► Solutions!

## CBV (Eager) and CBN (Lazy)

When performing beta reduction with arguments that can be evaluated further, you have two options: Eager Evaluation or Lazy Evaluation Take the following example:

```
(λx . a x a) ((λy . y y) z)    (taken from Cliff's notes)
```



In **eager/call-by-value evaluation**, you start by reducing the argument if possible.

```
(λx . a x a) ((λy . y y) z) (we'll evaluate eagerly!)
(λx . a x a) (z z) ((λy . y y) z gets evaluated to (z z))
a (z z) a (pass (z z) into (λx . a x a))
```

In **lazy/call-by-name evaluation**, you don't reduce the argument, you just pass it in.

```
(λx . a x a) ((λy . y y) z) (we'll evaluate lazily!)
a ((λy . y y) z) a (passing ((λy . y y) z) in as x)
a (z z) a (evaluate (λy . y y) z)
```

Lazy and eager will evaluate to the same beta normal form, if there is one. If there's an infinite loop, we say there is no beta normal form, and lazy and eager may evaluate differently. Take the following example:

```
(λx. y)((λx. x x)(λx. x x))
```

Try evaluating it with both lazy and eager evaluation! What happens?

## ► Solution!






Bonus: alpha equivalence problem. Fully reduce the following example to beta normal form. Be careful!

```
((λx . λy. x y) y) z
```

► Solution!

# Rust

## Rust References

- [Rust Book](https://doc.rust-lang.org/book/)  [\(https://doc.rust-lang.org/book/\)](https://doc.rust-lang.org/book/)
- [Simple Version of Rust Book](https://www.cs.brandeis.edu/~cs146a/rust/doc-02-21-2015/book/README.html)   [\(https://www.cs.brandeis.edu/~cs146a/rust/doc-02-21-2015/book/README.html\)](https://www.cs.brandeis.edu/~cs146a/rust/doc-02-21-2015/book/README.html)
- [Rust Standard Library](https://doc.rust-lang.org/std/index.html)   [\(https://doc.rust-lang.org/std/index.html\)](https://doc.rust-lang.org/std/index.html)
- [Online Rust Playground](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)   [\(https://play.rust-lang.org/?version=stable&mode=debug&edition=2021\)](https://play.rust-lang.org/?version=stable&mode=debug&edition=2021)
- [Anwar's Rust Intro](https://bakalian.cs.umd.edu/assets/slides/00-rust-introduct.pdf)   [\(https://bakalian.cs.umd.edu/assets/slides/00-rust-introduct.pdf\)](https://bakalian.cs.umd.edu/assets/slides/00-rust-introduct.pdf)

## Properties of Rust

Rust is type-safe, meaning a well-typed program has a defined behavior and does not get "crash".

Languages like C are not type-safe. This is due to the ability for pointers to point to garbage parts of memory.

- Rust's type safety limits some of its capabilities for the sake of this safety
- The **unsafe** keyword can be used to write unsafe Rust.

## Rust Syntax

### Statements

```
// Here is how print statements will be written:
println!("This statement will be printed!");
```

Note that the '!' symbol is placed at this "function" call, denoting that this is not a normal function. This is actually a **macro**, meaning the code will be replaced at compile time.

As with most languages, string literals are wrapped with quotation marks.

The end of this statement has a semicolon, denoting the end of the expression.

### Functions

```
// This is the main function.
fn main() {
}
```

The main function is where every Rust program starts.

Similar to most languages you've seen before, functions in Rust are denoted using parentheses for their input parameters and curly braces wrapping the body.

```
fn add1(x: i32, y: i32) -> i32 {
    x + y // Note lack of semi-colon. That means this expression is returned! Equivalent to `return x + y;`
}
// to call this:
add1(2, 3) // will return 5
```


## Variables and Bindings

```
let x = 5; // x: i32
```

In this example, we are binding the value 5 to the variable x. Rust has type inference, so we don't need to explicitly state the type of x here.

However, we can still explicitly give the type:

```
let x: i32 = 5;
```

[Here](https://doc.rust-lang.org/book/ch03-02-data-types.html)  (<https://doc.rust-lang.org/book/ch03-02-data-types.html>) is some more information Rust's primitive types.



As with OCaml, bindings are effectively a type of pattern matching. Because of this, we can still do things like this:

```
let (x, y) = (1, 2);
```

## Mutable Variables

In Rust, bindings are not mutable by default. This following code will not compile:

```
let x = 5;
x = 6;
```

Attempting to compile this code will give this error:

```
error: re-assignment of immutable variable `x`
    x = 6;
    ^~~~~
```

If you want to make a variable mutable, you must use the **mut** keyword.

```
let mut x = 5; // x is now mutable
x = 6; // this is a valid statement
```

## Conditionals and Loops

```
let mut x = 2;
x = if x < 0 { 10 } else { x }
```

If expressions in Rust do not have parentheses around the guard clause. Note that conditional branches must evaluate to the same type as is in OCaml.

```
let mut x = 10;
loop {
    if x <= 0 {
        break;
    }
    println!("This statement is printed 10 times.");
    x = x - 1;
}
```

Using the generic "loop" is an infinite loop, which can only be exited using break. Similar to other languages, `break;` exits from the loop and `continue;` jumps to the next loop iteration.

```
let mut x = 10;
while x > 0 {
    println!("This statement is printed 10 times.");
    x = x - 1;
}
```



Similar to if statements, while loops don't have parentheses around the guard. Observe that iterating using `loop {}` is the same as iterating with `while true {}`.

```
for x in 0..10 {
    println!("{}", x); // prints numbers 0, 1, 2, ..., 8, 9
}
```

For loops in Rust are analogous to "for each" loops in Java. If we are looking for variables in a range of numbers, remember that it is **inclusive** of the first number and **exclusive** of the last number.

Abstractly, they follow the structure:

```
for var in expression {
    // code
}
```

Additionally, you can loop through an iterable object in the following way:

```
some_list.iter().foreach(|x| //do something )
```

This syntax is called a closure, and (for our purposes) is equivalent to ocaml's anonymous function `(fun x -> //do something)`

We can also call map/fold like in ocaml to do iteration!

```
let x = vec![1,2,3]; // a vec is a list of a variable size! This line is creating a vec of the list 1,2,3

let y: Vec<i32> = x.iter().map(|x| x + 1).collect(); //collect() is called to turn it from a "map object" type to the type we want (if it can)
let z: i32 = x.iter().fold(0, |acc, ele| acc + ele); // Here, 0 is the init value

println!("{:?}", y); // [2, 3, 4]
println!("{:?}", z); // 6
```

## Pattern Matching

```
let x = 5;

match x {
  1 => println!("one"),
  2 => println!("two"),
  3 => println!("three"),
  4 => println!("four"),
  5 => println!("five"),
  _ => println!("something else"),
}
```

Match expressions work similarly to those in OCaml, but the syntax is slightly different.

In Rust, match expressions enforce **exhaustive** checking. If the matches are not exhaustive, you will give the following error:



```
error: non-exhaustive patterns: `_` not covered
```

## Ownership

We implement copy traits (like implementing interfaces in java). Redefining variables with copy traits (primitives, bool, char) creates copy of values.

```
let x = 5; // x owns 5
let y = x; // y owns 5 (a different 5 than x)

println!("{}", y); //ok
println!("{}", x); //ok
```

Redefining variables without copy traits transfers ownership

```
let x = String::from("hello");
let y = x; // x transfers ownership to y

println!("{}", y); // ok
println!("{}", x); // fails
```

## References


When defining variables, take note of the following:

- **mut**: an object/primitive has the ability to update (ex. Array or Integer)
- **&**: Borrowing a reference for a variable, useful when you want to pass a variable to a function without updating it
- **&mut**: A mutable reference to a variable (must be defined mut). Comes particularly handy when you want to pass a variable to another function and have it updated.

At any given time, you can have either *but not both* of **one mutable reference** or **any number of immutable references**.

## Rules of References

1) At any given time, you can have either one mutable reference XOR any number of immutable references. 2) References must always be valid (no dangling references - will cover later).

Refer to [here](https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html)  (<https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html>) for more information!

An example of borrowing a reference.

```
fn main() {
    let s1 = String::from("hello");
    let len = calc_len(&s1); // lends reference
    println!("the length of {} is {}", s1, len);
}

fn calc_len(s: &String) -> usize {
    // s.push_str("hi"); fails! refs are immutable
    s.len() // s dropped; but not its referent
}
```



An example of borrowing a mutable reference.

```
fn main() {
    let mut s1 = String::from("hello");
    append_world(&mut s1); // lends mutable reference, note s1 must be mutable
    println!("s1 is now {}", s1);
}

fn append_world(s: &mut String) {
    s.push_str(" world!");
}
```

## Scopes

**Scopes** of a variable extend as far as defined/freed (between curly braces). **Lifetimes** span as long as the variable lives. A variable's lifetime ends the line after the last time it is used.

```
{
    let x = 3;
    { // x lifetime ends
        let y = 4;
        println!("{}", y);
    }
}
```



```

    } // scope of y ends, y lifetime ends

    let z = 5;
    println!("Hello World!"); // z lifetime ends
} // scope of x, z ends

```

We cannot have an immutable/mutable reference at the same time.

```

fn main() {
    let mut x = String::from("Hello");
    x.push_str(" World");
    {
        let y = &x;
        // x.push_str(", I am an alien!"); // error! x is turned to an immutable reference
        println!("{}", and {} can only read, no write", x, y);
        x.push_str(", I am an alien!"); // y lifetime ends so this is ok
    }

    x.push_str("!");
    println!("{}", is still valid", x);
}

```

In the above, x becomes immutable for as long as y lives (until line 6). Here is another example with &mut.

```

fn main() {
    let mut s1 = String::from("hello");
    {
        let s2 = &s1;
        //s2.push_str(" there"); //disallowed; s2 immutable
    } //s2 dropped
    let s3 = &mut s1; //ok since s1 mutable
    s3.push_str(" there"); //ok since s3 mutable
    // println!("String is {}",s1); //NOT OK, s3 has mutable borrow
    println!("String is {}",s3); //ok;
    println!("String is {}",s1); //ok; s3 dropped
}

```



## Compiling Rust

### Using Rustc

Rust files are compiled in the command line using **rustc**, a compiler that was built in Rust.

```
$ rustc main.rs
```

Similar to compiling C with gcc, this creates an executable file.

```
$ ls
main  main.rs
```

To execute the file, simply run `./main`.

### Using Cargo

Cargo is a Rust package manager. It can download Rust crates and invoke the compiler (rustc) on them.

Basic Cargo commands:

- `cargo check` runs a syntax check on the code
- `cargo build` compiles the code
- `cargo test` will run tests
- `cargo run` will run the code directly
  - If your code has changed since last time it was compiled, `cargo run` will recompile the code and then run it.


## Exercises

1. Write a function `is_prime` that given an integer, returns true if the integer is prime and false if the integer is composite. Then, write a `main` function that tests your `is_prime` function by printing `i is prime!` or `i is composite!` for integers `1 <= i <= 500`. Then, print the number of integers `i` we found to be prime in the format `We found [answer] primes from 1 to 500!`. Finally, compile and run your code!








Some expressions that might be useful:

- `fn is_prime(n: u32) -> bool` should be the function header
- `(n as f64).sqrt() as u32` will find the square root of n and floor it
- `%` is the modulus operator

[Click here for the solution! \(I would highly encourage you to try this problem first!! :D\)](https://github.com/cmssc330spring25/spring25/blob/main/discussions/d11_rust_basics/solution.rs)   
[\(https://github.com/cmssc330spring25/spring25/blob/main/discussions/d11\\_rust\\_basics/solution.rs\)](https://github.com/cmssc330spring25/spring25/blob/main/discussions/d11_rust_basics/solution.rs)

## Additional Readings & Resources

- [Cliff's Lambda Calculus Notes](https://bakalian.cs.umd.edu/assets/notes/lambdacalc.pdf)  [\(https://bakalian.cs.umd.edu/assets/notes/lambdacalc.pdf\)](https://bakalian.cs.umd.edu/assets/notes/lambdacalc.pdf)
- [Spring 2021 - Lambda Calculus Basics](https://www.cs.umd.edu/class/spring2021/cmssc330/lectures/24-lambda-calc-1.pdf)  [\(https://www.cs.umd.edu/class/spring2021/cmssc330/lectures/24-lambda-calc-1.pdf\)](https://www.cs.umd.edu/class/spring2021/cmssc330/lectures/24-lambda-calc-1.pdf)
- [Fall 2022 - Discussion 10 \(Lambda Calculus\)](https://github.com/umd-cmssc330/fall2022/tree/main/discussions/discussion10#lambda-calculus)  [\(https://github.com/umd-cmssc330/fall2022/tree/main/discussions/discussion10#lambda-calculus\)](https://github.com/umd-cmssc330/fall2022/tree/main/discussions/discussion10#lambda-calculus)
- [Types and Programming Languages \(Pierce 2002\)](https://www.cs.sjtu.edu.cn/~kzhu/cs383/Pierce_Types_Programming_Languages.pdf)  [\(https://www.cs.sjtu.edu.cn/~kzhu/cs383/Pierce\\_Types\\_Programming\\_Languages.pdf\)](https://www.cs.sjtu.edu.cn/~kzhu/cs383/Pierce_Types_Programming_Languages.pdf)
  - See Chapter 5: *The Untyped Lambda-Calculus*
- [nmittu.github.io - Beta Reduction Practice](https://nmittu.github.io/330-problem-generator/beta_reduction.html)  [\(https://nmittu.github.io/330-problem-generator/beta\\_reduction.html\)](https://nmittu.github.io/330-problem-generator/beta_reduction.html)

- [Online Interactive Rust Environment](https://play.rust-lang.org/) ↗ (https://play.rust-lang.org/)
- [Cliff's Rust Notes](https://bakalian.cs.umd.edu/assets/notes/rust.pdf) ↗ (https://bakalian.cs.umd.edu/assets/notes/rust.pdf)
- [Anwar's Rust Notes](https://github.com/anwarmamat/cmsc330spring2024/blob/main/rust.md) ↗ (https://github.com/anwarmamat/cmsc330spring2024/blob/main/rust.md)

