

---

# Lambda Calculus

Prepared by Mark on September 25, 2025

---

Beware of the Turing tar pit, in which everything is possible but nothing of interest is easy.

Alan Perlis, *Epigrams of Programming*, #54

## Part 1: Introduction

*Lambda calculus* is a model of computation, much like the Turing machine. As we're about to see, it works in a fundamentally different way, which has a few practical applications we'll discuss at the end of class.

A lambda function starts with a lambda ( $\lambda$ ), followed by the names of any inputs used in the expression, followed by the function's output.

For example,  $\lambda x.x + 3$  is the function  $f(x) = x + 3$  written in lambda notation.

Let's dissect  $\lambda x.x + 3$  piece by piece:

- “ $\lambda$ ” tells us that this is the beginning of an expression.  
 $\lambda$  here doesn't have a special value or definition;  
it's just a symbol that tells us “this is the start of a function.”
- “ $\lambda x$ ” says that the variable  $x$  is “bound” to the function (i.e, it is used for input).  
Whenever we see  $x$  in the function's output, we'll replace it with the input of the same name.  
This is a lot like normal function notation: In  $f(x) = x + 3$ ,  $(x)$  is “bound” to  $f$ , and we replace every  $x$  we see with our input when evaluating.
- The dot tells us that what follows is the output of this expression.  
This is much like  $=$  in our usual function notation:  
The symbols after  $=$  in  $f(x) = x + 3$  tell us how to compute the output of this function.

### Problem 1:

Rewrite the following functions using this notation:

- $f(x) = 7x + 4$
- $f(x) = x^2 + 2x + 1$

To evaluate  $\lambda x.x + 3$ , we need to input a value:

$$(\lambda x.x + 3) 5$$

This is very similar to the usual way we call functions: we usually write  $f(5)$ .

Above, we define our function  $f$  “in-line” using lambda notation, and we omit the parentheses around 5 for the sake of simpler notation.

We evaluate this by removing the “ $\lambda$ ” prefix and substituting 3 for  $x$  whenever it appears:

$$(\lambda x.x + 3) 5 = 5 + 3 = 8$$

**Problem 2:**

Evaluate the following:

- $(\lambda x.2x + 1) 4$
- $(\lambda x.x^2 + 2x + 1) 3$
- $(\lambda x.(\lambda y.9y)x + 3) 2$

*Hint:* This function has a function inside, but the evaluation process doesn’t change. Replace all  $x$  with 2 and evaluate again.

As we saw above, we denote function application by simply putting functions next to their inputs. If we want to apply  $f$  to 5, we write “ $f 5$ ”, without any parentheses around the function’s argument.

You may have noticed that we’ve been using arithmetic in the last few problems. This isn’t fully correct: addition is not defined in lambda calculus. In fact, nothing is defined: not even numbers! In lambda calculus, we have only one kind of object: the function. The only action we have is function application, which works by just like the examples above.

Don’t worry if this sounds confusing, we’ll see a few examples soon.

**Definition 3:**

The first “pure” functions we’ll define are  $I$  and  $M$ :

- $I = \lambda x.x$
- $M = \lambda x.xx$

Both  $I$  and  $M$  take one function ( $x$ ) as an input.

$I$  does nothing, it just returns  $x$ .

$M$  is a bit more interesting: it applies the function  $x$  on a copy of itself.

Also, note that  $I$  and  $M$  don’t have a meaning on their own. They are not formal functions.

Rather, they are abbreviations that say “write  $\lambda x.x$  whenever you see  $I$ .”

**Problem 4:**

Reduce the following expressions.

*Hint:* Of course, your final result will be a function.

Functions are the only objects we have!

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

**Example Solution****Solution for  $(I I)$ :**

Recall that  $I = \lambda x.x$ . First, we rewrite the left  $I$  to get  $(\lambda x.x) I$ .

Applying this function by replacing  $x$  with  $I$ , we get  $I$ :

$$I I = (\lambda x.x) I = I$$

In lambda calculus, functions are left-associative:

$(f g h)$  means  $((f g) h)$ , not  $(f (g h))$

As usual, we use parentheses to group terms if we want to override this order:  $(f (g h)) \neq ((f g) h)$

In this handout, all types of parentheses (  $()$ ,  $[\ ]$ , etc ) are equivalent.

**Problem 5:**

Rewrite the following expressions with as few parentheses as possible, without changing their meaning or structure. Remember that lambda calculus is left-associative.

- $(\lambda x.(\lambda y.\lambda z.((xz)(yz))))$
- $((ab)(cd))((ef)(gh))$
- $(\lambda x.((\lambda y.(yx))(\lambda v.v)z)u)(\lambda w.w)$

**Definition 6: Equivalence**

We say two functions are *equivalent* if they differ only by the names of their variables:

$$I = \lambda a.a = \lambda b.b = \lambda \heartsuit.\heartsuit = \dots$$

**Definition 7:**

Let  $K = \lambda a.(\lambda b.a)$ . We'll call  $K$  the "constant function function."

**Problem 8:**

That's not a typo. Why does this name make sense?

*Hint:* What is  $K x$ ?

**Problem 9:**

Show that associativity matters by evaluating  $((M K) I)$  and  $(M (K I))$ .

What would  $M K I$  reduce to?

**Currying:**

In lambda calculus, functions are only allowed to take one argument.

If we want multivariable functions, we'll have to emulate them through *currying*<sup>1</sup>

The idea behind currying is fairly simple: we make functions that return functions.

We've already seen this on the previous page:  $K$  takes an input  $x$  and uses it to construct a constant function. You can think of  $K$  as a "factory" that constructs functions using the input we provide.

**Problem 10:**

Let  $C = \lambda f. [\lambda g. (\lambda x. [ f(g(x)) ] )]$ . For now, we'll call it the "composer."

*Note:* We could also call  $C$  the "right-associator." Why?

$C$  has three "layers" of curry: it makes a function  $(\lambda g)$  that makes another function  $(\lambda x)$ .

If we look closely, we'll find that  $C$  pretends to take three arguments.

What does  $C$  do? Evaluate  $(C a b x)$  for arbitrary expressions  $a, b$ , and  $x$ .

*Hint:* Evaluate  $(C a)$  first. Remember, function application is left-associative.

**Problem 11:**

Using the definition of  $C$  above, evaluate  $C M I \star$

Then, evaluate  $C I M I$

*Note:*  $\star$  represents an arbitrary expression. Treat it like an unknown variable.

As we saw above, currying allows us to create multivariable functions by nesting single-variable functions. You may have notice that curried expressions can get very long. We'll use a bit of shorthand to make them more palatable: If we have an expression with repeated function definitions, we'll combine their arguments under one  $\lambda$ .

For example,  $A = \lambda f. [\lambda a. f(f(a))]$  will become  $A = \lambda f a. f(f(a))$

**Problem 12:**

Rewrite  $C = \lambda f. \lambda g. \lambda x. (g(f(x)))$  from ?? using this shorthand.

Remember that this is only notation. **Curried functions are not multivariable functions, they are simply shorthand!** Any function presented with this notation must still be evaluated one variable at a time, just like an un-curried function. Substituting all curried variables at once will cause errors.

<sup>1</sup>After Haskell Brooks Curry<sup>2</sup> a logician that contributed to the theory of functional computation.

<sup>2</sup>There are three programming languages named after him: Haskell, Brook, and Curry.

Two of these are functional, and one is an oddball GPU language last released in 2007.

**Problem 13:**

Let  $Q = \lambda abc.b$ . Reduce  $(Q\ a\ c\ b)$ .

*Hint:* You may want to rename a few variables.

The  $a, b, c$  in  $Q$  are different than the  $a, b, c$  in the expression!

**Problem 14:**

Reduce  $((\lambda a.a)\ \lambda bc.b)\ d\ \lambda eg.g$

## Part 2: Combinators

### Definition 15:

A *free variable* in a  $\lambda$ -expression is a variable that isn't bound to any input. For example,  $b$  is a free variable in  $(\lambda a.a) b$ .

### Definition 16: Combinators

A *combinator* is a lambda expression with no free variables.

Notable combinators are often named after birds.<sup>3</sup> We've already met a few:

The *Idiot*,  $I = \lambda a.a$

The *Mockingbird*,  $M = \lambda f.f f$

The *Cardinal*,  $C = \lambda f g x.( f(g(x)) )$  The *Kestrel*,  $K = \lambda a b.a$

### Problem 17:

If we give the Kestrel two arguments, it does something interesting:

It selects the first and rejects the second.

Convince yourself of this fact by evaluating  $(K \heartsuit \star)$ .

### Problem 18:

Modify the Kestrel so that it selects its **second** argument and rejects the first.

### Problem 19:

We'll call the combinator from ?? the *Kite*,  $KI$ .

Show that we can also obtain the kite by evaluating  $(K I)$ .

## Part 3: Boolean Algebra

The Kestrel selects its first argument, and the Kite selects its second. Maybe we can somehow put this “choosing” behavior to work...

Let  $T = K = \lambda ab.a$

Let  $F = KI = \lambda ab.b$

**Problem 20:**

Write a function NOT so that  $(\text{NOT } T) = F$  and  $(\text{NOT } F) = T$ .

*Hint:* What is  $(T \heartsuit \star)$ ? How about  $(F \heartsuit \star)$ ?

**Problem 21:**

How would “if” statements work in this model of boolean logic?

Say we have a boolean  $B$  and two expressions  $E_T$  and  $E_F$ . Can we write a function that evaluates to  $E_T$  if  $B$  is true, and to  $E_F$  otherwise?

**Problem 22:**

Write functions AND, OR, and XOR that satisfy the following table.

$A$	$B$	$(\text{AND } A B)$	$(\text{OR } A B)$	$(\text{XOR } A B)$
F	F	F	F	F
F	T	F	T	T
T	F	F	T	T
T	T	T	T	F

**Problem 23:**

To complete our boolean algebra, construct the boolean equality check EQ. What inputs should it take? What outputs should it produce?

## Part 4: Numbers

Since the only objects we have in  $\lambda$  calculus are functions, it's natural to think of quantities as *adverbs* (once, twice, thrice,...) rather than *nouns* (one, two, three ...)

We'll start with zero. If our numbers are *once*, *twice*, and *twice*, it may make sense to make zero *don't*. Here's our *don't* function: given a function and an input, don't apply the function to the input.

$$0 = \lambda f a.a$$

If you look closely, you'll find that 0 is equivalent to the false function  $F$ .

### Problem 24:

Write 1, 2, and 3. We will call these *Church numerals*.<sup>4</sup>

*Note:* This problem read aloud is "Define *once*, *twice*, and *thrice*."

### Problem 25:

What is  $(4 I) \star$ ?

### Problem 26:

What is  $(3 NOT T)$ ?

How about  $(8 NOT F)$ ?

---

<sup>4</sup>after Alonzo Church, the inventor of lambda calculus and these numerals. He was Alan Turing's thesis advisor.

**Problem 27:**

Peano's axioms state that we only need a zero element and a "successor" operation to build the natural numbers. We've already defined zero. Now, create a successor operation so that  $1 := S(0)$ ,  $2 := S(1)$ , and so on.

*Hint:* A good signature for this function is  $\lambda n f a$ , or more clearly  $\lambda n. \lambda f a$ . Do you see why?

**Problem 28:**

Verify that  $S(0) = 1$  and  $S(1) = 2$ .

Assume that only Church numerals will be passed to the functions in the following problems. We make no promises about their output if they're given anything else.

**Problem 29:**

Define a function ADD that adds two Church numerals.

**Problem 30:**

Design a function MULT that multiplies two numbers.

*Hint:* The easy solution uses ADD, the elegant one doesn't. Find both!

**Problem 31:**

Define the functions  $Z$  and  $NZ$ .  $Z$  should reduce to  $T$  if its input was zero, and  $F$  if it wasn't.  $NZ$  does the opposite.  $Z$  and  $NZ$  should look fairly similar.

**Problem 32:**

Design an expression  $\text{PAIR}$  that constructs two-value tuples.

For example, say  $A = \text{PAIR } 1 \ 2$ . Then,

$(A \ T)$  should reduce to 1 and  $(A \ F)$  should reduce to 2.

From now on, I'll write  $(\text{PAIR } A \ B)$  as  $\langle A, B \rangle$ .

Like currying, this is only notation. The underlying logic remains the same.

**Problem 33:**

Write a function  $H$ , which we'll call "shift and add."

It does exactly what it says on the tin:

Given an input pair, it should shift its second argument left, then add one.

$H \langle 0, 1 \rangle$  should reduce to  $\langle 1, 2 \rangle$

$H \langle 1, 2 \rangle$  should reduce to  $\langle 2, 3 \rangle$

$H \langle 10, 4 \rangle$  should reduce to  $\langle 4, 5 \rangle$

**Problem 34:**

Design a function  $D$  that un-does  $S$ . That means

$D(1) = 0$ ,  $D(2) = 1$ , etc.  $D(0)$  should be zero.

*Hint:*  $H$  will help you make an elegant solution.

## Part 5: Recursion

Say we want a function that computes the factorial of a positive integer. Here's one way we could define it:

$$x! = \begin{cases} x \times (x - 1)! & x \neq 0 \\ 1 & x = 0 \end{cases}$$

We cannot re-create this in lambda calculus, since we aren't given a way to recursively call functions.

One could think that  $A = \lambda a.A$   $a$  is a recursive function. In fact, it is not.

Remember that such "definitions" aren't formal structures in lambda calculus.

They're just shorthand that simplifies notation.

### Problem 35:

Write an expression that resolves to itself.

*Hint:* Your answer should be quite short.

This expression is often called  $\Omega$ , after the last letter of the Greek alphabet.

$\Omega$  useless on its own, but it gives us a starting point for recursion.

**Definition 36:**

This is the *Y-combinator*. You may notice that it's just  $\Omega$  put to work.

$$Y = \lambda f.(\lambda x.f(x x))(\lambda x.f(x x))$$

**Problem 37:**

What does this thing do?

Evaluate  $Yf$ .

## Part 6: Challenges

Do ?? first, then finish the rest in any order.

**Problem 38:**

Design a recursive factorial function using  $Y$ .

**Problem 39:**

Design a non-recursive factorial function.

This one is easier than ??, but I don't think it will help you solve it.

**Problem 40:**

Solve ?? without using  $H$ .

In ??, we created the “decrement” function.

**Problem 41:**

Using pairs, make a “list” data structure. Define a GET function, so that GET  $L n$  reduces to the  $n$ th item in the list. GET  $L 0$  should give the first item in the list, and GET  $L 1$ , the *second*.

Lists have a defined length, so you should be able to tell when you're on the last element.

**Problem 42:**

Write a lambda expression that represents the Fibonacci function:

$$f(0) = 1, f(1) = 1, f(n + 2) = f(n + 1) + f(n).$$

**Problem 43:**

Write a lambda expression that evaluates to  $T$  if a number  $n$  is prime, and to  $F$  otherwise.

**Problem 44:**

Write a function MOD so that  $(\text{MOD } a \ b)$  reduces to the remainder of  $a \div b$ .

**Problem 45: Bonus**

Play with *Lamb*, an automatic lambda expression evaluator.

<https://git.betalupi.com/Mark/lamb>