

Lambda Calculus

Programming Languages (Fall 2026)

zeph (Zephyr.S.Lucas.GR@dartmouth.edu)

Oct 13/15, 2026

We have seen a few different types of computation: imperative languages that are likely more familiar to the average programmer, and functional languages like haskell, scheme, or lisp. One can notice that some constructs in languages can be easily re-written in terms of other constructs. For example, it is possible to re-write every `for` loop as a `while` loop, so, in some ways the existence of a `for` loop is unnecessary. As we delve further, it becomes interesting to ask: what is really necessary for computation? It turns out the answer is pretty simple: `lambda` is all you need!

1 Intro

In the early 1900s, there was a revolution as ideas of formalizing math and computation began propagating through the discourse of the time. There were several models proposed as a representation of “an algorithm”, the *Turing Machine*, *Generalized Recursive Functions*, and *Lambda Calculus* being the three most prominent. In 1952, the three were formally unified by Church, Turing, and Kleene in the famous *Church-Turing Thesis*. This statement proved the three models were equivalent and encouraged the idea that any *effectively calculable* notion of a program would be equivalent to this definition as well. Lambda Calculus continues to influence the development of functional languages, type systems, and software verification.

1.1 Curry’s Paradox

One of the first few attempts to formalize lambda calculus resulted in paradoxes forming, the most frequent was Curry’s paradox. If you can make a statement that references itself, then you can prove anything true: Let $X := X \implies Y$ We then end with the following chain of reasoning:

$$\begin{array}{ll} X \implies X & (Reflexivity) \\ X \implies (X \implies Y) & (Substitution) \\ X \implies Y & (Contraction) \\ X & (Substitution) \\ Y & (Implication) \end{array}$$

Thankfully, Church’s implementation of lambda calculus remains free of this paradox, thanks to a very small set of primitives.

1.2 Lambda functions

Within Church’s formulation of lambda calculus, there are two primitives:

1. **Abstraction** defines a function by abstracting a variable (the function’s parameter) away from a context (the function’s return value). This is equivalent to the `lambda` command in lisp / scheme / python (and many others).
2. **Application** applies one function to another function. Although normally this can be thought of as $f(x) = f$ applied to x , or evaluating a function f with a parameter x , in lambda calculus x is always another function.

With this we can construct everything we need. The function:

$$I := \lambda x. x$$

Is the identity function. It takes one parameter and returns that value. We can then apply it to another function (say the identity function):

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

We can also define a function that “stores” a parameter and will return it upon another function application.

$$K := \lambda x. (\lambda y. (x))$$

If we apply this function to anything, we end up with a function that ignores its parameter and returns that thing:

$$(\lambda x. (\lambda y. (x))) X = \lambda y. (X)$$

Doing this to the identity function gives us a new combinator called KI

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

1.3 Reductions

There are three types of reductions, each of which we can use to rewrite a lambda expression.

α **Alpha reductions** $\boxed{\lambda x. (\dots x \dots) = \lambda y. (\dots y \dots)}$ Alpha reductions assert that the variable names. If we define a function and use a parameter, then that’s the same as if we defining the same function but name the parameter differently.

β **Beta reductions** $\boxed{(\lambda x. (\dots x \dots)) y = (\dots y \dots)}$ Beta reductions capture the idea of function application. Applying a function to a variable is the same as the body of the function after re-writing all instances of the function’s parameter with the variable the function is applied to.

η **Eta reductions** $\boxed{\lambda x. (fx) = f}$ Eta reductions assert that a function which applies another function to it’s parameter is the same as that other functions. Beta reductions tell us that $(\lambda x. (fx)) y = fy$, and eta reductions tell us we can still perform that reduction even when there is no y to beta-reduce.

These three reductions perform re-writing rules to allow us to simplify expressions. Alpha and beta reductions exhibit what is called “weak confluence” meaning if a series beta reductions would simplify an expression to a point no more beta reductions apply to it, then that simplified form is unique (up to alpha reductions).

1.4 Currying

Currently we have only talked about functions with one parameter, and this is because in pure lambda calculus, functions only have one parameter! You could imagine a function like $+$ that takes more than one parameter but it turns out representing it as one-parameter functions is sufficient for computation, in fact, some languages do this. If you look at $+$ ’s type signature in Haskell it is:

$$+ : [\text{Num} \rightarrow \text{Num} \rightarrow \text{Num}]$$

What this means is $+$ is a function that takes in a `Num` and returns a function that maps `Num` to `Num`. In python we could represent this as:

```
def add(a: int) -> typing.Callable[[int], int]:  
    return lambda b: a + b
```

What does the function `add` do? It takes an `int` (a) and returns another function, which itself takes an `int` (b) and returns an `int` ($a + b$). So if we apply this function to an integer (say, for example 3) we find¹:

```
add(3) # returns a function: lambda b: 3 + b
```

Then we can apply this function to another number (say 5 or 9) and we get:

```
(add(3))(5) # returns: 8
(add(3))(9) # returns: 12
```

Instead of `add` applying to the integers to both 3 and 5 at the same time and returning 8, it is first applied to 3 returning a function which “adds three” and that function is then applied to 5 giving the result 8.

1.5 De Bruijn notation

In some ways α -reductions hint that this whole idea of parameters and naming is unnecessary. De Bruijn show that it is, the only thing that matters is scopes. We replace every parameter by the number of lambda out the variable is, in other words instead of writing x we write how many lambda back you’d need to go to find the λx which defines x . For example:

$$\begin{aligned}\lambda x. (\lambda y. (x)) &= \lambda \lambda 2 \\ \lambda x. (\lambda y. (1)) &= \lambda \lambda 1 \\ \lambda b. (\lambda x. (\lambda y. (b \ x \ y))) &= \lambda \lambda \lambda 3 2 1 \\ \lambda x. (x \ \lambda y. (x \ x)) &= \lambda 1 \lambda 2 2 \\ \lambda f. (\lambda x. (f \ (f \ (f \ (f \ x)))) &= \lambda \lambda 2 \ (2 \ (2 \ (2 \ 1)))\end{aligned}$$

This notation ends up being really useful when writing evaluation engines, because keeping around lists of bindings become unnecessary, and determining equality of functions becomes easier without the need for performing α -reductions to pattern match.

2 Building up computation

This is great, but can we actually do anything with lambda calculus?

2.1 Boolean Logic

Now that we have an understanding of functions and application, we can start building some functions that are useful for computation. The simplest type is probably booleans, so let’s build a function for both true and false:

$$\begin{aligned}\mathbf{True} &:= \lambda x. (\lambda y. (x)) \\ \mathbf{False} &:= \lambda x. (\lambda y. (y))\end{aligned}$$

This choice is arbitrary, though this choice does give us some nice properties (and it is also the definition Church chose). Why is this choice useful, well it makes branching on an if very useful! Consider the following definition for `Not`:

$$\mathbf{Not} := \lambda b. (b \ \mathbf{False} \ \mathbf{True})$$

¹Technically, what is returned is not a function, but what is called a “closure” (which is a function with a binding list), so instead of returning: the function `lambda b: 3 + b`, what is returned is `lambda b: a + b` with the binding list `[a := 3]`. For most cases this distinction doesn’t matter, but it is important to note the substitution does not actually occur until the lambda is evaluated.

We can then apply `Not` to both `True` and `False` to understand better how the boolean expressions work:

<pre>Not True = (λb. (b False True)) True = True False True = (λx. (λy. (x))) False True = (λy. (False)) True = False</pre>	<pre>Not False = (λb. (b False True)) False = False False True = (λx. (λy. (y))) False True = (λy. (y)) True = True</pre>
---	---

This gives us a nice framework for what it means to use booleans. If we want to differentiate a true and a false expression, we can simply apply the boolean to two values. A `True` will always return the first value, while a `False` will always return the second value. We can use this to build any construct that we want. For example, `And`:

$$\text{And} := \lambda a. (\lambda b. (a \ b \ a))$$

Notice this function's behavior:

- When a is true, this function will return b (as `Anding` a boolean with true will just return that boolean).
- Similarly, when a is false, this function will return a , (as `Anding` a boolean with false will return false).

Going through the casework explicitly, we see that this works:

<pre>And True True = (λa. (λb. (a b a))) True True = (λb. (True b True)) True = True True True = (λx. (λy. (x))) True True = (λy. (True)) True = True</pre>	<pre>And True False = (λa. (λb. (a b a))) True False = (λb. (True b True)) False = True False True = (λx. (λy. (x))) False True = λy. (False) True = False</pre>
<pre>And False True = (λa. (λb. (a b a))) False True = (λb. (False b False)) True = False True False = (λx. (λy. (y))) True False = (λy. (y)) False = False</pre>	<pre>And False False = (λa. (λb. (a b a))) False False = (λb. (False b False)) False = False False False = (λx. (λy. (y))) False False = λy. (y) False = False</pre>

We know that NAND gates can generate any boolean expression by themselves². This means we can build any boolean function we want, although there is a much easier way of generating OR than combining NOT and AND.

2.2 Arithmetic

While we could represent numbers as functions that are sequences of booleans and doing everything with binary, this comes with some inelegancies³ that we can avoid. The solution Church decided upon was

²Although it is known that NAND gates are Turing complete by themselves, this is true *if and only if* you are allowed to loop the wires back into the gates themselves. The above lambda expressions do lack this capability so generating loops and memory states is impossible.

³For example, we then have to bound our `ADD` operation by the number parameters it takes in... This is not all that different from the representation of numbers on a computer (after all, an `int` is actually a `unit32_t` or similar in its underlying representation).

representing the naturals as repeated function application. Consider:

$$\begin{aligned}
0 &= \lambda f. (\lambda z. (z)) & 4 &= \lambda f. (\lambda z. (f (f (f (f z)))))) \\
1 &= \lambda f. (\lambda z. (f z)) & 5 &= \lambda f. (\lambda z. (f (f (f (f (f z)))))) \\
2 &= \lambda f. (\lambda z. (f (f z))) & 6 &= \lambda f. (\lambda z. (f (f (f (f (f (f z)))))) \\
3 &= \lambda f. (\lambda z. (f (f (f z)))) & & \dots
\end{aligned}$$

These are the *Church Numerals*. Each natural number n is represented by a two-parameter function that takes a function f and an initial value z , and returns the value when f is applied iteratively to z n times. This is nice in that our natural numbers come “preloaded” with a for loop: just give the number a loop body and initial state, and then the number itself will perform the loop with the correct number of iterations. If we want to do arithmetic, we should probably define a successor function. This is to say, a way to increment a number. There are a few different functions that will perform this task, though the two most straightforward are as follows.

$$\begin{aligned}
\text{Succ} &:= \lambda n. (\lambda f. (\lambda z. (n f (f z)))) \\
\text{Succ_Alt} &:= \lambda n. (\lambda f. (\lambda z. (f (n f z))))
\end{aligned}$$

These both are functions that receive a church numeral, n , and then return a new church numeral that uses n to apply f to z and then also add in one more application of f (in **Succ**’s case the application is on the inside, whereas **Succ_Alt** puts the extra f application on the outside). For example:

$$\begin{aligned}
\text{Succ } 0 &= (\lambda n. (\lambda f. (\lambda z. (n f (f z)))) 0 & \text{Succ } 2 &= (\lambda n. (\lambda f. (\lambda z. (n f (f z)))) 2 \\
&= \lambda f. (\lambda z. (0 f (f z))) & &= \lambda f. (\lambda z. (2 f (f z))) \\
&= \lambda f. (\lambda z. ((\lambda f'. (\lambda z'. (z'))) f (f z))) & &= \lambda f. (\lambda z. ((\lambda f'. (\lambda z'. (f' (f' z')))) f (f z))) \\
&= \lambda f. (\lambda z. ((\lambda z'. (z')) (f z))) & &= \lambda f. (\lambda z. ((\lambda z'. (f (f z'))) (f z))) \\
&= \lambda f. (\lambda z. (f z)) & &= \lambda f. (\lambda z. (f (f (f z)))) \\
&= 1 & &= 3
\end{aligned}$$

2.2.1 Addition

Since addition can be thought of as repeated application of the successor function, we could derive a definition of **Plus** that looks something like this:

$$\text{Plus} := \lambda n. (\lambda m. (n \text{ Succ } m))$$

This says that **Plus** takes two parameters, n and m . Assuming n is a church numeral, **Plus** will apply the successor function n times to m . For example:

$$\begin{aligned}
\text{Plus } 2 \ 3 &= \lambda n. (\lambda m. (n \text{ Succ } m)) \ 2 \ 3 \\
&= \lambda m. (2 \text{ Succ } m) \ 3 \\
&= 2 \text{ Succ } 3 \\
&= (\lambda f. (\lambda z. (f (f z)))) \text{ Succ } 3 \\
&= (\lambda z. (\text{Succ } (\text{Succ } z))) \ 3 \\
&= \text{Succ } (\text{Succ } 3) \\
&= \text{Succ } 4 \\
&= 5
\end{aligned}$$

Astute observers will note that we can actually simplify the expression some; the “ $\lambda m. (n \text{ Succ } m)$ ” inside here can be η -reduced! Recall that application is left-associative, so really we are returning “ $(n \text{ Succ } m)$ ” (because n is curried, it is not actually a two parameter function, it receives a f and then returns a new function that receives a z). So, after η -reducing, we have:

$$\lambda m. (n \text{ Succ } m) = n \text{ Succ}$$

This means that we could have simply defined `Plus` as:

$$\text{Plus} := \lambda n. (n \text{ Succ})$$

This is maybe a little more confusing to read, but the meaning is the same: addition by n is the same as simply repeating the successor function n times. Where is the second parameter to plus? It is the z parameter of the church numeral n so there is no need to explicitly add it here.

2.2.2 Multiplication

Similarly to addition being repeated successor, multiplication can be thought of as repeated addition. Unlike the successor function, which increments a single church numeral, `Plus` takes two arguments. So, the function that we need to repeat n times to add n and m is “ $\lambda k. (\text{Plus } m \ k)$ ”. This makes the multiplication implementation simply:

$$\text{Mult} := \lambda n. (\lambda m. (n (\lambda k. (\text{Plus } m \ k)) \ 0))$$

This function takes two parameters n and m and applies the “ $\lambda k. (\text{Plus } m \ k)$ ” function⁴ n times, starting with 0. But, like before, we can η -reduce this inner function. This gives us a simplified definition for multiplication:

$$\text{Mult} := \lambda n. (\lambda m. (n (\text{Plus } m) \ 0))$$

Note that the earlier statement “`Plus` takes two arguments” was a lie; `Plus` is a curried function, so it takes one argument and returns a new function that takes another argument. By applying `Plus` to m we turn it into a function that takes a single argument and returns the result of adding m to it. This is one of the advantages of currying and one of the reasons languages like Haskell do this by default. To better understand `Plus` we can try it out on an example:

$$\begin{aligned} \text{Mult } 3 \ 2 &= (\lambda n. (\lambda m. (n (\text{Plus } m) \ 0))) \ 3 \ 2 \\ &= (\lambda m. (3 (\text{Plus } m) \ 0)) \ 2 \\ &= 3 (\text{Plus } 2) \ 0 \\ &= (\lambda f. (\lambda z. (f (f (f \ z))))) (\text{Plus } 2) \ 0 \\ &= (\lambda z. ((\text{Plus } 2) ((\text{Plus } 2) ((\text{Plus } 2) \ z)))) \ 0 \\ &= \text{Plus } 2 (\text{Plus } 2 (\text{Plus } 2 \ 0)) \\ &= \text{Plus } 2 (\text{Plus } 2 \ 2) \\ &= \text{Plus } 2 \ 4 \\ &= 6 \end{aligned}$$

This makes sense as $3 \cdot 2$ is, in fact, 6.

2.2.3 Exponentiation (and beyond)

Exponentiation follows relatively trivially from the method used to construct multiplication. The only question is whether or not you want “`Pow` $n \ m$ ” to be n^m or m^n . We chose the former (i.e. n^m) because this is the most common choice among functional programming languages.

$$\text{Pow} := \lambda n. (\lambda m. (m (\text{Mult } n) \ 1))$$

It should be somewhat believable that most of the other basic arithmetical functions can be built using these definitions, boolean predicates, and maybe some type of lambda structure to store internal calculations. It might currently be a stretch to see how one could build a predecessor⁵. If f^{-1} (some inverse f function) exists, then this is fairly trivial because we could use the same approach as we did for `Succ` adding an additional f^{-1} instead of an additional f . Unfortunately, not every function is reversible, so we will need a different approach. Section 2.3.2 gives one such construction of a predecessor.

⁴This is the single parameter function that adds m to its parameter.

⁵That is to say, given that you can do something n times, is it possible to do it $n - 1$ times?

2.2.4 Alternate Formulations

Note how the definition of **Succ** ended up looking quite different from the definitions of the other common operations. **Succ** took a church numeral and constructed a new one there, creating a new “ $\lambda f.\lambda z$ ” and defining a function body that would always “look” like a new church numeral (with the correct number of f applications). This approach can be taken with the three functions above to arrive at some alternate definitions.

$$\begin{aligned}\text{Plus_Alt} &:= \lambda n. (\lambda m. (\lambda f. (\lambda z. (m \ f \ (n \ f \ z)))))) \\ \text{Mult_Alt} &:= \lambda n. (\lambda m. (\lambda f. (\lambda z. (m \ (\lambda x. (n \ f \ x)) \ z)))) \\ \text{Pow_Alt} &:= \lambda n. (\lambda m. (\lambda f. (\lambda z. ((m \ (\lambda g. (\lambda x. (n \ g \ x))) \ (\lambda y. (f \ y))) \ z))))\end{aligned}$$

It might be worth working through an example or two to see how these functions work. After some η -reductions, these simplify pretty nicely:

$$\begin{aligned}\text{Plus_Alt} &:= \lambda n. (\lambda m. (\lambda f. (\lambda z. (m \ f \ (n \ f \ z)))))) \\ \text{Mult_Alt} &:= \lambda n. (\lambda m. (\lambda f. (m \ (n \ f)))) \\ \text{Pow_Alt} &:= \lambda n. (\lambda m. (m \ n))\end{aligned}$$

Although these will work as alternate arithmetic functions, they cannot be β -reduced to the more natural definitions. This is because while **Add** and **Add_Alt** behave the same on church numerals they function differently on generic functions. For those reading ahead, there is also a similar “structural” way to formulate the predecessor function:

$$\text{Pred_Alt} := \lambda n. (\lambda f. (\lambda z. ((n \ (\lambda g. (\lambda h. (h \ (g \ f)))) \ \lambda x. (z)) \ \lambda y. (y))))$$

The intuition behind this is that the first time we apply the function, we ignore the parameter and return z . Each successive time we apply the function, we wrap another f around the value. Consider: the first application of “ $\lambda g. (\lambda h. (h \ (g \ f)))$ ” to “ $\lambda x. (z)$ ” creates a new function “ $\lambda h. (h \ z)$ ”. This is because “ $\lambda x. (z)$ ” ignores its input and returns z , so one application of f is dropped. Now, when we perform our second application of “ $\lambda g. (\lambda h. (h \ (g \ f)))$ ” (meaning it is being applied to the function “ $\lambda h. (h \ z)$ ” instead), the “ $g \ f$ ” expression results in one application of f onto z , giving us the function “ $\lambda h. (h \ (f \ z))$ ”. Applying “ $\lambda g. (\lambda h. (h \ (g \ f)))$ ” a third time will result in “ $\lambda h. (h \ (f \ (f \ z)))$ ”. At the very end, we are left with a function that is *almost* the church numeral for $n - 1$, but it is wrapped in an extra lambda: “ $\lambda h. (h \ [n - 1 \ \text{body}])$ ”. We can remove this by applying it to the identity, correctly returning $n - 1$. In my opinion, this is not the most straightforward formulation of **Pred**, but going solely by the number of λ s in the expression, it is the smallest that I have found/seen.

2.3 Lists

With some basic algebra and boolean expressions, the next obvious thing to look at is having some data structures to store and operate on data. As languages like scheme and lisp show, you can get a lot done with just a basic linked list.

2.3.1 Cons Cells

The most fundamental building block of lists is a *cons cell*, the basic memory object that stores two objects. We can do this by making a two parameter function that returns a function storing the two parameters:

$$\text{Cons} := \lambda a. (\lambda b. (\lambda f. (f \ a \ b)))$$

The returned value takes an f and applies it to the two stored values. In this way, the various functions we pass in as f can be thought of as *methods* for the *object* returned by **Cons**. For cons cells the two most obvious methods are the two accessor functions: **Car** and **Cdr**:

$$\begin{aligned}\text{Car} &:= \lambda c. (c \ \lambda a. (\lambda b. (a))) \\ \text{Cdr} &:= \lambda c. (c \ \lambda a. (\lambda b. (b)))\end{aligned}$$

We can see these functions working in practice by example by first constructing a cons cell:

$$\begin{aligned}\text{Cons } 8 \ 5 &= (\lambda a. (\lambda b. (\lambda f. (f \ a \ b)))) \ 8 \ 5 \\ &= (\lambda b. (\lambda f. (f \ 8 \ b))) \ 5 \\ &= \lambda f. (f \ 8 \ 5)\end{aligned}$$

and then applying them to the cons cell:

$$\begin{array}{ll}\text{Car } (\text{Cons } 8 \ 5) = \text{Car } \lambda f. (f \ 8 \ 5) & \text{Cdr } (\text{Cons } 8 \ 5) = \text{Cdr } (\lambda f. (f \ 8 \ 5)) \\ = (\lambda c. (c \ \lambda a. (\lambda b. (a)))) \ \lambda f. (f \ 8 \ 5) & = (\lambda c. (c \ \lambda a. (\lambda b. (a)))) \ (\lambda f. (f \ 8 \ 5)) \\ = (\lambda f. (f \ 8 \ 5)) \ \lambda a. (\lambda b. (a)) & = (\lambda f. (f \ 8 \ 5)) \ \lambda a. (\lambda b. (b)) \\ = (\lambda a. (\lambda b. (a))) \ 8 \ 5 & = (\lambda a. (\lambda b. (b))) \ 8 \ 5 \\ = (\lambda b. (8)) \ 5 & = (\lambda b. (b)) \ 5 \\ = 8 & = 5\end{array}$$

Notice that there was nothing special about 8 or 5; whatever we passed into the **Cons** function will get stored. **Car** will always return the first parameter to **Cons** and **Cdr** will always return the second parameter to **Cons**. Similarly, while the **Car** and **Cdr** functions are useful, we can construct any method to operate on a cons block. For example, we could make a **Swap** function that reverses the first and second objects of a **Cons** cell:

$$\text{Swap} := \lambda c. (\lambda f. (f \ (\text{Cdr } c) \ (\text{Car } c)))$$

We can see this applied to our example:

$$\begin{aligned}\text{Swap } (\text{Cons } 8 \ 5) &= (\lambda c. (\lambda f. (f \ (\text{Cdr } c) \ (\text{Car } c)))) \ (\text{Cons } 8 \ 5) \\ &= \lambda f. (f \ (\text{Cdr } (\text{Cons } 8 \ 5)) \ (\text{Car } (\text{Cons } 8 \ 5))) \\ &= \lambda f. (f \ 5 \ 8) \\ &= \text{Cons } 5 \ 8\end{aligned}$$

We have some basic operations for **Cons** cells, but the same idea works for generic objects.

2.3.2 Predecessor

With some basic structures, it is also possible to build a fairly intuitive function for the predecessor function. The main idea is built off of a *shift-and-increment* function:

$$\text{Sai} := \lambda c. (\lambda f. (f \ (\text{Succ } (\text{Car } c)) \ (\text{Car } c)))$$

This does what you expect:

$$\begin{array}{ll}\text{Sai } (\text{Cons } 8 \ 5) = \text{Cons } 8 \ 8 & \text{Sai } (\text{Cons } 2 \ 1) = \text{Cons } 3 \ 2 \\ \text{Sai } (\text{Cons } 0 \ 0) = \text{Cons } 1 \ 0 & \text{Sai } (\text{Cons } 3 \ 2) = \text{Cons } 4 \ 3\end{array}$$

Notice that if we start with **(Cons 0 0)**, the first instance of **Sai** does not change the **Cdr** of the cell. Each application of **Sai** thereafter will increment the **Cdr**:

$$\begin{aligned}\text{Sai } (\text{Cons } 0 \ 0) &= (\text{Cons } 1 \ 0) \\ \text{Sai } (\text{Sai } (\text{Cons } 0 \ 0)) &= (\text{Cons } 2 \ 1) \\ \text{Sai } (\text{Sai } (\text{Sai } (\text{Cons } 0 \ 0))) &= (\text{Cons } 3 \ 2) \\ \text{Sai } (\text{Sai } (\text{Sai } (\text{Sai } (\text{Cons } 0 \ 0)))) &= (\text{Cons } 4 \ 3)\end{aligned}$$

This gives us a definition for the predecessor function:

$$\text{Pred} := \lambda n. (\text{Cdr } (n \ \text{Sai } (\text{Cons } 0 \ 0)))$$

This definition comes with the natural consequence that **Pred 0 = 0**. This is what most systems define for the predecessor function.

2.4 Recursion

There has been one piece we have been missing so far: recursion. Without self reference, how can we loop recursively? We can apply a function to itself. Consider the following implementation of the factorial function:

$$\text{Fact}_2 := \lambda f. (\lambda n. (\text{If } (\text{Zero? } n) \ 1 \ (\text{Mult } n \ (f \ f \ (\text{Pred } n))))))$$

When we pass Fact_2 to itself, it ends up with another instance of Fact_2 on itself:

$$\text{Fact}_2 \text{ Fact}_2 = \lambda n. (\text{If } (\text{Zero? } n) \ 1 \ (\text{Mult } n \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } n))))$$

This has another instance of $\text{Fact}_2 \text{ Fact}_2$ inside itself. This means we have an infinite loop. For example:

$$\begin{aligned} \text{Fact}_2 \text{ Fact}_2 \ 2 &= (\lambda n. (\text{If } (\text{Zero? } n) \ 1 \ (\text{Mult } n \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } n)))) \ 2 \\ &= \text{If } (\text{Zero? } 2) \ 1 \ (\text{Mult } 2 \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } 2))) \\ &= \text{If False } 1 \ (\text{Mult } 2 \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } 2))) \\ &= \text{Mult } 2 \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } 2)) \\ &= \text{Mult } 2 \ (\text{Fact}_2 \text{ Fact}_2 \ 1) \\ &= \text{Mult } 2 \ ((\lambda n. (\text{If } (\text{Zero? } n) \ 1 \ (\text{Mult } n \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } n)))) \ 1) \\ &= \text{Mult } 2 \ (\text{If } (\text{Zero? } 1) \ 1 \ (\text{Mult } 1 \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } 1)))) \\ &= \text{Mult } 2 \ (\text{If False } 1 \ (\text{Mult } 1 \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } 1)))) \\ &= \text{Mult } 2 \ (\text{Mult } 1 \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } 1))) \\ &= \text{Mult } 2 \ (\text{Mult } 1 \ (\text{Fact}_2 \text{ Fact}_2 \ 0)) \\ &= \text{Mult } 2 \ (\text{Mult } 1 \ ((\lambda n. (\text{If } (\text{Zero? } n) \ 1 \ (\text{Mult } n \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } n)))) \ 0)) \\ &= \text{Mult } 2 \ (\text{Mult } 1 \ (\text{If } (\text{Zero? } 0) \ 1 \ (\text{Mult } 0 \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } 0))))) \\ &= \text{Mult } 2 \ (\text{Mult } 1 \ (\text{If True } 1 \ (\text{Mult } 0 \ (\text{Fact}_2 \text{ Fact}_2 \ (\text{Pred } 0))))) \\ &= \text{Mult } 2 \ (\text{Mult } 1 \ 1) \\ &= \text{Mult } 2 \ 1 \\ &= 2 \end{aligned}$$

This works, but it can be generalized further. A *fixed-point combinator* is a function that, when applied to another function, returns another instance of itself applied to its parameter. There are a few examples of this, but the most common is the Y-combinator.

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

We can see why it is a fixed point by applying it to another function:

$$\begin{aligned} Y \ g &= (\lambda f. ((\lambda x. (f \ (x \ x))) \ (\lambda x. (f \ (x \ x)))) \ g \\ &= (\lambda x. (g \ (x \ x))) \ (\lambda x. (g \ (x \ x))) \\ &= g \ ((\lambda x. (g \ (x \ x))) \ (\lambda x. (g \ (x \ x)))) \\ &= g \ (Y \ g) \end{aligned}$$

This lets us do any recursion we want! For example:

$$\begin{aligned} \text{Fact}_b &:= \lambda f. (\lambda n. (\text{If } (\text{Zero? } n) \ 1 \ (\text{Mult } n \ (f \ (\text{Pred } n)))))) \\ \text{Fact} &:= Y \ \text{Fact}_b \end{aligned}$$

And to see it in action, we can try it on 3:

```
Fact 3 = (Y Factb) 3
        = (Y Factb) 3
        = (Factb (Y Factb)) 3
        = (λf. (λn. (If (Zero? n) 1 (Mult n (f (Pred n)))))) (Y Factb) 3
        = (λn. (If (Zero? n) 1 (Mult n (Y Factb (Pred n)))) 3
        = If (Zero? 3) 1 (Mult 3 (Y Factb (Pred 3)))
        = If False 1 (Mult 3 (Y Factb (Pred 3)))
        = Mult 3 (Y Factb (Pred 3))
        = Mult 3 (Y Factb 2)
        = Mult 3 (Factb (Y Factb) 2)
        = Mult 3 ((λf. (λn. (If (Zero? n) 1 (Mult n (f (Pred n)))))) (Y Factb) 2)
        = Mult 3 (λn. (If (Zero? n) 1 (Mult n (Y Factb (Pred n)))) 2)
        = Mult 3 (If (Zero? 2) 1 (Mult 2 (Y Factb (Pred 2))))
        = Mult 3 (If False 1 (Mult 2 (Y Factb (Pred 2))))
        = Mult 3 (Mult 2 (Y Factb (Pred 2)))
        = Mult 3 (Mult 2 (Y Factb 1))
        = Mult 3 (Mult 2 (Factb (Y Factb) 1))
        = Mult 3 (Mult 2 ((λf. (λn. (If (Zero? n) 1 (Mult n (f (Pred n)))))) (Y Factb) 1))
        = Mult 3 (Mult 2 ((λn. (If (Zero? n) 1 (Mult n (Y Factb (Pred n)))) 1))
        = Mult 3 (Mult 2 (If (Zero? 1) 1 (Mult 1 (Y Factb (Pred 1)))))
        = Mult 3 (Mult 2 (If False 1 (Mult 1 (Y Factb (Pred 1)))))
        = Mult 3 (Mult 2 (Mult 1 (Y Factb (Pred 1))))
        = Mult 3 (Mult 2 (Mult 1 (Y Factb 0)))
        = Mult 3 (Mult 2 (Mult 1 (Factb (Y Factb) 0)))
        = Mult 3 (Mult 2 (Mult 1 ((λf. (λn. (If (Zero? n) 1 (Mult n (f (Pred n)))))) (Y Factb) 0)))
        = Mult 3 (Mult 2 (Mult 1 ((λn. (If (Zero? n) 1 (Mult n (Y Factb (Pred n)))) 0)))
        = Mult 3 (Mult 2 (Mult 1 (If (Zero? 0) 1 (Mult 0 (Y Factb (Pred 0)))))
        = Mult 3 (Mult 2 (Mult 1 (If True 1 (Mult 0 (Y Factb (Pred 0)))))
        = Mult 3 (Mult 2 (Mult 1 1))
        = Mult 3 (Mult 2 1)
        = Mult 3 2
        = 6
```

This is correct, so that is exciting! In some ways f is the “self-reference” but only because we can pass a fixed-point combinator into it. For more complicated functions that have multiple cases where recursion can occur, f can be used in each of them. This method works when we get to choose the order we evaluate arguments. If we always evaluate the arguments before application⁶ the Y-combinator could loop infinitely instead of reducing. For systems that always evaluate arguments before application, the Z-combinator (this will η -reduce to the Y-combinator) can be used:

$$Z := \lambda f. ((\lambda x. (f \lambda y. (x x y))) (\lambda x. (f \lambda y. (x x y))))$$

This is not necessary for the lambda calculus assignment, but is a nice tool to have if you do need to use lambda calculus for something.

⁶This is how many languages (like Lisp) operate. Some functional languages (like Haskell) instead lazily evaluate, meaning they never expand a function unless it is needed for an application.