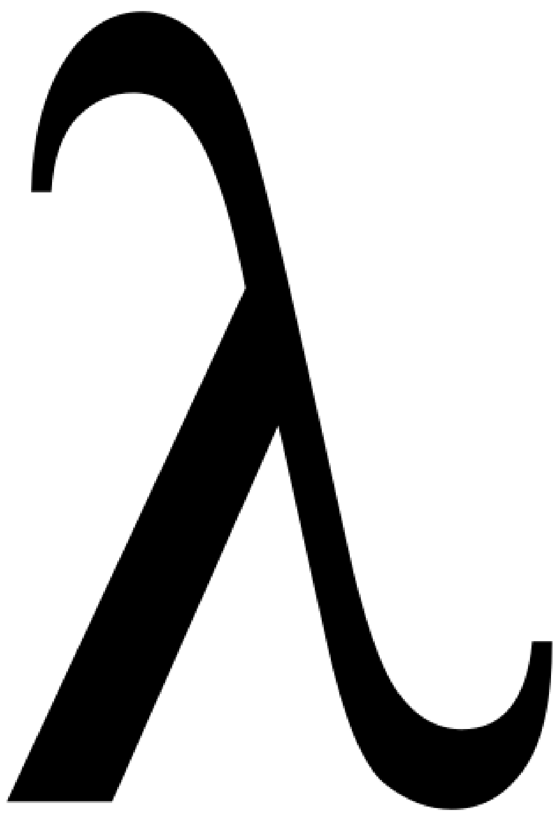
An Introduction to the Lambda Calculus, Church Encodings, and the Y Combinator

Eric Breyer

October 2021 - January 2022



Contents

Qualifications

I am not an expert in any of these concepts in any way. I have just become interested in learning these concepts on my own lately, and wanted to write down my thoughts, what I have learned, and how I have learned it.

1 What is the Lambda Calculus

1.1 Overview

The Lambda Calculus is a model of computing created by Alonzo Church. Essentially, in the Lambda Calculus, everything is a function. With a few simple rules, everything from Booleans and numbers, to recursion can be represented by pure functions.

1.2 The Rules

1.2.1 Lambda Terms

1. Variables
   - character or string representing a parameter or value
   - Syntax: x
2. Abstractions
   - A function definition that binds a variable in the body
   - Syntax: $\lambda x.M$ (M is an expression - the function body)
3. Application
   - Application of a function to an argument
   - Syntax: M I
   - Note: unless otherwise grouped by parenthesis, function application is done left to right. eg. M I K applies I to M then K to the result

1.2.2 Reduction Operations

1. α-conversion (alpha conversion)
   - Simply changing the names of all variables in an abstraction
   - Example: λx.x is α-equivalent to λy.y
2. β-reduction (beta reduction)
   - applying the function, replace all bound variables with the argument of the application
   - Example: $(\lambda x.xy)M$ is equivalent to My (the value of M is passed to x and the expression is reduced)
3. η-reduction (eta reduction)
   - a way to drop an abstraction over a function to simplify it. $(\lambda x.Mx)$ can be η-reduced to M, assuming M does not bind this x
   - Example: $(\lambda x.Mx)I$ is equivalent to MI (prove this to yourself using both β-reduction and η-reduction)

1.2.3 Important Notes

- In pure Lambda Calculus, lambda abstractions (functions) do not have names, however we will give some names here for the sake of convenience in referencing them later
- I will use capital letters and names (I, M, ADD, etc.) to reference previously defined lambda expressions and lowercase letters (x, y, n, etc.) to represent variables

2 Combinators

2.1 Overveiw

What is a combinator? A combinator is simply a lambda abstraction with no free variables, meaning all variables in the body are bound by the abstraction. For example $\lambda xyz.xzy$ is a combinator, but $\lambda xy.xyz$ is not (z is a free variable)

Why? Combinators are used in a branch of mathematics and computing called combinatorial logic and form the basis of computation in Lambda Calculus. For our purposes they will get us acquainted with Lambda calculus and many will have interesting applications down the road.

2.2 The I Combinator

"Idiot Bird"

$$\lambda x.x$$

The identity function simply returns its input.

$I\ I$	
$I\ I$	expand the representation of I
$(\lambda x.x)\ I$	β-reduction (Apply I as x)
I	$I\ I \rightarrow I$

Table 2.2.1: The I Combinator

2.3 The M Combinator

"Mockingbird"

$$\lambda x.xx$$

The mockingbird accomplishes self application of a function to itself

$M\ I$	
$M\ I$	expand the representation of M
$(\lambda x.x\ x)\ I$	β-reduction (Apply I as x)
$I\ I$	β-reduction
I	$M\ I \rightarrow I$

Table 2.3.1: The M Combinator

$M\ M$	
$M\ M$	expand the representation of M
$(\lambda x.x\ x)\ M$	β-reduction (Apply M as x)
$M\ M$	β-reduction
$M\ M$	β-reduction
...	
$M\ M$	this is interesting: a small
...	glimpse into recursion

Table 2.3.2: The M Combinator - Infinite expansion

2.4 The K Combinator

"Kestral"

$$\lambda x.\lambda y.x$$

The Kestrel takes two arguments and returns the first

$K\ M\ I$	
$K\ M\ I$	expand the representation of K
$(\lambda x.\lambda y.x)\ M\ I$	β-reduction (Apply M as x)
$(\lambda y.M)\ I$	β-reduction (Apply I as y)
M	$K\ M\ I \rightarrow\ M$

Table 2.4.1: The K Combinator

Note on currying: All lambda abstractions by definition only take one argument, but we can "curry" them to accept more. This function takes an x and returns a function that takes a y and "captures" what x was, thus allowing us to effectively take 2 arguments. This is called currying. It is important to realize that all functions are curried, but we can simplify our syntax for functions by letting them accept two arguments "at once."

Thus the K combinator can be written in simplified syntax:

$$\lambda xy.x$$

This makes our β-reductions faster and simpler.

eg.

$K\ M\ I$	
$K\ M\ I$	expand the representation of K
$(\lambda xy.x)\ M\ I$	β-reduction (Apply M as x and I as y)
M	$K\ M\ I \rightarrow\ M$

Table 2.4.2: The K Combinator - Condensed notation

However, remember that passing two arguments at once is essentially a syntactic shortcut, all lambda abstractions with multiple inputs are actually curried.

2.5 The KI Combinator

"Kite"

$$\lambda xy.y$$

The Kestrel takes two arguments and returns the second

$KI\ M\ I$	
$KI\ M\ I$	expand the representation of KI
$(\lambda xy.y)\ M\ I$	β-reduction (Apply M as x and I as y)
I	$K\ M\ I \rightarrow\ I$

Table 2.5.1: The KI Combinator

Interesting Note: There is significance to this being called the KI combinator. We can construct the Kite with the Kestrel and the Identity function.

eg. (a and b are arbitrary)

$K\ I\ a\ b$	
$K\ I\ a\ b$	expand the representation of K
$(\lambda xy.x)\ I\ a\ b$	β-reduction (Apply I as x and a as y)
$I\ b$	β-reduction
b	$K\ I\ a\ b \rightarrow\ b$

Table 2.5.2: The KI Combinator as $(K\ I)$

Thus (K I) and KI are equivalent functions

2.6 The C Combinator

"Cardinal"

$$\lambda fab.fba$$

The Cardinal takes a function and two arguments, than applies the two arguments to the function in reverse order

$C\ K\ I\ M$	
$C\ K\ I\ M$	expand the representation of C
$((\lambda fab.fba)\ K)\ I\ M$	β-reduction (Apply K as f)
$(\lambda ab.Kba)\ I\ M$	β-reduction (Apply I as a and M as b)
$K\ M\ I$	β-reduction (apply M and I to K)
M	$C\ K\ I\ M \rightarrow M$

Table 2.6.1: The C Combinator

We have just given (C K) two inputs and picked the second

$\therefore (C\ K)$ is also equivalent to KI

2.7 The V Combinator

"Virio"

$$\lambda xyf.fxy$$

The Virio takes a function and two arguments, then applies the two arguments to the function.

$V\ I\ M\ K$

$V\ I\ M\ K$	expand the representation of V
$(\lambda xyf.fxy)\ I\ M\ K$	β-reduction (Apply I as x and M as y)
$(\lambda f.f\ I\ M)\ K$	β-reduction (Apply K as f)
$K\ I\ M$	β-reduction (apply I and M to K)
I	$V\ I\ M\ K \rightarrow I$

Table 2.7.1: The V Combinator

Why is this useful?

Creating Pairs The fact that we pass in arguments first (before the function) will allow us to "pair" and x and a y together and "save" them for later use. Remember, since functions are curried, we need not pass in all elements at the same time. The function we get by only passing two arguments to V is a pair.

eg.

$V\ I\ M$

$V\ I\ M$	expand the representation of V
$(\lambda xyf.fxy)\ I\ M$	β-reduction (Apply I as x and M as y)
$(\lambda f.f\ I\ M)$	we have now formed a pair of I and M

Table 2.7.2: Making a pair example

Using Pairs Now that we have a pair, we can pass a lambda expression (namely K or KI) to it as its final argument to pull elements out.

eg. Getting the first element of the I/M pair we created

$(I/M$ pair$)\ K$

$(I/M$ pair$)\ K$	expand the representation of the I/M pair
$(\lambda f.f\ I\ M)\ K$	β-reduction (Apply K as f)
$K\ I\ M$	β-reduction (Apply I and M to K)
I	We have retrieved the first element of the pair

Table 2.7.3: Using a pair example

We will see the full power of pairs later, and how they allow us to create data structures like lists.

3 Church Encoding - Booleans

3.1 Overveiw

Now we can start to do some computation with lambda expression. Booleans (True, False) and Boolean logic (And, Or, Not) allow us to perform simple aritmetic and are the basis for control flow operations (If Then Else).

"Church" Encoding? Alsonzo Church was the creator of lambda calculus. He discovered how to represent many things such as Booleans and numbers as lambda expressions. We call his representations Church Encodings.

3.2 T - True

$$\lambda ab.a$$

The encoding of true is a function that takes two inputs and returns the first. (look familiar?)

3.3 F - False

$$\lambda ab.b$$

The encoding of true is a function that takes two inputs and returns the second. (look familiar?)

3.4 NOT - Boolean Negation

$$\lambda p.pFT$$

NOT takes a Boolean and returns the Boolean not of that input (True evaluates to False and visa versa). p is a Boolean which is passed the arguments of F (False) and T (True). If p is True, it will take the first of the two arguments F and T and evaluate to F. Likewise if p is False, it will take the second of the two arguments F and T and evaluate to T.

$NOT\ F$

$NOT\ F$	expand the representation of NOT
$(\lambda p.pFT)\ F$	β-reduction (Apply F as p)
$F\ F\ T$	expand the representation of F
$(\lambda ab.b)\ F\ T$	β-reduction (Apply F as a and T as b)
T	$NOT\ F \rightarrow T$

Table 3.4.1: NOT of F

3.5 AND - Boolean And

$$\lambda pq.pqp$$

AND takes a Boolean p and q. It then passes q and p as arguments to p itself. Since p is a Boolean, it will either "pick" q or p. If p is false we expect $(AND\ p\ q)$ to be False. In this case p, which is False, will "pick" the value of p, which is False. If p is True, we expect $(AND\ p\ q)$ to be True if q is True and False if q is False. This is just the same as evaluating to q if p is True, so when p is True, it will "pick" the value of q.

$AND\ F\ T$

$AND\ F\ T$	expand the representation of AND
$(\lambda pq.pqp)\ F\ T$	β-reduction (Apply F as p and T as q)
$F\ T\ F$	β-reduction (Apply T and F to F)
F	$AND\ F\ T \rightarrow F$

Table 3.5.1: AND of F and T

$AND\ T\ F$

$AND\ T\ F$	expand the representation of AND
$(\lambda pq.pqp)\ T\ F$	β-reduction (Apply T as p and F as q)
$T\ F\ T$	β-reduction (Apply F and T to T)
F	$AND\ T\ F \rightarrow F$

Table 3.5.2: AND of T and F

$AND\ T\ T$

$AND\ T\ T$	expand the representation of AND
$(\lambda pq.pqp)\ T\ T$	β-reduction (Apply T as p and T as q)
$T\ T\ T$	β-reduction (Apply T and T to T)
T	$AND\ T\ T \rightarrow T$

Table 3.5.3: AND of T and T

3.6 OR - Boolean Or

$$\lambda pq.ppq$$

In a similar vein to AND. If p is true, $OR\ p\ q$ is True, so we return p, which is True. If p is False, $OR\ p\ q$ is True if q is True and False if q is False. This is just the same as evaluating to q if p is False.

$OR\ T\ F$	
$OR\ T\ F$	expand the representation of OR
$(\lambda pq.ppq)\ T\ F$	β-reduction (Apply T as p and F as q)
$T\ T\ F$	β-reduction (Apply T and F to T)
T	$OR\ T\ F \rightarrow T$

Table 3.6.1: OR of T and F

$OR\ F\ T$	
$OR\ F\ T$	expand the representation of OR
$(\lambda pq.ppq)\ F\ T$	β-reduction (Apply F as p and T as q)
$F\ F\ T$	β-reduction (Apply F and T to F)
T	$OR\ F\ T \rightarrow T$

Table 3.6.2: OR of F and T

$OR\ F\ F$	
$OR\ F\ F$	expand the representation of OR
$(\lambda pq.ppq)\ F\ F$	β-reduction (Apply F as p and F as q)
$F\ F\ F$	β-reduction (Apply F and F to F)
F	$OR\ F\ F \rightarrow F$

Table 3.6.3: OR of F and F

3.7 BEQ - Boolean equality

$$\lambda pq.pq(NOT\ q)$$

We can again follow similar logic to AND and OR when checking for equality. See if you can work through this one yourself.

$BEQ\ T\ F$	
$BEQ\ T\ F$	expand the representation of BEQ
$(\lambda pq.pq(NOT\ q))\ T\ F$	β-reduction (Apply T as p and F as q)
$T\ F\ (NOT\ F)$	β-reduction (Apply F to NOT)
$T\ F\ T$	β-reduction (Apply F and T to T)
F	$BEQ\ T\ F \rightarrow F$

Table 3.7.1: BEQ of T and F

$BEQ\ F\ F$	
$BEQ\ F\ F$	expand the representation of BEQ
$(\lambda pq.pq(NOT\ q))\ F\ F$	β-reduction (Apply F as p and F as q)
$F\ F\ (NOT\ F)$	β-reduction (Apply F to NOT)
$F\ F\ T$	β-reduction (Apply F and T to F)
T	$BEQ\ F\ F \rightarrow T$

Table 3.7.2: BEQ of F and F

4 Church Encoding - Natural Numbers

4.1 Overview

This is where things really start to get magical. How can we represent, add, and multiply numbers with only functions?

4.2 ZERO

$$\lambda fx.x$$

4.3 ONE

$$\lambda fx.fx$$

4.4 TWO

$$\lambda fx.f(fx)$$

4.5 THREE

$$\lambda fx.f(f(fx))$$

4.6 FOUR

$$\lambda fx.f(f(f(fx)))$$

......

4.7 FIFTEEN

$$\lambda fx.f(f(f(f(f(f(f(f(f(f(f(f(f(f(fx))))))))))))))$$

See a pattern? Church numerals are just repeated application of a function to an argument. It can be helpful to think of these functions not in terms of "one, two, three ...", but in terms of "once, twice, thrice..."

5 Arithmetic with Church Numerals

5.1 Overview

The first function we want to define is a successor function, one that will add one to our church numeral. All other arithmetic operations can be reduced down to repeated applications of +1.

5.2 SUCC - Successor (n + 1)

$$\lambda nfx.f(nfx)$$

n in this case is the church numeral we want to increment. $SUCC$ will pass f and x straight to n, but it will also apply another application of f to x after n has applied all of its applications. This one extra application makes $SUCC\ n$ equivalent to $n+1$

$SUCC\ TWO$	
$SUCC\ TWO$	expand the representations of $SUCC$ and TWO
$(\lambda nfx.f(nfx))\ (\lambda fx.f(fx))$	α-convert
$(\lambda nfx.f(nfx))\ (\lambda ab.a(ab))$	β-reduction (Apply $(\lambda ab.a(ab))$ as n)
$\lambda fx.f(\ (\lambda ab.a(ab))fx\ )$	β-reduction (Apply f as a and x as b)
$\lambda fx.f(f(fx))$	represent as a previously defined expression
$THREE$	$SUCC\ TWO \rightarrow \lambda fx.fffx = THREE$

Table 5.2.1: Successor of TWO

5.3 ADD - Addition

$$\lambda mn.m\ SUCC\ n$$

Getting $SUCC$ was the hard part, now we just have to find a way to add one to a number n, m times, Lucky we already have this function: m itself! Add applies the arguments $SUCC$ and n to m. m will apply $SUCC$ to n m number of times. In other words n will be incremented m times, which is the same as adding the two numbers.

$ADD\ TWO\ THREE$	
$ADD\ TWO\ THREE$	expand the representation of ADD
$(\lambda mn.m\ SUCC\ n)\ TWO\ THREE$	β-reduction (apply functions as m & n)
$TWO\ SUCC\ THREE$	expand the representation of TWO
$(\lambda fx.f(fx))\ SUCC\ THREE$	β-reduction (apply functions as f & x)
$SUCC\ (SUCC\ THREE)$	β-reduction (apply $THREE$ to $SUCC$)
$SUCC\ FOUR$	β-reduction (apply $FOUR$ to $SUCC$)
$FIVE$	$ADD\ TWO\ THREE \rightarrow FIVE$

Table 5.3.1: Addition of TWO and $THREE$

5.4 MULT - Multiplication

$$\lambda mn.m\ (ADD\ n)\ ZERO$$

$MULT$ works in a similar way to add. We just have to add a number n m times. Since functions are curried, if we give ADD a single argument n, we have effectively created an $ADDn$ expression. For example, if we make a function $ADD5$ that is equal to (ADD 5), any argument we pass to $ADD5$ will be like the second argument to ADD in ADD 5. Therefore any argument passed to $ADD5$ will become itself + 5.

In $MULT$, we create an $ADDn$ function, and apply it m times. If we add n m times we get $m * n$.We need to kick off this repeated adding with something to add too, that is where the $ZERO$ comes in.

$MULT\ TWO\ THREE$	
$MULT\ TWO\ THREE$	expand the representation of $MULT$
$(\lambda mn.m\ (ADD\ n)\ ZERO)\ TWO\ THREE$	β-reduction (apply functions as m and n)
$TWO\ (ADD\ THREE)\ ZERO$	expand the representation of TWO
$(\lambda fx.f(fx))\ (ADD\ THREE)\ ZERO$	β-reduction (apply as f and x)
$(ADD\ THREE)\ ((ADD\ THREE)\ ZERO)$	β-reduction (apply functions to ADD)
$(ADD\ THREE)\ THREE$	β-reduction (apply functions to ADD)
SIX	$MULT\ TWO\ THREE \rightarrow SIX$

Table 5.4.1: Multiplication of TWO and $THREE$

5.5 POW - Exponentiation

$$\lambda mn.nm$$

$POW\ m\ n$ represents m^n. It has a very nice representation in Lambda Calculus. Let's convince ourselves why it works.

This one is a lot but see if you can work through it and really convince yourself that POW works.

$POW\ TWO\ THREE$

$POW\ TWO\ THREE$	expand the representation of POW
$(\lambda mn.nm)\ TWO\ THREE$	β-reduction (apply functions as m and n)
$THREE\ TWO$	expand the representation of $THREE$
$(\lambda fx.f(f(fx)))\ TWO$	α-convert
$(\lambda fa.f(f(fa)))\ TWO$	β-reduction (apply TWO as f)
$\lambda a.TWO(TWO(TWO(a)))$	expand the representation of TWO
$\lambda a.TWO(TWO((\lambda fx.f(fx))(a)))$	β-reduction (apply a as f)
$\lambda a.TWO(TWO(\lambda x.a(ax)))$	expand the representation of TWO
$\lambda a.TWO((\lambda cd.c(cd))(\lambda x.a(ax)))$	β-reduction (apply $(\lambda x.a(ax))$ as c)
$\lambda a.TWO(\lambda d.(\lambda x.a(ax))((\lambda x.a(ax))d))$	β-reduction (apply d as x) also α-convert
$\lambda a.TWO(\lambda d.(\lambda b.a(ab))(a(ad)))$	β-reduction (apply $(a(ad))$ as b)
$\lambda a.TWO(\lambda d.a(a(a(ad))))$	β-reduction (apply $(a(ad))$ as b)
$\lambda a.TWO(\lambda d.a(a(a(ad))))$	expand the representation of TWO
$\lambda a.(\lambda gh.g(gh))(\lambda d.a(a(a(ad))))$	β-reduction (apply $(\lambda d.a(a(a(ad))))$ as g)
$\lambda a.\lambda h.(\lambda d.a(a(a(ad))))((\lambda d.a(a(a(ad))))h)$	β-reduction (apply h as d) also α-convert
$\lambda a.\lambda h.(\lambda j.a(a(a(aj))))(a(a(a(ah))))$	β-reduction (apply $(a(a(a(ah))))$ as j)
$\lambda a.\lambda h.a(a(a(a(a(a(ah))))))$	α-convert and simplify representation
$\lambda fx.f(f(f(f(f(f(fx))))))$	represent as previously defined expression
$EIGHT$	$POW\ TWO\ THREE \rightarrow EIGHT$

Table 5.5.1: Exponentiation of TWO to the $THREE$ power

6 Encoding Pairs and Lists

6.1 Overview

Now it is time to implement some data structures, all with pure functional logic.

6.2 PAIR - The simplest data structure

$$\lambda xyf.fxy$$

We have seen Pair before as the V combinator ("Virio"). To refresh: The Virio takes a function and two arguments, then applies the two arguments to the function. However the fact that we pass in arguments first (before the function) will allow us to "pair" and x and a y together and save them for later use.

Building a pair Lets build a pair of ONE and TWO:

$PAIR\ ONE\ TWO$	
$PAIR\ ONE\ TWO$	expand the representation of $PAIR$
$(\lambda xyf.fxy)\ ONE\ TWO$	β-reduction (Apply ONE as x and TWO as y)
$\lambda f.f\ ONE\ TWO$	$PAIR\ ONE\ TWO \rightarrow \lambda f.f\ ONE\ TWO$
	this lambda expression is a container for ONE and TWO

Table 6.2.1: Pairing ONE and TWO

Accessing elements of a pair We access the elements of this pair by passing the pair the K or KI combinators. We will see this with $FIRST$ and $SECOND$.

6.3 FIRST - Access the first element of a pair

$$\lambda p.pK$$

Evaluates to the first element of a pair

$FIRST\ (PAIR\ ONE\ TWO)$	
$FIRST\ (PAIR\ ONE\ TWO)$	expand the representation of $FIRST$
$\lambda p.pK\ (PAIR\ ONE\ TWO)$	expand the representation of $(PAIR\ ONE\ TWO)$
$\lambda p.pK\ (\lambda f.f\ ONE\ TWO)$	β-reduction (Apply $(\lambda f.f\ ONE\ TWO)$ as p)
$(\lambda f.f\ ONE\ TWO)K$	β-reduction (Apply K as f)
$K\ ONE\ TWO$	β-reduction (Apply ONE and TWO to K)
ONE	$FIRST\ (PAIR\ ONE\ TWO) \rightarrow ONE$

Table 6.3.1: Getting first element of a pair

6.4 SECOND - Access the second element of a pair

$$\lambda p.p\ KI$$

Evaluates to the second element of a pair

$SECOND\ (PAIR\ ONE\ TWO)$	
$FIRST\ (PAIR\ ONE\ TWO)$	expand the representation of $SECOND$
$\lambda p.p\ KI\ (PAIR\ ONE\ TWO)$	expand the representation of $(PAIR\ ONE\ TWO)$
$\lambda p.p\ KI\ (\lambda f.f\ ONE\ TWO)$	β-reduction (Apply $(\lambda f.f\ ONE\ TWO)$ as p)
$(\lambda f.f\ ONE\ TWO)\ KI$	β-reduction (Apply KI as f)
$KI\ ONE\ TWO$	β-reduction (Apply ONE and TWO to KI)
TWO	$SECOND\ (PAIR\ ONE\ TWO) \rightarrow TWO$

Table 6.4.1: Getting second element of a pair

6.5 NIL - an empty pair

$$\lambda f.T$$

We need a way to define an empty pair or "nil" value when it comes time to build lists. This choice is fairly arbitrary and there are other definitions for a NIL value, but all that is important is that we can test for its existence.

6.6 NIL? - is a pair the NIL value

$$\lambda p.p(\lambda xy.F)$$

Nil? checks if a pair is nil. If the pair is a real pair, it will apply $(\lambda xy.F)$ and evaluate to False, however a NIL pair will just evaluate to True (the definition of NIL).

$NIL?\ (PAIR\ ONE\ TWO)$

$NIL?\ (PAIR\ ONE\ TWO)$	expand the representation of $NIL?$
$(\lambda p.p(\lambda xy.F))\ (PAIR\ ONE\ TWO)$	β-reduction (apply $(PAIR\ ONE\ TWO)$ as p)
$(PAIR\ ONE\ TWO)\ (\lambda xy.F)$	expand representation of $(PAIR\ ONE\ TWO)$
$(\lambda f.f\ ONE\ TWO)\ (\lambda xy.F)$	β-reduction (Apply $(\lambda xy.F)$ as f)
$(\lambda xy.F)\ ONE\ TWO$	β-reduction (Apply ONE, TWO to $(\lambda xy.F)$)
F	$NIL?\ (PAIR\ ONE\ TWO) \to F$

Table 6.6.1: Checking if a (non-nil) pair is NIL

$NIL?\ NIL$

$NIL?\ NIL$	expand the representation of $NIL?$
$(\lambda p.p(\lambda xy.F))\ NIL$	β-reduction (apply NIL as p)
$NIL\ (\lambda xy.F)$	expand representation of NIL
$(\lambda f.T)\ (\lambda xy.F)$	β-reduction (Apply $(\lambda xy.F)$ as f)
T	$NIL?\ NIL \to T$

Table 6.6.2: Checking if a (nil) pair is NIL

6.7 Lists - pairs of pairs of pairs of...

Pairs We can build a pair by calling PAIR on 2 arguments. PAIR 2 3 gives us a pair of (2 3). To retrieve elements from this pair we can use FIRST and SECOND.

Lists To build a list, we can create a pair where the first element is a value, and the second element of a pair. This pair's first element is a value, and its second is a pair, and so on. We can end this "chain" of pairs with a value of NIL as the second element in a pair. So a list with this encoding could look like (1 (2 (3 (4 NIL)))) and could be built by $PAIR\ ONE$ ($PAIR\ TWO$ ($PAIR\ THREE$ ($PAIR\ FOUR\ NIL$))). Calling first on this list will give us the first value, and calling second will give us the rest of the list. This model lends itself to recursive funtions.

Using a List This is very rough psudocode and is in no way pure lambda calculus (but we will make use of some of our previously defined lambda expressions), but it should be sufficient to give us a rough idea for now of how we might use these lists. To add all the elements in a list like the one we have just generated, we might try:

Define a function *addList* that takes a list: *list* as argument:
| if NIL? *list*:
| | return $ZERO$
| else:
| | return ADD ($FIRST$ *list*) (*addList*($SECOND$ *list*))

The last line adds the first element of list, to the sum of the rest of the list (the rest of the list is ($SECOND$ *list*))

7 Revisiting Arithmetic with Church Numerals - Subtraction

7.1 Overview

We never defined subtraction before. Now that we know pairs, we finally have the tools to implement it.

7.2 The Φ Combinator

$$\lambda p.PAIR\ (SECOND\ p)\ (SUCC(SECOND\ p))$$

The Φ Combinator simply takes a pair, "copies" the second element to the first, and increments the second element. This is a weird combinator but will allow us to implement subtraction by "remembering" a previous value as we count.

$\Phi\ (PAIR\ ONE\ TWO)$	
(1) $\Phi\ (PAIR\ ONE\ TWO)$	expand the representation of Φ
(2) $(\lambda p.PAIR\ (SECOND\ p)\ (SUCC(SECOND\ p)))$ $(PAIR\ ONE\ TWO)$	β-reduction (apply pair as p)
(3) $PAIR$ $(SECOND\ (PAIR\ ONE\ TWO))$ $(SUCC\ (SECOND\ (PAIR\ ONE\ TWO)))$	β-reduction (apply pairs to $SECOND$)
(4) $PAIR\ \ TWO\ (SUCC\ TWO)$	β-reduction (apply TWO to $SUCC$)
(5) $PAIR\ \ TWO\ THREE$	$\Phi\ (PAIR\ ONE\ TWO) \rightarrow$ $PAIR\ \ TWO\ THREE$

Table 7.2.1: Shift and increment a pair of ONE and TWO

7.3 PRED - Predecessor

$$\lambda n.FIRST\ (n\ \Phi\ (PAIR\ ZERO\ ZERO))$$

The predecessor (n -1) function works by shifting and incrementing the pair (0 0) n times. As we increment (0 0), the second number will count up, with the first always being one less. If we shift and increment n times, the pair will end up being (n-1 n). We can pick the first element of this pair to get n-1.

$PRED\ FOUR$	
$PRED\ FOUR$	expand representation of $PRED$
$(\lambda n.FIRST(n\Phi(PAIR\ ZERO\ ZERO)))\ FOUR$	β-reduction (apply $FOUR$ as n)
$FIRST\ (FOUR\ \Phi\ (PAIR\ ZERO\ ZERO))$	expand representation of $FOUR$
$FIRST((\lambda fx.f(f(f(fx))))\Phi(PAIR\ ZERO\ ZERO))$	β-reduction (apply as f and x)
$FIRST\ (\Phi(\Phi(\Phi(\Phi(PAIR\ ZERO\ ZERO)))))$	β-reduction (apply pair to Φ)
$FIRST\ (\Phi(\Phi(\Phi(PAIR\ ZERO\ ONE))))$	β-reduction (apply pair to Φ)
$FIRST\ (\Phi(\Phi(PAIR\ ONE\ TWO)))$	β-reduction (apply pair to Φ)
$FIRST\ (\Phi(PAIR\ TWO\ THREE))$	β-reduction (apply pair to Φ)
$FIRST\ (PAIR\ THREE\ FOUR)$	β-reduction (apply pair to $FIRST$)
$THREE$	$PRED\ FOUR \rightarrow THREE$

Table 7.3.1: Shift and increment a pair of ONE and TWO

7.4 SUB - Subtraction

$$\lambda mn.n\ PRED\ m$$

Subtracts n from m. Same concept as ADD: apply $PRED$ to m, n number of times. Note, if n is larger than m, we will end up with $ZERO$. In our current representation we do not have negative church numerals. However we will be able to use this ”min-zero” property to our advantage to create some predicates like greater-than-or-equal-to.

$SUB\ FOUR\ TWO$	
$SUB\ FOUR\ TWO$	expand the representation of SUB
$(\lambda mn.n\ PRED\ m)\ FOUR\ TWO$	β-reduction (apply functions as m & n)
$TWO\ PRED\ FOUR$	expand the representation of TWO
$(\lambda fx.f(fx))\ PRED\ FOUR$	β-reduction (apply functions as f & x)
$PRED\ (PRED\ FOUR)$	β-reduction (apply $FOUR$ to $PRED$)
$PRED\ THREE$	β-reduction (apply $THREE$ to $PRED$)
TWO	$SUB\ FOUR\ TWO \rightarrow TWO$

Table 7.4.1: Subtractions of $FOUR$ and TWO

8 Revisiting Booleans - Control Flow

8.1 Overview

Booleans are generally used with if else statements and different tests to create branches in our code. Lets see how this works in the Lambda Calculus.

Predicates A predicate is a functions that asks question about an input and returns a Boolean answer. For example, *isZero* would be a predicate that returns true if its input is zero and false otherwise

8.2 NIL?

We have already defined a predicate in NIL? (for definition see subsection 6.6). NIL? asks if the given input list is NIL, returns T is it is and F if it isn't.

8.3 ZERO? - is a numeral ZERO?

$$\lambda n.n(\lambda x.F)T$$

We pass a lambda expression and a value to the given church numeral. If the numeral is zero, it will disregard the function and take the value as is: it will evaluate to T. However, any other church numeral will evaluate the function at least once, which will always evaluate to F.

$ZERO?\ FOUR$

$ZERO?\ FOUR$	expand the representation of $ZERO$?
$(\lambda n.n(\lambda x.F)T)\ FOUR$	β-reduction (apply $FOUR$ as n)
$FOUR\ (\lambda x.F)\ T$	expand the representation of $FOUR$
$(\lambda fx.f(f(f(fx))))\ (\lambda x.F)\ T$	β-reduction (apply $(\lambda x.F)$ as f and T as x)
$(\lambda x.F)((\lambda x.F)((\lambda x.F)((\lambda x.F)T)))$	β-reduction (apply T as x)
$(\lambda x.F)((\lambda x.F)((\lambda x.F)F))$	β-reduction (apply F as x)
$(\lambda x.F)((\lambda x.F)F)$	β-reduction (apply F as x)
$(\lambda x.F)F$	β-reduction (apply F as x)
F	$ZERO?\ FOUR \rightarrow F$

Table 8.3.1: Checking if a (non-zero) number is $ZERO$

$ZERO?\ ZERO$

$ZERO?\ ZERO$	expand the representation of $ZERO$?
$(\lambda n.n(\lambda x.F)T)\ ZERO$	β-reduction (apply $ZERO$ as n)
$ZERO\ (\lambda x.F)\ T$	expand the representation of $ZERO$
$(\lambda fx.x)\ (\lambda x.F)\ T$	β-reduction (apply $(\lambda x.F)$ as f and T as x)
T	$ZERO?\ ZERO \rightarrow T$

Table 8.3.2: Checking if a (zero) number is $ZERO$

8.4 LEQ? - is a number less than or equal to another?

$$\lambda mn.ZERO?\ (SUB\ m\ n)$$

$LEQ?$ Asks if m is less than or equal to n. Since the minimum we can get from SUB is $ZERO$, if n is greater than or equal to m, $(SUB\ m\ n)$ will be $ZERO$. Thus we can just check if SUB is $ZERO$, if it is, m is less than or equal to n.

$LEQ?\ FOUR\ TWO$

$LEQ?\ FOUR\ TWO$	expand the representation of $LEQ?$
$(\lambda mn.ZERO?\ (SUB\ m\ n))\ FOUR\ TWO$	β-reduction (apply functions as m and n)
$ZERO?\ (SUB\ FOUR\ TWO)$	β-reduction (apply functions to SUB)
$ZERO?\ TWO$	β-reduction (apply TWO to $ZERO$)
F	$LEQ?\ FOUR\ TWO \rightarrow F$

Table 8.4.1: Checking if a (greater) number is less than or equal to another

$LEQ?\ TWO\ TWO$

$LEQ?\ TWO\ TWO$	expand the representation of $LEQ?$
$(\lambda mn.ZERO?\ (SUB\ m\ n))\ TWO\ TWO$	β-reduction (apply functions as m and n)
$ZERO?\ (SUB\ TWO\ TWO)$	β-reduction (apply functions to SUB)
$ZERO?\ ZERO$	β-reduction (apply $ZERO$ to $ZERO$)
T	$LEQ?\ TWO\ TWO \rightarrow T$

Table 8.4.2: Checking if a (equal) number is less than or equal to another

$LEQ?\ TWO\ FOUR$

$LEQ?\ TWO\ FOUR$	expand the representation of $LEQ?$
$(\lambda mn.ZERO?\ (SUB\ m\ n))\ TWO\ FOUR$	β-reduction (apply functions as m and n)
$ZERO?\ (SUB\ TWO\ FOUR)$	β-reduction (apply functions to SUB)
$ZERO?\ ZERO$	β-reduction (apply $ZERO$ to $ZERO$)
T	$LEQ?\ TWO\ FOUR \rightarrow T$

Table 8.4.3: Checking if a (lesser) number is less than or equal to another

8.5 EQ? - is a number equal to another?

$$\lambda mn.AND\ (LEQ?\ m\ n)\ (LEQ?\ n\ m)$$

If m is less than or equal to n and n is also less than or equal to m, m and n must be equal.

$EQ?\ FOUR\ TWO$	
$EQ?\ FOUR\ TWO$	expand representation of $EQ?$
$(\lambda mn.AND(LEQ?\ m\ n)(LEQ?\ n\ m))\ FOUR\ TWO$	β-reduction
$AND(LEQ?\ FOUR\ TWO)(LEQ?\ TWO\ FOUR)$	β-reduction (evaluate $LEQ?$s)
$AND\ F\ T$	β-reduction (evaluate AND)
F	$EQ?\ FOUR\ TWO \to F$

Table 8.5.1: Checking if a (unequal) numbers are equal

$EQ?\ TWO\ TWO$	
$EQ?\ TWO\ TWO$	expand representation of $EQ?$
$(\lambda mn.AND(LEQ?\ m\ n)(LEQ?\ n\ m))\ TWO\ TWO$	β-reduction
$AND(LEQ?\ TWO\ TWO)(LEQ?\ TWO\ TWO)$	β-reduction (evaluate $LEQ?$s)
$AND\ T\ T$	β-reduction (evaluate AND)
T	$EQ?\ TWO\ TWO \to T$

Table 8.5.2: Checking if a (equal) numbers are equal

Control Flow Now we need a way to use our Booleans and predicates. We can create if-then-else statements with lambdas.

8.6 IFTHENELSE - control flow statements

$$\lambda pab.pab$$

IFTHENELSE takes in a Boolean p and two expressions a and b. Since a Boolean value takes two arguments and returns one, we can simply pass a and b to our Boolean. a is the "then" case (as T picks the first value) and b is the "else" case (as F picks the second value.) We can also pass (evaluated) predicates to IFTHENELSE.

$IFTHENELSE\ (EQ?\ TWO\ FOUR)\ (SUCC\ TWO)\ (PRED\ TWO)$	
$IFTHENELSE\ (EQ?\ TWO\ FOUR)(SUCC\ TWO)(PRED\ TWO)$	expand $IFTHENELSE$
$(\lambda pab.pab)\ (EQ?\ TWO\ FOUR)\ (SUCC\ TWO)\ (PRED\ TWO)$	β-reduction
$(EQ?\ TWO\ FOUR)\ (SUCC\ TWO)\ (PRED\ TWO)$	β-reduction (eval. $EQ?$)
$F\ (SUCC\ TWO)\ (PRED\ TWO)$	β-reduction (eval. F)
$PRED\ TWO$	β-reduction (eval. $PRED$)
ONE	yay

Table 8.6.1: $IFTHENELSE$ example

8.7 Control flow statements continued

If we look at the general behavior of $IFTHENESLE$:
eg.

$IFTHENELSE\ b\ x\ y$	
$IFTHENELSE\ b\ x\ y$	expand $IFTHENELSE$
$(\lambda pab.pab)\ b\ x\ y$	β-reduction
$b\ x\ y$	$IFTHENELSE\ b\ x\ y \to b\ x\ y$

Table 8.7.1: general $IFTHENELSE$ example

We can see that $IFTHENELSE$ dosen't actually do anything for us besides add more text. We apply the arguments in the same order, and a Boolean can pick an if or else clause on its own. So we can cut out $IFTHENELSE$ and just use Booleans (and predicates) for control flow.

eg.

$(EQ?\ TWO\ FOUR)\ (SUCC\ TWO)\ (PRED\ TWO)$	
$(EQ?\ TWO\ FOUR)\ (SUCC\ TWO)\ (PRED\ TWO)$	β-reduction (eval. $EQ?$)
$F\ (SUCC\ TWO)\ (PRED\ TWO)$	β-reduction (eval. F)
$PRED\ TWO$	β-reduction (eval. $PRED$)
ONE	yay

Table 8.7.2: Control flow without $IFTHENELSE$

This is almost identical to the flow of this example with $IFTHENELSE$, but we have cut out the first two steps.

This behavior can be further proven using η-reduction

$IFTHENELSE$	
$IFTHENELSE$	expand $IFTHENELSE$
$\lambda pab.pab$	expand further
$\lambda p.(\lambda a.(\lambda b.pab))$	η-reduction (reduce $\lambda b.pab$)
$\lambda p.(\lambda a.pa)$	η-reduction (reduce $\lambda a.pa$)
$\lambda p.p$	substitute for previously defined function
I	$IFTHENELSE$ is nothing more than a fancy identity

Table 8.7.3: $IFTHENELSE$ η-reduction

9 Naming Expressions and Defining Variables

Overview If we remember, lambda abstractions actually do not have names, we have been naming an using some expressions so far for ease of use, but this is technically cheating. As long as we recognize that named lambda expressions don't really exist, it's fine to use them in notation. However it is fun to try to figure out how we would "name" expressions in pure lambda calculus.

Let expressions Lets say we have the function $\lambda x.x$ and we want to name it I. What if we take the lambda expression $(\lambda I.\{someBody\})(\lambda x.x)$. We have effectively named the identity function. If someBody wants to use the identity function, it can use I (which has been bound by the abstraction) instead of $\lambda x.x$

	$(\lambda I.I\ I)(\lambda x.x)$
$(\lambda I.I\ I)(\lambda x.x)$	β-reduction (apply $(\lambda x.x)$ as I)
$(\lambda x.x)(\lambda x.x)$	α-conversion
$(\lambda y.y)(\lambda x.x)$	β-reduction (apply $(\lambda x.x)$ as y)
$(\lambda x.x)$	

Table 9.0.1: Simple let expression

We have just named I and used it in a (very simple) computation with pure lambda calculus!

10 The Y Combinator

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

Overview The Y combinator makes it possible to implement recursion in Lambda Calculus. Remember that Lambda expressions do not have names, we can not simply have a function reference itself. The Y combinator gives us what is called a fixed point, allowing us to reference the function itself. When we apply the Y combinator to a function (Y f) it will evaluate to f (Y f). Essentially, calling Y f gives us a "copy" of f that we can use and pass more arguments to.

Evaluation expansion Let us go through an example to see how the Y combinator actually evaluates. We need to give the Y combinator a "not-quite-recursive" function to make recursive. We will call this function "G"

$Y\ G$	
$Y\ G$	expand representation of Y?
$(\lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ G$	β-reduction (apply G as f)
$(\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x))$	α-conversion
$(\lambda z.\ G\ (z\ z))\ (\lambda x.\ G\ (x\ x))$	β-reduction (apply $(\lambda x.\ G\ (x\ x))$ as z)
$G\ ((\lambda x.\ G\ (x\ x))\ (\lambda x.\ G\ (x\ x)))$	substitute a previously defined expression (step 3)
$G\ (Y\ G)$	$Y\ G \rightarrow G\ (Y\ G)$

Table 10.0.1: Y Combinator of a general function

Practical usage This is cool but what does it mean? First let us define a "traditionally" recursive addition function. I will be using scheme syntax here.

```
(define add
 (lambda (x)
   (lambda (y)
     (cond
       ((zero? y) x)
       (#t ((add (+ x 1) (- y 1)))))))
```

The problem is that the define statement is not allowed in the lambda calculus. To fix this, lets abstract the problem out one layer and have the function require us to pass a reference to itself.

```
(lambda (f)
  (lambda (x)
    (lambda (y)
      (cond
        ((zero? y) x)
        (#t ((f (+ x 1) (- y 1)))))))
```

This is our "not-quite-recursive" function that we can pass to the Y combinator! In lambda calculus notation we can represent add as

$$\lambda fxy.(ZERO?y)\ x\ (f\ (SUCC\ x)\ (PRED\ y))$$

Let's call this function ADDr. If we pass ADDr to Y, we will end up with the function ADDr (Y ADDr). However, ADDr takes 3 arguments, f, x, and y. (Y ADDr) will get passed in as f, but we need to pass in two numbers too. So we can take the function (Y ADDr) and give it an x and y. Evaluating like this: (Y ADDr) x y → ADDr (Y ADDr) x y. The function ADDr now has its function reference and numeric input! Let's go through an example of actually calculating the addition this way. First, let's turn ADDr into a recursive function. (this example is very involved, a cleaner version is on the next page)

$Y\ ADDr$

$Y\ ADDr$	expand representation of $Y\ ADDr$
$ADDr\ (Y\ ADDr)$	expand representation of $ADDr$
$(\lambda fxy.(ZERO?y)x(f\ (SUCCx)(PREDy)))(Y\ ADDr)$	β-reduction (apply $(Y\ ADDr)$ as f)
$\lambda xy.(ZERO?y)x((Y\ ADDr)\ (SUCCx)(PREDy))$	

Now lets pass some numbers to our newly recursive add function - $(Y\ ADDr)$

$(\lambda xy.(ZERO?y)x((Y\ ADDr)\ (SUCCx)(PREDy)))\ TWO\ TWO$

$(\lambda xy.(ZERO?y)x((Y\ ADDr)\ (SUCCx)(PREDy)))\ TWO\ TWO$	β-reduction
$(ZERO?\ TWO)\ TWO\ ((Y\ ADDr)\ (SUCC\ TWO)(PRED\ TWO))$	β-reduction (eval. $ZERO?$)
$F\ TWO\ ((Y\ ADDr)\ (SUCC\ TWO)(PRED\ TWO))$	β-reduction (eval. F)
$(Y\ ADDr)\ (SUCC\ TWO)(PRED\ TWO)$	β-reduction (eval. $Y\ ADDr$)
$Y\ (Y\ ADDr)\ (SUCC\ TWO)(PRED\ TWO)$	β-reduction (eval. $SUCC$ & $PRED$)
$Y\ (Y\ ADDr)\ THREE\ ONE$	expand representation of $ADDr\ (Y\ ADDr)$
$\lambda xy.(ZERO?y)x((Y\ ADDr)\ (SUCCx)(PREDy))\ THREE\ ONE$	

Now we are back exactly where we started with 2 new arguments. This is recursion! Lets keep going

$(\lambda xy.(ZERO?y)x((Y\ ADDr)\ (SUCCx)(PREDy)))\ THREE\ ONE$	β-reduction
$(ZERO?\ ONE)\ THREE\ ((Y\ ADDr)\ (SUCC\ THREE)(PRED\ ONE))$	β-reduction (eval. $ZERO?$)
$F\ TWO\ ((Y\ ADDr)\ (SUCC\ THREE)(PRED\ ONE))$	β-reduction (eval. F)
$(Y\ ADDr)\ (SUCC\ THREE)(PRED\ ONE)$	β-reduction (eval. $Y\ ADDr$)
$Y\ (Y\ ADDr)\ (SUCC\ THREE)(PRED\ ONE)$	β-reduction (eval. $SUCC$ & $PRED$)
$ADDr\ (Y\ ADDr)\ FOUR\ ZERO$	expand representation of $ADDr\ (Y\ ADDr)$
$\lambda xy.(ZERO?y)x((Y\ ADDr)\ (SUCCx)(PREDy))\ FOUR\ ZERO$	

Once More!

$(\lambda xy.(ZERO?y)x((Y\ ADDr)\ (SUCCx)(PREDy)))\ FOUR\ ZERO$	β-reduction
$(ZERO?\ ZERO)\ FOUR\ ((Y\ ADDr)\ (SUCC\ FOUR)(PRED\ ZERO))$	β-reduction (eval. $ZERO?$)
$T\ FOUR\ ((Y\ ADDr)\ (SUCC\ FOUR)(PRED\ ZERO))$	β-reduction (eval. T)
$FOUR$	

Table 10.0.2: Verbose Y Combinator making ADDr recursive

$(Y\ ADDr)\ FOUR\ FIVE$	
$(Y\ ADDr)\ FOUR\ FIVE$	expand representation of $(Y\ ADDr)$
$ADDr\ (Y\ ADDr)\ FOUR\ FIVE$	expand representation of $ADDr$
$(\lambda fxy.(ZERO?y)\ x\ (f\ (SUCC\ x)\ (PRED\ y)))\ (Y\ ADDr)\ FOUR\ FIVE$	β-reduction (apply all)
$(ZERO?FIVE)\ FOUR\ ((Y\ ADDr)\ (SUCC\ FOUR)\ (PRED\ FIVE))$	β-reduction (eval $(ZERO?FIVE)$)
$(Y\ ADDr)\ (SUCC\ FOUR)\ (PRED\ FIVE)$	β-reduction (eval $SUCC$ & $PRED$)
$(Y\ ADDr)\ FIVE\ FOUR$	expand representation of $(Y\ ADDr)$
$ADDr\ (Y\ ADDr)\ FIVE\quad FOUR$	β-reduction (apply arguments to $ADDr$)
$ADDr\ (Y\ ADDr)\ SIX\quad THREE$	β-reduction (apply arguments to $ADDr$)
$ADDr\ (Y\ ADDr)\ SEVEN\quad TWO$	β-reduction (apply arguments to $ADDr$)
$ADDr\ (Y\ ADDr)\ EIGHT\quad ONE$	β-reduction (apply arguments to $ADDr$)
$ADDr\ (Y\ ADDr)\ NINE\quad ZERO$	β-reduction (apply arguments to $ADDr$)
$NINE$	$(Y\ ADDr)\ FOUR\ FIVE \rightarrow NINE$

Table 10.0.3: Recursive addition of $FOUR$ and $FIVE$

11 Church Encoding - Signed Numbers

11.1 Overview

Since, we can't apply a function a negative number of times, we need a different way to represent signed numbers than the encoding we currently have.

Strategy We will encode our new numbers as a pair of two church numerals. The first element will be a positive part and the second element will be a negative part. So our number is the first numeral minus the second numeral.

Examples

- $(PAIR\ FOUR\ ZERO)$ = 4
- $(PAIR\ ZERO\ FOUR)$ = -4
- $(PAIR\ TWO\ FOUR)$ = -2
- $(PAIR\ FOUR\ TWO)$ = 2
- $(PAIR\ FOUR\ FOUR)$ = 0

11.2 CONVERTs - Convert to signed number

$$\lambda x.V\ x\ ZERO$$

Convert a church numeral to a signed number. The church numeral becomes the positive part of the signed numeral and the negative part can just be zero.

note: remember V is equivalent to $PAIR$, I will use V to denote pairs from now on for simplicity. Essentially (V x y) denotes an x/y pair

CONVERTs FOUR

$CONVERTs\ FOUR$	expand $CONVERTs$
$(\lambda x.V\ x\ ZERO)\ FOUR$	β-reduction (apply $FOUR$ as x)
$(V\ FOUR\ ZERO)$	$CONVERTs\ FOUR \rightarrow (V\ FOUR\ ZERO)$
	$(V\ FOUR\ ZERO)$ represents 4

Table 11.2.1: Convert a numeral to a signed number

11.3 NEGs - Negate a number

$$\lambda x.V\ (SECOND\ x)\ (FIRST\ x)$$

We can negate a signed number by simply switching the positive and negative parts

NEGs (V FOUR ZERO)

$NEGs\ (V\ FOUR\ ZERO)$	expand $NEGs$
$(\lambda x.V\ (SECOND\ x)\ (FIRST\ x))\ (V\ FOUR\ ZERO)$	β-reduction (apply $(V\ FOUR\ ZERO)$ as x)
$(V\ (SECOND\ (V\ FOUR\ ZERO))\ (FIRST\ (V\ FOUR\ ZERO)))$	β-reduction (eval $FIRST$ and $SECOND$)
$(V\ ZERO\ FOUR)$	$NEGs\ (V\ FOUR\ ZERO) \rightarrow (V\ ZERO\ FOUR)$
	$(V\ ZERO\ FOUR)$ represents -4

Table 11.3.1: Negate a signed number

11.4 SIMPLIFYs - Simplify a signed number

Overview A signed number really only needs one non-zero value in the pair. For example ($V\ FOUR\ TWO$) (which represents 2) can and should be simplified to ($V\ TWO\ ZERO$).

SIMs Simplify will be a recursive function that we need to build using the Y combinator. We will first build the not-quite-recursive function $SIMs$

$$\lambda f.\lambda x.(OR\ (ZERO?\ (FIRST\ x))\ (ZERO?\ (SECOND\ x)))\ x\ (f\ (V\ (PRED\ (FIRST\ x))\ (PRED\ (SECOND\ x))))$$

This is a lot so lets go through it step by step

1. "$(OR\ (ZERO?\ (FIRST\ x))\ (ZERO?\ (SECOND\ x)))$" Check if either element of the pair is 0
2. "x" if either element is 0 we can return the pair, it is simplified
3. "$(f\ (V\ (PRED\ (FIRST\ x))\ (PRED\ (SECOND\ x))))$", otherwise run the function again, decrementing each element of the pair. Since we subtract one from the positive and negative parts simultaneously, we aren't actually changing the represented number itself, just simplifying its representation.

SIMPLIFYs To make simplify, we just make our not-quite-recursive simplify recursive with the Y combinator.

$$Y\ SIMs$$

$SIMPLIFYs\ (V\ FOUR\ TWO)$	
$SIMPLIFYs\ (V\ FOUR\ TWO)$	β-reduction (neither element is $ZERO$ so decrement pair)
$SIMPLIFYs\ (V\ THREE\ ONE)$	β-reduction (neither element is $ZERO$ so decrement pair)
$SIMPLIFYs\ (V\ TWO\ ZERO)$	β-reduction (one element is $ZERO$ so return pair)
$(V\ TWO\ ZERO)$	$SIMPLIFYs\ (V\ FOUR\ TWO) \rightarrow (V\ TWO\ ZERO)$

Table 11.4.1: Simplify representation of a signed number

11.5 PLUSs - Add two signed numbers

$$\lambda xy.SIMPLIFYs\ (V\ (ADD\ (FIRST\ x)\ (FIRST\ y))\ (ADD\ (SECOND\ x)\ (SECOND\ y)))$$

To add a signed number, we can simply add the positive and negative parts together (and simplify it at the end). eg.

$$x + y = [x_p, x_n] + [y_p, y_n] = x_p - x_n + y_p - y_n = (x_p + y_p) - (x_n + y_n) = [x_p + y_p, x_n + y_n]$$

$PLUSs\ (V\ FOUR\ ZERO)\ (V\ ZERO\ TWO)$	
$PLUSs\ (V\ FOUR\ ZERO)\ (V\ ZERO\ TWO)$	β-reduction (add the matching elements of the pairs)
$SIMPLIFYs\ (V\ FOUR\ TWO)$	β-reduction (simplify the number)
$(V\ TWO\ ZERO)$	$4 + (-2) = 2$

Table 11.5.1: add two signed numbers

11.6 MULTs - Multiply two signed numbers

$$\lambda xy.SIMPLIFYs\ (V\ [ADD\ (MULT\ (FIRST\ x)\ (FIRST\ y))\ (MULT\ (SECOND\ x)\ (SECOND\ y))]$$
$$[ADD\ (MULT\ (FIRST\ x)\ (SECOND\ y))\ (MULT\ (SECOND\ x)\ (FIRST\ y))])$$

Signed multiplication is defined:

$$x - y = [x_p, x_n] * [y_p, y_n] = (x_p - x_n) * (y_p - y_n) = (x_p * y_p + x * n * y_n) - (x_p * y_n + x_n * y_p)$$
$$= [x_p * y_p + x * n * y_n, x_p * y_n + x_n * y_p]$$

12 Glossary of common definitions

12.1 Combinators

The I Combinator Identity

$$\lambda x.x$$

The M Combinator "Mockingbird"

$$\lambda x.xx$$

The K Combinator "Kestral"

$$\lambda x.\lambda y.x$$

The KI Combinator "Kite"

$$\lambda xy.y$$

Alternate definitions

$$K\ I$$

$$C\ K$$

The C Combinator "Cardinal"

$$\lambda fab.fba$$

The V Combinator "Virio"

$$\lambda xyf.fxy$$

The Φ Combinator Shift and increment pair

$$\lambda p.PAIR\ (SECOND\ p)\ (SUCC(SECOND\ p))$$

The Y Combinator Recursion

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

12.2 Church Booleans

T True

$$\lambda ab.a$$

F False

$$\lambda ab.b$$

NOT Boolean not

$$\lambda p.pFT$$

AND Boolean and

$$\lambda p.pqp$$

OR Boolean or

$$\lambda p.ppq$$

BEQ Boolean equality

$$\lambda p.pq(NOT\ q)$$

12.3 Church Numerals

ZERO 0

$$\lambda fx.x$$

ONE 1

$$\lambda fx.fx$$

TWO 2

$$\lambda fx.f(fx)$$

THREE 3

$$\lambda fx.f(f(fx))$$

n a positive integer N

$$\lambda fx.f^n(x)$$

12.4 Natural Number Arithmetic

SUCC successor

$$\lambda nfx.f(nfx)$$

ADD addition

$$\lambda mn.m\ SUCC\ n$$

MULT multiplication

$$\lambda mn.m\ (ADD\ n)\ ZERO$$

POW exponentiation

$$\lambda mn.nm$$

PRED $n - 1$

$$\lambda n.FIRST\ (n\ \Phi\ (PAIR\ ZERO\ ZERO))$$

SUB subtraction

$$\lambda mn.n\ PRED\ m$$

12.5 Pairs and Lists

PAIR Pair two expressions

$$\lambda xyf.fxy$$

FIRST Access first element of pair

$$\lambda p.pK$$

SECOND Access second element of pair

$$\lambda p.p\ KI$$

NIL Empty pair

$$\lambda f.T$$

12.6 Predicates and Control Flow

NIL? checks if a pair is NIL

$$\lambda p.p(\lambda xy.F)$$

ZERO? checks if a numeral is ZERO

$$\lambda n.n(\lambda x.F)T$$

LEQ? checks if a numeral is less than or equal to another

$$\lambda mn.ZERO?\ (SUB\ m\ n)$$

EQ? checks if a numeral equal to another

$$\lambda mn.AND\ (LEQ?\ m\ n)\ (LEQ?\ n\ m)$$

IFTHENELSE control flow statements

$$\lambda pab.pab$$

12.7 Signed Numbers and Arithmetic

CONVERTs convert to signed number

$$\lambda x.V\ x\ ZERO$$

NEGs negate a number

$$\lambda x.V\ (SECOND\ x)\ (FIRST\ x)$$

SIMPLIFYs simplify a signed number

$$Y\ (\lambda f.\lambda x.(OR\ (ZERO?\ (FIRST\ x))\ (ZERO?\ (SECOND\ x)))\ x\ (f\ (V\ (PRED\ (FIRST\ x))\ (PRED\ (SECOND\ x)))))$$

PLUSs add two signed numbers

$$\lambda xy.SIMPLIFYs\ (V\ (ADD\ (FIRST\ x)\ (FIRST\ y))\ (ADD\ (SECOND\ x)\ (SECOND\ y)))$$

MULTs multiply two signed numbers

$$\lambda xy.SIMPLIFYs\ (V\ [ADD\ (MULT\ (FIRST\ x)\ (FIRST\ y))\ (MULT\ (SECOND\ x)\ (SECOND\ y))]\ [ADD\ (MULT\ (FIRST\ x)\ (SECOND\ y))\ (MULT\ (SECOND\ x)\ (FIRST\ y))])$$

13 Sources/Further Reading/Links

1. The videos that initially sparked my interest
   Part 1: https://www.youtube.com/watch?v=6BnVo7EHO_8
   Part 2: https://www.youtube.com/watch?v=pAnLQ9jwN-E
2. The article that finally helped me understand the Y-combinator
   https://sookocheff.com/post/fp/recursive-lambda-functions/
3. Wikipedia Pages
   Lambda Calculus: https://en.wikipedia.org/wiki/Lambda_calculus
   Church Encoding: https://en.wikipedia.org/wiki/Church_encoding