Chapter 1

Lambda Calculus

Baaaa

Sheep

1.1 Intro

According to Wikipedia[1], "*Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.*"

While correct, we will be treating it as a Turing complete language which is the basis of many functional languages. So we will go over how to create statements in this language, and how to evaluate them.

Let us begin by giving the grammar for the language:

$$\begin{aligned} e \Rightarrow \quad & x \\ & |\lambda x.e \\ & |ee \end{aligned}$$

x here is a variable. That's it. A very small language grammar. Using this grammar the following statements are valid:

x	$x\ y$	$\lambda x.x$	$\lambda x.y$
xyz	xx	$\lambda x.\lambda y.x$	$\lambda x.\lambda y.xy$

1.2 Turing Complete

As we saw earlier, finite state machine can only solve certain types of problems. Different machines have different levels of computational power. Checking if a string is accepted by a regular expression requires a Finite State Machine. Checking if a string is accepted by a Context Free Languages requires a Push Down Automata (PDA)[2]. Recursively enumerable languages need a Turing machine. A Turing Machine[3], can solve any computable problem. Some problems, we know cannot be solved: like the halting problem, thus a Turing machine cannot solve this problem. But if we know the problem can be solved, then a Turing machine, can model it.

It is important to note the difference of syntax and semantics here. CFGs and regular expressions only describe the syntax of the language. Since they only describe sets of strings, they say nothing about what those strings can express (in this case, a language is just a set of strings that is allowed). What a language is capable of expressing (or its semantics) is a different question (in this case, a language is a way to express ideas). We will now pull away from the former way of speaking about languages, and start discussing the latter. One property the semantics of a language could have is Turing

[1]https://en.wikipedia.org/wiki/Lambda_calculus
[2]We will not be covering PDAs in this course
[3]initially called a-machines or atomic machines

Completeness.

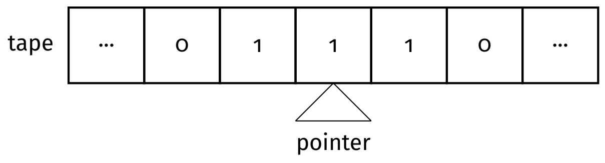
When we say a language is Turing complete, we mean that language can model or simulate a Turing machine.. A Universal Turing Machines (UTM), is a Turing machines that can produce other Turing machines, where each machine solves a particular problem (a machine that ANDs can't be used to OR). What does a Turing complete language look like? Well, despite the fact that the syntax of Lambda Calculus can be represented as a CFG, the semantics of the language is enough to be Turing complete.

1.3 Turing Machines

We said that a Turing complete language is one which can simulate a Turing machine. What exactly is a Turing Machine? It is a machine that has the following properties:

- Has an infinite ticker tape (a tape with an infinite number of cells)
- Each cell on the ticker tape stores a symbol from a finite alphabet (typically either a 1 or a 0)
- Has some "pointer" that can points to a cell on the tape
- The pointer needs to be able to move left or right one cell
- Has a writer and reader on the pointer to read the value in the cell or overwrite its value
- has a list of "states" that tell the pointer's reader and writer what to do and what other state to move to

Here is an example of a Turing Machine that will zero out the tape until a 0 is read in on both sides of the starting point:

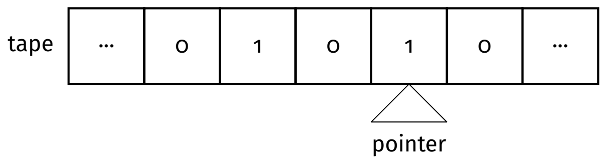


Read in	State A			State B		
	Write	Move	New State	Write	Move	New State
0	0	L	B	0	R	Halt
1	0	L	A	1	R	A

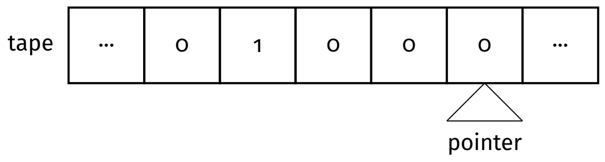
Assuming we start in the cell seen above and start in State A, here is the next few things that will happen:

1. We read in a 1, and are in state A so we write a 0 and move left, staying in state A
2. We read in another 1, and are in state A so we write a 0 and move left, staying in state A
3. We read in 0, and are in state B so we write a 0 and move Right and Halt.

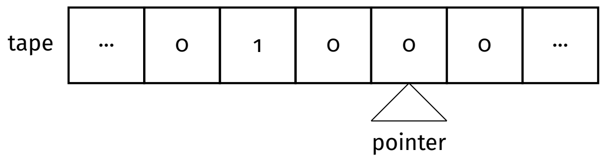
These steps can be visualized here:
After Step 1:



After Step 2:



After Step 3:



This process gets very complicated when writing larger programs, but this machine can be used to implement any computer algorithm. Thus any language that can simulate this machine (much like we simulated a FSM as a project in the course), then we say that language is Turing complete. Lambda Calculus is one such language that is Turing Complete.

1.4 Lambda Calculus Semantics

As we saw in the intro section, the syntax of the language is very small. It's now time to figure out what do those lambda calculus sentences mean.

1.4.1 Variables

Let's start with the basic 'x' sentence. In languages like C, a variable by itself means whatever value is bound to that variable.

```
int x = 3;
x; //here x means 3
```

In lambda calculus, this meaning stays the same, however what the variable is bound too is sometimes not given.

```
  y //y not defined, but understood to be a variable
/*
Analogous to the following C code
void foo(){
  y; //not defined in this scope, but understood to be defined elsewhere.
}
*/
```

1.4.2 Function Definitions

The next type of sentence we is referred to as a lambda function. A lambda function takes the form $\lambda x.e$, where x is a variable and e is another lambda expression. As seen in the intro section, $\lambda y.x$ is a valid lambda expression, where y is a variable and x is another lambda expressions (in this case, a variable). This is called a function because it has the same semantics of a function we may see in other languages. Let's first break down it's structure though.

$$\lambda\boxed{\text{y}}.\underline{x}$$

This is a lambda function where the $\boxed{\text{y}}$ can be thought of as the parameter to the function, and $\underline{x}$ can be thought of as the body.
Since lambda functions are also lambda expressions, they can be placed as the body of another function: $\lambda x.\lambda y.y$. This one is more complicated, so let's look at it's structure:

$$\lambda\boxed{\text{x}}.\underline{\lambda\boxed{\text{y}}.\underline{y}}$$

In this case, the λx function has parameter $'x'$ and body of $\lambda y.y$. The λy function has parameter $'y'$ and body $'y'$.

Here is something analogous in other languages:

For reference: lambda calculus: $\lambda x.\lambda y.y$	`# Python`
	`lambda x: lambda y: y`

`(* Ocaml *)`
`fun x -> fun y -> y`
	`# Ruby`
	`Proc.new{|x| Proc.new{|y| y}}`

`// C`
`void* bar(void* y) { return y;}`
`void* (*foo(void* x))(void*){ return bar; }`
	`// Java`
	`interface Lambda{ Object run(int i); }`
	`public static Lambda foo(Object x){`
	`  return (Lambda)((y) -> y);`
	`}`

Each thing here is a function, that takes in one parameter, and then returns a function (where that returned function is the identity function).

As you can imagine, we can nest these even more: $\lambda x.\lambda y.\lambda z.z$. This becomes important when we change the body of the nested function: $\lambda x.\lambda y.x$ and start to do function application. More on this in a few sections.

1.4.3 Function Application

As the name suggests the last part is a way to call a function. However, this is also a bit of a misnomer, since sometimes we cannot apply a function. Let's expand on this.

$$(\lambda x.x)a$$

This is a lambda expression that consists of 2 sub-expressions: $(\lambda x.x)$ and a. When we have this structure, we call the left sub-expression using the right sub-expression as its input. Specifically, a will be used in the $(\lambda x.x)$ function. Since this function is the identity function, we just get back a.

$$(\lambda x.x)a \Rightarrow a$$

This is equivalent to calling a function and getting the return value. Here are some analogous examples in some programming languages:

For reference: lambda calculus: $(\lambda x.x)a$	`# Python`
	`(lambda x: x)(a)`

`(* Ocaml *)`
`(fun x -> x) a`
	`# Ruby`
	`Proc.new{|x| x}.call(a)`

`// C`
`void* foo(void* y) { return y;}`
`foo(a);`
	`// Java`
	`interface Lambda{ Object run(Object i); }`
	`((Lambda)(x) -> x).run(a)`

Notice, that we have an argument, and it is being replaced in the body with whatever the input is. Thus when given something like $(\lambda x.y)a$, we get y back. To see this in more detail, consider the following:

```
int foo(int x){
  return x;
}
foo(3) //returns 3, because 3 is our input, where ever we see x, replace with 3.

int bar(int x){
  return 4;
}
bar(3) //returns 4, because 3 is our input, but we don't use x anywhere
```

To see more complicated examples, we first need to consider the following lambda expression:

$$a\ b$$

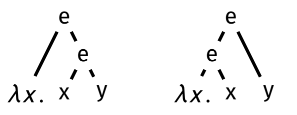
This lambda expression has two sub-expressions: a and b. Here however, a is not a lambda function, so we have no idea how to call a with b as input. In this case, since we cannot call a function, we say this expression means exactly what it says: $a\ b$.
To see this side by side:
$(\lambda x.x)\ a \Rightarrow a \ \big|\ (\lambda x.y)\ a \Rightarrow y \ \big|\ a\ b \Rightarrow a\ b$

1.5 Removing Ambiguity

Now that we have the basics down, let's consider something more complicated:

$$\lambda x.x\ y$$

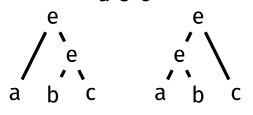
We can see this is an ambiguous statement. Either $x\ y$ the body of the λx function, or y is being applied to the $\lambda x.x$ function.



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

Here is another example of an ambiguous expression:

$$a\ b\ c$$



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

It is initially unclear if we are calling a with the input $b\ c$ or if we are calling a with input b and then calling c on the result of $a\ b$.
To help remove ambiguity, there are some explicit rules that Lambda Calculus follows.

- Expressions are left associative
- The scope of a function goes until the end of the entire expression or until a (unmatched) parenthesis is reached

1.5.1 Left Associative

When we say expressions are left associative, that means when given a series of expressions, for example three: $e_1\ e_2\ e_3$, then we group the left two items together before grouping the next item: $(e_1\ e_2)\ e_3$. This means that we take the right tree in the above example with $a\ b\ c$. As you can imagine, we chain this rule with larger expressions. Thus: $a\ b\ c\ d\ e\ f$ has implicit parenthesis: $(((((a\ b)\ c)\ d)\ e)\ f)$.

For functions, this is important since it tells us what order we should apply things:

$$\begin{aligned} & (\lambda x.\lambda y.y)\ a\ b \\ \Rightarrow\ & ((\lambda x.\lambda y.y)\ a)\ b \\ \Rightarrow\ & (\lambda y.y)\ b \\ \Rightarrow\ & b \end{aligned}$$

1.5.2 Function Scope

The next important part is the scope of a lambda function. The body of of a lambda function is whatever follows the . symbol until the end of the entire expression is reached or a (unmatched) parenthesis is reached. It is important to note the parenthesis must be unmatched from the viewpoint of the lambda. Consider the following functions and their body (which is underlined):

$$\lambda x.\underline{a\ b\ y} \quad \lambda x.\underline{(a\ b)\ y} \quad \lambda x.\underline{a\ (b\ y)} \quad (\lambda x.\underline{(a\ b)})\ y \quad (\lambda x.\underline{(a\ b)\ y})\ z \quad (\lambda x.\underline{(a\ b)})\ y\ z$$

This rule continues even when we nest lambda functions:

$$\lambda x.\underline{\lambda y.\underline{a\ b\ y}} \quad \lambda x.\underline{\lambda y.\underline{(a\ b)\ y}} \quad \lambda x.\underline{\lambda y.\underline{a\ (b\ y)}} \quad \lambda x.\underline{(\lambda y.\underline{a\ b})\ y} \quad (\lambda x.\underline{(\lambda y.\underline{a})\ b})\ y$$

1.6 Reduction

The process of of applying a function is called reducing. In particular, we call a function call a beta reduction (β-reduction). A single function call is a reduction. So the following is actually performing two reductions.

$$\begin{array}{lll} & (\lambda x.\lambda y.y)\ a\ b & \\ \Rightarrow & ((\lambda x.\lambda y.y)\ a)\ b & \\ \Rightarrow & (\lambda y.y)\ b & \text{reduced the x function} \\ \Rightarrow & b & \text{reduced the y function} \end{array}$$

When you cannot reduce any further, we say the expression is in beta normal form. So b is the result of reducing $(\lambda x.\lambda y.y)\ a\ b$ to beta normal form.

When performing a beta reduction, consider there is the function being called, and the expression passed in to the function. For example: $(\lambda x.\ x)\ y$ can be beta reduced by calling the $\lambda x.\ x$ function with the input y.

What happens when the input can also be reduced? $(\lambda x.\ a\ x\ a)\ ((\lambda y.\ y\ y)\ z)$ We have two options:

- We evaluate the argument first. That is, evaluate $((\lambda y.\ y\ y)\ z)$ to $(z\ z)$ and pass that into $(\lambda x.\ a\ x\ a)$
- We shove $((\lambda y.\ y\ y)\ z)$ into $(\lambda x.\ a\ x\ a)$ first and get $a\ ((\lambda y.\ y\ y)\ z)\ a$

The first is an example of **eager** evaluation (sometimes called call-by-value), where we evaluate the argument first before passing it into the function.

The second is an example of **lazy** evaluation (sometimes called call-by-name), where we only evaluate when we need to.

For example:

```
def foo(x):
  return "str"

foo(3+4)
# using lazy evaluation, 3 + 4 is never calculated
# using eager, 3 + 4 is calculated and then never used.
# in this case lazy does less work

# Try running print(foo((lambda x: print(x))(4))).
# Does python use eager or lazy evaluation?
```

1.7 Variable Semantics

Up until this point, we have been using simple functions, but typically more complicated or harder to read functions are used. For example:

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

To figure out how to reduce this problem, we need to discuss variables and their binding.

Variables fall under two categories: free or bound. A bound variable is one who's value is dependent on a parameter of a lambda function.

$$(\lambda x.\underline{x}\ \boxed{a})\ \boxed{b}$$

In the above example, x is bound to the input parameter since they share the same name and x is in the body of the λx function. a and b are then what are known as free variables, variables not dependent on the parameter. b is not bound because it falls outside the body of the function, and does not share the same name as the parameter. a is not bound because it does not share the same name as the parameter. Consider the following C program:

```c
int a = 6; // free
int foo(int x){ // x is name of the parameter
  x == 5; // this x is bound to the parameter
  int y = 3; // free
}
```

In regards to the `foo` function, a and y are free since they are not bound to the parameter x.

This becomes important when we nest functions:

$$(\lambda x.\lambda y.\underline{x}\ \boxed{a}\ \underline{y})\ \boxed{b}$$

Here, x is bound to the outer λx parameter and the y is bound to the inner λy parameter. This gets a tad confusing when we shadow the variable:

$$(\lambda x.\lambda x.\underline{x})\ \boxed{a}$$

In these cases, we know that x is bound, but to what? In lambda calculus (and most languages that I know), the x is bound to the inner λ function. This is because the parameter is being shadowed by the inner function. Consider the following C code:

```c
int a = 6; // free
{
  int a = 5;
  printf("%d\n",a); //prints 5 here since a is shadowed by the previous line
}
printf("%d\n",a); //prints 6, now that the inner 5 is out of scope
```

Thus, we can now beta reduce complicated functions if we keep this is mind:

$$\begin{aligned} & (\lambda x.(\lambda x.x\ x)\ x)\ a \\ \Rightarrow\ & (\lambda x.x\ x)\ a \\ \Rightarrow\ & a\ a \end{aligned}$$

Here, the right most x is bound to the left most λx where as the inner two x's are bound to the inner λ function.

$$(\lambda \underline{x}.(\lambda \boxed{x}.\boxed{x}\ \boxed{x})\ \underline{x})$$

Just reiterated, where all boxed variables are related and all underlined variables are related. Here's one more for practice:

$$\begin{aligned} & (\lambda y.(\lambda x.x\ x)\ y\ x)\ a \\ \Rightarrow\ & (\lambda x.x\ x)\ a\ x \\ \Rightarrow\ & (a\ a)\ x \end{aligned}$$

To help make things more readable, there is this concept of alpha equivalence (α-equivalence). Alpha equivalence should not change the meaning of the initial statement, but rather make sure things are more readable, or to preserve the semantics of the initial statement.

To do so, an alpha-conversion (α-conversion) is the process of renaming all the variables that are bound together along with the bounded parameter to a different name.

$$(\lambda x.x)a \Rightarrow (\lambda y.y)a$$

Again, this does not change the semantics of the expression, but makes things more readable. Consider the C code:

```c
int foo(int x){
  return x + 1;
}
int bar(int y){
  return y + 1;
}
```

There is no difference between `foo` and `bar`. These two functions are α-equivalent. So let's consider our initial statement and alpha-convert it to be more readable:

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

It is important to note that you **cannot** convert free variables. You can only convert bound variables. Thus $(\lambda x.x)\ x$ is not alpha equivalent to $(\lambda x.x)\ a$.

For the most part, this just helps with readability, but sometimes it is important to alpha convert to keep the semantics. Consider the following **incorrect** reduction:

$$\begin{array}{rl} & (\lambda y.(\lambda x.y))\ x \\ \Rightarrow & (\lambda x.x) \end{array}$$

This now reduces to the identity function, but this is incorrect since the initial outer x was free and now becomes bound. To keep the semantics, free variables cannot become bound, and bound variables cannot become free. To make this correct, we must alpha convert:

$$\begin{array}{rl} & (\lambda y.(\lambda x.y))\ x \\ \Rightarrow & (\lambda y.(\lambda a.y))\ x \\ \Rightarrow & (\lambda a.x) \end{array}$$

1.8 Church Encodings

Now that we have an idea of how to evaluate Lambda Calculus, let's discuss about how we can use this as a language to calculate information.

The words we use to represent concepts are ultimately arbitrary. We all came together and agree that words like 'true' or 'false' mean something. However, these things are ultimately arbitrary when figuring out the string of symbols to represent these concepts. In OCaml for example, we say `true` and `false`, but in Python we say `True` and `False`. Even in C, there is no boolean, it's (typically) just checking if a number is 0 or not.

It then stands to reason that in lambda calculus, while we may not have strings like "true" or "false", we do have something that represents these ideas. This concept is called encoding. We want to encode information into a text string valid in the language.

The encodings that exist in Lambda Calc were made by Alonzo Church who also introduced the idea of Lambda Calculus. He encoded '`true`' and '`false`' as follows:

- True: $\lambda x.\lambda y.x$
- False: $\lambda x.\lambda y.y$

That is, the OCaml program: `true` is analogous to the lambda calculus program: $\lambda x.\lambda y.x$

This may make sense once we introduce the `if guard then true_branch else false_branch` equivalent. To write `if a then b else c` in Lambda Calculus, we just write: `a b c`. This may be confusing so consider:

- `if true then false else true`: $(\lambda x.\lambda y.x)(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)$
- `if false then false else false`: $(\lambda x.\lambda y.y)(\lambda x.\lambda y.y)(\lambda x.\lambda y.y)$
- `if false then true else false`: $(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)(\lambda x.\lambda y.y)$

To see these encodings at work, consider `if true then false else true` should evaluate to `false`. So let's see what `if true then false else true` looks like in lambda calc

$$\begin{aligned}\text{if true then false else true} &= (\lambda x.\lambda y.x)\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)\\ &\Rightarrow ((\lambda x.\lambda y.x)\ (\lambda x.\lambda y.y))\ (\lambda x.\lambda y.x)\\ &\Rightarrow (\lambda y.(\lambda x.\lambda y.y))\ (\lambda x.\lambda y.x)\\ &\Rightarrow (\lambda x.\lambda y.y))\\ &= \text{false}\end{aligned}$$

There are other encodings that exist as well, such as pairs, numbers, addition and multiplication of numbers, and,or,not on booleans, etc. See Appendix ... TODO for more examples.

1.9 Looping

One such properties of Turing Completeness is the ability to jump, conditionally, or unconditionally. This ultimately allows for looping to exist so let's examine looping in Lambda Calc.

Let us consider the following lambda calculus expression:

$$(\lambda x.xx)(\lambda x.xx)$$

If were to beta reduce this, we would get back the exact same thing. This is one such example of a lambda calculus expression which would loop to infinity and never reach a beta normal form. This particular expression is called the Ω-combinator. This expression by itself is not truly useful, but we can exploit this structure and insert a modification to achieve a conditional or at least "recursive" looping structure.

Since lambda calculus doesn't have named functions, we cannot get recursion in the same way would could in typical programming languages. However, as we saw in OCaml, functions are pieces of data and the same is true for lambda calculus. The trick here is to pass in the recursive function into a wrapper function. In lambda calculus, we will be using the Y-combinator (sometimes called a fixpoint combinator). It is as follows:

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

To see this, suppose we have a function **F**. A recursive call may look something like **`F(F(F(F(...F(`**`value`**`)...))))`**. We can obtain that using the following

$$\begin{aligned}& (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ F\\ \Rightarrow\ & (\lambda x.F(xx))(\lambda x.F(xx))\\ \Rightarrow\ & (F((\lambda x.F(xx))(\lambda x.F(xx))))\\ \Rightarrow\ & (F(F((\lambda x.F(xx))(\lambda x.F(xx)))))\\ \Rightarrow\ & (F(F(F((\lambda x.F(xx))(\lambda x.F(xx))))))\\ \Rightarrow\ & \ldots\end{aligned}$$

If we wanted to make this easier to read, let Y = $(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$ and Y F = $((\lambda x.F(xx))(\lambda x.F(xx)))$

$$\begin{aligned}Y\ F &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ F\\ &\Rightarrow (\lambda x.F(xx))(\lambda x.F(xx))\\ &\Rightarrow (F((\lambda x.F(xx))(\lambda x.F(xx))))\\ &\Rightarrow (F(Y\ F))\\ &\Rightarrow \ldots\end{aligned}$$

So now suppose we could write encode numbers in lambda calculus much like we could "true" and "false" (we can![4]). Also suppose we could encode things like "=0", "n*m" and "n-1" like we could "if then else" and "and" and "or" (we can![5]). This could mean we could write a function G like factorial in the form $G = \lambda f.(\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (f(n-1)))$
Now we can use G and a factorial number to place into Y.

$$\begin{aligned}
(Y\ G)3 = \quad & ((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ G)3 \\
\Rightarrow \quad & ((\lambda x.G(xx))(\lambda x.G(xx)))3 \\
\Rightarrow \quad & (G(\lambda x.G(xx))(\lambda x.G(xx)))3 \\
\Rightarrow \quad & (G(Y\ G))3 \\
\Rightarrow \quad & \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * ((YG)2) \\
\Rightarrow \quad & 3 * ((YG)2) \\
\Rightarrow \quad & 3 * (((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ G)2) \\
\Rightarrow \quad & 3 * (((\lambda x.G(xx))(\lambda x.G(xx)))2) \\
\Rightarrow \quad & 3 * ((G(\lambda x.G(xx))(\lambda x.G(xx)))2) \\
\Rightarrow \quad & 3 * ((G(Y\ G))2) \\
\Rightarrow \quad & 3 * (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * ((YG)1)) \\
\Rightarrow \quad & 3 * (2 * ((YG)1)) \\
\Rightarrow \quad & 3 * (2 * (((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ G)1)) \\
\Rightarrow \quad & 3 * (2 * (((\lambda x.G(xx))(\lambda x.G(xx)))1)) \\
\Rightarrow \quad & 3 * (2 * ((G(Y\ G))1)) \\
\Rightarrow \quad & 3 * (2 * \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * ((YG)0)) \\
\Rightarrow \quad & 3 * (2 * (1 * ((YG)0))) \\
\Rightarrow \quad & 3 * (2 * (1 * (((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ G)0))) \\
\Rightarrow \quad & 3 * (2 * (1 * (((\lambda x.G(xx))(\lambda x.G(xx)))9))) \\
\Rightarrow \quad & 3 * (2 * (1 * ((G(Y\ G))0))) \\
\Rightarrow \quad & 3 * (2 * (1 * \text{if } 0 = 0 \text{ then } 1 \text{ else } 1 * ((YG)0))) \\
\Rightarrow \quad & 3 * (2 * (1 * 1)) \\
\Rightarrow \quad & 3 * (2 * (1)) \\
\Rightarrow \quad & 3 * (2) \\
\Rightarrow \quad & 6
\end{aligned}$$

While this does look gross, I would highly recommend you trace through this on your own.

[4]See Appendix ... TODO
[5]See Appendix ... TODO