Lambda calculus

Vukašin S. Tasić

Github: kaseen
Linkedin: ka200seen

In 1936, mathematician Alonzo Church published a paper titled "An Unsolvable Problem of Elementary Number Theory," which served as an introduction to lambda calculus. Church's primary goal was to grasp the essence of solvability in mathematical problems. Through his work on lambda calculus, he aimed to explore the limits of computation and create a precise mathematical model to differentiate between computable and non-computable functions. This can be understood as considering the concept of a function from a computational perspective. The power of lambda calculus lies in its ability to encode any computation, making it a foundational tool for exploring the theoretical aspects of computability. This implies that any program written in any programming language can, to some extent, be encoded in Lambda calculus, which is why Lambda calculus has been referred to as 'the smallest universal programming language in the world.'

While Church did not directly solve the problem of mathematical unsolvability, his research significantly advanced our understanding of it. This, in turn, influenced Alan Turing, who was supervised by Church during his PhD and eventually provided a solution inspired by Church's contributions, known as 'The Halting Problem'. While Turing machines represent a machine-based model of computability, Church's model is purely mathematical. Despite their distinct natures — Church's lambda calculus involves functions without internal states, while Turing machines have internal states — these two models have been proven equivalent, a concept known as the Church-Turing hypothesis. Any Turing machine program can be translated into an equivalent Lambda calculus program, and vice versa.

Lambda calculus consists of three fundamental elements: variables, function construction (abstraction), and function application. Formally, lambda expressions are mathematical expressions that represent functions and are defined inductively as follows:

1. Variables: Countably many variables are considered valid λ-expressions,
   Example: x, y

2. Abstractions: If E is an expression and x is a variable, then $(\lambda x.E)$ is a λ-expression. Formally, this is called the abstraction of expression E per variable x,
   Example: $(\lambda x.x + 1)$ - *abstraction of the expression* $x + 1$ *per variable* x

3. Applications: If E and F are λ-expressions, then (EF) is also a λ-expression. Formally, this is called an application of F on E.

Example: $((\lambda x.x+1)2)$ - *application of the abstraction* $(\lambda x.x+1)$ *on the argument 2, resulting 3*

We can omit the external parentheses on rules 2 and 3, where EFG can be interpreted as $((EF)G)$, since left associativity implicitly holds.

Before we proceed further, let's draw an analogy between functions in mathematics and lambda functions, using an example of a function that increments the input value by 1. Both types of functions are defined by formulas that specify how the output value relates to the input value.

In mathematics, functions are usually assigned names such as f, g, h etc. For example, our demonstration function can be represented as $f(x) = x+1$. Therefore, applying f to an input of 6 is expressed as $f(6) = 6+1 = 7$.

In lambda calculus, functions do not have names and therefore referred to as anonymous functions. For instance, the function that increments a value by 1 is represented as $\lambda x.x + 1$. To calculate the result for an input 6, we apply this anonymous function as $(\lambda x.x + 1)6 = 6 + 1 = 7$. We will delve into this process in more detail later on.

We can distinguish between free and bound variables according to two fundamental rules: when we perform function abstraction per variable, that variable becomes bound, and any variables not bound in this process are free or unbound. This concept is not exclusive to lambda calculus. For instance, quantifiers like $\forall x$ or $\exists x$ perform a similar thing, as does a mathematical function like $f(x) = x + y$, where x is bound while y remains free.

For example, consider the lambda expression $\lambda x.x+y$. In this expression, the free variable is y because it is not locally bound within the expression $x + y$. Similarly, x is considered a bound variable as it is utilized by the function abstraction, thus being bound by it.

Furthermore, it's important to remember that variables are bound to the closest occurrence of themselves following a λ sign. For instance, in the expression $\lambda x.x(\lambda x.x)$, the placeholder x is utilized in the inner function definition and isn't associated with the one used in the outer definition. Formally, there is a need for renaming the variables in the outer definition, and $\lambda x.x(\lambda x.x)$ should be understood as $\lambda x.x(\lambda y.y)$. Additionally, in the expression $\lambda x.(\lambda y.xy)$, the variable x is unbound within the inner lambda expression, but it is bound by the outer lambda expression. This concept has a tight connection with function scopes in programming languages.

As you've likely observed, functions can be assigned to variables, used as input values for other functions, or returned as output values from functions. This is why functions are referred to as 'first-class citizens' in lambda calculus. This concept is particularly useful in writing functions that accept two (or more) arguments, given that lambda calculus lacks a straightforward means of doing so.

The technique is called 'currying', named after the mathematician Haskell Curry and derived from mathematical logic. It facilitates the transformation of a function that accepts multiple arguments into a series of functions, each accepting only one argument. This approach enhances flexibility in combining functions and executing operations.

For instance, $\lambda x.(\lambda y.(\lambda z.x + y + z))$ is functionally equivalent to $\lambda xyz.x + y + z$. If we

express these two lambda expressions in the style of a JavaScript programming language, the computation will yield the same result.

```
// called as sum1(x)(y)(z)
function sum1(x) {
    return function(y) {
        return function(z) {
            return x+y+z;
        }
    }
}

// called as sum2(x, y, z)
function sum2(x, y, z) {
    return x+y+z;
}
```

Lambda calculus also includes a set of transformation rules for manipulating the lambda terms.

1. α-conversion: it involves renaming bound variables within a lambda expression to avoid variable capture and ensure clarity in variable naming. Formally, $\lambda x.E \xrightarrow{\alpha} \lambda y.E[x := y]$, where y is a 'fresh' variable.
   Example: $(\lambda x.x + y)[x := t] \xrightarrow{\alpha} \lambda t.t + y$

2. β-conversion: it involves applying a lambda abstraction to an argument, resulting in the substitution of the argument for the bound variable within the body of the abstraction. Formally, $(\lambda x.E)F \xrightarrow{\beta} E[x := F]$.
   Example: $(\lambda x.x + y)t \xrightarrow{\beta} (x + y)[x := t] = t + y$

Some examples of α and β reductions:

$(\lambda x.xx)xy \overset{\beta}{=} xx[x := x]y = xxy$

$(\lambda x.xx)(xy) \overset{\beta}{=} xx[x := xy] = xyxy$

$(\lambda x.xx)(\lambda y.y) \overset{\beta}{=} xx[x := (\lambda y.y)] = (\lambda y.y)(\lambda y.y) \overset{\alpha}{=} (\lambda y.y)(\lambda z.z) \overset{\beta}{=} y[y := (\lambda z.z)] = \lambda z.z$

$(\lambda x.x(\lambda x.x))y \overset{\alpha}{=} (\lambda x.x(\lambda z.z))y \overset{\beta}{=} x[x := y](\lambda z.z) = y(\lambda z.z)$

$(\lambda x.(\lambda y.xy))yz \overset{\alpha}{=} (\lambda x.(\lambda t.xt))yz \overset{\beta}{=} (\lambda t.xt)[x := y]z = (\lambda t.yt)z \overset{\beta}{=} yz$

Note that any lambda expression enclosed within parentheses is considered as a single unit and processed accordingly. Rearranging the order of evaluation produces the same lambda expression:

$(\lambda x.(\lambda y.xy))((\lambda x.x)t) \overset{\beta}{=} (\lambda y.xy)[x := ((\lambda x.x)t)] = \lambda y.((\lambda x.x)t)y \overset{\beta}{=} \lambda y.(x[x := t])y = \lambda y.ty$

$(\lambda x.(\lambda y.xy))((\lambda x.x)t) \overset{\beta}{=} (\lambda x.(\lambda y.xy))(x[x := t]) = (\lambda x.(\lambda y.xy))t \overset{\beta}{=} (\lambda y.xy)[x := t] = \lambda y.ty$

As demonstrated in the example above, evaluation persists until no further reduction is possible. By definition, we consider lambda expressions E and F to be equivalent if there exists a sequence of reductions that both expressions undergo, resulting in the same lambda expression. If a lambda expression has a normal form (although it is not required to have one), then it is unique.

This lambda expression does not have normal form:

$$(\lambda x.xxy)(\lambda x.xxy) \stackrel{\beta}{=} xxy[x := (\lambda x.xxy)] = (\lambda x.xxy)(\lambda x.xxy)y = ... = (\lambda x.xxy)(\lambda x.xxy)y^n$$

But, if a lambda subexpression lacks a normal form, it does not necessarily imply that the entire expression will also lack one:

$$(\lambda u.v)((\lambda x.xxy)(\lambda x.xxy)) \stackrel{\beta}{=} v[u := ((\lambda x.xxy)(\lambda x.xxy))] = v$$

If we want to utilize logical value like True and False, we require a method to represent them in a manner that enables decision-making between two alternatives. Therefore, the concept of making choices between two options is used to encode True or False. The trick is to express them as follows:

True: $\lambda xy.x$, function which selects the first input
False: $\lambda xy.y$, function which selects the second input

This enables us to construct statement of *if M, then N, else L.*

$$\top NL = (\lambda xy.x)NL = (\lambda y.x[x := N])L = (\lambda y.N)L = N[y := L] = N, \qquad \textit{if } M = \textit{True}$$
$$\bot NL = (\lambda xy.y)NL = (\lambda y.y[x := N])L = (\lambda y.y)L = y[y := L] = L, \qquad \textit{if } M = \textit{False}$$

Functionally complete set of logical connectives is one that can be used to express any logical function, and the 'if-then-else' construction enables exactly that. We'll see this in practice problems shortly.

Also, to perform any computation using lambda calculus, numbers are crucial, necessitating their encoding as lambda expressions. This encoding process is similar to the Peano definition of natural numbers, encapsulated in the Peano axioms established in his 1889 paper, "Arithmetices principia, nova methodo exposita." Church further generalized this notion. Church numerals, also known as Church encoding, offer a method of expressing natural numbers within lambda calculus entirely through functions.

The fundamental concept is that each natural number n can be expressed as a function with two parameters: a function f and an initial value x, denoted as $\lambda f.\lambda x.expression$. This function f is then applied n times to the initial value x. The first parameter f can represent the *increment* function to be used, while the second parameter x can represent the value that serves as zero.

Before deriving the generalized formula, let's have an example how we define the numeral 2. Considering numerals have the form $\lambda f.\lambda x.expression$, and our aim is to represent the number 2, the *expression* must involve applying the function f to x twice. Hence, we arrive at the form $\lambda f.\lambda x.f(f(x))$, or in abbreviated form $\lambda fx.f(f(x))$. This lambda expression will

be denoted as $\overline{2}$, since it represents the number 2 as Church numeral. If we want to obtain the number two, we can evaluate this lambda expression using the function *increment* (inc) and the constant 0.

$$\overline{2}(inc)0 = (\lambda fx.f(f(x)))(inc)0 \overset{\beta}{=} (\lambda x.inc(inc(x)))0 \overset{\beta}{=} inc(inc(0)) = inc(1) = 2$$

Bringing everything together, we can express numerals in the following format:

$\overline{0} : \lambda fx.x$
$\overline{1} : \lambda fx.f(x)$
$\overline{2} : \lambda fx.f(f(x))$
$\overline{3} : \lambda fx.f(f(f(x)))$

$\vdots$

$\overline{n} : \lambda fx.f^n(x)$, where parentheses can be omitted.

Building on our previous example, the next equality holds true:

$$\overline{n}fx = (\overline{n}f)x \overset{\beta}{=} (\lambda x.f^n x)x \overset{\beta}{=} f^n x$$

How can we perform binary operations, such as addition, using Church numerals? The initial challenge arises from the fact that addition does not operate on a single argument, unlike the *increment* function which simply adds one. Binary operations, however, necessitate two arguments. In the context of Church numbers, currying is used in defining binary operations.

Keeping this concept in mind, we can construct a function that accepts one argument (the first Church numeral) and returns another function that accepts the next argument (the second Church numeral), ultimately returning the result. This function is represented as $((plus\ \overline{n})\ \overline{m})$. All we need is to define that *plus* function as a lambda expression.

Exercise 1. Define a lambda expression designed to perform addition on encoded numerals.

We express $\overline{n} = \lambda fx.f^n(x)$, and $\overline{m} = \lambda fx.f^m(x)$,
Their sum should be of the form, $\overline{n} + \overline{m} = \lambda fx.f^{n+m}(x)$.
Let's explore result further, using exponentiation rules (1) and equality $\overline{n}fx = f^n x$ (2)

$\lambda fx.f^{n+m}(x) \overset{(1)}{=} \lambda fx.f^n(f^m(x)) \overset{(2)}{=} \lambda fx.f^n(\overline{m}fx) \overset{(2)}{=} \lambda fx.\overline{n}f(\overline{m}fx)$
Therefore, the outcome should be represented by the lambda expression: $\lambda nmfx.nf(mfx)$

Let's check this solution on numerals $\overline{2}$ and $\overline{3}$:

$(\lambda nmfx.nf(mfx))\ \overline{2}\ \overline{3}$
$= (\lambda nmfx.nf(mfx))(\lambda fx.f^2x)(\lambda fx.f^3x),$ *associativity EFG=(EF)G*
$= ((\lambda nmfx.nf(mfx))(\lambda fx.f^2x))(\lambda fx.f^3x),$ *β-reduction on n*
$= (\lambda mfx.(\lambda fx.f^2x)f(mfx))(\lambda fx.f^3x),$ *β-reduction on m*
$= \lambda fx.(\lambda fx.f^2x)f((\lambda fx.f^3x)fx),$ *β-reduction on f*
$= \lambda fx.(\lambda fx.f^2x)f((\lambda x.f^3x)x),$ *β-reduction on x*
$= \lambda fx.(\lambda fx.f^2x)f(f^3x),$ *β-reduction on f*

$= \lambda fx.(\lambda x.f^2x)(f^3x),$ *β-reduction on x*
$= \lambda fx.f^2f^3x = \lambda fx.f^5x = \overline{5}$

Exercise 2. Define a lambda expression designed to perform logical NOT operation.

If we examine this lambda function, it should reverse $\top$ when received, as well as $\bot$. Our 'if-then-else' structure should be: *if M, then* $\bot$*, else* $\top$. By incorporating our previous analysis, we obtain: $M\bot\top = M(\lambda xy.y)(\lambda xy.y)$. Therefore, our lambda expression should be: $\lambda M.M(\lambda xy.y)(\lambda xy.x)$

Verification of solutions using two different methods:

$$(\lambda M.M(\lambda xy.y)(\lambda xy.x))\top \stackrel{\beta}{=} M(\lambda xy.y)(\lambda xy.x)[M := \top] = \top(\lambda xy.y)(\lambda xy.x) =$$

$$(\lambda xy.x)(\lambda xy.y)(\lambda xy.x)\top \stackrel{\alpha}{=} (\lambda xy.x)(\lambda zt.t)(\lambda pq.p) \stackrel{\beta}{=} (\lambda y.x)[x := (\lambda zt.t)](\lambda pq.p) =$$

$$(\lambda y.(\lambda zt.t))(\lambda pq.p) \stackrel{\beta}{=} (\lambda zt.t)[y := (\lambda pq.p)] \stackrel{(*)}{=} \lambda zt.t \stackrel{\alpha}{=} \lambda xy.y = \bot$$

* - In our lambda expression there is no y to reduce, so we can disregard it.

$$(\lambda M.M(\lambda xy.y)(\lambda xy.x))\bot \stackrel{\beta}{=} M(\lambda xy.y)(\lambda xy.x)[M := \bot] = \bot(\lambda xy.y)(\lambda xy.x) =$$

$$(\underbrace{\lambda xy.y}_{\text{function}})(\underbrace{\lambda xy.y}_{\text{param1}})(\underbrace{\lambda xy.x}_{\text{param2}}) \stackrel{(**)}{=} (\lambda xy.x) = \top$$

** - The function $(\lambda xy.x)$ takes two parameters and selects the first one. We will use this method in future exercises.

Exercise 3. Define a lambda expression designed to perform logical AND operation.

In the logical AND operation, if both inputs are true, the result is true. Applying the same principle, if the first input is true, we return the second input, as the result depends on it. However, if the first input is false, we don't need to evaluate the second input due to lazy evaluation, returning false immediately. Our 'if-then-else' structure should appear as follows: *if M, then N, else* $\bot$. Therefore, our lambda expression should be: $\lambda MN.MN(\lambda xy.y)$

Verification of solution:

$$(\lambda MN.MN(\lambda xy.y))\bot\top = \bot\top(\lambda xy.y) = (\lambda xy.y)(\lambda xy.x)(\lambda xy.y) = (\lambda xy.y) = \bot$$

Exercise 4. Define a lambda expression designed to perform logical OR operation.

The structure of the 'if-then-else' should be as follows: *if M, then* $\top$*, else N*. Therefore, our lambda expression should be: $\lambda MN.M(\lambda xy.x)N$, similar to the AND operation where if the first input is true, we return true, otherwise, we return the second, as the result depends on it.

Exercise 5. Define a lambda expression designed to perform multiplication on encoded numerals.

We express $\overline{n} = \lambda fx.f^n(x)$, and $\overline{m} = \lambda fx.f^m(x)$,
Their product should be of the form,
$\overline{n} \cdot \overline{m} = \lambda fx.f^{n \cdot m}(x) = \lambda fx.\underbrace{f^n(f^n(...(f^n(x))...))}_{m \text{ times}} = \lambda fx.(f^n)^m(x)$
$\overset{(*)}{=} \lambda fx.\overline{m}f^n x \overset{(**)}{=} \lambda fx.\overline{m}(\overline{n}f)x$

* - We know that $f^n x = \overline{n}fx$, and when we have $(f^n)^m(x)$, it becomes $\overline{m}f^n x$
** - We need to be careful here since when we apply following lambda expression to an argument, $(\lambda x.f^n(x))a$, it results in $f^n(a)$. This implies that $\lambda f^n(x) = \overline{n}fx$, while $\lambda x.f^n(x) = \overline{n}f$, demonstrating that these lambda expressions are not the same.

Combining all of the points discussed, the result can be represented by the lambda expression: $\lambda nmfx.m(nf)x$

Let's check this solution on numerals $\overline{2}$ and $\overline{3}$:

$((\lambda nmfx.m(nf)x)\overline{2})\overline{3} = \lambda fx.\overline{3}(\overline{2}f)x = \lambda fx.(\lambda fx.f^3x)((\lambda fx.f^2x)f)x =$
$\lambda fx.(\lambda fx.f^3x)(\lambda x.f^2x)x \overset{(*)}{=} \lambda fx.(\lambda x.(f^2)^3x)x = \lambda fx.f^6x = \overline{6}$

* - For a formal proof, we can analyze the lambda expression for a numeral, $\lambda fx.f^n x$, in the form $\lambda fx.f(f...(f(x))...)$ to obtain the same result:

$$\lambda fx.(\lambda fx.f^3x)(\lambda x.f^2x)x = \lambda fx.\Big(\lambda fx.f(f(f(x)))\Big)\Big(\lambda x.f(f(x))\Big)x =$$
$$\lambda fx.\Bigg(\lambda x.\Big(\lambda x.f(f(x))\Big)\Bigg(\Big(\lambda x.f(f(x))\Big)\Bigg(\Big(\lambda x.f(f(x))\Big)x\Bigg)\Bigg)\Bigg)x =$$
$$\lambda fx.\Bigg(\lambda x.\Big(\lambda x.f(f(x))\Big)\Bigg(\Big(\lambda x.f(f(x))\Big)\Big(f(f(x))\Big)\Bigg)\Bigg)x =$$
$$\lambda fx.\Bigg(\lambda x.\Big(\lambda x.f(f(x))\Big)\Big(f(f(f(f(x))))\Big)\Bigg)x =$$
$$\lambda fx.\Bigg(\lambda x.\Big(f(f(f(f(f(x))))\Big)\Bigg)x = \lambda fx.f(f(f(f(f(x))))) = \lambda fx.f^6(x)$$

Exercise 6. Define a lambda expression designed to perform the *sgn* operation on encoded numerals, returning false if the number is zero, and true otherwise.

If we examine the numeral $\overline{n} = \lambda fx.f^n x$, we observe that when $\overline{n} = 0$, according to the definition, f is not present in the expression. Additionally, for this exercise, we need to define a function that always returns true, which has the form $(\lambda x.\top)$.

Given the branching in relation to n, where n is a function with two arguments, if n equals zero, it takes on an equivalent form to logical value 'False', which selects the second input. With this in mind, we can construct a lambda expression for that case.

$\overline{n} = 0 \Rightarrow \overline{n}(something)\bot = (\lambda fx.x)(something)\bot = \bot$

When $\overline{n} > 0$, the *something* must be a function that always returns true upon evaluation. Upon examination, for example, of numeral 3 using the same format as in the previous exercise, we find:

$(\lambda fx.f(f(f(x))))(\lambda x.\top)\bot = (\lambda x.\top)((\lambda x.\top)((\lambda x.\top)x)) = (\lambda x.\top)((\lambda x.\top)\bot) = (\lambda x.\top)\top = \top$

Notice that the right lambda expression will always return $\top$, regardless of the number of evaluations. To sum up, the solution is $sgn(\overline{n}) = \overline{n}(\lambda x.\top)\bot$. If we want to define the 'ISZERO' lambda expression, the solution would be the reverse, like $\overline{n}(\lambda x.\bot)\top$.

Exercise 7. Define a lambda expression designed to create a pair of encoded numerals, formally represented as: $n, m \mapsto (n, m)$.

The pair (n, m) can be represented by a lambda expression using two functions f and g, symbolizing the symbolizing the pair creation operation, and producing their composition. It could be expressed as $\overline{(n, m)} := \lambda fgx.f^n(g^m(x))$.

Similarly to before, if we further examine the lambda expression of a pair, we obtain:

$$\lambda fgx.f^n(g^m(x)) = \lambda fgx.f^n(mgx) = \lambda fg.nf(mgx)$$

Therefore, the result expression is: $\lambda nmfgx.nf(mgx)$. It's noteworthy that the result yields a similar outcome to the addition operation, with the only difference being the inclusion of the g function within the parentheses.

For example:
$((\lambda nmfgx.nf(mgx)\overline{2})\overline{3} = \lambda fgx.\overline{2}f(\overline{3}gx) = \lambda fgx.\overline{2}f((\lambda fx.f^3x)gx) = \lambda fgx.\overline{2}f(g^3x) =$
$\lambda fgx.(\lambda fx.f^2x)f(g^3x) = \lambda fgx.f^2g^3x$

Exercise 8. Define a lambda expressions that, given a pair of encoded numerals, perform extraction of the first and second element.

For this exercise, we'll utilize the identity function $\lambda x.x$ to evaluate the unnecessary part of the pair so we can eliminate it. For instance, the lambda expression $(\lambda x.x)^n x$, upon evaluation, should result in x.

Given that our desired output is a lambda expression that accepts a pair p and returns a Church numeral, the structure should look like $\lambda pfx.p(placeholder_1)(placeholder_2)x$. Here, one placeholder would be replaced with the identity function if the corresponding variable of the pair should be removed, while the other would be replaced with f to preserve it.

So, our results are, $first : \lambda pfx.pf(\lambda x.x)x$ and $second : \lambda pfx.p(\lambda x.x)fx$. From the last exercise, we saw that $\overline{(2,3)} := \lambda fgx.f^2(g^3(x))$, and we will use that expression to verify if our results are correct.

$(\lambda pfx.pf(\lambda x.x)x)\overline{(2,3)} = (\lambda pfx.pf(\lambda x.x)x)(\lambda fgx.f^2g^3x) = \lambda fx.(\lambda fgx.f^2g^3x)f(\lambda x.x)x =$

$\lambda fx.\big(\lambda gx.f^2g(g(g(x)))\big)(\lambda x.x)x = \lambda fx.\Big(\lambda x.f^2\Big((\lambda x.x)\Big((\lambda x.x)\big((\lambda x.x)x\big)\Big)\Big)\Big)x =$

$\lambda fx.\Big(\lambda x.f^2\Big((\lambda x.x)\big((\lambda x.x)x\big)\Big)\Big)x = \lambda fx.\Big(\lambda x.f^2\Big((\lambda x.x)x\Big)\Big)x = \lambda fx.\Big(\lambda x.f^2x\Big)x =$

$\lambda fx.f^2x = \overline{2}$

$(\lambda pfx.p(\lambda x.x)fx)\overline{(2,3)} = (\lambda pfx.p(\lambda x.x)fx)(\lambda fgx.f^2g^3x) = \lambda fx.(\lambda fgx.f^2g^3x)(\lambda x.x)fx =$

$\lambda fx.\Big(\lambda fgx.f\big(f(g^3x)\big)\Big)(\lambda x.x)fx = \lambda fx.\Big(\lambda gx.(\lambda x.x)\big((\lambda x.x)(g^3x)\big)\Big)fx =$

$\lambda fx.\Big(\lambda gx.(\lambda x.x)\big(g^3x\big)\Big)fx = \lambda fx.\big(\lambda gx.g^3x\big)fx = \lambda fx.f^3x = \overline{3}$

Exercise 9. Define a lambda expression that for a given encoded numeral returns it's predecessor.

The idea here is to generate a pair $(n, n-1)$ and subsequently select the second element of this pair as the result. To accomplish this, we define an operation G and iterate it n times, after which we select the second element of the resulting pair.

$$(0,0) \xrightarrow{G} (1,0) \xrightarrow{G} (2,1) \xrightarrow{G} (3,2) \xrightarrow{G} (4,3) \xrightarrow{G} ...$$

Additionally, we require a successor function, defined by the lambda expression (similar to the procedure in Exercise 1 for addition), which takes the form $\lambda nfx.f(nfx)$.

$(\lambda nfx.f(nfx))\overline{n} = \lambda fx.f(\overline{n}fx) = \lambda fx.f((\lambda fx.f^nx)fx) = \lambda fx.f(f^nx) = \lambda fx.f^{n+1}x = \overline{n+1}$

We can define G as a lambda expression that accepts a pair p, retrieves the first element of the given pair, and constructs a new pair where the first element is that retrieved element incremented by one, while the second element is that retrieved element unchanged.

Let's initially confirm that $G = pair(succ(first(p)), first(p))$. If we have an encoded pair $p := \overline{(n, n-1)} = \lambda fgx.f^ng^{n-1}x$, it should return $p' := \overline{(n+1, n)} = \lambda fgx.f^{n+1}g^nx$

$$\begin{aligned}
p &= \lambda fgx.f^ng^{n-1}x, \\
first(p) &= (\lambda pfx.pf(\lambda x.x)x)(\lambda fgx.f^ng^{n-1}x) = \\
&\quad \lambda fx.(\lambda fgx.f^ng^{n-1}x)f(\lambda x.x)x = \\
&\quad \lambda fx.f^n(\lambda x.x)^{n-1}x = \\
&\quad \lambda fx.f^nx = \overline{n}, \\
succ(first(p)) &= (\lambda nfx.f(nfx))first(p) = \\
&\quad (\lambda nfx.f(nfx))(\lambda fx.f^nx) = \\
&\quad \lambda fx.f((\lambda fx.f^nx)fx) =
\end{aligned}$$

$$\begin{aligned}
& \lambda fx.f(f^n x) = \\
& \lambda fx.f^{n+1}x = \overline{n+1}, \\
pair(succ(first(p)), first(p)) = pair(\lambda fx.f^{n+1}x, \lambda fx.f^n x) = \\
& (\lambda nmfgx.nf(mgx))(\lambda fx.f^{n+1}x)(\lambda fx.f^n x) = \\
& \lambda fgx.(\lambda fx.f^{n+1}x)f((\lambda fx.f^n x)gx) = \\
& \lambda fgx.(\lambda fx.f^{n+1}x)f(g^n x) = \\
& \lambda fgx.f^{n+1}g^n x = \overline{(n+1,n)}
\end{aligned}$$

From this point forward, we'll concentrate on the concept, as the evaluation of the formula for G is extensive, as demonstrated here:

$$\lambda p.\ \underbrace{\Big(\lambda nmfgx.nf(mgx)\Big)}_{\text{pair}}\ \Big(\ \underbrace{\Big(\lambda nfx.f(nfx)\Big)}_{\text{succ}}\ \Big(\ \underbrace{\Big(\lambda pfx.pf(\lambda x.x)x\Big)}_{\text{first}}\ p\Big)\Big)\Big(\ \underbrace{\Big(\lambda pfx.pf(\lambda x.x)x\Big)}_{\text{first}}\ p\Big)$$

With this in mind, our objective is to devise a function that applies G to the encoded numeral for zero n times, eventually returning the second element. The formula for this is:

$$second(G^n(0,0)) = second(nG(0,0)) = second(nG(\lambda fgx.x)) = (\lambda pfx.p(\lambda x.x)fx)(nG(\lambda fgx.x))$$

The solution to this exercise can be expressed as follows: $\lambda n.(\lambda pfx.p(\lambda x.x)fx)(nG(\lambda fgx.x))$. Solution will be validated using numeral zero, where the result will be zero, and numeral three, where the result will be numeral two. This operation is denoted as δ which returns 0 when $n = 0$, and $n - 1$ when $n > 0$.

$$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)(nG(\lambda fgx.x))\Big)\overline{0} = (\lambda pfx.p(\lambda x.x)fx)((\lambda fx.x)G(\lambda fgx.x)) =$$
$$(\lambda pfx.p(\lambda x.x)fx)(\lambda fgx.x) = \lambda fx.(\lambda fgx.x)(\lambda x.x)fx = \lambda fx.x = \overline{0}$$

$$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)(nG(\lambda fgx.x))\Big)\overline{3} = (\lambda pfx.p(\lambda x.x)fx)((\lambda fx.f^3x)G(\lambda fgx.x)) =$$
$$(\lambda pfx.p(\lambda x.x)fx)(G^3(\lambda fgx.x)) = (\lambda pfx.p(\lambda x.x)fx)\Big(G(G(G(\lambda fgx.x)))\Big) =$$
$$(\lambda pfx.p(\lambda x.x)fx)\Big(G(G(\lambda fgx.fx))\Big) = ... = (\lambda pfx.p(\lambda x.x)fx)(\lambda fgx.f^3g^2x) =$$
$$\lambda fx.(\lambda fgx.f^3g^2x)(\lambda x.x)fx = \lambda fx.(\lambda x.x)^2f^2x = \lambda fx.f^2x = \overline{2}$$

Exercise 10. Define a lambda expression designed to perform monus operation (subtraction defined on neutral numbers) on encoded numerals.

The idea here is that if we want to subtract m from n, we can apply m times the δ function to n. Therefore, the subtraction equals $n - m = \delta^m(n) = m\delta n$, where n and m are encoded numerals. Thus, the result is $\lambda nm.m\delta n$.

$\overline{3} - \overline{2} = ((\lambda nm.m\delta n)\overline{3})\overline{2} = \overline{2}\delta\overline{3} = (\lambda fx.f^2x)\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)(\lambda fx.f^3x) =$

$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)\Big(\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)(\lambda fx.f^3x)\Big) =$

$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)\Big((\lambda pfx.p(\lambda x.x)fx)\Big((\lambda fx.f^3x)G\big(\lambda fgx.x\big)\Big)\Big) =$

$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)\Big((\lambda pfx.p(\lambda x.x)fx)\big(G^3(\lambda fgx.x)\big)\Big) \overset{(ex.9)}{=}$

$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)\Big((\lambda pfx.p(\lambda x.x)fx)(\lambda fgx.f^3g^2x)\Big) =$

$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)\Big(\lambda fx.(\lambda fgx.f^3g^2x)(\lambda x.x)fx\Big) =$

$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)\Big(\lambda fx.(\lambda x.x)^3f^2x\Big) =$

$\Big(\lambda n.(\lambda pfx.p(\lambda x.x)fx)\big(nG(\lambda fgx.x)\big)\Big)(\lambda fx.f^2x) =$

$(\lambda pfx.p(\lambda x.x)fx)\Big((\lambda fx.f^2x)G(\lambda fgx.x)\Big) = (\lambda pfx.p(\lambda x.x)fx)(G^2(\lambda fgx.x)) \overset{(ex.9)}{=}$

$(\lambda pfx.p(\lambda x.x)fx)(\lambda fgx.f^2gx) = \lambda fx.(\lambda fgx.f^2gx)(\lambda x.x)fx = \lambda fx.(\lambda x.x)^2fx = \lambda fx.fx = \overline{1}$

Now that we have defined addition and monus operations, we can utilize the following identity: $|n - m| = (n - m) + (m - n)$. This allows us to define the absolute value operation.

Exercise 11. Define a lambda expression that, when given two encoded numerals, outputs the maximum of the two.

Upon examination, we can notice that this problem involves choosing between two alternatives: if $n > m$, then select n, otherwise select m. Therefore, our 'if-then-else' condition should appear as follows: $if(n > m),\ then\ n,\ else\ m$.

However, since we haven't defined the relational operator $>$, we can utilize the δ operator to determine whether the result of subtraction is greater than or less than zero. Our condition would appear as follows: $if((n - m) > 0),\ then\ n,\ else\ m$. Yet again, we can circumvent the operator $>$ by using the *sgn* operation defined in Exercise 6. This operation returns false if the number is zero and true otherwise.

Ultimately, our 'if-then-else' condition would look like: $if(sgn(n-m)),\ then\ n,\ else\ m$, and our resulting lambda expression should look like $\lambda nm.sgn(n - m)nm$.

$max(\overline{3}, \overline{2}) = ((\lambda nm.sgn(n - m)nm)\overline{3})\overline{2} = sgn(\overline{3} - \overline{2})\overline{3}\ \overline{2} \overset{(ex.10)}{=} sgn(\overline{1})\overline{3}\ \overline{2} \overset{(ex.6)}{=}$
$(\overline{1}(\lambda x.\top)\bot)\overline{3}\ \overline{2} = ((\lambda fx.fx)(\lambda x.\top)\bot)\overline{3}\ \overline{2} = ((\lambda x.\top)\bot)\overline{3}\ \overline{2} = \top\ \overline{3}\ \overline{2} = (\lambda xy.x)\overline{3}\ \overline{2} = \overline{3}$

Similarly, we can express the minimum function in the same manner.