The Austalian National University Semester 1, 2024
Research School of Computer Science
Dr. Ranald Clouston

COMP1130: Lambda Calculus - Worksheet 3

- **Harder Beta Reduction**

  Reduce the following terms to normal form (apply as many beta reductions as possible). If you are comfortable with alpha conversions you do not need to write them out now.

  1. $(\lambda x.xx)(\lambda y.y)$
     Solution.
     $$(\lambda x.xx)(\lambda y.y) \underset{\beta}{\rightarrow} (\lambda y.y)(\lambda y.y) \underset{\beta}{\rightarrow} \lambda y.y$$
  2. $(\lambda x.\lambda y.yxy)(xx)z$
     Solution.
     $$(\lambda x.\lambda y.yxy)(xx)z \underset{\beta}{\rightarrow} (\lambda y.y(xx)y)z \underset{\beta}{\rightarrow} z(xx)z$$
  3. $(\lambda x.\lambda y.x)(\lambda x.\lambda y.y)w$
     Solution.
     $$(\lambda x.\lambda y.x)(\lambda x.\lambda y.y)w \underset{\beta}{\rightarrow} (\lambda y.(\lambda x.\lambda z.z))w \underset{\beta}{\rightarrow} \lambda x.\lambda z.z$$
  4. $(\lambda x.(\lambda x.x)(\lambda z.zy)x)(\lambda z.zw)$
     Solution.
     $$\begin{aligned} &(\lambda x.(\lambda x.x)(\lambda z.zy)x)(\lambda z.zw) \\ \underset{\beta}{\rightarrow}&(\lambda x.x)(\lambda z.zy)(\lambda z.zw) \\ \underset{\beta}{\rightarrow}&(\lambda z.zy)(\lambda z.zw) \\ \underset{\beta}{\rightarrow}&(\lambda z.zw)y \\ \underset{\beta}{\rightarrow}&yw \end{aligned}$$
  5. $(\lambda x.xx)(\lambda y.\lambda z.y)$
     Solution.
     $$(\lambda x.xx)(\lambda y.\lambda z.y) \underset{\beta}{\rightarrow} (\lambda y.\lambda z.y)(\lambda y.\lambda z.y) \underset{\beta}{\rightarrow} \lambda x.\lambda y.\lambda z.y$$
  6. $(\lambda x.xxx)(\lambda y.yw)$
     Solution.
     $$\begin{aligned} &(\lambda x.xxx)(\lambda y.yw) \\ \underset{\beta}{\rightarrow}&(\lambda y.yw)(\lambda y.yw)(\lambda y.yw) \\ \underset{\beta}{\rightarrow}&(\lambda y.yw)w(\lambda y.yw) \\ \underset{\beta}{\rightarrow}&ww(\lambda y.yw) \end{aligned}$$

- **Encoding Pairs**

  Recall that

$$\begin{aligned}(a, b) &:= \lambda x.\texttt{if}\ \ x\ \ \texttt{then}\ \ a\ \ \texttt{else}\ \ b \\ \texttt{fst} &= \lambda p.p\ \ \texttt{true} \\ \texttt{snd} &= \lambda p.p\ \ \texttt{false}\end{aligned}$$

1. Find a lambda expression for the function `pairmap`, that applies a function to each element of a pair, that is
$$\texttt{pairmap}\, f\ (a, b) \underset{\beta}{\rightarrow} (fa, fb)$$
Solution. The solution to this is to write a function that extracts out each part of the pair, applies f to it, and then packs the two parts back together in a pair
$$\texttt{pairmap}\ (a, b) = (fa, fb)$$
Replacing (a, b) with a generic variable p, we would then need to use `fst` and `snd` to extract each part out.
$$\texttt{pairmap}\, p = (f(\texttt{fst}\, p), f(\texttt{snd}\, p))$$
We can now use abstraction to move the p over,
$$\texttt{pairmap} = \lambda p.(f(\texttt{fst}\, p), f(\texttt{snd}\, p))$$
and then replace each function we used to build `pairmap` with it's definition
$$\begin{aligned}\texttt{pairmap} &= \lambda p.(f((\lambda q.q\ \ \texttt{true})p), f((\lambda q.q\ \ \texttt{false})p)) \\ &\underset{\beta}{\rightarrow} \lambda p.(f(p\ \ \texttt{true}), f(p\ \ \texttt{false})) \\ &= \lambda p.(f(p(\lambda x.\lambda y.x)), f(p(\lambda x.\lambda y.y))) \\ &= \lambda p.\lambda x.\texttt{if}\ \ x\ \ \texttt{then}\ \ f(p(\lambda x.\lambda y.x))\ \ \texttt{else}\ \ f(p(\lambda x.\lambda y.y)) \\ &= \lambda p.\lambda x.x(f(p(\lambda x.\lambda y.x)))(f(p(\lambda x.\lambda y.y)))\end{aligned}$$

2. Find a lambda expression for the function `swap`, that swaps the elements of a pair, that is,
$$\texttt{swap}\ (a, b) \underset{\beta}{\rightarrow} (b, a)$$
Solution. We can use the same construction as before, but swap the position of `fst` and `snd`, and remove the application of the function f.
After doing so, we obtain as the final result
$$\texttt{swap} = \lambda p.\lambda x.x(p\lambda x.\lambda y.y)(p\lambda x.\lambda y.x)$$

3. What could be a sensible definition of a 3-tuple (a, b, c)?
Solution. One option is to nest the pairs that already exist.
$$(a, b, c) := ((a, b), c)$$

4. Based on your definition above, try to define the functions `fst3, snd3, trd3` such that
$$\begin{aligned}&\texttt{fst3}(a, b, c) \underset{\beta}{\rightarrow} a \\ &\texttt{snd3}(a, b, c) \underset{\beta}{\rightarrow} b \\ &\texttt{trd3}(a, b, c) \underset{\beta}{\rightarrow} c\end{aligned}$$

Solution. Since my 3-tuples are just made out of pairs underneath, it's clear how to define these using `fst` and `snd`.

$$\begin{aligned}\texttt{fst3} &= \lambda p.(\texttt{fst}(\texttt{fst}\ p))\\ \texttt{snd3} &= \lambda p.(\texttt{snd}(\texttt{fst}\ p))\\ \texttt{trd3} &= \texttt{snd}\end{aligned}$$

5. Define a function `rot3` such that $\texttt{rot3}(a, b, c) = (b, c, a)$.
Solution. We can use the above definitions to do this.

$$\texttt{rot3} = \lambda p.(\texttt{snd3}\ p, \texttt{trd3}\ p, \texttt{fst3}\ p)$$

- **Encoding Numerals (Church)**

Recall from lectures that the Church encoding of natural numbers is

$$\begin{aligned}\texttt{0} :=& \lambda f.\lambda x.x\\ \texttt{1} :=& \lambda f.\lambda x.fx\\ \texttt{2} :=& \lambda f.\lambda x.f(fx)\\ \dots&\\ \texttt{n} :=& \lambda f.\lambda x.f^n(x)\\ \texttt{succ} :=& \lambda n.\lambda f.\lambda x.f(nfx)\\ \texttt{iszero} :=& \lambda z.z(\lambda y.\texttt{false})\texttt{true}\end{aligned}$$

where $f^n(x)$ is shorthand for the function f applied to x, n times over.

1. Verify that $\texttt{iszero}\ \texttt{0} \underset{\beta}{\rightarrow} \texttt{true}$ and $\texttt{iszero}\ \texttt{1} \underset{\beta}{\rightarrow} \texttt{false}$

Solution.

$$\begin{aligned}\texttt{iszero}\ \texttt{0} &= (\lambda z.z(\lambda y.\texttt{false})\texttt{true})(\lambda f.\lambda x.x)\\ &\underset{\beta}{\rightarrow} (\lambda f.\lambda x.x)(\lambda y.\texttt{false})\texttt{true}\\ &\underset{\beta}{\rightarrow} (\lambda x.x)\texttt{true}\\ &\underset{\beta}{\rightarrow} \texttt{true}\end{aligned}$$

$$\begin{aligned}\texttt{iszero}\ \texttt{1} &= (\lambda z.z(\lambda y.\texttt{false})\texttt{true})(\lambda f.\lambda x.fx)\\ &\underset{\beta}{\rightarrow} (\lambda f.\lambda x.fx)(\lambda y.\texttt{false})\texttt{true}\\ &\underset{\beta}{\rightarrow} (\lambda x.(\lambda y.\texttt{false})x)\texttt{true}\\ &\underset{\beta}{\rightarrow} (\lambda y.\texttt{false})\texttt{true}\\ &\underset{\beta}{\rightarrow} \texttt{false}\end{aligned}$$

2. Verify that $\texttt{succ}\ \texttt{0} \underset{\beta}{\rightarrow} \texttt{1}$ and $\texttt{succ}\ \texttt{1} \underset{\beta}{\rightarrow} \texttt{2}$.

Solution.

$$\begin{aligned}\texttt{succ}\ \texttt{0} &= (\lambda n.\lambda f.\lambda x.f(nfx))(\lambda f.\lambda x.x)\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.f((\lambda f.\lambda x.x)fx)\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.f((\lambda x.x)x)\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.fx = \texttt{1}\end{aligned}$$

$$\begin{aligned}\texttt{succ 1} &= (\lambda n.\lambda f.\lambda x.f(nfx))(\lambda f.\lambda x.fx)\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.f((\lambda f.\lambda x.fx)fx)\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.f((\lambda x.fx)x)\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.f(fx) = \texttt{2}\end{aligned}$$

3. Suppose you were given a function[1] `pred` that satisfied the property that

$$\texttt{pred n} = \begin{cases}\texttt{n-1} & \texttt{n} \underset{\alpha}{\neq} \texttt{0}\\ \texttt{0} & \texttt{n} \underset{\alpha}{=} \texttt{0}\end{cases}$$

for any natural number `n`.
Using `pred` and previously defined functions, try to define the `isOne` function satisfying

$$\texttt{isOne n} = \begin{cases}\texttt{true} & \texttt{n} \underset{\alpha}{=} \texttt{1}\\ \texttt{false} & \texttt{n} \underset{\alpha}{\neq} \texttt{1}\end{cases}$$

Solution. The easiest way to do this is to first check if the number is zero, and return false if so. If the number is not so, decrement, and then return the result obtained when checking if the new number is zero. Formally,

$$\texttt{isOne} = \lambda n.\texttt{if (iszero } n\texttt{) then false else iszero(pred n)}$$

4. Check that `succ` actually does what we claim, that `succ n = n+1`. (Hint: Try to evaluate `succ` given an arbitrary Church numeral `n` as input, and use the fact that $f(f^n(x)) = f^{n+1}(x)$.)
Solution.

$$\begin{aligned}\texttt{succ n} &= (\lambda n.\lambda f.\lambda x.f(nfx))\\ &\underset{\beta}{\rightarrow} (\lambda n.\lambda f.\lambda x.f(nfx))(\lambda f.\lambda x.f^n(x))\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.f((\lambda f.\lambda x.f^n(x))fx)\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.f((\lambda x.f^n(x))x)\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.f(f^n(x))\\ &\underset{\beta}{\rightarrow} \lambda f.\lambda x.(f^{n+1}(x)) = \texttt{n+1}\end{aligned}$$

5. Note that our definition of numbers is a function that takes two arguments, and applies the first to the second, many times over. With that in mind, let `foo` $:= \lambda n.$`2 succ` n. What do you think the `foo` function does when given Church numerals as input?
Solution. We can expand out the definition of `foo`,

$$\begin{aligned}\texttt{foo} &:= \lambda n.\texttt{2 succ } n\\ &= \lambda n.(\lambda f.\lambda x.f(fx))\texttt{succ } n\\ &\underset{\beta}{\rightarrow} \lambda n.(\lambda x.\texttt{succ (succ } x))n\\ &\underset{\beta}{\rightarrow} \lambda n.\texttt{succ (succ } n) = \texttt{n+2}\end{aligned}$$

It looks like `foo` takes a natural number and adds 2 to it.

[1]You can try to derive the definition of `pred` yourself, but it's very difficult!

6. Using the previous exercise as a starting point, try to define the function `plus` that satisfies the property `plus n m = n+m` for any Church numerals `n,m`.

   Solution. We can generalise the previous example by replacing `2` with one of the inputs to the function.

$$\texttt{plus} = \lambda n.\lambda m.n\ \texttt{succ}\ m$$

7. Define `bar` := $\lambda n.n$ `(plus 2) 0`. What do you think that `bar` does when given Church numerals as input?

   Solution. We already know that natural numbers act to apply a function many times to an argument. Consider the case where `bar` is given an input of `3`. (We can also use the known properties of `plus` to ease evaluation.) Then,

$$\begin{aligned}
&(\lambda n.n\ (\texttt{plus 2})\ \texttt{0})\ \texttt{3}\\
&\xrightarrow[\beta]{} (\lambda f.\lambda x.f(f(f(x))))\ (\texttt{plus 2})\ \texttt{0}\\
&\xrightarrow[\beta]{} (\lambda x.(\texttt{plus 2})((\texttt{plus 2})((\texttt{plus 2})x)))\ \texttt{0}\\
&\xrightarrow[\beta]{} (\texttt{plus 2})((\texttt{plus 2})((\texttt{plus 2})\texttt{0}))\\
&\xrightarrow[\beta]{} (\texttt{plus 2})((\texttt{plus 2})(\texttt{0+2}))\\
&\xrightarrow[\beta]{} (\texttt{plus 2})((\texttt{plus 2})(\texttt{2}))\\
&\xrightarrow[\beta]{} (\texttt{plus 2})(\texttt{2+2})\\
&\xrightarrow[\beta]{} (\texttt{plus 2})\texttt{4}\\
&\xrightarrow[\beta]{} \texttt{4+2} = \texttt{6}
\end{aligned}$$

   It looks like `bar` takes a natural number and doubles it.

8. Using the previous exercise as a starting point, try to define the function `mul` that satisfies the property `mul n m = n*m` for any Church numerals `n,m`.

   Solution. By modifying the above example function `bar`, we can replace the `2` with another input variable, and obtain

$$\texttt{mul} = \lambda n.\lambda m.n\ (\texttt{plus}\ m)\texttt{0}$$

9. Try and define a function `pow` with the property that

$$\texttt{pow n m} = \texttt{n}^{\texttt{m}}$$

   for any natural numbers `n,m`.

   Solution. The construction is similar to building `mul` from `plus`, by noting that $\texttt{n}^{\texttt{m}}$ is the same as `n` multiplied by itself, `m` times over. Also note that `1` is the multiplicative identity, so that will be our base case to start from.

$$\texttt{pow} = \lambda n.\lambda m.m\ (\texttt{mul}\ n)\ \texttt{1}$$

- **Writing functions with a Fixed Point Combinator**

  To write functions like the factorial function, we write an intermediate function that has the body of the factorial function, but it takes the function to call again as an argument

$$F := \lambda f.\lambda n.\ \texttt{if (isZero}\ n\texttt{) then 1 else (mul}\ n\ \texttt{(}f\ \texttt{(pred}\ n\texttt{)))}$$

  We then use any fixed point combinator (for example, Turing's fixed point combinator Θ) to give us the recursion, as it will call F over and over again.

$$\texttt{fac} = \Theta F$$

1. Verify that `fac 3` $= 6$. (You may assume that Θ, `isZero`, `mul`, `pred` ect. do what they should do, rather than evaluating the whole thing tediously by hand.)
   Solution.

$$\begin{aligned}
&\texttt{fac 3}\\
=&\Theta F \texttt{3}\\
\xrightarrow[\beta]{}&F(\Theta F)\texttt{3}\\
=&(\lambda f.\lambda n.\ \texttt{if (iszero } n\texttt{) then 1 else (mul } n\ (f\ \texttt{(pred } n\texttt{)))}(\Theta F)\texttt{3}\\
\xrightarrow[\beta]{}&\ \texttt{if (iszero 3) then 1 else (mul 3 (}\Theta F\ \texttt{(pred 3)))}\\
\xrightarrow[\beta]{}&\ \texttt{if (false) then 1 else (mul 3 (}\Theta F\ \texttt{(pred 3)))}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (}\Theta F\ \texttt{2)}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (}F(\Theta F)\ \texttt{2)}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (if (iszero 2) then 1 else (mul 2 (}\Theta F\ \texttt{(pred 2)))}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (mul 2 (}\Theta F\ \texttt{(pred 2)))}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (mul 2 (mul 1 (}\Theta F\ \texttt{0)))}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (mul 2 (mul 1 (}F(\Theta F)\ \texttt{0)))}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (mul 2 (mul 1 (if (iszero 0) then 1 else (mul 3 (}\Theta F\ \texttt{(pred 0))))))}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (mul 2 (mul 1 (if (true) then 1 else (mul 3 (}\Theta F\ \texttt{(pred 0))))))}\\
\xrightarrow[\beta]{}&\texttt{mul 3 (mul 2 (mul 1 1))}\ \xrightarrow[\beta]{}\texttt{6}
\end{aligned}$$

With the help of previously defined functions (like `succ`, `pred`, `isZero`), try to write the following functions with help of Θ.
(Hint: It might help to write the Haskell definition first!)

1. Define `sum` with the property that for all natural numbers `n` representing values of n, that

$$\texttt{sum n} = 1 + 2 + \ldots + n$$

   Solution.
   We first write it in Haskell, to identify the body of the code.

```
sum :: Int -> Int
sum n
    |n == 0 = 0
    |otherwise = n + sum (n-1)
```

   This indicates we should use

$$G = \lambda f.\lambda n.\texttt{if (iszero } n\texttt{) then 0 else (plus n (}f\ \texttt{(pred n)))}$$

   and then define `sum` using the Θ combinator, as before

$$\texttt{sum} = \Theta G$$

2. Define `geq` with the property that

$$\texttt{geq n m} = \begin{cases} \texttt{true} & n \geq m \\ \texttt{false} & \text{otherwise} \end{cases}$$

for all natural numbers `n,m` representing values n, m respectively. (Note that this is given in lecture slides; you could try to do it without consulting the slides. Can you give a more efficient solution?)

Solution.

```
geq :: Int -> Int -> Int
geq n m
    |n == 0 && m == 0 = True
    |n == 0 = False
    |m == 0 = True
    |otherwise = geq (n-1) (m-1)
```

$$\begin{aligned} G = \lambda f.\lambda n.\lambda m.&\texttt{if (and (iszero } n\texttt{) (iszero } m\texttt{)) then true else(} \\ &\texttt{if (iszero } n\texttt{) then false else(} \\ &\texttt{if (iszero } m\texttt{) then true else(} \\ &f\ \texttt{(pred n) (pred m))))} \end{aligned}$$

$$\texttt{geq} = \Theta G$$

3. Define `strgeq` with the property that

$$\texttt{strgeq n m} = \begin{cases} \texttt{true} & n > m \\ \texttt{false} & \text{otherwise} \end{cases}$$

for all natural numbers `n,m` representing values n, m respectively.
(Hint: you may be able to use functions you have written previously here, along with the equals function from the lecture.)
Solution. Note that

$$(n > m) \equiv (n \geq m) \wedge (n \neq m)$$

Hence,

$$\texttt{strgeq} = \lambda n.\lambda m.\texttt{and (geq } n\ m\texttt{) (not (eq } n\ m\texttt{))}$$

4. Define `sub` to compute saturated subtraction, that is, for any natural numbers `n,m` representing values n, m we have that

$$\texttt{sub n m} = \begin{cases} 0 & n < m \\ n - m & n \geq m \end{cases}$$

Solution.
Notice that $n - m = (\texttt{pred}\ n) - (\texttt{pred}\ m)$, where so we can use this to define `sub` in terms of `pred`.

```
sub :: Int -> Int -> Int
sub n m
    |m == 0 = n
    |otherwise = sub (pred n) (pred n)
```

and convert, as before,

$$G = \lambda f.\lambda n.\lambda m.\texttt{if (iszero } m\texttt{) then n else(} f \texttt{ (pred } n\texttt{) (pred } m\texttt{))}$$

5. (Tricky) Define `quotRem` to compute integer division with remainder [2]. Formally, it should have the property that `quotRem` returns a pair of natural numbers

$$\texttt{quotRem n d} = (\texttt{q}, \texttt{r})$$

satisfying the property that $qd + r = n$.
(Hint: You might need to write a "helper" function with an extra parameter to keep track of the state of q during computation, as you would in Haskell.)
Solution.
As the hint suggests, we need a helper function to keep track of the current quotient q. We implement division via repeated subtraction, and stop when the dividend n is strictly smaller than the divisor d. We return a tuple of both the quotient q and the remainder r.

```
quotRem :: Int -> Int -> (Int,Int)
quotRem n d = helper 0 n d
    where
    helper :: Int -> Int -> Int -> (Int,Int)
    helper q n d
        |d > n = (q,n)
        |otherwise = helper (q+1) (n-d) d
```

We define

$$\texttt{helper} := \lambda f.\lambda q.\lambda n.\lambda d \texttt{ if (strgeq } d\ n\texttt{) then (}q\texttt{, } n\texttt{) else (}f \texttt{ (succ } q\texttt{) (sub } n\ d\texttt{) } d\texttt{)}$$

and then use the Θ combinator to give us the recursion, initializing the value of q to zero.

$$\texttt{quotRem} := \lambda n.\lambda d(\Theta\ \texttt{helper}\ 0\ n\ d)$$

[2]The name `quotRem` comes from "quotient" and "remainder". The function is also defined already in the Haskell prelude.