Computation Theory
Solution Notes for Example Sheet 3

In this document you will find some solution notes for the problems of Example Sheet 3 of Computation Theory. If you find any mistake or any typos, please do let me know. Also, I am happy to hear (and include them in the notes (with credit) if you want) about alternative solutions to the problems or variations of a problem that you came up with.

Lecture 7

Exercise 1

(a) Define $\texttt{proj}_i^n$, $\texttt{succ}$ and $\texttt{zero}$.

(b) Show that all of these are RM computable.

(a) The projection function $\texttt{proj}_i^n : \mathbb{N}^n \to \mathbb{N}$ (for $n \in \mathbb{N}$) is defined as $\texttt{proj}_i^n(x_1, \ldots, x_n) \triangleq x_i$.
The successor function $\texttt{succ} : \mathbb{N} \to \mathbb{N}$ is defined as $\texttt{succ}(x) \triangleq x + 1$.
The zero function $\texttt{zero}^n : \mathbb{N}^n \to \mathbb{N}$ (for $n \in \mathbb{N}$) is defined as $\texttt{zero}(x_1, \ldots, x_n) \triangleq 0$.

See **Lecture 7 slide 12**.

(b) See Example Sheet 1.

Exercise 2

(a) What is *Kleene equivalence* of two expressions?

(b) Define *composition* of multi-dimensional functions.

(c) Show that composition of RM computable functions is RM computable.

(a) Kleene equivalence $A \equiv B$ of two possible undefined expressions A, B, means that either both A and B are undefined or they are both defined and equal.

See **Lecture 7 slide 14**.

(b) The composition of functions $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g_1, \ldots, g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$, if the partial function $h \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in N$,

$$h(x_1, \ldots, x_m) \equiv f(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$$

Usually, h is denoted by $f \circ [g_1, \ldots, g_n]$.

See **Lecture 7 slide 16**.

(c) The idea is to compute $y_1 = g_1(x_1, \ldots, x_m)$, $y_2 = g_1(x_1, \ldots, x_m)$ and so on, and then compute $\texttt{R}_0 = f(x_1, \ldots, x_n)$. We need to be careful to erase the contents of the registers used in the computation of g (say $\texttt{R}_0, \ldots \texttt{R}_n$).

See **Lecture 7 slide 18**.

Lecture 8

Exercise 3

(a) Define *primitive recursion* (See **[2017P6Q4 (a)]**, **[2014P6Q4 (a)]**, **[1999P4Q1 (a)]**).

(b) Define *primitive recursive functions* PRIM (See **[2017P6Q4 (b)(i)]**, **[2014P6Q4 (b)]**, **[2011P6Q4 (a)]**, **[2006P4Q9 (a)]**, **[1995P4Q9 (a)]**).

(c) Prove that PRIM functions are total (See **[2006P4Q9 (b)]**). Deduce that there exist computable functions that are not PRIM.

(d) Are all total functions primitive recursive? (See **[2017P6Q4 (b)(iii)]**)

(e) Show that the functions add, pred, mult, tsub, exp are primitive recursive (See **[2014P6Q4 (c)]**, [**2006P4Q9 (c)**]).

(f) Show that the following functions are primitive recursive:

i.
$$\mathrm{Eq}_0(x, y, z) = \begin{cases} y & \text{if } x = 0 \\ z & \text{otherwise.} \end{cases}$$

ii. The bounded summation function for $g : \mathbb{N}^{n+1} \to \mathbb{N}$ and $f : \mathbb{N}^{n+1} \to \mathbb{N}$,

$$g(\vec{x}, x) = \begin{cases} 0 & \text{if } x = 0 \\ f(\vec{x}, 0) & \text{if } x = 1 \\ f(\vec{x}, 0) + \ldots + f(\vec{x}, x-1) & \text{if } x > 1 \end{cases}$$

(g) Show that the functions $\mathrm{square}(x) = x^2$ and $\mathrm{fact}(x) = x!$ are primitive recursive functions (See [**2011P6Q4 (c)**])

(a) Given functions $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g \in \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$, the primitive recursion $\rho^n(f, g) \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ of f and g, is defined as

$$(\rho^n(f, g))(\vec{x}, 0) \equiv f(\vec{x})$$
$$(\rho^n(f, g))(\vec{x}, x + 1) \equiv g(\vec{x}, x, h(\vec{x}, x))$$

where $\equiv$ is Kleene equivalence: either both the left hand and right hand sides of the equation are undefined expressions, or they are both defined and equal.

See **Lecture 8 slide 8**.

(b) The class of primitive recursive functions is the smallest set (with respect to subset inclusion) of numerical functions containing the basic functions (projections, successor, zero) and that is closed under the operations of primitive recursion and composition.

See **Lecture 8 slide 13**.

(c) Every $f \in \mathrm{PRIM}$ is total, because the basic functions (projections, successor, zero) are total, the composition of two total functions is total (since if f and $g_1, \ldots, g_n$ are total, then $g_1 x_1, \ldots, g_n x_n$ are defined and so $f(g_1 x_1, \ldots, g_n x_n)$ is defined) and primitive recursion of two total functions is total (since every branch of $\rho^n(f, g)$ is function application of f or g).

There exist (strictly) partial functions that are computable.

See **Lecture 8 slide 16**.

(d) Primitive recursive functions are countable since they are a subset of the RM computable functions, which are countable. The total functions on the other hand are uncountable as shown in Part IA Discrete Mathematics.

We can also define a way to count the primitive recursive functions. We assign encode the lists $\ulcorner[0, i, n]\urcorner$ for $\mathtt{proj}_i^n$, $\ulcorner[1]\urcorner$ for $\mathtt{succ}$ and $\ulcorner[2, n]\urcorner$ for $\mathtt{zero}^n$. For composition of f and $g_1, \ldots, g_n$, we use $\ulcorner[3, \ulcorner f\urcorner, \ulcorner g_1\urcorner, \ldots, \ulcorner g_n\urcorner]\urcorner$. For primitive recursion of $\rho^n(f, g)$, we use $\ulcorner[4, n, \ulcorner f\urcorner, \ulcorner g\urcorner]\urcorner$. (This mapping is not bijective).

(e) The addition function is given by $\rho^1(\mathtt{proj}_1^1, \mathtt{succ} \circ \mathtt{proj}_3^3)$, since

$$\mathtt{add}(x_1, 0) = x_1$$
$$\mathtt{add}(x_1, x_2 + 1) = \mathtt{add}(x_1, x_2) + 1$$

See **Lecture 8 slide 9**.

The predecessor function is given by $\rho^0(\mathtt{zero}^0, \mathtt{proj}_1^2)$.

$$\mathtt{pred}(0) = 0$$
$$\mathtt{pred}(x_1 + 1) = x + 1$$

See **Lecture 8 slide 10**.

The multiplication function is given by $\rho^1(\mathtt{zero}^1, \mathtt{add} \circ [\mathtt{proj}_1^3, \mathtt{proj}_3^3])$

$$\mathtt{mult}(x_1, 0) = 0$$
$$\mathtt{mult}(x_1, x_2) = \mathtt{mult}(x_1, x_2) + x_1$$

See **Lecture 8 slide 11**.

The truncated subtraction function is given by $\rho^1(\texttt{proj}_1^1, \texttt{pred} \circ \texttt{proj}_3^3)$.

$$\texttt{tsub}(x, 0) = x$$
$$\texttt{tsub}(x, y+1) = \texttt{pred}(\texttt{tsub}(x, y))$$

See official solution notes

The exponentiation function is given by $\rho^2(\texttt{succ} \circ \texttt{zero}^1, \texttt{mult} \circ [\texttt{proj}_1^3, \texttt{proj}_3^3])$

$$\texttt{exp}(x, 0) = 0$$
$$\texttt{exp}(x, y+1) = \texttt{mult}(x, \texttt{exp}(x, y))$$

(f) i. $\mathrm{Eq}_0(x, y, z) \triangleq \rho^2(\texttt{proj}_1^2, \texttt{proj}_2^4) \circ [\texttt{proj}_2^3, \texttt{proj}_3^3, \texttt{proj}_1^3]$.

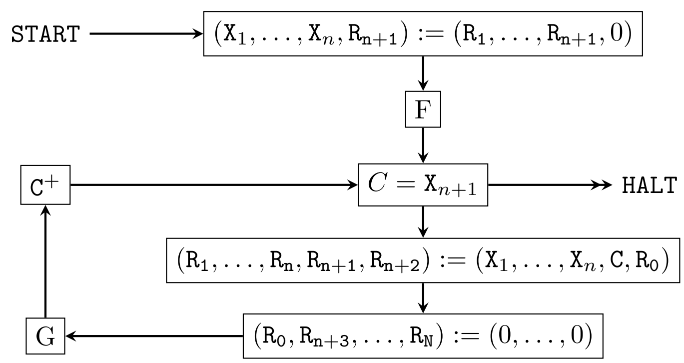
ii. The bounded summation $\rho^n(\texttt{zero}^n, \texttt{add} \circ [\texttt{proj}_{n+1}^{n+2}, \texttt{proj}_{n+2}^{n+2}])$.

(g) The square function is defined as $\texttt{sqr} \triangleq \texttt{mult} \circ [\texttt{proj}_1^2, \texttt{proj}_2^2]$.

The factorial function is defined as $\texttt{fact}(x) = \rho^2(\texttt{succ} \circ \texttt{zero}^2, \texttt{mult} \circ [\texttt{proj}_1^2, \texttt{proj}_2^2])$

Exercise 4 [RMs implement PRIM] Show that primitive recursion is implementable in RMs. Deduce that PRIM functions are computable.

We have already shown in Example Sheet 1, all the basic functions are RM computable (Exercise 1). We know that composition is also primitive recursive (Exercise 2(c)). Hence, we need to show that primitive recursion of RM computable functions is RM computable. We can compute the recursive equation, using a simple for-loop, starting from the base case $x = 0$ and progressing to $x = 1, 2, \ldots$ until we reach the target value, in which case we halt. In each iteration we need to zero all registers $\texttt{R}_0, \ldots, \texttt{R}_n$ used by the program.



See **Lecture 8 slide 15**.

Exercise 5 [Minimisation]

(a) Define *minimisation*.

(b) Why might we want to define minimisation?

(c) Implement div.

(d) Show that minimisation is implementable using RMs.

(a) Given a partial function $f \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$, the minimisation of f, $\mu^n f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is defined as $\mu^n f(x)$ is the smallest x such that $f(\vec{x}, x) = 0$ and for each $i = 0, \ldots, x-1$, $f(\vec{x}, i)$ is defined and is positive. If no such x exists, then it is undefined.

See **Lecture 8 slide 18**.

(b) Minimisation is the missing part to make primitive recursion equivalent to TMs and to RMs (and to λ-calculus).

(c) The result of integer division between x_1 and x_2 is the least x_3 such that $x_1 < x_2(x_3+1)$. Let's consider an example $x_1 = 70$ and $x_2 = 8$. For $x_3 = 7$ we have $x_2(x_3+1) = 8 \cdot 8 = 64 < 70$ (so all values $x_3 < 7$ we have < 70, for $x_3 = 8$ we have $x_2(x_3+1) = 8 \cdot 9 = 72 > 70$.

Hence, we can define integer division as $\mu^2\texttt{tsub}(x_1, x_2)$, so for $x_1 < x_2(x_3+1)$ this will be 0 and otherwise it will be 1. When $x_2 = 0$, this will always be 0 and hence the minimisation will be undefined.

See **Lecture 8 slide 20**.

(d) The idea is to loop over $x = 0, 1, \ldots$ evaluate $f(\vec{x}, x)$ and if it is 0 return x, otherwise continue with $x+1$.

See **Lecture 8 slide 24**.

Exercise 6

(a) Define *partial recursive (PR) functions*. (See [**2018P6Q5 (a)**], [**2016P6Q3 (a)**], [**2006P4Q9 (d)**], [**1995P4Q9 (a)**])

(b) Show that PR functions are RM computable. (See [**2016P6Q3 (b)**], [**1999P4Q1 (b)**])

(c) Describe in high-level terms why every computable function is also PR (See [**1995P4Q9 (b),(c)**]).

(a) A partial function f is partial recursive ($f \in \text{PR}$) if it can be built up in finitely many steps from the basic functions (projection, successor, zero) by use of the operations of composition, primitive recursion and minimisation.

See **Lecture 8 slide 22**.

(b) We need to show that the basic functions (projection, successor, zero) (done in Exercise 1), composition (done in Exercise 2(c)), primitive recursion (done in Exercise 4) and minimisation (done in Exercise 5(d)) are all RM computable.

(c) See Computability: An introduction to recursive function theory (p.106) for showing that the $\texttt{next}_M$ is partial recursive.

See **Lecture 8 slide 25**.

Exercise 7 Attempt [**2018P6Q5**].

See official solution notes

Exercise 8 Attempt [**2014P6Q4 (e)**].

See official solution notes

Lecture 9 (first part)

Exercise 9

(a) Define the *Ackermann function.*

(b) In what sense does it grow faster than any primitive recursive function?

(c) (optional - advanced) Read this proof for the Ackermann's function growing faster than any primitive recursive function.

(a) The Ackermann function $\texttt{ack} : \mathbb{N}^2 \to \mathbb{N}$ is defined recursively as

$$\texttt{ack}(x_1, x_2) = \begin{cases} x_2 + 1 & \text{if } x_1 = 0 \\ \texttt{ack}(x_1 - 1, 1) & \text{if } x_2 = 0 \\ \texttt{ack}(x_1 - 1, \texttt{ack}(x_1, x_2 - 1)) & \text{otherwise} \end{cases}$$

See **Lecture 9 slide 6**.

(b) It means that for every primitive recursive function f, $\exists N_f . \forall x_1, x_2 . f(x_1, x_2) < \texttt{ack}(x_1, x_2)$.

See **Lecture 9 slide 6**.

Exercise 10 Attempt [**2001P4Q8**].

(a) For $y = 0$, $g_{n+1}(y) = f(n+1, 0) = f(n, 1) = g_n^{(1)}(1)$. Assume, it is true for $y = k$ and consider $y = k+1$. Then, we have,

$$g_{n+1}(k+1) = f(n+1, k+1) = f(n, f(n+1, k)) = f(n, g_{n+1}^{(k+1)}(1)) = g_{n+1}(g_{n+1}^{(k+1)}(1)) = g_{n+1}^{(k+2)}(1).$$

Hence, by the principle of natural induction, this holds for all y.

(b) For $n = 0$, g_n is just the sum function. Assume that g_n is primitive recursive. Consider the partial recursive function $\rho^1(g_n(\underbrace{\texttt{succ} \circ \ldots \circ \texttt{succ}}_{n \text{ times}} 0), g_n \circ \texttt{proj}_2^2)$. **Note:** This only works because we are considering a fixed n.

(c) Assume `ack` is not total, then there exists a smallest x_1 such that $\texttt{ack}(x_1, x_2)$ is not defined. Note that $x_1 > 0$. Hence, then $g_{x_1}(x_2)$ is not defined and so is $g_{x_1-1}^{x_2+1}(1)$. But this contradicts the minimality of x_1. So, `ack` is total.

(d) As stated in the lecture notes, the Ackermann function is growing faster than any primitive recursive function.

Exercise 11 [`ack` **is RM computable**] Recall the definition of Ackermann's function ack (slide 102). Sketch how to build a register machine M that computes $\mathrm{ack}(x_1, x_2)$ in R_0 when started with x_1 in R_1 and x_2 in R_2 and all other registers zero. [*Hint*: here's one way; the next question steers you another way to the computability of `ack`. Call a finite list $L = [(x_1, y_1, z_1), (x_2, y_2, z_2), ...]$ of triples of numbers suitable if it satisfies

- if $(0, y, z) \in L$, then $z = y + 1$
- if $(x+1, 0, z) \in L$, then $(x, 1, z) \in L$
- if $(x+1, y+1, z) \in L$, then there is some u with $(x+1, y, u) \in L$ and $(x, u, z) \in L$.

The idea is that if $(x, y, z) \in L$ and L is suitable then $z = \mathrm{ack}(x, y)$ and L contains all the triples $(x', y', \mathrm{ack}(x, y'))$ needed to calculate $\mathrm{ack}(x, y)$. Show how to code lists of triples of numbers as numbers in such a way that we can (in principle, no need to do it explicitly!) build a register machine that recognises whether or not a number is the code for a suitable list of triples. Show how to use that machine to build a machine computing $\mathrm{ack}(x, y)$ by searching for the code of a suitable list containing a triple with x and y in it's first two components.]

[**Exercise 9 in Lecturer's handout**]

Exercise 12 Give an example of a function that is not in PRIM. (See [**2014P6Q4 (d)**])

Any non-computable language is not in PRIM. Also, Ackermann's function is not in PRIM, because it grows faster than any language in PRIM.

Lecture 9 (second part)

Further reading:

- Foundations of Functional Programming.

Exercise 13

(a) How are λ-terms defined? (See [**2016P6Q4 (a)**])

(b) What notational conventions do we follow?

(c) **Exercise 1.4 in Hindley and Seldin (2008)** Insert the full amount of parentheses in the following abbreviated terms:

   i. $xyz(yx)$,

ii. $\lambda x.uxy$,
iii. $\lambda u.u(\lambda x.y)$,
iv. $ux(yz)(\lambda v.vy)$,
v. $(\lambda xyz.xz(yz))uvw$,
vi. $w(\lambda xyz.xz(yz))uv$.

(d) What does $x\#M$ mean?

(e) What do the terms *bound variable*, *body*, *binding*, *bound*, *free*, $\mathrm{FV}(\cdot)$, $\mathrm{BV}(\cdot)$ and *closed term* mean?

(f) Determine the free variables and bound variables in the following expressions:
i. $\lambda u.\lambda u.\lambda y.u\lambda u.\lambda y.u$.
ii. $(\lambda x\lambda u.y)((xx)x)((vy)\lambda u.u)$.
iii. $(((\lambda z.z)z)\lambda y.\lambda v.v)(\lambda v.\lambda y.v)$.
iv. $(\lambda x.((xv)u)(\lambda z.z\lambda y.y))$
v. You can generate more practice questions here.

(a) λ-terms are build by a collection of variables and two operations:

- (λ-abstraction) $\lambda x.M$, where x is a variable and M is a λ term.
- (application) $M\ N$, where M and N are λ terms.

Alternatively, we can use rule induction to define these. Let V be the set of variables,

$$\frac{}{x}\,x \in V \qquad \frac{M}{\lambda x.M}\,x \in V \qquad \frac{M \quad N}{M\ N}$$

See **Lecture 9 slide 13**.

(b) The following notational conventions where introduced by the lecturer:

- $(\lambda x_1, \ldots, x_n.M)$ stands for $(\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.M)\ldots))$.
- $(M_1 \ldots M_n)$ stands for $(\ldots(M_1\ M_2)\ldots M_n)$. This is similar to function application in OCaml (and it can be stated concisely as function application is left-associative).
- $\lambda x.M$ stands for $(\lambda x.M)$ (so we drop the outermost parentheses.

See **Lecture 9 slide 14**.

(c) i. $((x\ y)\ z)\ (y\ x)$
ii. $(\lambda x.((u\ x)\ y))$
iii. $(\lambda u.(u(\lambda x.y)))$
iv. $(((u\ x)\ (y\ z))\ (\lambda v.(v\ y)))$
v. $((((\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z)))))\ u)\ v)\ w)$
vi. $(((w\ (\lambda x.(\lambda y.(\lambda z.((x\ z)\ (y\ z))))))\ u)\ v)$

(d) $x\#M$ means that the variable x does not occur anywhere in the λ-term M.

See **Lecture 9 slide 14**.

(e)
- $\lambda \underbrace{x}_{\text{bound variable}} . \underbrace{M}_{\text{body of }\lambda\text{-abstraction}}$
- *binding variable*: an occurrence of variable x in $\lambda x.\ldots$.
- *bound variable*: an occurrence of variable x in the body M of some $\lambda x.M$.
- *free variable*: an occurrence of variable x that is neither bounding nor bound.
- The set of *free variables* is defined as $\mathrm{FV}(x) = \{x\}$, $\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) - \{x\}$ and $\mathrm{FV}(M\ N) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$.
- The set of *bound variables* is defined as $\mathrm{BV}(x) = \emptyset$, $\mathrm{BV}(\lambda x.M) = \mathrm{BV}(M) \cup \{x\}$ and $\mathrm{BV}(M\ N) \cup \mathrm{BV}(N)$.

- *closed term/combinator*: a λ-term without any free variables.

See **Lecture 9 slide 15**.

See **Lecture 9 slide 17**.

(f) The free variables are shown in red and the bound variables are shown in blue:

i. $\lambda u.\lambda u.\lambda y.u\lambda u.\lambda y.u$.

ii. $(\lambda x\lambda u.y)((x\ x)\ x)((v\ y)\lambda u.u)$.

iii. $(((\lambda z.z)\ z)\lambda y.\lambda v.v)(\lambda v.\lambda y.v)$.

iv. $(\lambda x.((x\ v)\ u)(\lambda z.z\ \lambda y.y))$

Exercise 14 [**α-equivalence**]

(a) Intuitively, what does α-*equivalence* try to capture?

(b) Define what $M\{z/x\}$ means.

(c) Define formally α-equivalence.

(d) Show that the following pairs are α-equivalent:

i. $A \triangleq \lambda xy.x(xy)$ and $B \triangleq \lambda uv.u(uv)$,

ii. $A \triangleq (\lambda xyz.y(zx(\lambda k.k)))(\lambda xy.yx)$ and $B \triangleq (\lambda k\ell m.\ell(mk(\lambda a.a)))(\lambda yx.xy)$.

(e) (optional) Show that α-equivalence is an equivalence relation.

(a) Intuitively, two λ-terms are α-equivalent if they are equal up to renaming variables (variables bound to the same binding variable should be renamed together).

(b) $M\{z/x\}$ represents substituting all free occurrences of x with z (and $z\#M$ as otherwise for the $M = \lambda z.xz$, we would get $M\{z/x\} = \lambda z.zz$, which is not the same function).

See **Lecture 9 slide 18**.

(c) The α-equivalence is defined inductively as

$$\frac{}{x =_\alpha x} \qquad \frac{z\#(M\ N) \qquad M\{z/x\} =_\alpha N\{z/y\}}{\lambda x.M =_\alpha \lambda y.N} \qquad \frac{M =_\alpha M' \quad N =_\alpha}{M\ N =_\alpha M'\ N'}$$

See **Lecture 9 slide 19**.

(d) i.

$$\dfrac{z\#(\lambda y.x(xy))\ (\lambda v.u(uv)) \qquad \dfrac{w\#(z\ (z\ y))\ z\ (z\ v) \qquad \dfrac{z =_\alpha u \qquad \dfrac{z =_\alpha z \qquad w =_\alpha w}{(z\ w) =_\alpha (z\ w)}}{z\ (z\ w) =_\alpha z\ (z\ w)}}{(\lambda y.z(z\ y)) =_\alpha (\lambda v.z(z\ v))}}{\lambda xy.x\ (x\ y) =_\alpha \lambda uv.u\ (u\ v)}$$

ii.

$$\dfrac{\dfrac{A}{(\lambda xyz.y(zx(\lambda k.k))) =_\alpha (\lambda k\ell m.\ell(mk(\lambda a.a)))} \qquad \dfrac{z\#(\lambda y.yx)\ (\lambda x.xy) \qquad \dfrac{w\#(y\ z)\ (x\ z) \qquad \dfrac{w =_\alpha w \qquad z =_\alpha z}{w\ z =_\alpha w\ z}}{(\lambda y.yz)\ (\lambda x.xz)}}{(\lambda xy.yx) =_\alpha (\lambda yx.xy)}}{(\lambda xyz.y(zx(\lambda k.k)))(\lambda xy.yx) =_\alpha (\lambda k\ell m.\ell(mk(\lambda a.a)))(\lambda yx.xy)}$$

$$\dfrac{w\#\ldots \qquad \dfrac{u\#\ldots \qquad \dfrac{v\#\ldots \qquad \dfrac{u =_\alpha u \qquad \dfrac{v =_\alpha v \qquad \dfrac{w =_\alpha w \qquad \dfrac{b\#k\ a \qquad b =_\alpha b}{\lambda k.k =_\alpha \lambda a.a}}{w(\lambda k.k) =_\alpha w(\lambda a.a)}}{vw(\lambda k.k) =_\alpha vw(\lambda a.a)}}{u(vw(\lambda k.k)) =_\alpha u(vw(\lambda a.a))}}{\lambda z.u(zw(\lambda k.k)) =_\alpha (\lambda m.u(mw(\lambda a.a)))}}{\lambda yz.y(zw(\lambda k.k)) =_\alpha (\lambda \ell m.\ell(mw(\lambda a.a)))}}{A}$$

Lecture 10

Exercise 15 [Substitution]

(a) Define the *substitution operation* $N[M/x]$.

(b) **Exercise 1.14 in Hindley and Seldin (2008)** Evaluate the following substitutions:

i. $(\lambda y.x\ (\lambda w.v\ w\ x))[(u\ v)/x]$

ii. $(\lambda y.x(\lambda x.x))[(\lambda y.x\ y)/x]$

iii. $(y\ (\lambda v.x\ v))[(\lambda y.vy)/x]$

iv. $(\lambda x.z\ y)[(u\ v)/x]$

(a) The substitution operation is defined as:

- $x[M/x] = M$
- $y[M/x] = y$ if $y \neq x$
- $(\lambda y.N)[M/x] = \lambda y.N[M/x]$ if $y\#(M\ x)$
- $(N_1\ N_2)[M/x] = N_1[M/x]N_2[M/x]$

See **Lecture 10 slide 3**.

(b) i. $(\lambda y.u\ v\ (\lambda w.v\ w\ (u\ v)))$

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

iii. $(y\ (\lambda z.(\lambda y.vy)\ z))$

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

Exercise 16

(a) Define one-step β-*reduction.*

(b) Define the many-step β-*reduction.* (See **[2015P6Q4 (a)]**)

(c) Define the β-*conversion.* (See **[2019P6Q6 (a)(i)]**)

(a) The one-step β-reduction is defined inductively as follows:

$$\frac{}{(\lambda x.M)\ N \to M[N/x]} \qquad \frac{M \to M'}{\lambda x.M \to \lambda x.M'} \qquad \frac{M \to M'}{M\ N \to M'\ N'}$$

$$\frac{M \to M'}{N\ M \to N\ M'} \qquad \frac{N =_\alpha M \quad M \to M' \quad M' =_\alpha N'}{N \to N'}$$

See **Lecture 10 slide 13**.

(b) The many-step β-reduction is defined as follows:

$$\frac{M =_\alpha M'}{M \twoheadrightarrow M'} \qquad \frac{M \to M'}{M \twoheadrightarrow M'} \qquad \frac{M \twoheadrightarrow M' \quad M' \twoheadrightarrow M''}{M \twoheadrightarrow M''}$$

See **Lecture 10 slide 19**.

(c)

$$\frac{M =_\alpha M'}{M =_\beta M'} \qquad \frac{M \to M'}{M =_\beta M'} \qquad \frac{M =_\beta M'}{M' =_\beta M}$$

$$\frac{M =_\beta M' \quad M' =_\beta M''}{M =_\beta M''} \qquad \frac{M =_\beta M'}{\lambda x.M =_\beta \lambda x.M'} \qquad \frac{M =_\beta M' \quad N =_\beta N'}{M\ N =_\beta M'\ N'}$$

See **Lecture 10 slide 24**.

Exercise 17

(a) State the Church-Rosser Theorem and prove its corollary.

(b) Attempt [**2019P6Q6 (a)(iii)**].

(a) The Church-Rosser theorem states that: The $\twoheadrightarrow$ relation is *confluent* meaning that if $M_1 \twoheadleftarrow M \twoheadrightarrow M_2$, then there exists M' such that $M_1 \twoheadrightarrow M' \twoheadleftarrow M_2$.

The corollary states that two terms M_1 and M_2 are β-convertible iff there exists a term M such that $M_1 \twoheadrightarrow M$ and $M_2 \twoheadrightarrow M$.

($\Leftarrow$) Assume $\exists M.(M_1 \twoheadrightarrow M \twoheadleftarrow M_2)$. Since $A \twoheadrightarrow B$ implies that $A =_\beta B$, $M_1 =_\beta M$ and $M_2 =_\beta M$, so (by transitivity) $M_1 =_\beta M_2$.

($\Rightarrow$) We will prove by rule induction that for every two β-equivalent λ-terms M_1 and M_2, there exists M such that $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$.

- $\frac{M_1 =_\alpha M_2}{M_1 =_\beta M_2}$: If $M_1 =_\alpha M_2$, then we can transform M_1 to M_2 using substitutions and M_2 to M_1 using substitutions. Hence, we can take $M = M_1$ (or $M = M_2$) and then $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$.
- $\frac{M_1 \to M_2}{M_1 =_\beta M_2}$: If $M_1 \to M_2$, then $M_1 \twoheadrightarrow M_2$, so we pick $M = M_2$.
- $\frac{M_1 =_\beta M_2}{M_2 =_\beta M_1}$: By inductive hypothesis, there exists M such that $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$. So, $M_2 \twoheadrightarrow M \twoheadleftarrow M_1$.
- $\frac{M_1 =_\beta M_2 \quad M_2 =_\beta M_3}{M_1 =_\beta M_3}$: !! By inductive hypothesis, there exist M_4 and M_5, such that $M_1 \twoheadrightarrow M_4 \twoheadleftarrow M_2$ and $M_2 \twoheadrightarrow M_5 \twoheadleftarrow M_3$. So $M_4 \twoheadleftarrow M_2 \twoheadrightarrow M_5$. By the Church-Rosser theorem, there exists M such that $M_4 \twoheadrightarrow M \twoheadleftarrow M_5$. Hence, $M_1 \twoheadrightarrow M_4 \twoheadrightarrow M \twoheadleftarrow M_5 \twoheadleftarrow M_3$. So, $M_1 \twoheadrightarrow M \twoheadleftarrow M_3$ by transitivity of $\twoheadleftarrow$.
- $\frac{M_1 =_\beta M_2}{\lambda x.M =_\beta \lambda x.M_2}$: By inductive hypothesis, there exists M such that $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$. Note that if $N \twoheadrightarrow N'$, then $\lambda x.N \twoheadrightarrow \lambda x.N'$, by performing the $\to$ steps in the body of the function. Hence, $\lambda x.M_1 \twoheadrightarrow \lambda x.M \twoheadleftarrow \lambda x.M_2$.
- $\frac{M_1 =_\beta M_2 \quad M_3 =_\beta M_4}{M_1\ M_3 =_\beta M_2\ M_4}$: By inductive hypothesis, there exist M_5 and M_6 such that $M_1 \twoheadrightarrow M_5 \twoheadleftarrow M_2$ and $M_3 \twoheadrightarrow M_6 \twoheadleftarrow M_4$. So, $M_1\ M_3 \twoheadrightarrow M_5\ M_6 \twoheadrightarrow M_2\ M_4$.

See **Lecture 10 slide 29**.

(b) By the corollary of the Church-Rosser Theorem, we have $M_1 \twoheadrightarrow M$ and $M_2 \twoheadrightarrow M$. Since they are in both in β-nf, it means that the reductions to M are just α-equivalence steps (if there was a β reduction, they would not be in β-nf).

See official solution notes

Exercise 18

(a) Define the β-*normal form.* (See [**2019P6Q6 (a)(ii)**], [**2013P6Q4 (a)(i)**])

(b) What properties does this form have?

(c) Do all terms have a β-*normal form*? (See [**2013P6Q4 (a)(iii)**])

(d) Show that there exists λ-terms that have both a β-normal form and an infinite chain of reductions from it.

(e) **Exercise 1.28 in Hindley and Seldin (2008)** Find the β-normal form for the following terms (if it exists):

i. $(\lambda x.x(xy))z$,

ii. $(\lambda x.y)z$,

iii. $(\lambda x.(\lambda y.yx)z)v$,

iv. $(\lambda x.xxy)(\lambda x.xxy)$,

v. $(\lambda x.xy)(\lambda u.vuu)$,

vi. $(\lambda x.x(x(yz))x)(\lambda u.uv)$,

vii. $(\lambda xy.xyy)(\lambda u.uyx)$,

viii. $(\lambda xyz.xz(yz))((\lambda xy.yx)u)((\lambda xy.yx)v)w$.

(a) A λ-term is in β-normal form if it contains no β-redexes (i.e. no sub-terms of the form $(\lambda x.M)\ M'$). A term M has a β-normal form N if N is in β-normal form and $M =_\beta N$.

See **Lecture 10 slide 32**.

(b) The β-normal form (if it exists) is unique up to α-equivalence.

See **Lecture 10 slide 32**.

(c) The example given in the lecture notes was the term $(\lambda x.y)\ \Omega$, where if we β-reduce Ω, we get the same term $(\lambda x.y)\ \Omega$ (so we can get chains of unbounded length); otherwise we get y (which is β-nf).

See **Lecture 10 slide 34**.

(d) i. $(\lambda x.x(xy))z \twoheadrightarrow z\ (z\ y)$

ii. $(\lambda x.y)z \twoheadrightarrow y$,

iii. $(\lambda x.(\lambda y.yx)z)v \twoheadrightarrow z\ v$,

iv. $(\lambda x.xxy)(\lambda x.xxy) \twoheadrightarrow (\lambda x.xxy)(\lambda x.xxy)\ y \twoheadrightarrow (\lambda x.xxy)(\lambda x.xxy)\ y\ y \twoheadrightarrow \ldots$ (each time the only possible reduction will be $(\lambda x.xxy)(\lambda x.xxy)$, hence this term has no β-normal form),

v. $(\lambda x.x\ y)(\lambda u.v\ u\ u) \twoheadrightarrow (\lambda u.v\ u\ u)\ y \twoheadrightarrow v\ y\ y$,

vi.

$$\begin{aligned}(\lambda x.x(x(yz))x)(\lambda u.uv) &\twoheadrightarrow (\lambda u.uv)((\lambda u.uv)(y\ z))(\lambda u.uv) \twoheadrightarrow ((\lambda u.uv)(y\ z))\ v(\lambda u.uv)\\ &\twoheadrightarrow ((y\ z)\ v)\ v(\lambda u.uv)\end{aligned}$$

vii. $(\lambda xy.xyy)(\lambda u.uyx) \twoheadrightarrow \lambda a.(\lambda u.u\ y\ x)\ a\ a \twoheadrightarrow \lambda a.a\ y\ x\ a$

viii.

$$\begin{aligned}(\lambda xyz.xz(yz))((\lambda xy.yx)u)((\lambda xy.yx)v)w &\twoheadrightarrow (\lambda yz.((\lambda xy.yx)u)z(yz))((\lambda xy.yx)v)w\\ &\twoheadrightarrow (\lambda z.((\lambda xy.yx)u)z(((\lambda xy.yx)v)z))w\\ &\twoheadrightarrow ((\lambda xy.yx)u)w(((\lambda xy.yx)v)w)\\ &\twoheadrightarrow (\lambda y.y\ u)w(((\lambda xy.yx)v)w)\\ &\twoheadrightarrow (w\ u)(((\lambda xy.yx)v)w)\\ &\twoheadrightarrow (w\ u)((\lambda y.y\ v)w) \twoheadrightarrow (w\ u)(w\ v)\end{aligned}$$

Exercise 19

(a) Define *normal-order* reduction.

(b) Is it similar to *call-by-name*?

(c) Is there an evaluation analogous to *call-by-value*? Which one is preferred?

(a) Normal-order reduction refers to the deterministic strategy of first reducing left-most and then outer-most terms in a λ-term. This reduction has the property that it will find the β-nf if it exists.

See **Lecture 10 slide 35**.

(b) It is similar to call by name in the sense that it does not evaluate the arguments of a function before calling the function, but only when the argument is about to be applied.

(c) Yes, there is an analogous, but it has the disadvantage that if you have e.g. `zero` Ω, then it will try to evaluate Ω and loop forever, even though the term reduces to 0. A practical disadvantage of the call-by-name is that the is that evaluates a certain argument multiple times (e.g. $\lambda x.x + x + x$ will evaluate x three times). In practice, there exist some hybrid schemes like *call-by-need* (but this also has problems of its own).

Exercise 20 [Lambda functions in OCaml] (optional) In this exercise, you will implement β-reduction for lambda terms in OCaml.

(a) Define a type for lambda terms in OCaml.

(b) Define the function `substitute n m x` that replaces all occurrences of variable `x` with `m` inside `n`.

(c) Define the function `single_step_reduce m` that returns `(m', reduced)` the reduced term (or the original term) and whether a reduction was applied.

(d) Define the function `multi_step_reduce m` that calls `single_step_reduce` until `reduced` is false. Verify the reduction works as expected by applying it on the above examples.

```
type lambda = Var of string | App of lambda * lambda | Lambda of string * lambda;;

let rec substitute n m x = match n with
| Var(y) ->
   if x = y then m
   else n
| App(ell1, ell2) -> App(substitute ell1 m x, substitute ell2 m x)
| Lambda(z, ell) ->
   if x = z then n
   else Lambda(z, substitute ell m x);;

let sub_term = Lambda("y", App(Var("x"), Lambda("w", App(App(Var("v"), Var("w")),
    Var("x")))));;
substitute sub_term (App(Var("u"), Var("v"))) "x";;

let rec single_step_beta = function
| Var(x) -> (Var(x), false)
| App(Lambda(x, m), n) -> (substitute m n x, true)
| App(ell1, ell2) ->
   let (left, reduced_left) = single_step_beta ell1 in
      if reduced_left then (App(left, ell2), true)
      else let (right, reduced_right) = single_step_beta ell2 in
         (App(ell1, right), reduced_right)
| Lambda(x, ell) -> let (v, reduced) = single_step_beta ell in
   (Lambda(x, v), reduced);;

let term = App(Lambda("x", App(Var("x"), Var("y"))), App(Lambda("y", Lambda("z",
    Var("z"))), Var("u")));;

let rec multi_step_beta n =
   let (v, reduced) = single_step_beta n in
      if reduced then multi_step_beta v
      else v;;
```

Lecture 11

Exercise 21

(a) Define *Church's numerals*. (See **[2020P6Q6 (a)]**, **[2016P6Q4 (b)]**, **[2010P6Q4 (a)]**)

(b) What is the difference between ffx and $f(f(x))$.

(c) Show that $\underline{n}\ M\ N =_\beta M^n\ N$.

(d) Prove by induction that $(\lambda x_1 x_2.\lambda f x.x_1 f(x_2 f x))\ \underline{n}\ \underline{m}$ represents addition.

(a) Church's numerals are lambda terms used to represent the natural numbers. In this representation, $\underline{n} \triangleq \lambda f\,x.\underbrace{f(\ldots(f}_{n \text{ times}}\ x))$ (Of course, this is not the only valid representation)

(b) The first term corresponds to $(ff)\ x$, while the second corresponds to $f(fx)$.

(c) For $\underline{n}$, $\underline{n}\ M\ N =_\beta (\lambda fx.f^n\ x)\ M\ N =_\beta M^n\ N$.

(d)

$$\begin{aligned}\lambda fx.\underline{n}f(\underline{m}\ f\ x) &=_\beta \lambda fx.\underline{n}\ f\ (f^m x) \text{(By property (c))}\\ &=_\beta \lambda fx.f^n\ (f^m x) \text{(By property (c))}\\ &= \lambda fx.f^{n+m}x\end{aligned}$$

The last step is an equality because $f^n(f^m x)$ is just a different expression for f^{n+m} or $f\ldots f$.

See **Lecture 11 slide 6**.

Exercise 22 Define λ-*definable* functions. (See [**2020P6Q6 (c)**], [**2018P6Q6 (c)**], [**2010P6Q4 (b)**])

A function $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is λ-definable if there is a closed λ-term F that represents it: for all $X_1, \ldots, x_n) \in \mathbb{N}^n$ and $y \in \mathbb{N}$:

- If $f(x_1, \ldots, x_n) = y$, then $F\underline{x_1} \ldots \underline{x_n} =_\beta \underline{y}$.
- If $f(x_1, \ldots, x_n) \uparrow$, then $F\underline{x_1} \ldots \underline{x_n}$ has no β-nf.

See **Lecture 11 slide 6**.

Exercise 23

(a) Show that `proj`, `succ` and `zero` are λ-definable. (See [**2020P6Q6 (d)**], [**2010P6Q4 (c)**])

(b) Show how to represent *composition*. What is the problem here? (See [**2013P6Q4 (b)(ii),(iii)**])

(c) Define λ-terms for **True**, **False** and **If**. (See [**2020P6Q6 (b)**], [**2019P6Q6 (b)**])

(d) Prove that **If True** $MN \equiv_\beta M$ and **If False** $M\ N =_\beta N$.

(e) Define λ-terms for **And**, **Or** and **Not**.

(f) Show that testing for equality with 0 is λ-definable.

(g) Define λ-terms for **Pair**, **Fst** and **Snd**. Show that **Fst** (**Pair** $M\ N) =_\beta M$ (See [**2020P6Q6 (e)**]).

(h) Define the `pred` function and prove by induction that it works.

(i) Attempt [**2020P6Q6 (f),(g)**].

(j) Attempt [**2016P6Q4 (c)**].

(a) The projection function is simply defined as $\texttt{proj}_i^n \triangleq \lambda x_1 \ldots x_n.x_i$.

The zero function is $\texttt{zero}^n \triangleq \lambda x_1, \ldots, x_n.\underline{0}$.

The successor function $\texttt{succ} \triangleq \lambda v\ f\ x.f\ (v\ f\ x)$.

(b) Composition between total functions can be presented as $\lambda x_1 \ldots x_n.F(G_1 x_1 \ldots x_n) \ldots (G_m x_1 \ldots x_n)$. The problem is that this construction does not work for when F and G can be partial. A concrete example is $F \triangleq \texttt{zero}^1$ and $G \triangleq \lambda x.\Omega$. The composition of this function should be undefined, but $F\ G\ \underline{0} = (\texttt{zero}^1\ (\lambda x.\Omega))\ \underline{0} =_\beta \underline{0}\ \underline{0}$ which is in β-nf. See Exercise 27 for how to fix this.

See **Lecture 11 slide 14**.

(c) These are defined as:

- **True** $\triangleq \lambda xy.x$
- **False** $\triangleq \lambda xy.y$
- **If** $\triangleq \lambda fxy.f\ x\ y$

See **Lecture 11 slide 19**.

(d) **If True** $M\ N = (\lambda fxy.fxy)\ (\lambda xy.x)\ M\ N =_\beta (\lambda xy.x)\ M\ N =_\beta M$
If False $M\ N = (\lambda fxy.fxy)\ (\lambda xy.y)\ M\ N =_\beta (\lambda xy.y)\ M\ N =_\beta N$

See **Lecture 11 slide 19**.

(e) We give the following definitions:

- **And** $\triangleq \lambda f_1 f_2.\lambda xy.f_1\ (f_2\ x\ y)\ y$
  This way if either of f_1 or f_2 returns y, there is no way for the other function to change the result.
- **Or** $\triangleq \lambda f_1 f_2.\lambda xy.f_1\ x\ (f_2\ x\ y)$
  This way if either of f_1 or f_2 returns x, there is no way for the other function to change the result.
- **Not** $\triangleq \lambda f.\lambda xy.f\ y\ x$.
  This way the outcome of f is reversed.

(f) The idea is that since $\underline{n} \triangleq \lambda fx.f^n\ x$ (and $\underline{0} = \lambda fx.x$) we are going to choose an f so that when it is applied (even once), we get **False** otherwise we get **True**. A suitable function is $\lambda y.\mathbf{False}$, so that

$$\underline{0}\ (\lambda y.\mathbf{False})\ \mathbf{True} = (\lambda fx.x)\ (\lambda y.\mathbf{False})\ \mathbf{True} =_\beta \mathbf{True}$$

and

$$\begin{aligned}\underline{n+1}\ (\lambda y.\mathbf{False})\ \mathbf{True} = (\lambda fx.f^{n+1}x)\ (\lambda y.\mathbf{False})\ \mathbf{True} &=_\beta (\lambda y.\mathbf{False})^{n+1}\ \mathbf{True} \\ =_\beta (\lambda y.\mathbf{False})\ \mathbf{True} =_\beta \mathbf{False}.\end{aligned}$$

(g) The definitions are as follows:

- $\mathbf{Pair} \triangleq \lambda xyf.f\ x\ y$
- $\mathbf{Fst} \triangleq \lambda f.f\ \mathbf{True}$
- $\mathbf{Snd} \triangleq \lambda f.f\ \mathbf{False}$

$\mathbf{Fst}(\mathbf{Pair}\ M\ N) =_\beta \mathbf{Fst}(\lambda f.\ M\ N) =_\beta (\lambda f.\ M\ N)\ \mathbf{True} =_\beta \mathbf{True}MN =_\beta M$

See **Lecture 11 slide 22**.

(h) The difficulty with defining the predecessor is that in Church's numerals we only have access to a function that is applied n times. The idea is to apply the function $(x, y) \to (f\ x, x)$, so inductively we will be storing $(f^{n+1}x_0, f^n x_0)$. Hence, in the end we can just take the item of the pair. A formal proof was given in the slides.

$$\begin{aligned}\mathbf{Pred} &\triangleq \lambda yfx.\mathbf{Snd}(y\ (G\ f)\ (\mathbf{Pair}\ x\ x)) \\ G &\triangleq \lambda fp.\mathbf{Pair}(f\ (\mathbf{Fst}\ p))(\mathbf{Fst}\ p)\end{aligned}$$

See **Lecture 11 slide 23**.

(i) This represents the predecessor function. The proof follows by induction as in the lecture notes.

For constructing the predicate **Leq** we use **Pred** and $\mathbf{Eq}_0$, Namely, we define

$$\mathbf{Leq}\ \underline{m}\ \underline{n} \triangleq \lambda xy.\mathbf{Eq}_0(y\ \mathbf{Pred}\ x)$$

This works because,

$$\mathbf{Leq}\ \underline{m}\ \underline{n} =_\beta \mathbf{Eq}_0(\underline{n}\ \mathbf{Pred}\ \underline{m}) =_\beta \mathbf{Eq}_0(\mathbf{Pred}^n\ \underline{m}) =_\beta \mathbf{Eq}_0(\underline{n \dot{-} m}) =_\beta \begin{cases}\mathbf{True} & \text{if } m \leq n \\ \mathbf{False} & \text{otherwise}\end{cases}$$

See **Lecture 11 slide 23**.

See official solution notes

(j)

Exercise 24 If you are still not fed up with Ackermann's function $\mathtt{ack} \in \mathbb{N}^2 \to \mathbb{N}$, show that the λ-term $\mathbf{Ack} \triangleq \lambda x.x(\lambda fy.y\ f\ (f\ \underline{1}))\ \mathbf{Succ}$ represents `ack` (where **Succ** is as on slide 123).

[Exercise 11 in Lecturer's handout]

We will prove each of the three branches in the `ack` definition. Then an inductive argument over all x_1 and x_2 (which proceeds in a diagonal direction with base case $\mathtt{ack}\ \underline{0}\ \underline{0}$), proves that $\mathbf{Ack}\ \underline{x_1}\ \underline{x_2} = \underline{\mathtt{ack}(x_1, x_2)}$.

$$\begin{aligned}\mathbf{Ack}\ \underline{0}\ \underline{x_2} &=_\beta (\lambda x.x(\lambda fy.y\ f\ (f\ \underline{1}))\ \mathbf{Succ})\ \underline{0}\ \underline{x_2} \\ &=_\beta \underline{0}\ (\lambda fy.y\ f\ (f\ \underline{1}))\ \mathbf{Succ}\ \underline{x_2} \\ &= (\lambda fx.x)\ (\lambda fy.y\ f\ (f\ \underline{1}))\ \mathbf{Succ}\ \underline{x_2}\ \text{(By definition of } \underline{0}) \\ &=_\beta \mathbf{Succ}\ \underline{x_2} \\ &=_\beta \underline{x_2+1}\ \text{(By properties of } \mathbf{Succ})\end{aligned}$$

Consider $x_2 = 0$,

$$\begin{aligned}
\textbf{Ack}\ \underline{x_1+1}\ \underline{0} &=_\beta (\lambda x.x\ (\lambda fy.y\ f\ (f\ \underline{1}))\ \textbf{Succ})\ \underline{x_1+1}\ \underline{0} \\
&=_\beta \underline{x_1+1}\ (\lambda fy.y\ f\ (f\ \underline{1}))\ \textbf{Succ}\ \underline{0} \\
&= (\lambda fx.f^{x_1+1}\ x)\ (\lambda fy.y\ f\ (f\ \underline{1}))\ \textbf{Succ}\ \underline{0} \\
&=_\beta (\lambda fy.y\ f\ (f\ \underline{1}))^{x_1+1}\ \textbf{Succ}\ \underline{0} \\
&=_\beta (\lambda fy.y\ f\ (f\ \underline{1}))\ ((\lambda fy.y\ f\ (f\ \underline{1}))^{x_1}\ \textbf{Succ})\ \underline{0} \\
&=_\beta \underline{0}\ ((\lambda fy.y\ f\ (f\ \underline{1}))^{x_1}\ \textbf{Succ})\ (((\lambda fy.y\ f\ (f\ \underline{1}))^{x_1}\ \textbf{Succ})\ \underline{1})\ \text{(by definition of iteration)} \\
&=_\beta ((\lambda fy.y\ f\ (f\ \underline{1}))^{x_1}\ \textbf{Succ})\ \underline{1} \\
&=_\beta \textbf{Ack}\ \underline{x_1}\ \underline{0}\ \text{(Since it is the same as step 2 for } x_1\text{)}
\end{aligned}$$

Consider the case $x_1 + 1$ and $x_2 + 1$, and let $F \triangleq ((\lambda fy.y\ f\ (f\ \underline{1}))^{x_1}\ \textbf{Succ})$,

$$\begin{aligned}
\textbf{Ack}\ \underline{x_1+1}\ \underline{x_2+1} &=_\beta \underline{x_2+1}\ F\ (F\ \underline{1})\ \text{(Following the same steps as above)} \\
&=_\beta F^{x_2+1}\ (F\ \underline{1}) \\
&=_\beta F^{x_2+2}\ \underline{1} \\
&=_\beta F\ (F^{x_2+1}\ \underline{1})\ \text{(We are trying to form the } \texttt{ack}(x_1,\ldots) \text{ part)} \\
&=_\beta F\ (\underline{x_2}\ F\ (F\ \underline{1})) \\
&=_\beta F\ ((\lambda fy.y\ f\ (f\ \underline{1}))\ F\ \underline{x_2}) \\
&=_\beta F\ (((\lambda fy.y\ f\ (f\ \underline{1}))^{x_1+1}\ \textbf{Succ})\ \underline{x_2}) \\
&=_\beta F\ (\textbf{Ack}\ \underline{x_1+1}\ \underline{x_2}) \\
&=_\beta \textbf{Ack}\ \underline{x_1}\ (\textbf{Ack}\ \underline{x_1+1}\ \underline{x_2})
\end{aligned}$$

Exercise 25 Give a definition of a function that is λ-definable but not primitive recursive. [**2011P6Q4 (d)**].

The Ackermann function is known to grow faster than any primitive recursive function, so it is not primitive recursive. In Exercise 11, we showed that this is RM-computable, so it is also λ-definable (or otherwise you can directly define it using lambda terms (see Exercise 26)).

Exercise 26 Attempt [**2010P6Q4 (d)**].

See official solution notes

Exercise 27 [**Correct composition**] Let I be the λ-term $\lambda x.x$.

(a) Show that $\underline{n}\ I =_\beta I$ holds for every Church numeral $\underline{n}$.

(b) Now consider $B \triangleq \lambda fgx.g\ x\ I\ (f\ (g\ x))$. Assuming the fact about normal order reduction mentioned on **Lecture 10 slide 35**, show that if partial functions $f, g \in \mathbb{N} \rightharpoonup \mathbb{N}$ are represented by closed λ-terms F and G respectively, then their composition $(f \circ g)(x) \equiv f(g(x))$ is represented by $B\ F\ G$.

(c) How does this solve the problem mentioned on **Lecture 11 slide 14**?

[**Exercise 12 in Lecturer's handout**]

(a) $\underline{n}\ I =_\beta (\lambda fx.f^n x)\ I =_\beta \lambda x.I^n x =_\beta \lambda x.x =_\beta I$, where we used that $I^n x =_\beta x$.

This can be proven more formally using induction. The bases case $I^0 x =_\beta x$ holds. Assume true for n, then $I^{n+1}\ x =_\beta I^n(Ix) =_\beta I^n x =_\beta x$.

(b) According to the slides, normal-order reduction always finds the β-nf if it exists. If $g\ \underline{x}$ is defined, then it evaluates to some $\underline{n}$ and so normal-order reduction gives:

$$(\lambda fgx.g\ x\ I\ (f\ (g\ x)))\ f\ g\ \underline{x} =_\beta g\ \underline{x}\ I\ (f\ (g\ \underline{x})) =_\beta \underline{n}\ I\ (f\ (g\ \underline{x})) =_\beta I\ (f\ (g\ \underline{x})) =_\beta f\ (g\ \underline{x}) =_\beta f\ \underline{n}$$

So the composition has a β-nf iff $f\ \underline{n}$ has a β-nf, as desired.

Otherwise, if $g\ \underline{x}$ is undefined, i.e. it does not have a β-nf, then when attempting normal-order reduction on $g\ \underline{x}\ I\ (f\ (g\ \underline{x}))$, we will not be able to find one.

(c) The problem with the way composition was defined in the slides was that when the outer function is undefined and the inner is defined, we could still end up with a defined function.

Lecture 12

Exercise 28

(a) Why do we need the fixed point combinator in showing that primitive recursion is λ-definable? How is it used?

(b) Define Curry's fixed point combinator Y and show that it satisfies the desired property.

(c) Define Turing's combinator and show that it satisfies the desired property. (See [**2015P6Q4 (c)**])

(d) Attempt [**2019P6Q6 (d),(e)**].

(e) Show that the `square` and `fact` are λ-definable. (See [**2011P6Q4 (c)**])

(a) The combinator Y is needed in order to make a recursive function call. More specifically, because it satisfies $Y\ M =_\beta M(Y\ M)$, given a λ-term M, we can apply it recursively. In implementing primitive recursion, we define $M \triangleq (\lambda z\vec{x}x.\mathbf{If}(\mathbf{Eq}_0 x)(F\vec{x})(G\ \vec{x}\ (\mathbf{Pred}\ x)(z\ \vec{x}\ (\mathbf{Pred}\ x))))$ and primitive recursion is defined as $Y\ M$.

See **Lecture 12 slide 15**.

(b) Curry's fixed point combinator Y is defined $Y \triangleq \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$.

$Y\ M \to (\lambda x.M\ (x\ x))(\lambda x.M\ (x\ x)) \to M\ ((\lambda x.M\ (x\ x))\ (\lambda x.M\ (x\ x)))$

and similarly,

$$M(Y\ M) = M((\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)))M) \to M((\lambda x.M(x\ x))(\lambda x.M(x\ x)))$$

Hence, $Y\ M =_\beta M(Y\ M)$.

See **Lecture 12 slide 11**.

(c) Turing's fixed point combinator is given by $\Theta \triangleq A\ A$, where $A \triangleq \lambda xy.y(x\ x\ y)$. It satisfies the desired property, since

$$\Theta\ M = A\ A\ M = (\lambda xy.y\ (x\ x\ y))\ A\ M \to M\ (A\ A\ M)$$

See **Lecture 12 slide 14**.

(d) As in the hint, consider the λ-term $M\ (Y\ K)$, where $K = (\lambda x.\mathbf{If}\ (M\ x)\ B\ A)$. Now assume that $M\ (Y\ K) =_\beta \mathbf{True}$, then

$$\begin{aligned} M\ (Y\ K) &=_\beta M\ (K\ (Y\ K)) \text{ (since } Y \text{ is a fixed-point combinator)} \\ &=_\beta M\ (K\ (Y\ K)) = M\ ((\lambda x.\mathbf{If}\ (M\ x)\ B\ A)\ (Y\ K)) \\ &=_\beta M\ (\mathbf{If}\ (M\ (Y\ K))\ B\ A) \\ &=_\beta M\ (\mathbf{If}\ (M\ \mathbf{True})\ B\ A) \text{ (by assumption)} \\ &=_\beta M\ B =_\beta \mathbf{False} \end{aligned}$$

which implies that $M\ (Y\ K) =_\beta \mathbf{False}$, which leads to a contradiction. Similarly, assume $M\ (Y\ K) =_\beta \mathbf{False}$,

$$\begin{aligned} M\ (Y\ K) &=_\beta M\ (\mathbf{If}\ (M\ (Y\ K))\ B\ A) \\ &=_\beta M\ (\mathbf{If}\ (M\ \mathbf{False})\ B\ A) \text{ (by assumption)} \\ &=_\beta M\ A =_\beta \mathbf{True} \end{aligned}$$

which implies that $M\ (Y\ K) =_\beta \mathbf{True}$, which leads to a contradiction.

(e) Square can be defined as $\lambda x.\mathbf{Mult}\ x\ x$.
The factorial function can be defined recursively: $M \triangleq \lambda zx.\mathbf{If}\ \mathbf{Eq}_0(x)\ \underline{1}\ (\mathbf{Mult}\ x\ (z\ (\mathbf{Pred} x)))$, where z is the recursive function. Hence, $\mathbf{Fact} \triangleq \lambda x.Y\ M\ x$.

Exercise 29

(a) Explain how fixed-point combinators are used in the λ-definition of minimisation.

(b) Deduce that every total recursive function is λ-definable. Collect the arguments and make an outline of the proof.

(a) Minimisation is defined as $\lambda x.Y\ (\lambda z\vec{x}x.\mathbf{If}\ (\mathbf{Eq}_0(F\ \vec{x}\ x))\ x\ (z\ \vec{x}\ (\mathbf{Succ}\ x)))\vec{x}\ \underline{0}$. The fixed-point combinator allows the inner λ to be applied recursively with access to the recursive function through variable z.

(b) We have show that the basic functions (projection, successor, zero) in Exercise 23(a), composition in Exercise 27, primitive recursion in Exercise 28(a) and minimisation in part (b) are λ-definable. Hence, every total recursive function is λ-definable.

Note: We have not shown the definition of minimisation for partial functions, hence we cannot make the conclusion that all partial recursive functions are representable in λ calculus. See Theorem 4.23 in Hindley and Seldin's "Lambda-calculus and combinators, an introduction" (2008).

Exercise 30 Give a high-level argument for why every λ-definable function is RM computable. (See [**2018P6Q6 (d)**])

$\boxed{\text{See official solution notes}}$

Exercise 31 Describe the *Church-Turing thesis*. Why is this not called a theorem? What examples did you come across in the lectures?

It is a high-level claim, that any formal definition of an algorithm will lead to model that is equivalent to Turing Machines (and λ-calculus, partial recursive functions, register machines). This means that the other formalisms will also have the same limitations (uncomputable functions and undecidable problems).