Computation Theory

Supervision 3 – Solutions

9. Lambda calculus

1. Given a set $\mathcal{V} = \{x, y, \ldots\}$ of variables, define the set $\mathcal{T}$ of λ-terms

   a) as an inductively defined set (see IA Formal Languages course).

   > One axiom for variables, and two rules:
   >
   > $$\frac{}{x \in \mathcal{T}}\ (x \in \mathcal{V}) \qquad \frac{M \in \mathcal{T}}{(\lambda x.\, M) \in \mathcal{T}}\ (x \in \mathcal{V}) \qquad \frac{M \in \mathcal{T} \qquad N \in \mathcal{T}}{(MN) \in \mathcal{T}}$$
   >
   > The $\in \mathcal{T}$ is often omitted.

   b) using Backus–Naur form (see IB Semantics course).

   > BNF often makes the set where variables come from, and set of terms being constructed implicit.
   >
   > $$M, N ::= x \mid (\lambda x.\, M) \mid (MN)$$

   c) using a recursive set comprehension (see Lecture 7 of the IB Logic course).

   > Hierarchical construction:
   >
   > $$\mathcal{T}_0 \triangleq \mathcal{V}, \quad \mathcal{T}_{k+1} \triangleq \{(\lambda x.\, M) \mid x \in \mathcal{V} \wedge M \in \mathcal{T}_k\} \cup \{(MN) \mid M, N \in \mathcal{T}_k\}, \quad \mathcal{T} \triangleq \bigcup_{k \in \mathbb{N}} \mathcal{T}_k$$
   >
   > Recursive set comprehension:
   >
   > $$\mathcal{T} \triangleq \mathcal{V} \cup \{(\lambda x.\, M) \mid x \in \mathcal{V} \wedge M \in \mathcal{T}\} \cup \{(MN) \mid M, N \in \mathcal{T}\}$$

2. a) Simplify the following λ-terms (as much as possible, but without evaluating them) using the notational conventions described on Slide 105:

   $$(\lambda x.\, ((ux)y)) \qquad (((\lambda u.\, (\lambda v.\, (vu)u))z)y) \qquad ((((\lambda x.\, (\lambda y.\, (\lambda z.\, ((xz)(yz)))))u)v)w)$$

   > Application is left-associative, dot after binding extends as far to the right as possible.
   >
   > $$\lambda x.\, uxy \qquad (\lambda uv.\, vuu)zy \qquad (\lambda xyz.\, xz(yz))uvw$$

   b) Expand the following simplified λ-terms (as much as possible) using the notational conventions described on Slide 105, inserting all parentheses and λ's:

   $$xyz(yx) \qquad \lambda u.\, u(\lambda x.\, y) \qquad \lambda xy.\, ux(yz)(\lambda v.\, vy)$$

> Note that the syntax includes parentheses around every compound lambda term, so the whole expression should be wrapped in parentheses.
>
> $$(((xy)z)(yx)) \qquad (\lambda u.\,(u(\lambda x.\,y))) \qquad (\lambda x.\,(\lambda y.\,(((ux)(yz))(\lambda v.\,(vy)))))$$

3. Give a recursive definition of the function $len(M)$ denoting the *length* of the λ-term M given by the total number of variables in M. For example, $len(x(\lambda y.\,yux)) = 5$.

> Straightforward recursive function that pattern-matches on the shape of the argument.
>
> $$\begin{aligned} len(x) &\triangleq 1 \\ len(\lambda x.\,M) &\triangleq 1 + len(M) \\ len(MN) &\triangleq len(M) + len(N) \end{aligned}$$

4. a) Define the *subterm relation* $M \sqsubseteq N$ by recursion on N. For example,

$$x \sqsubseteq \lambda y.\,ux \qquad \lambda x.\,y \sqsubseteq \lambda x.\,y \qquad xy \sqsubseteq (\lambda x.\,xy)z \qquad z \sqsubseteq x(\lambda z.\,y)$$

but $uv \not\sqsubseteq \lambda x.\,xu(vy)$.

> By recursion on N, we have three cases:
>
> - $M \sqsubseteq x$ if $M = x$ (and x is a variable)
> - $M \sqsubseteq \lambda x.\,N$ if $M = x$ or $M \sqsubseteq N$
> - $M \sqsubseteq NN'$ if $M \sqsubseteq N$ or $M \sqsubseteq N'$ The relation is also reflexive, so we have $M \sqsubseteq M$ for all M.

b) We say there is *an occurrence of* M *in* N if $M \sqsubseteq N$.

(i) Mark all occurrences of xy in $(xy)(\lambda x.\,xy)$.
(ii) Mark all occurrences of x in $(xy)(\lambda x.\,xy)$.
(iii) Mark all occurrences of xy in $\lambda xy.\,xy$.
(iv) Mark all occurrences of uv in $x(uv)(\lambda u.\,v(uv))uv$.
(v) Does $\lambda u.\,u$ occur in $\lambda u.\,uv$?

> 1) $(\underline{xy})(\lambda x.\,\underline{xy})$
> 2) $(\underline{x}y)(\lambda \underline{x}.\,\underline{x}y)$
> 3) Since $\lambda xy.\,xy = \lambda x.\,\lambda y.\,xy$, we only have one occurrence: $\lambda xy.\,\underline{xy}$.
> 4) If we add one omitted pair of parentheses, we get $(x(uv)(\lambda u.\,v(uv))u)v$, so the last "$uv$" is not actually an occurrence: $x(\underline{uv})(\lambda u.\,v(\underline{uv}))uv$.
> 5) No, since $\lambda u.\,uv = \lambda u.\,(uv)$, not $(\lambda u.\,u)v$.

5. Let M be the λ-term $\lambda xy.\,x(\lambda z.\,zu)y$.

a) What is the β-normal form of the term $N = M(\lambda vw.\,v(wb))(\lambda xy.\,yaz)$?

> We calculate the normal form by successive β reductions and substitutions. We may start by α-renaming the second argument term, but performing both substitutions at

once in the first step, any name clashes are avoided.

$$\begin{aligned} M(\lambda vw.\ v(wb))(\lambda xy.\ yaz) &=_\beta (\lambda xy.\ x(\lambda z.\ zu)y)(\lambda vw.\ v(wb))(\lambda xy.\ yaz) \\ &=_\beta (\lambda vw.\ v(wb))(\lambda z.\ zu)(\lambda xy.\ yaz) \\ &=_\beta (\lambda z.\ zu)((\lambda xy.\ yaz)b) \\ &=_\beta ((\lambda xy.\ yaz)b)u \\ &=_\beta (\lambda y.\ yaz)u \\ &=_\beta uaz \end{aligned}$$

b) Apply the simultaneous substitution $\sigma = [x/y, (\lambda xy.\ zy)/u]$ to M and N, and find the β-normal form of $N[\sigma]$.

Substitutions are only performed for *free* variables: even if the names match, bound variables are unaffected by substitutions and can always be arbitrarily renamed. To avoid the capture of z, we rename the bound z to w in M first.

$$M[\sigma] = (\lambda xy.\ x(\lambda w.\ w\underline{u})y)[x/y, (\lambda xy.\ zy)/u] = \lambda xy.\ x(\lambda w.\ w(\underline{\lambda xy.\ zy}))y$$

Since substitution distributes over application, we can apply it to the two arguments of N first – however, neither of them contain free occurrences of y or u, so the substitution is a no-op. Hence:

$$\begin{aligned} N[\sigma] &= M[\sigma](\lambda vw.\ v(wb))(\lambda xy.\ yaz) \\ &= (\lambda xy.\ x(\lambda w.\ w(\lambda xy.\ zy))y)(\lambda vw.\ v(wb))(\lambda xy.\ yaz) \end{aligned}$$

To find the normal form of $N[\sigma]$, we could evaluate the above expression. However, we can once again make use of the fact that substitution distributes over application, so the normal form of $N[\sigma]$ will be the normal form of N with σ applied to it.

$$N[\sigma] =_\beta (uaz)[x/y, (\lambda xy.\ zy)/u] = (\lambda xy.\ zy)az = zz$$

c) Give 2 terms α-equivalent to M. Give 2 other terms β-equivalent to M.

The α-equivalent terms will have the same structure, but renamed (bound) variables.

$$\lambda fa.\ f(\lambda z.\ zu)a \qquad \lambda xy.\ x(\lambda ☺.\ ☺u)y$$

The β-equivalent terms all evaluate to M (since M is in β-nf, there are no shorter β-equivalent terms). Remember that the rules allow for evaluation inside the body of a λ-binding. An infinite number of possibilities here of course – for "yeah okay fair enough" points you can recall that $=_\beta$ is reflexive so M is β-equivalent to M.

$$(\lambda x.\ x)M \qquad \lambda xy.\ (\lambda abc.\ bca)(\lambda z.\ zu)yx$$

h. We define η-equivalence as: $M =_\eta \lambda x.\ Mx$ for any λ-term M. Give a shorter and a longer term η-equivalent to M. What is the use of η-equivalence in functional programming?

A longer term can simply surround M with an application and a binding: $\lambda w.\,(\lambda xy.\,x(\lambda z.\,zu)y)w$. For a shorter term we notice that $M = \lambda x.\,\lambda y.\,x(\lambda z.\,zu)y$, and the body of the outermost binding has the right shape to be η-reduced: $\lambda y.\,x(\lambda z.\,zu)y =_\eta x(\lambda z.\,zu)$. Hence $M =_\eta \lambda x.\,x(\lambda z.\,zu)$. This is an example of a term which is in β-normal form, but not in η-normal form, but $\lambda x.\,x(\lambda z.\,zu)$ cannot be reduced any further with either method, so it is in $\beta\eta$-normal form.

In functional programming η-reduction corresponds to *pointfree programming*: a style of writing functions without mentioning (all of the) arguments, and instead concentrating on how the functions can be combined and composed. For instance, we can write a function to get the last element of a list as the head of its reverse:

```
let last xs = hd (rev xs)
```

However, we can use the o operator to compose hd and rev:

```
let last xs = (hd o rev) xs
```

We can rewrite this as a value equalling an anonymous function

```
let last = fun xs -> (hd o rev) xs
```

But the anonymous function is precisely of the form that admits η-reduction: it is a function that takes an argument and applies another function to the argument. Extensionally, the two functions are equal, so we can also write

```
let last = hd o rev
```

6. What are some differences between the lambda calculus as defined in this course, and the functional subset of L2 from the IB Semantics course?

The basic syntactic constructs are the same: we have terms e made up of variables x, functional bindings $\mathbf{fn}\ x : T \Rightarrow e$ and application $e_1\ e_2$. The main difference is that L2 is *typed* while the (untyped) λ-calculus presented in this course is not. This has many consequences, most notably a severe restriction on what kinds of terms we may construct: while the grammar of λ-terms is unrestricted (so e.g. xx is a perfectly reasonable term), in a simply typed setting every subterm is required to have a unique type (so self-application is not possible). Another difference is the evaluation strategy: in L2 we use call-by-value (reduce argument first, then substitute). In UTLC, the β-reduction is nondeterministic (albeit confluent), and also allows reduction in the body of a lambda term. We can restrict it to normal-order reduction (left-most, outer-most), which corresponds to call-by-name evaluation: application is performed without evaluating the argument.

10. Lambda-definable functions

1. Give a complete proof of the correctness of Church addition from Slide 119. *Hint:* a formal justification of one of the steps will require mathematical induction.

$$\mathbf{Plus}\,\underline{m}\,\underline{n} =_\beta \underline{m+n}$$

> The full calculation is given in the notes, the only step that requires a more formal proof is $f^m(f^n x) = f^{m+n}$ – we remedy that now.
>
> We prove that for all $m, n \in \mathbb{N}$ and λ-terms F and M, we have that $F^m(F^n M) = F^{m+n}M$ by mathematical induction on m.
>
> **Base case**: $m = 0$. We have $F^0(F^n M) = F^n M = F^{0+n}M$, as required.
>
> **Inductive step**: $m = k+1$. Assume that the IH holds for $m = k$: $F^k(F^n M) = F^{k+n}M$. Then we have: $F^{k+1}(F^n M) = F(F^k(F^n M)) = F(F^{k+n}M) = F^{(k+1)+n}M$, where we applied the IH in the second equality.

2. Define the λ-terms **Times** and **Exp** representing multiplication and exponentiation of Church numerals respectively. Prove the correctness of your definitions.

$$\mathbf{Times}\,\underline{m}\,\underline{n} =_\beta \underline{m \times n} \qquad \mathbf{Exp}\,\underline{m}\,\underline{n} =_\beta \underline{m^n}$$

> Addition applies the successor function n then m times. Multiplication applies the "apply the successor function n times" function m times.
>
> $$\mathbf{Times} \triangleq \lambda mn.\, \lambda f\,x.\, m(nf)x \qquad (=_\eta \lambda mn.\, \lambda f.\, m(nf))$$
>
> We can calculate (noting the difference between definitional equality and β-equivalence):
>
> $$\begin{aligned} \mathbf{Times}\,\underline{m}\,\underline{n} &=_\beta \lambda f\,x.\, \underline{m}(\underline{n}f)x \\ &=_\beta \lambda f\,x.\, (f^n)^m x \\ &= \lambda f\,x.\, f^{n\times m}\,x \\ &= \underline{n \times m} \end{aligned}$$
>
> The property we use in the third step is $(F^n)^m M = F^{n\times m}M$ for all terms F, M and naturals m, n. We prove it by induction on m:
>
> **Base case**: $m = 0$. We have $(F^n)^0 M = M = F^{n\times 0}$, as required.
>
> **Inductive step**: $m = k+1$. Assume that the IH holds for $m = k$: $(F^n)^k M = F^{n\times k}M$. Then:
>
> $$(F^n)^{k+1}M = F^n((F^n)^k M) =_{\text{IH}} F^n(F^{n\times k}M) = F^{n\times k+n}M = F^{n\times(k+1)}M$$
>
> where we made use of the addition property above in the third step.
>
> Exponentiation of Church numerals is remarkably simple, as their definition automatically

performs the "exponentiation":

$$\mathbf{Exp} \triangleq \lambda mn.\, \lambda f\, x.\, n\, m\, f\, x \qquad (=_\eta \lambda mn.\, n\, m)$$

We can calculate:

$$\begin{aligned} \mathbf{Exp}\, \underline{m}\, \underline{n} &=_\beta \lambda f\, x.\, \underline{n}\, \underline{m}\, f\, x \\ &=_\beta \lambda f\, x.\, (\underline{m})^n\, f\, x \\ &= \lambda f\, x.\, (\underline{m^n})\, f\, x \\ &= \lambda f\, x.\, f^{m^n}\, x \\ &= \underline{m^n} \end{aligned}$$

The property we use in the third step is $(\underline{m})^n\, M = \underline{m^n}\, M$ for all terms M and $m, n \in \mathbb{N}$. We proceed by induction on n:

Base case: $n = 0$. We have $(\underline{m})^0\, M = M = \underline{1}\, M = \underline{m^0}\, M$, as required.

Inductive step: $n = k+1$. Assume that the IH holds for $n = k$: $(\underline{m})^k\, M = \underline{m^k}\, M$. Then:

$$(\underline{m})^{k+1} M = \underline{m}\, ((\underline{m})^k M) =_{\text{IH}} \underline{m}\, ((\underline{m^k})\, M) = \mathbf{Times}\, \underline{m}\, \underline{m^k}\, M = \underline{m \times m^k}\, M = \underline{m^{k+1}}\, M$$

3. Show that the λ-term $\mathbf{Ack} \triangleq \lambda x.\, x\, T\, \mathbf{Succ}$, where $T \triangleq (\lambda f\, y.\, y\, f\, (f\, \underline{1}))$ represents Ackermann's function $ack \in \mathbb{N}^2 \to \mathbb{N}$. *Hint*: you will need to use nested induction; consider deriving a simplified form for the outer inductive case before starting the nested proof.

For brevity I will write A for $\mathbf{Ack}$ and S for $\mathbf{Succ}$. We need to show that for all naturals $m, n \in \mathbb{N}$, $A\, \underline{m}\, \underline{n} =_\beta \underline{ack(m,n)}$ by induction on m.

Base case: $m = 0$.

$$A\, \underline{0}\, \underline{n} = \underline{0}\, T\, S\, \underline{n} = T^0\, S\, \underline{n} = S\, \underline{n} = \underline{n+1} = \underline{ack(0,n)}$$

Inductive step. Assume

$$\forall n \in \mathbb{N}.\, A\, \underline{m}\, \underline{n} =_\beta \underline{ack(m,n)} \tag{IH1}$$

and prove that for all $n \in \mathbb{N}$, $A\, \underline{m+1}\, \underline{n} =_\beta \underline{ack(m+1,n)}$. Before attempting this proof, it's worth showing some simple lemmas. First, we calculate the partial application:

$$A\, \underline{m+1} =_\beta \underline{m+1}\, T\, S =_\beta T\, (T^m\, S) =_\beta T\, (\underline{m}\, T\, S) =_\beta T\, (A\, \underline{m}) \tag{†}$$

This gives us

$$\begin{aligned} A\, \underline{m+1}\, \underline{n} &=_\beta T\, (A\, \underline{m})\, \underline{n} && \text{(by †)} \\ &= (\lambda f\, y.\, y\, f\, (f\, \underline{1}))\, (A\, \underline{m})\, \underline{n} \\ &=_\beta \underline{n}\, (A\, \underline{m})\, (A\, \underline{m}\, \underline{1}) \\ &=_\beta \underline{n}\, (A\, \underline{m})\, \underline{ack(m,1)} && \text{(by IH1)} \end{aligned}$$

Thus we have that $A\,\underline{m+1}\,\underline{n} =_\beta \underline{n}\,(A\,\underline{m})\,\underline{ack(m,1)}$ and can proceed with a proof of

$$\forall n \in \mathbb{N}.\ A\,\underline{m+1}\,\underline{n} =_\beta \underline{ack(m+1,n)}$$

by (nested) induction on $n \in \mathbb{N}$.

Base case: $n = 0$. We have

$$A\,\underline{m+1}\,\underline{0} =_\beta \underline{0}\,(A\,\underline{m})\,\underline{ack(m,1)} =_\beta \underline{ack(m,1)}$$

and $ack(m+1,0) \triangleq ack(m,1)$ by definition.

Inductive step. Assume

$$A\,\underline{m+1}\,\underline{n} =_\beta \underline{ack(m+1,n)} \qquad \text{(IH2)}$$

We need to prove $A\,\underline{m+1}\,\underline{n+1} =_\beta \underline{ack(m+1,n+1)}$:

$$\begin{aligned}
A\,\underline{m+1}\,\underline{n+1} &=_\beta \underline{n+1}\,(A\,\underline{m})\,\underline{ack(m,1)} && \text{(by †)}\\
&=_\beta (A\,\underline{m})^{n+1}\,\underline{ack(m,1)} && \text{(def. of Church numerals)}\\
&= A\,\underline{m}\,(\underline{n}\,(A\,\underline{m})\,\underline{ack(m,1)}) && \text{(by def. of exponentiation)}\\
&=_\beta A\,\underline{m}\,(A\,\underline{m+1}\,\underline{n}) && \text{(by †)}\\
&=_\beta A\,\underline{m}\,\underline{ack(m+1,n)} && \text{(by IH2)}\\
&=_\beta \underline{ack(m,ack(m+1,n))} && \text{(by IH1 with } n = ack(m+1,n))\\
&=_\beta \underline{ack(m+1,n+1)} && \text{(by def. of Ackermann)}
\end{aligned}$$

All cases are covered, so the proof is finished. And what a proof it was.

4. Consider the following λ-terms:

$$\mathbf{I} \triangleq \lambda x.\, x \qquad \mathbf{B} \triangleq \lambda g f x.\, f\,x\,\mathbf{I}\,(g\,(f\,x))$$

a) Show that $\underline{n}\,\mathbf{I} =_\beta \mathbf{I}$ for every $n \in \mathbb{N}$.

Unsurprisingly, we prove this by induction on $n \in \mathbb{N}$. We also introduce an arbitrary λ-term M as the argument.

Base case: $n = 0$. We have $\underline{0}\,\mathbf{I}\,M =_\beta \mathbf{I}^0\,M = M =_\beta \mathbf{I}\,M$, as required.

Inductive step: $n = k+1$. Assume the IH holds for $n = k$: $\underline{k}\,\mathbf{I}\,M = \mathbf{I}\,M$. Now

$$\underline{k+1}\,\mathbf{I}\,M =_\beta \mathbf{I}\,(\underline{k}\,\mathbf{I}\,M) =_\beta \mathbf{I}\,(\mathbf{I}\,M) =_\beta \mathbf{I}\,M$$

b) Assuming the fact about normal order reduction mentioned on Slide 115, show that if partial functions $f, g : \mathbb{N} \rightharpoonup \mathbb{N}$ are represented by closed λ-terms F and G respectively, then their composition $(g \circ f)(x) \triangleq g(f(x))$ is represented by $\mathbf{B}\,G\,F$. Explain how this avoids the discrepancy with partial functions mentioned in Slide 125.

> Let F and G be closed λ-terms representing partial functions $f, g : \mathbb{N} \rightharpoonup \mathbb{N}$. We need to show that for all $n \in \mathbb{N}$ where $g(f(n))$ is defined, $\mathbf{B}\, G\, F\, \underline{n} =_\beta \underline{g(f(n))}$. If $g(f(n)) = m$ for some $m \in \mathbb{N}$, we must have a $k \in \mathbb{N}$ such that $f(n) = k$ and $g(k) = m$, so $F\, \underline{n} =_\beta \underline{k}$ and $G\, \underline{k} =_\beta \underline{m}$ by assumption. Then, using $\underline{k}\, \mathbf{I} =_\beta \mathbf{I}$ from (a):
>
> $$\mathbf{B}\, G\, F\, \underline{n} =_\beta F\, \underline{n}\, \mathbf{I}\, (G\, (F\, \underline{n})) =_\beta \underline{k}\, \mathbf{I}\, (G\, \underline{k}) =_\beta \mathbf{I}\, (G\, \underline{k}) =_\beta G\, \underline{k} =_\beta \underline{m}$$
>
> Now, if k exists but $g(k)$ is undefined, the last step will diverge. Since normal-order reduction will eventually try evaluating $G\, \underline{k}$ in $F\, \underline{n}\, \mathbf{I}\, (G\, \underline{k})$, the evaluation of $\mathbf{B}\, G\, F\, \underline{n}$ will also diverge. On the other hand, if $f(n)$ is undefined, then $F\, \underline{n}$ will diverge; since this is the outermost subterm of $F\, \underline{n}\, \mathbf{I}\, (G\, (F\, \underline{n}))$, normal-order reduction will also diverge. This differs from simply defining the composition as $G\, (F\, \underline{n})$, since normal-order reduction may not reach the evaluation of $(F\, \underline{n})$, e.g. if G is a constant function. The $\mathbf{B}$ combinator forces the evaluation of $F\, \underline{n}$ even if it wouldn't get encountered otherwise, and therefore it corresponds to composition of partial functions.

5. In the following questions you may use all of the λ-definable functions presented in the notes, as well as the terms you define as part of this exercise. You should explain your answers (possibly using some examples), but don't need to prove their correctness.

   a) Give a λ-term **Not** representing Boolean negation.

   > Since a Church Boolean corresponds to a branching operation, all we need to do is return false in the true branch, and true in the false branch:
   >
   > $$\mathbf{Not} \triangleq \lambda p.\ p\ \mathbf{False}\ \mathbf{True}$$

   b) Give λ-terms **And** and **Or** representing Boolean conjunction and disjunction.

   > Again, we use the standard definition of conjunction and disjunction in terms of Boolean branching, also exploiting the fact that Booleans are encoded as their own eliminators. For example, in $p \land q$, if p is true then we return q (and the truth value of q will determine the truth value of $\top \land q$), otherwise we return false. But, since p already denotes false (in its own 'else' branch), we just need to return p itself.
   >
   > $$\mathbf{And} \triangleq \lambda pq.\ p\, q\, p$$
   > $$\mathbf{Or} \triangleq \lambda pq.\ p\, p\, q$$

   c) Give a λ-term **Minus** representing truncated subtraction (i.e. $\mathbf{Minus}\, \underline{m}\, \underline{n} = 0$ if $m < n$).

   > Subtraction is repeated application of the predecessor function. Since $\mathbf{Pred}\ \underline{0} = \underline{0}$, it already performs the truncation.
   >
   > $$\mathbf{Minus} \triangleq \lambda mn.\ n\ \mathbf{Pred}\ m$$

   d) Give λ-terms **Eq**, **NEq**, **LT**, **LEq**, **GT**, **GEq**, representing the numeric comparison operations $=, \neq, <, \leq, >, \geq$ respectively. You can define them in any order you find most convenient.

> Most of these relations can be defined in terms of each other, so it's possible to define one as a starting point, then construct the rest from it. A good starting point is $\leq$ or $\geq$ which give you both the notion of ordering and equality. These are also simple to define in terms of truncated subtraction and zero check: $m \leq n$ iff $m \dot{-} n = 0$.
>
> $$\mathbf{LEq} \triangleq \lambda mn.\ \mathbf{Eq_0}\,(\mathbf{Minus}\ m\ n)$$
>
> From here, the remaining operations can be defined using Boolean operators. Here is one possible ordering, several others are possible:
>
> $$\begin{aligned} \mathbf{Eq} &\triangleq \lambda mn.\ \mathbf{And}\,(\mathbf{LEq}\ m\ n)\,(\mathbf{LEq}\ n\ m) \\ \mathbf{NEq} &\triangleq \lambda mn.\ \mathbf{Not}\,(\mathbf{Eq}\ m\ n) \\ \mathbf{GT} &\triangleq \lambda mn.\ \mathbf{Not}\,(\mathbf{LEq}\ m\ n) \\ \mathbf{GEq} &\triangleq \lambda mn.\ \mathbf{Or}\,(\mathbf{GT}\ m\ n)\,(\mathbf{Eq}\ m\ n) \\ \mathbf{LT} &\triangleq \lambda mn.\ \mathbf{Not}\,(\mathbf{GEq}\ m\ n) \end{aligned}$$

e) Define the λ-term **UCr** that represents the uncurrying higher-order function: if $\langle M, N\rangle$ denotes $\mathbf{Pair}\,M\,N$, then $\mathbf{UCr}\ F\ \langle M, N\rangle =_\beta F\,M\,N$.

> Nothing surprising here – use projections to get the two elements of the pair.
>
> $$\mathbf{UCr} \triangleq \lambda f\,p.\ f\,(\mathbf{Fst}\ p)\,(\mathbf{Snd}\ p)$$

f) Give a λ-term **MapPair** that applies a function to both elements of a pair: that is, $\mathbf{MapPair}\,F\ \langle M, N\rangle =_\beta \langle F\ M, F\ N\rangle$.

> Still nothing surprising!
>
> $$\mathbf{MapPair} \triangleq \lambda f\,p.\ \mathbf{Pair}\,(f\,(\mathbf{Fst}\ p))\,(f\,(\mathbf{Snd}\ p))$$

g) Give a λ-term **SqSum** which represents the function $\langle m, n\rangle \mapsto m^2 + n^2$.

> We can combine many of the functions we defined already to give a relatively clean, compositional definition.
>
> $$\mathbf{SqSum} \triangleq \lambda p.\ \mathbf{UCr}\ \mathbf{Plus}\,(\mathbf{MapPair}\,(\lambda k.\ \mathbf{Exp}\ k\ \underline{2})\ p)$$
>
> We can even go point-free by using the composition combinator **B** from above:
>
> $$\mathbf{SqSum} \triangleq \mathbf{B}\,(\mathbf{UCr}\ \mathbf{Plus})\,(\mathbf{MapPair}\,(\lambda k.\ \mathbf{Exp}\ k\ \underline{2}))$$

6. a) Explain why Curry's **Y** combinator is needed and how it works.

> Term definitions in the λ-calculus are simply abbreviations for larger λ-terms, and therefore they cannot be self-referential (otherwise trying to expand the term in its own definition would lead to an infinite expansion). This prevents us from writing "recursive" definitions such as $\mathbf{Fact}\ n \triangleq n \times (\mathbf{Fact}\ (n-1))$: all subterms of a term

should be well-defined. However, this doesn't mean that we cannot define recursive functions in the λ-calculus. We first note that a recursively defined term would have the general form $M \triangleq F\,M$, where the body of the definition of M itself takes the term M as an argument (e.g. $\mathbf{Fact} \triangleq (\lambda f.\, \lambda m.\, m \times f(m-1))\, \mathbf{Fact}$[a]). We can instead state this in terms of β-equivalence (as a *specification*, rather than *definition*):

$$M =_\beta F\,M$$

This form characterises M as a *fixed point* of F, where a fixed point of a function f is an argument x such that $f(x) = x$. Fixed points and recursion go hand-in-hand, since $M =_\beta F\,M =_\beta F(F\,M) =_\beta F(F \cdots (F\,M) \cdots)$, giving rise to the repetitive behavior that we are trying to capture. Hence, to define a term M that satisfies the "recursive" specification $M =_\beta F\,M$, we need to find a fixed point of F. The fixed point of the λ-term $(\lambda f\,m.\, m \times f(m-1))$ is precisely the factorial function.

The next question is: how do we find the fixed point of a λ-term? We can use *fixed-point combinators*, which are higher-order λ-terms that take a function and return its fixed point. One of the simplest fixed-point combinators is Curry's $\mathbf{Y}$ combinator, which satisfies the required fixed point property (stating that $\mathbf{Y}F$ is a FP of F):

$$\mathbf{Y}\,F =_\beta F(\mathbf{Y}F)$$

Curry defined $\mathbf{Y}$ as follows:

$$\mathbf{Y} \triangleq \lambda f.\, (\lambda x.\, f(xx))\, (\lambda x.\, f(xx))$$

It is crucially dependent on *self-application*, which is possible in the absence of types, but becomes impossible in a typed setting. $\mathbf{Y}$ is therefore slightly "magical" and doesn't correspond obviously to a natural computational construction, but it nevertheless satisfies the fixed point property above (see Slide 135). The $\mathbf{Y}$ combinator now allows us to define recursion indirectly: given the "intended" definition $M \triangleq F\,M$, we can define the λ-term that satisfies this as the fixed point $\mathbf{Y}\,F$ of F. For example,

$$\mathbf{Fact} \triangleq \mathbf{Y}\,(\lambda f\,m.\, m \times f(m-1))$$

[a] The base case in the definitions will be omitted for brevity – of course they are important, and are easy to define with $\mathbf{If}$ and $\mathbf{Eq}_0$.

b) Give a λ-term which is β-equivalent to the $\mathbf{Y}$ combinator, but only uses it's f argument once. *Hint*: see if you can exploit the symmetry of the $\mathbf{Y}$ combinator.

We can notice that the body of $\mathbf{Y} \triangleq \lambda f.\, (\lambda x.\, f(xx))\, (\lambda x.\, f(xx))$ is the same term $(\lambda x.\, f(xx))$ repeated twice. We can exploit this using the "self-application combinator" $\lambda x.\, xx$ to get $\mathbf{Y}'$, which reduces to $\mathbf{Y}$ in one step.

$$\mathbf{Y}' \triangleq \lambda f.\, (\lambda x.\, xx)\, (\lambda x.\, f(xx))$$

c) Define the λ-term $\mathbf{Fact}$ that computes the factorial of a Church numeral.

We expand our previous, informal example of fixpoint recursion. The λ-term **Fact** should satisfy the following equivalence:

$$\mathbf{Fact} =_\beta \lambda n.\ \mathbf{If}\,(\mathbf{Eq_0}\ n)\ \underline{1}\ (\mathbf{Times}\ n\ (\mathbf{Fact}\ (\mathbf{Pred}\ n)))$$

To make this into a well-formed term definition, we need to abstract out the recursive call, then "tie the knot" using the **Y** combinator.

$$\mathbf{Fact} \triangleq \mathbf{Y}\left(\lambda f.\ \lambda n.\ \mathbf{If}\,(\mathbf{Eq_0}\ n)\ \underline{1}\ (\mathbf{Times}\ n\ (f\,(\mathbf{Pred}\ n)))\right)$$

d) Define the λ-term **Fib** such that $\mathbf{Fib}\,\underline{n} =_\beta \underline{F_n}$ where F_n is the n^{th} Fibonacci number defined recursively as $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$.

Similarly, we start with the recursive specification, then turn it into a definition.

$$\mathbf{Fib} =_\beta \lambda n.\ \mathbf{If}\,(\mathbf{LEq}\ n\ \underline{1})\ n\ (\mathbf{Plus}\ (\mathbf{Fib}\ (\mathbf{Pred}\ n))\ (\mathbf{Fib}\ (\mathbf{Minus}\ n\ \underline{2})))$$

Rewriting with the fixed point combinator, we notice that the recursive call is made twice, which is captured by the repeated use of the higher-order argument f.

$$\mathbf{Fib} \triangleq \mathbf{Y}\left(\lambda f.\ \lambda n.\ \mathbf{If}\,(\mathbf{LEq}\ n\ \underline{1})\ n\ (\mathbf{Plus}\ (f\,(\mathbf{Pred}\ n))\ (f\,(\mathbf{Minus}\ n\ \underline{2})))\right)$$