Computation Answers 8: Lambda Calculus Part 2

1. (**Recap Question.**)

   (a) The result of performing $(\lambda x.xa)[x/a]$ should be something like $\lambda b.bx$.

   This is a "capture avoiding substitution". Recall slide 17 in your lecture notes. The only rules on that slide that can possibly apply are:

   $$\begin{array}{rcl} (\lambda x.N')[M/x] & = & \lambda x.N' \\ (\lambda y.N')[M/x] & = & \lambda z.N'[z/y][M/x] \\ \text{where} & & x \neq y, z \notin FV(M) \cup (FV(N') - \{y\}) \cup \{x\} \end{array}$$

   We can't use the first of these rules because it requires that we be substituting the variable that is bound by the lambda term. We have distinct x and a. We must therefore use the second rule.

   The second rule requires that we choose some z such that $z \notin FV(M) \cup (FV(N') - \{y\}) \cup \{x\}$ – which in this specific case is the same as $z \notin FV(x) \cup (FV(xa) - \{x\}) \cup \{a\}$. If we choose b as the name for this new variable then we get

   $$\begin{array}{rcl} (\lambda x.xa)[x/a] & = & \lambda b.(xa)[b/x][x/a] \\ & = & \lambda b.(ba)[x/a] \\ & = & \lambda b.bx \end{array}$$

   We could just as easily choose c, y or z. The two variable names we're absolutely not allowed to pick are x and a. Because of this restriction, no matter what variable name we pick, the results are all guaranteed to be α-equivalent to each other. If this seems opaque, consider what would happen if we lifted the restriction, and picked one of the two illegal values for z.

   (b) This should reduce to something α-equivalent to $\lambda c.c(\lambda b.bx)$

   The first and only beta reduction step requires

   $$(\lambda a.(\lambda x.xa))(\lambda b.bx) \longrightarrow_\beta (\lambda x.xa)[(\lambda b.bx)/a]$$

   We are now faced with a substitution problem similar to part (a). x is free in $(\lambda b.bx)$ but bound in $(\lambda x.xa)$. Before we can replace a with $\lambda b.bx$, we must rename the x in $(\lambda x.xa)$ to something that is not free in $(\lambda b.bx)$. If we choose c as our new variable name, then we get:

   $$\begin{array}{rcl} (\lambda x.xa)[(\lambda b.bx)/a] & = & \lambda c.(xa)[c/x][(\lambda b.bx)/a] \\ & = & \lambda c.(ca)[(\lambda b.bx)/a] \\ & = & \lambda c.c(\lambda b.bx) \end{array}$$

   If this is confusing, think about what would happen if we didn't rename x before performing the substitution. Would the β reduction preserve your intuitive understanding of the meaning of the function?

2. (**SKI Combinators.**)

$$
\begin{aligned}
&SKK \\
&= (\lambda xyz.\ (xz)(yz))KK \\
&\to (\lambda yz.\ (Kz)(yz))K \\
&\to \lambda z.\ (Kz)(Kz) \\
&= \lambda z.\ ((\lambda xy.\ x)z)(Kz) \\
&\to \lambda z.\ (\lambda y.\ z)(Kz) \\
&\to \lambda z.\ z
\end{aligned}
$$

Notice that we have shown that SKK is β-equivalent to I. Given our definitions of S and K, we could define $I \triangleq SKK$ (or indeed $I \triangleq SKS$ – can you see why?)

In fact, SKI and application are all we need to define *any* computable function. This is called the SKI combinator calculus, and it is Turing-complete.

3. (**Fixed Points**.)

$$
\begin{aligned}
&YM = ZZM = (\lambda zx.x(zzx))ZM \\
&\to (\lambda x.x(ZZx))M \\
&\to M(ZZM) = M(YM)
\end{aligned}
$$

What we have created is a "fixed-point" combinator. This means that YM will reduce to a value y such that $y = My$. We can say that "YM is a fixed-point of M".

This is useful for modelling recursion in the lambda calculus. For example, consider a function F (in a language like haskell) for calculating a factorial:

$$Fn \triangleq \text{ if } (n = 0) \text{ then } 1 \text{ else } n * (F(n-1))$$

We can represent numbers in the lambda calculus using Church encoding, and there are similar encodings for conditionals, multiplication, subtraction and equality testing. The problem with representing F in the lambda calculus is that it seems to need to refer to itself in its definition. We can circumvent this problem using the fixed point combinator, by defining a function G which performs a single step in the factorial calculation:

$$G \triangleq \lambda fn.\ \text{ if } (n = 0) \text{ then } 1 \text{ else } n * (f(n-1))$$

The function G takes as its first argument a function f. G performs the first step in calculating the factorial function, and then defers to f for the rest of the calculation. Therefore, if f is the factorial function, then Gf will also be a factorial function. In other words, the factorial function is a fixed point of G. We can show this using β-reduction:

$$
\begin{array}{rcll}
(YG)n & \twoheadrightarrow & (G(YG))n & \text{As above} \\
 & = & ((\lambda fn.\ \text{ if } (n=0) \text{ then } 1 \text{ else } n * (f(n-1)))(YG))n & \\
 & \to & (\lambda n.\ \text{ if } (n=0) \text{ then } 1 \text{ else } n * ((YG)(n-1)))n & \\
 & \to & \text{ if } (n=0) \text{ then } 1 \text{ else } n * ((YG)(n-1)) &
\end{array}
$$

If we abbreviate (YG) to F, then we have the recursive factorial function we were attempting to define.

[Aside: There are infinitely many fixed point combinators. The one we have defined is a variant of the Y combinator. If any of you are fans of Paul Graham's Seed-funding company `http://ycombinator.com/`, this is where he got the name from.]

4.
$$(\texttt{let}\ x = M\ \texttt{in}\ N) \triangleq (\lambda x.\, N)M$$

While it might be tempting to define it as simply $N[M/x]$, this involves computing on N, which makes the definition non-parametric. The above definition simply embeds the terms.

5. (a)
$$\mathsf{pair} \triangleq \lambda v_1\ v_2.\, (\lambda p.\, p\ v_1\ v_2)$$

(b) Given the pair $\lambda p.\, p\ v_1\ v_2$, we have

$$\begin{aligned}(\lambda p.\, p\ v_1\ v_2)(\lambda w_1\ w_2.\, w_1) &\to (\lambda w_1\ w_2.\, w_1)\ v_1\ v_2 \\ &\to (\lambda w_2.\, v_1)\ v_2 \\ &\to v_1\end{aligned}$$

(up to alpha conversion) so we want fst to be a function that applies its argument (the pair) to the term $(\lambda w_1\ w_2.\, w_1)$:

$$\mathsf{fst} \triangleq \lambda q.\, q(\lambda w_1\ w_2.\, w_1)$$

(c)
$$\mathsf{snd} \triangleq \lambda q.\, q(\lambda w_1\ w_2.\, w_2)$$

6. (a) Recall that $\underline{n} \triangleq \lambda f\ x.\, \underbrace{f(\ldots(f}_{n}\ x)\ldots)$.

$$\begin{aligned}\texttt{Succ}\ \underline{n} &= (\lambda r.\, (\lambda c_1\ c_2.\, c_1\ (r\ c_1\ c_2)))(\lambda f\ x.\, \underbrace{f(\ldots(f}_{n}\ x)\ldots)) \\ &\to \lambda c_1\ c_2.\, c_1\ ((\lambda f\ x.\, \underbrace{f(\ldots(f}_{n}\ x)\ldots))c_1 c_2) \\ &\to \lambda c_1\ c_2.\, c_1\ ((\lambda x.\, \underbrace{c_1(\ldots(c_1}_{n}\ x)\ldots))c_2) \\ &\to \lambda c_1\ c_2.\, c_1\ (\underbrace{c_1(\ldots(c_1}_{n}\ c_2)\ldots)) \\ &=_\alpha \lambda f\ x.\, \underbrace{f(\ldots(f}_{n+1}\ x)\ldots) \\ &= \underline{n+1}\end{aligned}$$

(b) i.
$$\texttt{True} \triangleq \lambda c_1\ c_2.\, c_1 \qquad \texttt{False} \triangleq \lambda c_1\ c_2.\, c_2$$

ii. For $\texttt{not}$, we want $\lambda b.\, \texttt{if}\ b\ \texttt{then False else True}$, which encodes as

$$\texttt{not} \triangleq \lambda b.\, b(\lambda c_1\ c_2.\, c_2)(\lambda c_1\ c_2.\, c_1)$$

For $\texttt{or}$, we want $\lambda b_1\ b_2.\,\texttt{if}\ \ b_1\ \ \texttt{then True else}\ \ b_2$, which encodes as

$$\texttt{or} \triangleq \lambda b_1\ b_2.\, b_1\ (\lambda c_1\ c_2.\, c_1)\ b_2$$

For $\texttt{and}$, we want $\lambda b_1\ b_2.\,\texttt{if}\ \ b_1\ \ \texttt{then}\ \ b_2\ \ \texttt{else False}$, which encodes as

$$\texttt{and} \triangleq \lambda b_1\ b_2.\, b_1\ b_2\ (\lambda c_1\ c_2.\, c_2)$$

(c) i.

$$\texttt{Left} \triangleq \lambda a.\, \lambda c_1\ c_2.\, c_1\ a \qquad \texttt{Right} \triangleq \lambda a.\, \lambda c_1\ c_2.\, c_2\ a$$

ii. Notice that $\texttt{Left}\ \ n = \lambda c_1\ c_2.\, c_1\ n$ and $\texttt{Right}\ \ n = \lambda c_1\ c_2.\, c_2\ n$, so what we want is a function which applies the identity function to n in the former case, and the $\texttt{Succ}$ function in the latter case:

$$\texttt{f} \triangleq \lambda x.\, x(\lambda y.\, y)(\lambda r.\, (\lambda c_1\ c_2.\, c_1\ (r\ c_1\ c_2)))$$