# Computation Exercises 8: Lambda Calculus Part 2

1. (**Recap Question.**)

   (a) Perform the following substitution: $(\lambda x.xa)[x/a]$

   (b) Reduce the following term to its normal form: $(\lambda a.(\lambda x.xa))(\lambda b.bx)$

2. (**SKI Combinators.**)
   Let $S \triangleq \lambda xyz.(xz)(yz)$ and $K \triangleq \lambda xy.x$. Reduce $SKK$ to normal form. (Hint: This can be messy if you are not careful. Keep the abbreviations $S$ and $K$ around as long as you can and replace them with their corresponding $\lambda$-terms only if you need to. This makes it much easier)

   [Aside:] The more eagle-eyed among you may notice that $S$ and $K$ form two thirds of the $SKI$ combinator calculus mentioned in Question 13 of Exercise Sheet 3. The third combinator is of course $I \triangleq \lambda x.x$

3. (**Fixed Points**.)
   Let $Z \triangleq \lambda zx.x(zzx)$ and let $Y \triangleq ZZ$. By performing a few $\beta$-reductions, show that for any term $M$, we have $YM =_\beta M(YM)$.

4. (**Let**.)

   Languages like Haskell and ML have a construct `let` $x = M$ `in` $N$, which reduces to $N[M/x]$. How can such a construct be expressed in the $\lambda$-calculus?

5. (**Pairs**.)

   Given two $\lambda$-terms, $v_1, v_2$, the pair of the two terms can be expressed in the $\lambda$-calculus as $\lambda p.\, p\ v_1\ v_2$ (where $p$ does not occur free in $v_1$ or $v_2$). Define the following functions as $\lambda$-terms:

   (a) pair, which takes two $\lambda$-terms and constructs the pair of them;

   (b) fst, which returns the first value in a pair;

   (c) snd, which returns the second value in a pair.

6. (**Datatypes.**)

   You should be familiar with Haskell datatypes, which are defined using the `data` keyword. For example, the following is a way of representing the natural numbers as a Haskell datatype:

```
data Nat = Succ Nat | Zero
```

   This datatype defines two constructors: `succ`, with one argument, which is recursive (it has type `Nat`, which is the type being defined), and `zero` with no arguments. Other datatypes, such as `Pair` have constructors with non-recursive arguments:

```
data Pair a b = Pair a b
```

(Note that while the type variables `a` and `b` *could* be instantiated to `Pair` types, they do not have to be, so they are not recursive.)

We have seen Church's encoding of natural numbers in the $\lambda$-calculus. This can also be applied to encode other simple recursive datatypes. Suppose that we have a datatype with $n$ constructors. If the $i$th constructor takes $j$ recursive arguments and $k$ non-recursive arguments, then it is represented as the $\lambda$-term:

$$\lambda r_1 \ldots r_j \; a_1 \ldots a_k . \, (\lambda c_1 \ldots c_n . \, c_i \; (r_1 \; c_1 \ldots c_n) \ldots (r_j \; c_1 \ldots c_n) \; a_1 \ldots a_k)$$

Thus, for encoding the natural numbers we have

$$\texttt{Succ} \triangleq \lambda r . \, (\lambda c_1 \; c_2 . \, c_1 \; (r \; c_1 \; c_2)) \qquad \texttt{Zero} \triangleq \lambda c_1 \; c_2 . \, c_2$$

(a) Notice that $\texttt{Zero} =_\alpha \underline{0}$. Show that these constructors really correspond to the Church numerals from the lectures by establishing that $\texttt{Succ} \; \underline{n} \twoheadrightarrow \underline{n+1}$ for all numbers $n$.

(b) i. Give the constructors for the datatype `Bool`, given in Haskell by

```
data Bool = True | False
```

ii. The Haskell construct **if** $b$ **then** $M$ **else** $N$ can be represented simply as $b \; M \; N$, where $b$ is the Church encoding of a Boolean. Use this to define functions `not`, `or` and `and` that operate on Boolean values with the expected results.

(c) i. Give the constructors for the datatype `Either`, given in Haskell by

```
data Either a b = Left a | Right b
```

(Note that the arguments of the constructors are non-recursive.)

ii. Define a lambda term `f` corresponding to the following Haskell function:

```
f (Left n) = n
f (Right m) = Succ m
```