## Problem Set 4

1. Reduce the following lambda terms to normal form:

   (a) $(\lambda xyz.zyx)aa(\lambda pq.q)$
   (b) $(\lambda yz.zy)((\lambda x.xxx)(\lambda x.xxx))(\lambda w.I)$
   (c) $SKSKSK$
   (d) $(\lambda x \lambda y \lambda z.xz(yz))(\lambda x.x)(\lambda x.x)x$
   (e) $(\lambda yz.zy)((\lambda x.xxx)(\lambda x.xxx))(\lambda w.\lambda x.x)$.

   Here $I = \lambda x.x$, $K = \lambda xy.x$ and $S = \lambda xyz.xz(yz)$.

2. Write a lambda term $f$ such that $f\ \overline{m}\ \overline{n} = \overline{m^n}$.

3. Recollect the relation between Church Numerals $\overline{n}$ and $foldn$. If $f_{\texttt{Nat}}$ and $id_{\texttt{Nat}}$ represent values (including function values) in the `Nat` domain, and $\overline{f}$ and $\overline{id}$ are the same functions expressed in lambda notation, then:

   $$foldn\ f_{\texttt{Nat}}\ id_{\texttt{Nat}}\ (Succ^n\ Zero) = \overline{n}\ \overline{f}\ \overline{id}$$

   Drawing an analogy, can you find a representation for lists in lambda calculus? In particular, what would be the representations of *cons* and *nil*? In other words, if Church numerals gave you a $foldn$-like behaviour, then the list representation through lambda calculus should give you a $foldr$ like behaviour. Test your answer by writing the *append* function in lambda calculus.

4. Write the following functions from $Nat$ to $Nat$: (a) $\lceil n/2 \rceil$ (b) $\lceil log_2 n \rceil$. In both cases assume that $n > 0$. Use Haskell notation. You can use any function whose lambda representation has been discussed during the course.

5. Imagine a typeless recursionless Haskell. How would you express the infinite list `[1,1,1,...]` in this language?

6. We want to express the relation $gt (>)$, as a lambda term.

   $true = \lambda x \lambda y.x$
   $false = \lambda x \lambda y.y$
   $pred = assume\ as\ given$
   $gt =$ ______
   $not =$ ______
   $if\_then\_else =$ ______
   $iszero =$ ______
   $sub =$ ______

7. Define a function *lambdafib* as a lambda term that will yield the Church representation of the $n$th fibonnacci number. Don't use $Y$.

8. Define *lambdafib* once again, *using* $Y$ this time.

9. Show that $Y_{funny}$ defined below is a fixed point combinator:

   $T = \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$
   $Y_{funny} = TTTTTTTTTTTTTTTTTTTTTTTTTT$

10. Write a function $max$ in lambda calculus which takes two Church numerals and returns the greater numeral. Do not use the $Y$ combinator.

11. Can you write a lambda term which will taken a Church numeral $\overline{n}$ and return the Church numeral corresponding to the smallest number whose factorial is larger than $n$.

12. Consider the datatype defined by: `data Nat = Zero | Succ Nat`. Write a function `mysqrt :: Nat -> Nat` which finds the integer square root of a number. The integer square root of $n$ is the largest integer whose square is less than or equal to $n$. Some helper functions are given:

```
foldn f id Zero = id
foldn f id (Succ n) = f (foldn f id n)
isZero n = n == Zero
mypred n = fst (foldn (\(a,b) -> (b, Succ b)) (Zero, Zero) n)
sub n m = foldn  mypred  n m
add n m = foldn  Succ  n m
mult n m = foldn (add n) Zero m
sqr x = mult x x
le n m = ___
eq n m = ____ && ____
lt n m = ____
mysqrt n = fst (foldn f ____  n)
             where f ____ = _____
```

   You are allowed to use the following from Haskell: `if`, boolean operators and pairing. There should be no recursion; all recursion should be through `foldn`.

13. Define using $foldn$ a function $f :: Nat \to Nat$ such that $f\ n$ is the highest $m$ which satisfies $2^m \leq n$. As an example, $f\ 15$ is 3.

14. Find closed terms $F$ such that

    (a) $F\ x = F$ . This term can be called the 'eater'.

    (b) $F\ x = x\ F$

15. Find a lambda term $F$ such that for all closed lambda terms $M$, $N$ and $L$, $F\,M\,N\,L = N(\lambda x.M)(\lambda yz.yLM)$.

16. We have seen the following representation of lists in lambda calculus:

$$\begin{aligned} \overline{nil} &= \lambda f \lambda id\,.\,f \\ \overline{cons} &= \lambda x \lambda xs \lambda f \lambda id\,.\,f\;x\,(xs\,f\;id) \end{aligned}$$

   Define $\overline{map}$ in terms of such a list representation. Do not use $Y$ or recursion.

17. You would have heard me mentioning that the combinators **S**, **K** and **I** are enough to write any program that you could write in Haskell. These combinators are defined as:

$$
\begin{aligned}
\mathbf{S} &= \lambda x \lambda y \lambda z . x\, z\, (y\, z) \\
\mathbf{K} &= \lambda x \lambda y . x \\
\mathbf{I} &= \lambda x . x
\end{aligned}
$$

Let us see why this is so through an example. Later, you have to generalize this example to define a scheme to translate any closed lambda term[1] to **SKI** combinators. Carefully observe the steps to translate $(\lambda x.x\ (\lambda y.xy))(\lambda z \lambda x.xz)$ to an equivalent term that uses **SKI** combinators only.

$$
\begin{aligned}
& \underline{(\lambda x.x\ (\lambda y.xy))}(\lambda z \lambda x.xz) \\
& \{\text{Considering the underlined term of the application}\} \\
= \;& \underline{\lambda x.x\ (\lambda y.xy)} && \tag{1} \\
= \;& \mathbf{S}\ \underline{(\lambda x.x)}\ (\lambda x.\underline{(\lambda y.xy)}) && \text{a. Why is (1) = (2)?} \tag{2} \\
= \;& \mathbf{S}\ \mathbf{I}\ (\lambda x.(\mathbf{S}\ \underline{(\lambda y.x)}\ \underline{(\lambda y.y)})) && \tag{3} \\
= \;& \mathbf{S}\ \mathbf{I}\ (\lambda x.(\mathbf{S}\ (\mathbf{K}\ x)\ \mathbf{I})) && \text{b. Why is (3) = (4)?} \tag{4} \\
= \;& \mathbf{S}\ \mathbf{I}\ \underline{(\lambda x.(\mathbf{S}\ (\mathbf{K}\ x))\ \mathbf{I})} \\
= \;& \mathbf{S}\ \mathbf{I}\ (\mathbf{S}\ (\lambda x.(\mathbf{S}\ (\mathbf{K}\ x)))\underline{(\lambda x.\mathbf{I})}) \\
= \;& \mathbf{S}\ \mathbf{I}\ (\mathbf{S}\ \underline{(\lambda x.(\mathbf{S}\ (\mathbf{K}\ x)))}(\mathbf{K}\ \mathbf{I})) \\
= \;& \mathbf{S}\ \mathbf{I}\ (\mathbf{S}\ (\mathbf{S}\ (\lambda x.\mathbf{S})\ \underline{(\lambda x.(\mathbf{K}\ x))})(\mathbf{K}\ \mathbf{I})) \\
= \;& \mathbf{S}\ \mathbf{I}\ (\mathbf{S}\ (\mathbf{S}\ \underline{(\lambda x.\mathbf{S})}\ (\mathbf{S}\ \underline{(\lambda x.\mathbf{K})}\ \underline{(\lambda x.x)}))(\mathbf{K}\ \mathbf{I})) \\
= \;& \mathbf{S}\ \mathbf{I}\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{S})\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{K})\ \mathbf{I}))(\mathbf{K}\ \mathbf{I}))
\end{aligned}
$$

c. In a similar manner, convert the right term $\lambda z \lambda x.xz$ into a term with **SKI** combinators.

d. What is the conversion of the original term $(\lambda x.x\ (\lambda y.xy))(\lambda z \lambda x.xz)$?

e. Complete the following translation scheme to translate a closed lambda term to an equivalent term made up of just **SKI** combinators :

i. $translate(M\ N) =$ ______

ii. $translate(\lambda x.M\ N) =$ ______

iii. $translate(\lambda x.x) =$ ______

iv. $translate(\lambda x.y) =$ ______

v. $translate(\lambda x.\mathbf{C}) =$ ______ $\quad$ **C** is one of **S**, **K** or **I**

vi. $translate(\lambda x.M) =$ ______ $\quad$ otherwise

(15 Marks)

[1] A closed lambda term is one which has no free variables