1. Draw the parse tree and the Stoy diagram of the λ-expression below, and on the parse tree, indicate each occurrence of an identifier as free or bound:

\[ (z \ (\lambda \ y \ ((\lambda \ z \ (x \ y)) \ z))) \]

2. Reduce the following expression to normal form if possible, otherwise indicate that a normal form does not exist:

\[ (y \ (\lambda \ x \ (x \ y))) \]

3. Write a pure lambda expression to define the macro \( M5? \) which accepts a single Church numeral as argument and returns a boolean TRUE or FALSE according as the integer corresponding to the Church numeral is a multiple of five or not. Do not use recursion or the Y-combinator.
4. What is the value of the following Scheme expression, assuming static scoping and applicative order evaluation?

\[
(\text{let } ((x 5))
  (\text{let } ((x 6) (f (\lambda (y z) (* y (- z x)))))
    (f x 8)))
\]

5. In class we have given a desugaring for the single \textit{let} form \((\text{let } I=E_a E_b)\). Give a desugaring for the Scheme multiple \textit{let} form

\[
(\text{let } ((I_1 E_1) \ldots (I_n E_n)) E_b)
\]

where no \(I_j\) can be used in any \(E_k\) unless \(I_j\) is bound in an outer scope, in which case it takes its value from that outer binding.
In class we saw how to desugar a single-parameter (\texttt{rec I E}) using the Y-combinator. We wish to extend this to a multi-parameter \texttt{letrec} form

\[ \texttt{letrec ((I1 E1)...(In En)) Eb} \]

where each \( E_k \) is a single-argument \texttt{proc} and mutual recursion is freely permitted (i.e., any \( I_j \) can be used in any \( E_k \) and does \textbf{not} take its value from an outer binding). Give a syntactic translation from \texttt{letrec} into core FLK without \texttt{rec}. 

\[ \text{4} \]
7. In Scheme, a recursive Fibonacci function may be written as follows:

```
(define (fib n)
  (cond
   ((= n 0) 0)
   ((= n 1) 1)
   (else (+ (fib (- n 1)) (fib (- n 2)))))
)
```

This is not tail-recursive, because the first `fib` call has to remember that the second `fib` and the addition are pending operations, and the second `fib` call must remember the pending addition. Convert the above code into continuation passing style by adding a continuation argument.