Review

Lambda Calculus

- Basic Syntax
    - know pure lambda calculus syntax
        - function abstraction
        - function application
        - variable
    - know Haskell Syntax
- Free/Bound/Alpha
    - identify bound variables from free variables
        - \x ➔ x y (x is bound, y is free)
    - identify bound variables' binding location (lambda binders)
        - \x ➔ (\x ➔ x) x (two binders, two bound x's, 1st x bound to 2nd binder, 2nd x bound to 1st binder, all counting from left to right)
    - perform alpha-renaming (rename bound variables consistently)
        - \x ➔ (\x ➔ x) y (can be renamed, for instance, to \x ➔ (\y ➔ y) y, but not to \y ➔ (\x ➔ \x) y)
- Beta-Reduction
    - know how to reduce function application (\x.e1) e2
        - please reduce (\x. x x) (\x. x x)
    - know what are expressions and values
        - is \x.x a value?
    - know what can be reduced, what can't
    - know the concept of evaluation/interpretation

- CBV/CBN
    - know CBV/CBN variants of beta-reduction
    - know their differences
    - know their indications (divergence)
        - evaluate (\x.\y. y) ((\x.x x) (\x. x x)) under CBV and CBN, respectively
- Substitution/Capture
    - know what is substitution (replace free variables with lambda expressions)
        - try \x.\y.xyz [\x.xy/z] (replace z with \x.xy)
    - know what is name collision/capture and how to avoid them
    - know that both substitution and environment can be used to interpret free variables.

Type Checking/Inference

- Unification (Substitution)
    - know what is unification problem
    - know what is the result of unification
    - know the invariant of unification (if s = unify e1 e2, then s e1 = s e2)
    - know how to perform unification on simple user defined datatypes with structures and variables
- Typing Rules
    - know the typings for common expressions
        - function abstraction
        - function application
        - base values and operators
        - if-then-else

  - let-in-end
- know how to do type inference using pen & paper
  - write the type for function composition operator `o`, as general as possible
  - can you write the type for omega (\x. x x)? why or why not
- know what will cause type error