Exercise 1 : Object encoding (10 points)

Consider the following classes written in Java:

```
class A extends Object {
  Object x;
  A(Object x) { super(); this.x = x; }
  Object foo(Object a) { return this.bar(a); }
  Object bar(Object b) { return b; }
  Object get() { return x; }
}

class B extends A {
  Object y;
  B(Object x, Object y) { super(x); this.y = y; }
  Object foo(Object a) { return y; }
  Object bar(Object b) { return super.foo(this.get()); }
}
```

During the lectures we have defined a mechanism to properly encode objects using simple typed lambda calculus extended with `records`, `let` and `fix` construct as well as `unit` value. Your task is to:

1. Translate classes `A` and `B` into such encoding. It is recommended that you use auxiliary names for types to make the encoding shorter.

2. Assuming the existence of values `v`, `w` and `z` having type `Object` what would be the result of `new B(v, w).foo(z)` and `new B(v, w).bar(z)` in your encoding? Please provide exact initial **3** reduction steps for the first expression.

Note: In the course you have seen that the naive encoding diverges during object creation. For the purpose of this exercise it is acceptable to use the call-by-need strategy to avoid divergence. You get 2 bonus points for a correct solution that encodes the objects under call-by-value without diverging.

Exercise 2 : Equivalences (12 points)

Several different equivalence relations can be defined on the terms of a language. The equivalence relations that we consider in this exercise are

1. structural equivalence wrt. α-renaming, denoted $\equiv$,
2. behavioral equivalence, denoted $\cong$, and
3. the smallest equivalence relation containing β-reduction, denoted $\cong_\beta$.

Since a relation is a set of pairs, the above relations are ordered as follows: $\equiv \;\subset\; \cong_\beta \;\subset\; \cong$

In each part of this exercise you are given two terms in the call-by-value lambda-calculus, unless specified otherwise. Indicate the smallest equivalence relation (that is, $\equiv$, $\cong_\beta$, or $\cong$) that relates the two terms, or indicate with "NONE" that the two terms are not related wrt. any of the above relations.

Note that to test for behavioral equivalence a term can be put into an arbitrary evaluation context. In particular if the language contains more expression forms than just pure lambda-terms, the context is not restricted to applications!

1. λ`x.` λ`y.` λ`z. x (y z)` and λ`f.` λ`g.` λ`x. f (g x)`
2. In *Scheme*, consider the following terms:

   ```
   (lambda x
     (lambda y
       (lambda z. (x (y z)))))
   ```

   and

   ```
   (lambda f
     (lambda g
       (lambda x. (f (g x)))))
   ```

   Hint: Scheme, and Lisp in general, allows to treat programs as data.
3. λ`y.` λ`x. y x` and λ`y. y`
4. In the *untyped* call-by-value lambda-calculus with numbers and arithmetic expressions (`succ` t etc.), consider the following terms:

   λ`y.` λ`x. y x` and λ`y. y`
5. (*twice* `f`) `x` and (*compose* `f f`) `x`

   where

   twice = λ`f.` λ`x. f (f x)`

   compose = λ`f.` λ`g.` λ`x. f (g x)`
6. (λ`b.` λ`f.` λ`s. b f s`) (λ`x.` λ`y. x`) and (λ`b.` λ`f.` λ`s. b s f`) (λ`x.` λ`y. y`)

Exercise 3 : Checked Error Handling (10 points)

In this exercise we use the Simply-Typed Lambda Calculus (STLC) extended with rules for error handling. In this language, terms may reduce to a normal form `error`, which is *not* a value. In addition, we add the new term form $\texttt{try}\ t_1\ \texttt{with}\ t_2$, which allows handling errors that occur while evaluating t_1.

Here is a summary of the extensions to syntax and evaluation:

$$
\begin{array}{rcll}
t & ::= & & \textbf{terms}: \\
 & | & \ldots & \\
 & | & \texttt{error} & \text{run-time error} \\
 & | & \texttt{try}\ t\ \texttt{with}\ t & \text{trap errors}
\end{array}
$$

New evaluation rules:

$$\text{(E-AppErr1)}\quad \texttt{error}\ t_2 \longrightarrow \texttt{error} \qquad\qquad \text{(E-AppErr2)}\quad v_1\ \texttt{error} \longrightarrow \texttt{error}$$

$$\text{(E-TryValue)}\quad \texttt{try}\ v_1\ \texttt{with}\ t_2 \longrightarrow v_1 \qquad\qquad \text{(E-TryError)}\quad \texttt{try}\ \texttt{error}\ \texttt{with}\ t_2 \longrightarrow t_2$$

$$\text{(E-Try)}\quad \frac{t_1 \longrightarrow t_1'}{\texttt{try}\ t_1\ \texttt{with}\ t_2 \longrightarrow \texttt{try}\ t_1'\ \texttt{with}\ t_2}$$

(Note that these extensions are exactly those summarized in Figures 14-1 and 14-2 on pages 172 and 174 of the TAPL book. However, also note that we will use *different* type rules.)

The goal of this exercise is to define typing rules for STLC with the above extensions such that the following progress theorem holds:

$$\text{If } \emptyset\ ;\ \texttt{false} \vdash t : T\text{, then either } t \text{ is a value or else } t \to t'.$$

The above theorem uses a typing judgment extended with a Boolean value E, written $\Gamma\ ;\ E\ \vdash\ t : T$ where $E \in \{\texttt{true}, \texttt{false}\}$. The theorem says that a well-typed term that is closed (that is, it does not have free variables, which is expressed using $\Gamma = \emptyset$) is either a value, or else it can be reduced *as long as* $E = \texttt{false}$.

Your task is to find out how the value of E can be used to distinguish the terms that may reduce to `error` from those terms that may never reduce to `error`. Note that `error` is a normal form, but it is *not* a value.

1. Specify typing rules of the form $\Gamma\ ;\ E\ \vdash\ t : T$ for all term forms of STLC with the above extensions such that the above progress theorem holds.
2. Prove the above progress theorem using structural induction. (You can use the canonical forms lemma for STLC as seen in the lecture without proof.)

Appendix: **The simply-typed lambda calculus**

$$
\begin{array}{lcll}
t & ::= & & \textbf{terms}: \\
 & | & x & \textit{variable} \\
 & | & \lambda\, x\ :\ \mathtt{T}.\ t & \textit{abstraction} \\
 & | & t\ t & \textit{application} \\
 & & & \\
v & ::= & & \textbf{values}: \\
 & | & \lambda\, x\ :\ \mathtt{T}.\ t & \textit{abstraction} - \textit{value} \\
 & & & \\
\mathtt{T} & ::= & & \textbf{types}: \\
 & | & \mathtt{T} \to \mathtt{T} & \textit{type of functions}
\end{array}
$$

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda\, x\colon \mathtt{T}_1.\ t_1)\ v_2 \longrightarrow [x \to v_2]\ t_1 \qquad \text{(E-AppAbs)}$$

Typing rules:

$$\frac{x\ :\ \mathtt{T} \in \Gamma}{\Gamma \vdash x\ :\ \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\ x\ :\ \mathtt{T}_1 \vdash t_2\ :\ \mathtt{T}_2}{\Gamma \vdash (\lambda\, x\colon \mathtt{T}_1.\ t_2)\ :\ \mathtt{T}_1 \to \mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1\ :\ \mathtt{T}_1 \to \mathtt{T}_2 \quad \Gamma \vdash t_2\ :\ \mathtt{T}_1}{\Gamma \vdash t_1\ t_2\ :\ \mathtt{T}_2} \qquad \text{(T-App)}$$