• Please read all instructions (including these) carefully.

• There are 4 questions on the exam, some with multiple parts. You have 90 minutes to work on the exam.

• The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.

• Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.

• Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: 

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>
1. **Lambda Calculus** (25 points)

Recall the following definitions of booleans, integers, and lists from homework 1:

```lisp
def T = λx. λy. x;
def F = λx. λy. y;
def not = λb. b F T
def inc = λn. λf. λx. f (n f x);
def 0 = λf. λx. x;
def cons = λh. λt. λf. λx. f h (t f x);
def nil = λf. λx. x;
```

For the following problems, feel free to define and use helper functions.

(a) Write a lambda expression `xor` that returns the xor of two booleans. Recall that `xor` is true iff exactly one of its two boolean arguments is true.

```lisp
def xor = λa. λb.__________________________
```

```lisp
def xor = λa. λb. a (not b) b;
```

(b) Write a lambda expression `num_odds` that returns the number of odd integers in a list of integers.

```lisp
def num_odds = λl.__________________________
```

```lisp
def is_odd = λn. n not F;
def num_odds = λl. (λn. λc.(is_odd n) (inc c) c) 0;
```
2. **Object Calculus** (10 points)

Consider the following object calculus, extended with some constants. Objects are defined by

\[ o = [\ldots, 1_i : \zeta(x) e_i, \ldots] \]

where the method bodies \( e_i \) can use variables \( x \), integers \( i \), sums of integers \( e + e' \), method selections \( e.l \), and method overrides \( e.l \leftarrow \zeta(y) e' \). Consider the following object calculus program:

\[ o = [a : \zeta(x) 0, b : \text{______________}] \]

Fill in the \( b \) field with a method that, when invoked, returns an object of the same kind with the \( a \) field changed to a method that returns the old value of \( a \) incremented by 1. For example, the expression \( o.b.b.b.b.b.a \) should evaluate to 5 (that is the number 5, not an encoding of 5).

\[ o = [a : \zeta(x)0, b : \zeta(y)a \leftarrow \zeta(y)a + 1] \]
3. **Simple Types** (30 points)

In this problem we extend the Simply Typed Lambda Calculus (STLC) with some new features. As a reminder, STLC has no polymorphic (quantified) types. The STLC’s syntax is:

**Constants**  
\[ c \ ::= \ 0 \mid 1 \mid \ldots \]

**Expressions**  
\[ e \ ::= \ x \mid \lambda x : \tau.e \mid e \ e \mid c \]

**Types**  
\[ \tau \ ::= \ \text{int} \mid \tau \to \tau \mid \alpha \]

Recall that the type checking rules \( \Gamma \vdash e : \tau \) for the STLC are:

\[
\begin{array}{c}
\Gamma \vdash c : \text{int} \\
\Gamma, x : \tau \vdash x : \tau \\
\Gamma \vdash \lambda x : \tau.e : \tau \to \tau' \\
\Gamma \vdash e_1 : \tau \to \tau' \\
\Gamma \vdash e_2 : \tau
\end{array}
\]

(a) We add *product* types to the STLC. Product types provide a way to construct and use pairs by packing two values together into a single value. Here we are introducing pairs as a new primitive type instead of using an encoding in the lambda calculus itself. Pairs are formed by the constructor \( \langle e_1, e_2 \rangle \), and are eliminated through the accessors \( \text{l} \) and \( \text{r} \). For example \( \langle 3, 4 \rangle.\text{x} \to 4 \) and \( \langle 3, 4 \rangle.\text{1} \to 3 \).

A pair of expressions \( \langle e_1, e_2 \rangle \) has type \( \tau_1 \times \tau_2 \) if \( e_1 : \tau_1 \) and \( e_2 : \tau_2 \). We extend the syntax of the STLC as follows:

**Expressions**  
\[ e \ ::= \ \ldots \mid \langle e, e \rangle \mid e.\text{l} \mid e.\text{r} \]

**Types**  
\[ \tau \ ::= \ \ldots \mid \tau \times \tau \]

Fill in the typing rules for pairs:

\[
\begin{array}{c}
\Gamma \vdash \langle e_1, e_2 \rangle : \boxed{} \\
\Gamma \vdash e.\text{l} : \boxed{} \\
\Gamma \vdash e.\text{r} : \boxed{}
\end{array}
\]
(b) Consider the following approach to adding pairs to the STLC by using an encoding instead of adding them as primitives to the language:

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_1 : \tau_2 \\
\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 & \quad \Gamma \vdash e.1 : \tau_1 \\
\Gamma \vdash e : \tau_1 \times \tau_2 & \quad \Gamma \vdash e.\text{r} : \tau_2
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{pair} &= \lambda x. \lambda y. \lambda f. f \ x \ y \\
\text{let } \text{dotl} &= \lambda p. p \ (\lambda x. \lambda y. x) \\
\text{let } \text{dotr} &= \lambda p. p \ (\lambda x. \lambda y. y) \\
\text{let } p &= \text{pair} \ 0 \ (\lambda f. f) \\
\text{let } x &= \text{dotl} \ p \\
\text{let } y &= \text{dotr} \ p
\end{align*}
\]

Is this program well typed? If so, provide the types of `pair`, `dotl` and `dotr`. If not, explain why the program cannot be typed. The program is not well-typed, because the third argument to `pair` cannot be both `\text{int} \to (\alpha \to \alpha) \to \text{int}` and `\text{int} \to (\alpha \to \alpha) \to (\alpha \to \alpha)`. 
4. Combinator Calculi (25 points)

(a) Show that the I combinator in the SKI calculus isn’t needed: Write a combinator using only S and K that implements I. There is a solution that uses three combinators. S K K

(b) Consider the following combinator language:

- $\langle x_1, \ldots, x_n \rangle$ is a list of length $n$
- $\text{fold } f \ y \langle x_1, \ldots, x_n \rangle = f \ x_1 (f \ x_2 (\ldots (f \ x_n \ y)\ldots))$
- $\text{cons } x_0 \langle x_1, \ldots, x_n \rangle = \langle x_0, x_1, \ldots, x_n \rangle$
- $\langle \rangle$ is the empty list, so we don’t need a separate empty list constructor

Note that $\text{fold}$ performs a reduction starting from the right end of the list using $y$ as the initial value. In the following problems, you may use $\text{fold}$, $\text{cons}$ and lambda expressions (including helper function definitions if you like) in your solution, but you may not use any form of iteration or recursion other than $\text{fold}$.

i. Write a function $\text{append}$ that appends an element $x$ to the right end of a list $l$.

$$\text{append } x \ l = \text{fold } \text{cons} \ \langle x \rangle \ l$$

ii. Using append, write a function $\text{reverse}$ that reverses a list $l$.

$$\text{reverse } l = \text{fold } \text{append} \ \langle \rangle \ l$$