You can turn in handwritten solutions to this assignment. To keep your graders happy, please write clearly, leave lots of whitespace, and use standard-sized (8.5 by 11in) paper! You may be penalized upto 20 points if you do not follow these instructions. Handwritten solutions should be submitted at Lindley Hall 301G by 5pm on the due date.

If you choose to typeset your solutions, you may use LaTeX or Word. If you use LaTeX, there is a template available for your use at the course website. (Remember to look in b522.sty for macros you can use.) A pdf file containing the solutions can be submitted online by midnight on the due date.

This homework is worth 90 points. It is a challenging assignment (especially Problem 1), so start early!

1. **State** (40 pts.)

   Writing programs that deal gracefully with failure can be difficult in the usual imperative style, especially in a distributed environment where failure is unavoidable. A failed subcomputation can leave the system in an unpredictable state where important invariants are violated. A *transaction* mechanism is one way to deal with this. Consider the following extension to uML!:

   $$e \ ::= \ \ldots \ \mid \ \textbf{transaction}\ e \ \mid \ \textbf{abort}\ e$$

   The idea is that a **transaction** expression evaluates e and if e evaluates successfully, the whole expression has the same result as e. However, if the evaluation of e leads to the evaluation of an expression **abort** e', then the most recently started transaction halts and its result is the result of evaluating e'. Since this is an extension to uML!, there is a state (store) that might be modified by expressions that are evaluated. When a transaction completes successfully, the resulting state is the same as the state after evaluating e. However, when a transaction is aborted, the resulting state then reverts to the state that existed at the beginning of the transaction. Transactions can be nested inside other transactions, so an **abort** only aborts the innermost ongoing transaction.

   (a) (20 pts) Give an operational semantics for this language. You may uniformly lift rules given in Homework 3 for uML!. (To see what we mean by uniformly *lifting* rules, see notes for Lecture 12, beginning of Section 2.1.)

   (b) (15 pts) Give a state-passing translation from this extended language to uML. (Hint: Make sure that you understand the state-passing translation discussed in class; see notes for Lecture 12, Section 2.2.)

   (c) (5 pts) Writing a formal semantics for this mechanism exposes some ambiguities in the above description, which entail some design choices. What are these choices, how did you make them, and where do these choices show up in the semantics you wrote? Discuss briefly.

2. **Induction** (20 pts.)

   Prove the following assertions using well-founded induction. Make sure to clearly identify what you are performing induction on, to state the induction hypothesis and point out where it is being used.

   (a) (6 pts) Given a term e in the untyped lambda calculus, show that it doesn't matter in what order you substitute closed terms. Specifically, prove the following lemma:

   Lemma A: Given a term e and closed terms e_1 and e_2, if $x \neq y$, then

   $$e\{e_1/x\}\{e_2/y\} = e\{e_2/y\}\{e_1/x\}$$

(b) (7 pts) In class, we said that $e \longrightarrow^* e'$ if and only if there exists some natural number n such that $e_0 \longrightarrow e_1 \longrightarrow \ldots \longrightarrow e_n$ where $e = e_0$ and $e' = e_n$. We call $\longrightarrow^*$ the multi-step evaluation relation.

For this problem, consider an alternative definition of multi-step evaluation for the untyped, call-by-value lambda calculus, where the relation $e \longrightarrow^* e'$ is defined by the following set of rules:

$$\frac{}{e \longrightarrow^* e} \text{ (M-Refl)} \qquad \frac{e \longrightarrow e' \qquad e' \longrightarrow^* e''}{e \longrightarrow^* e''} \text{ (M-Step)}$$

Note that the first premise of the M-Step rule uses the call-by-value, small-step relation ($\longrightarrow$) for the untyped lambda calculus.

Prove that the relation $\longrightarrow^*$ is transitive—that is, prove the following lemma:

Lemma B: If $e_1 \longrightarrow^* e_2$ and $e_2 \longrightarrow^* e_3$, then $e_1 \longrightarrow^* e_3$.

(c) (7 pts) Here is a fact that we use in the type soundness/safety proof of the simply-typed lambda calculus: the free variables of a well-typed term are always found in its typing environment. Prove the following lemma:

Lemma C: In the simply-typed lambda calculus with boolean values and conditionals, we have that

$$\Gamma \vdash e : T \implies FV(e) \subseteq \mathrm{dom}(\Gamma)$$

3. **CPS translation** (30 pts.)

In class we saw how to translate lambda-calculus terms to terms in continuation-passing style. For this problem, let us consider CPS translation of the following source language:

Source Terms	e	::=	$n \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid \mathbf{if0}(e_0, e_1, e_2) \mid (e_1,\, e_2) \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$
Source Values	v	::=	$n \mid \lambda x.\, e \mid (v_1,\, v_2)$
Primitive Operations	$\oplus$	::=	$+ \mid - \mid \times$

The source language terms include: integer literals (n); primitive operations ($\oplus$) on integers; a conditional $\mathbf{if0}(e_0, e_1, e_2)$ that tests if e_0 evaluates to 0, and evaluates the first branch (e_1) if it does, or else evaluates the second branch (e_2) if e_0 evaluates to an integer other than 0; pairs (e_1, e_2); and constructs (**fst**, **snd**) to extract the first and second components of a pair.

The small-step operational semantics of the source language is as follows:

Source Evaluation Contexts	E	::=	$[\cdot] \mid E\ e_2 \mid v_1\ E \mid E \oplus e_2 \mid v_1 \oplus E \mid \mathbf{if0}(E, e_1, e_2) \mid (E,\, e_2) \mid (v_1,\, E) \mid \mathbf{fst}\ E \mid \mathbf{snd}\ E$

Source Reductions

$$\begin{array}{rcll} (\lambda x.\, e)\ v & \longrightarrow & e\{v/x\} & \\ n_1 \oplus n_2 & \longrightarrow & n_3 & (\text{where } n_3 = n_1 \widehat{\oplus} n_2) \\ \mathbf{if0}(0, e_1, e_2) & \longrightarrow & e_1 & \\ \mathbf{if0}(n, e_1, e_2) & \longrightarrow & e_2 & (\text{where } n \neq 0) \\ \mathbf{fst}\ (v_1,\, v_2) & \longrightarrow & v_1 & \\ \mathbf{snd}\ (v_1,\, v_2) & \longrightarrow & v_2 & \end{array}$$

The continuation-passing style language that we'll use as the target of CPS translation is as follows:

$$
\begin{array}{lrcl}
\textit{Target Values} & v & ::= & n \mid x \mid (v_1, v_2) \mid \lambda(x, k).\, e \mid \underline{\lambda} x.\, e \mid \textbf{halt} \\
\textit{Target Declarations} & d & ::= & v \mid v_1 \oplus v_2 \mid \textbf{fst}\ v \mid \textbf{snd}\ v \\
\textit{Target Terms} & e & ::= & \textbf{let}\ x = d\ \textbf{in}\ e \mid v_0\ (v_1, v_2) \mid v_0\ v_1 \mid \textbf{if0}(v, e_1, e_2) \mid \textbf{halt}\ v \\
\textit{Primitive Operations} & \oplus & ::= & + \mid - \mid \times
\end{array}
$$

There are a few things to note about the target language. First, lambda abstractions that correspond to continuations are marked with an underline. Second, note that declarations cannot have declarations as subexpressions—d does not occur in its own definition. Third, ignoring the **if0** construct, terms in the target language are nearly linear in terms of control flow—that is, they consist of a series of let bindings followed by an application. The only exception to this is the **if0** construct, which forms a tree containing two subexpressions.

The small-step operational semantics of the target language is as follows:

Target Reductions

$$
\begin{array}{rcll}
\textbf{let}\ x = v\ \textbf{in}\ e & \longrightarrow & e\{v/x\} & \\
\textbf{let}\ x = n_1 \oplus n_2\ \textbf{in}\ e & \longrightarrow & e\{n_3/x\} & (\text{where } n_3 = n_1 \widehat{\oplus} n_2) \\
\textbf{let}\ x = \textbf{fst}\ (v_1, v_2)\ \textbf{in}\ e & \longrightarrow & e\{v_1/x\} & \\
\textbf{let}\ x = \textbf{snd}\ (v_1, v_2)\ \textbf{in}\ e & \longrightarrow & e\{v_2/x\} & \\
(\lambda(x, k).\, e)\ (v_1, v_2) & \longrightarrow & e\{v_1/x\}\{v_2/k\} & \\
(\underline{\lambda} x.\, e)\ v & \longrightarrow & e\{v/x\} & \\
\textbf{if0}(0, e_1, e_2) & \longrightarrow & e_1 & \\
\textbf{if0}(n, e_1, e_2) & \longrightarrow & e_2 & (\text{where } n \neq 0) \\
\textbf{halt}\ v & \longrightarrow & v &
\end{array}
$$

The CPS translation $\mathcal{C}[\![e]\!]$ takes a continuation k, computes the value of e, and passes that value to k. To translate a full program—a source term with no free variables—we define the CPS translation $\mathcal{C}^{\text{prog}}[\![e]\!]$, which calls the translation $\mathcal{C}[\![e]\!]$ with the special top-level continuation **halt** that accepts a final answer and halts. (An aside: Instead of adding the special continuation **halt** as a primitive to our target language, we could have defined the **halt** continuation as $\underline{\lambda} x.\, x.$)

The CPS translation for programs, integers, variables, λ-abstractions, and application is defined as follows:

$$
\begin{array}{lcl}
\mathcal{C}^{\text{prog}}[\![e]\!] & \stackrel{\text{def}}{=} & \mathcal{C}[\![e]\!](\underline{\lambda} x.\, \textbf{halt}\ x) \\
 & & \\
\mathcal{C}[\![n]\!]k & \stackrel{\text{def}}{=} & k\ n \\
\mathcal{C}[\![x]\!]k & \stackrel{\text{def}}{=} & k\ x \\
\mathcal{C}[\![\lambda x.\, e]\!]k & \stackrel{\text{def}}{=} & k\ (\lambda(x, k').\, \mathcal{C}[\![e]\!]k') \\
\mathcal{C}[\![e_1\ e_2]\!]k & \stackrel{\text{def}}{=} & \mathcal{C}[\![e_1]\!](\underline{\lambda} x_1.\, \mathcal{C}[\![e_2]\!](\underline{\lambda} x_2.\, x_1\ (x_2, k)))
\end{array}
$$

In the above translation, in order to avoid variable capture, we assume that x is fresh in the $\mathcal{C}^{\text{prog}}[\![]\!]$ case, that k' is fresh in the λ-abstraction case, and that x_1 and x_2 are fresh in the application case.

(a) (10 pts) Consider the following source language program:

$$(\lambda z.\, z\ 3)\ (\lambda y.\, y)$$

Show the CPS translation of the above program. Once you have completed the CPS translation, show the evaluation of the resulting target-level term. (You should show intermediate steps for both the translation and the evaluation.)

(b) (20 pts) The above definition of $\mathcal{C}[\![e]\!]k$ is incomplete—it only shows how to translate source-language integers, variables, λ-abstractions and application. Define the missing cases of the CPS translation.