You can turn in handwritten solutions to this assignment. Please write clearly and use standard-sized (8.5 by 11in) paper. If you choose to typeset your solutions using LaTeX, you may find the mathpartir.sty package useful.

1. **Induction** (30 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) (10 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) (10 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:

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

   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) (10 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

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

2. **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	$::=$	$n \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e_1 \oplus e_2 \mid \mathsf{if0}(e_0, e_1, e_2) \mid (e_1,\, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$
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 $\mathsf{if0}(e_0, e_1, e_2)$ 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:

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

Source Reductions

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

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

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

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

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

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:

$$\mathcal{C}^{\text{prog}}[\![e]\!] \stackrel{\text{def}}{=} \mathcal{C}[\![e]\!](\underline{\lambda} x.\, \mathsf{halt}\ x)$$

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

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.

3. **Well Typed** (20 points)

Below is the syntax, call-by-value operational semantics, and typing rules for the simply-typed λ-calculus with booleans.

Types	τ	$::=$	$\mathsf{Bool} \mid \tau_1 \rightarrow \tau_2$
Terms	e	$::=$	$x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$
Values	v	$::=$	$\mathsf{true} \mid \mathsf{false} \mid \lambda x\!:\!\tau.\, e$
Evaluation contexts	E	$::=$	$[\cdot] \mid E\ e \mid v\ E \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$

Evaluation rules:

$$\begin{array}{lcll} E[\lambda x\!:\!\tau.\, e\ v] & \longrightarrow & E[e[v/x]] & \text{(E-BETA)} \\ E[\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2] & \longrightarrow & E[e_1] & \text{(E-IFTRUE)} \\ E[\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2] & \longrightarrow & E[e_2] & \text{(E-IFFALSE)} \end{array}$$

Typing rules:

Term environments $\Gamma ::= \cdot \mid \Gamma, x:\tau$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau}\ \text{(T-VAR)} \qquad \frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda x\!:\!\tau_1.\, e : \tau_1 \rightarrow \tau_2}\ \text{(T-LAM)} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{(T-APP)}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ \text{(T-TRUE)} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}\ \text{(T-IF)}$$

For each of the following expressions, say if the expression is well typed or not. If it is well typed, provide a typing derivation; if it is not well typed, explain why in no more than 30 words.

(a) $\lambda x\!:\!\mathsf{Bool}.\, \lambda y\!:\!\mathsf{Bool}.\, \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ y$.

(b) $\lambda x\!:\!\mathsf{Bool}.\, \lambda y\!:\!\mathsf{Bool} \rightarrow \mathsf{Bool}.\, \mathsf{if}\ (y\ x)\ \mathsf{then}\ x\ \mathsf{else}\ y$.

(c) $\lambda x\!:\!\mathsf{Bool}.\, \lambda y\!:\!\mathsf{Bool} \rightarrow \mathsf{Bool}.\, \mathsf{if}\ x\ \mathsf{then}\ (y\ x)\ \mathsf{else}\ y$.

(d) $\lambda x\!:\!\mathsf{Bool}.\, \lambda y\!:\!\mathsf{Bool} \rightarrow \mathsf{Bool}.\, \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \lambda z\!:\!\mathsf{Bool}.\, x$.

4. **Type Soundness** (20 points)

Read Chapters 8 and 9 of Types and Programming Languages (TAPL). Make sure you understand the details of proving type soundness for Arith and STLC via progress and preservation.

We saw that the simply-typed λ-calculus ($\lambda^{\rightarrow}$) has a sound type system because it preserves types and guarantees progress of well-typed terms. Thus, well-typed terms do not get stuck (i.e., evaluation is *safe*). Let us add tagged sums to the call-by-value simply-typed λ-calculus.

$$\begin{array}{llll} \textit{Types} & \tau & ::= & \ldots \mid \tau_1 + \tau_2 \\ \textit{Terms} & e & ::= & \ldots \mid \mathsf{inl}_{\tau_1+\tau_2}\, e \mid \mathsf{inr}_{\tau_1+\tau_2}\, e \mid \mathsf{case}\ e\ \mathsf{of\ inl}\ x_1 \Rightarrow e_1 \mid\mid \mathsf{inr}\ x_2 \Rightarrow e_2 \\ \textit{Values} & v & ::= & \ldots \mid \mathsf{inl}_{\tau_1+\tau_2}\, v \mid \mathsf{inr}_{\tau_1+\tau_2}\, v \end{array}$$

New evaluation rules:

$$\frac{e \longrightarrow e'}{\mathsf{inl}_{\tau_1+\tau_2}\, e \longrightarrow \mathsf{inl}_{\tau_1+\tau_2}\, e'}\ \text{(E-Inl)} \qquad \frac{e \longrightarrow e'}{\mathsf{inr}_{\tau_1+\tau_2}\, e \longrightarrow \mathsf{inr}_{\tau_1+\tau_2}\, e'}\ \text{(E-Inr)}$$

$$\frac{e \longrightarrow e'}{\mathsf{case}\ e\ \mathsf{of\ inl}\ x_1 \Rightarrow e_1 \mid\mid \mathsf{inr}\ x_2 \Rightarrow e_2 \longrightarrow \mathsf{case}\ e'\ \mathsf{of\ inl}\ x_1 \Rightarrow e_1 \mid\mid \mathsf{inr}\ x_2 \Rightarrow e_2}\ \text{(E-Case)}$$

$$\frac{}{\mathsf{case}\ (\mathsf{inl}_{\tau_1+\tau_2}\, v)\ \mathsf{of\ inl}\ x_1 \Rightarrow e_1 \mid\mid \mathsf{inr}\ x_2 \Rightarrow e_2 \longrightarrow e_1[v/x_1]}\ \text{(E-Case-Inl)}$$

$$\frac{}{\mathsf{case}\ (\mathsf{inr}_{\tau_1+\tau_2}\, v)\ \mathsf{of\ inl}\ x_1 \Rightarrow e_1 \mid\mid \mathsf{inr}\ x_2 \Rightarrow e_2 \longrightarrow e_2[v/x_2]}\ \text{(E-Case-Inr)}$$

New typing rules:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\, e : \tau_1 + \tau_2}\ \text{(T-Inl)} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2}\, e : \tau_1 + \tau_2}\ \text{(T-Inr)}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of\ inl}\ x_1 \Rightarrow e_1 \mid\mid \mathsf{inr}\ x_2 \Rightarrow e_2 : \tau}\ \text{(T-Case)}$$

For this problem, you must extend the proofs of progress and preservation for STLC ($\lambda^{\rightarrow}$)—as well as the proofs of lemmas that these rely on—to demonstrate type soundness for this extended language ($\lambda^{\rightarrow +}$).

(a) State the inversion lemma.

(b) State and prove the canonical forms lemma.

(c) State the permutation and weakening lemmas.

(d) State and prove the substitution lemma.

(e) Prove the progress and preservation lemmas; their statements are as follows:

Lemma (Progress): If $\vdash e : \tau$ then *either* e is a value *or* there exists some e' such that $e \longrightarrow e'$.

Lemma (Preservation): If $\vdash e : \tau$ and $e \longrightarrow e'$, then $\vdash e' : \tau$.

Note: When proving preservation, use induction on the derivation of $e \longrightarrow e'$.

Note: For the proof portions only of parts (b), (d), and (e), you do not need to show the cases involving functions, application, and function types.