You can turn in handwritten solutions to this assignment. Please write clearly and use standard-sized (8.5 by 11in) paper. Solutions should be submitted before the beginning of class on the due date. If you choose to typeset your solutions using LaTeX, you may find the mathpartir.sty package useful.

1. **Names and scope** (20 pts.)

   Consider the following program:

   ```
   let x = 5 in
      let f = λy. x + y in
         let x = 4 in
            let g = (λz. let x = 3 in f(x)) in
               g(x) + f(x)
   ```

   (a) What does this program output using call-by-value semantics with static scope? Explain briefly.

   (b) What does this program output using call-by-value semantics with dynamic scope? Explain briefly.

   (c) What does this program output using call-by-name semantics with static scope? Explain briefly.

2. **Call-by-denotation** (25 pts.)

   In class we saw two different ways of evaluating the free variables in function bodies: static scoping and dynamic scoping. Static scoping uses the environment of the function definition (the lexical scope), and dynamic scoping uses the environment of the function evaluation (the dynamic scope).

   A similar distinction can by made with the evaluation of the actual arguments of a function: we could evaluate the free variables of actual arguments using either the environment at the function application (the lexical scope) or the environment at the evaluation of the actual arguments (the dynamic scope). In call-by-value semantics, since the actual arguments are evaluated before applying the function, the lexical and dynamic scope are the same. However, when the arguments are evaluated lazily, the distinction is important.

   We use *call-by-denotation* to refer to lazy evaluation of the actual arguments using dynamic scope, where even the choice of scope to use is lazy—the environment used to look up the values of variables is the one in force when the variable's value is needed. (We continue to use call-by-name to mean lazy evaluation of the actual arguments using the static scope.) For example, consider the following program. Using call-by-denotation semantics, the program evaluates to 1; using call-by-name semantics it evaluates to 2.

   ```
   let f = λy. let x = 0 in y in
      let x = 1 in
         f(x + 1)
   ```

   The TeX language has a semantics similar to call-by-denotation. For example, the following TeX code results in the text "inside", because the macro `foo` isn't expanded until after it is redefined.

   ```
   \def\fn#1{\def\foo{inside} #1}
   \def\foo{outside}
   \fn\foo
   ```

   The corresponding program in λ-calculus would be something like the following, and using call-by-value semantics would evaluate to "`outside`":

```
let fn = (λx. let foo = ``inside'' in x) in
   let foo = ``outside'' in
      fn foo
```

(a) What would the result of evaluating the following program be, using call-by-denotation semantics?

```
let x = 0 in
   let f = λy. x + y in
      let x = 1 in
         f(x + 1)
```

(b) Give a translation of dynamically scoped call-by-denotation lambda calculus into statically scoped uML, analogously to the translations given in class for dynamic and static scoping. That is, the source language uses dynamic scope to evaluate free variables in function bodies, and free variables in function arguments. Briefly explain the key differences between this translation and the translation for statically scoped, eager evaluation.

3. **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) (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:

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

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 $e_1 \longrightarrow^* e_3$.

4. **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:

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

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 $\mathsf{if0}$ 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 $\mathsf{if0}$ 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:

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

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.