1. Write down a sequence of reduction steps reducing each of the following terms to normal form. For this problem, reduction is allowed anywhere within a term, including under a $\lambda$.

(a) $(\lambda x(xy))(\lambda u.u)$
(b) $(\lambda xyz. zyx)aa(\lambda pq.q)$
(c) $(\lambda xyz. x(yz))(\lambda xy.x)(\lambda xy.x)$

Note: $\lambda xy.e$ is short for $\lambda x.\lambda y.e$. Remember also that the scope of $\lambda$ extends as far to the right as possible, and that application associates to the left.

2. For each type, construct a simply-typed lambda calculus term (variables, functions, and function application only) whose most general type is that type, or argue that no term has that type. (Hint: You can double-check your answers in OCaml.)

(a) $\alpha \rightarrow \beta \rightarrow \beta$
(b) $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \beta \rightarrow \alpha \rightarrow \gamma$
(c) $\alpha \rightarrow \beta$
(d) $\alpha \rightarrow \alpha \rightarrow \alpha$

**Update:** You should not put explicit type annotations anywhere in your terms, and your terms should have no free variables. For example, if we asked for a type $\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$, then your answer would be $\lambda x.\lambda y.y \ x$, and you can test this by typing “fun x -> fun y -> y x” into OCaml. So, another way of restating this question is: Construct OCaml terms consisting only of fun, variables, and application, with no type annotations, such that the OCaml interpreter outputs the given type; and then translate your answer into lambda calculus.

3. Does the simply-typed lambda calculus with integers have a subject expansion property, meaning if $A \vdash e : t$ and $e' \rightarrow e$, does $A \vdash e' : t$? Here $\rightarrow$ is reduction under call-by-value semantics. Either prove that subject expansion holds, or give a counterexample showing that it does not hold.

4. Consider the IMP language discussed in class. We give a variant of this language in Figure 1. In this variant of IMP, we do not syntactically distinguish integer- and boolean-valued expressions; rather, the syntactic class for expressions contains both the integer and boolean forms. Moreover, unlike the variant of IMP given in class where variables are always numbers, in this variant, we will allow variables to be assigned both integer as well as boolean values.

Now when we are given an expression $e$, we do not know based on the syntax whether $e$ will evaluate to a boolean value, to an integer value, or whether it will get stuck and fail to fully evaluate to any value at all! One consequence of this ambiguity is that conditional expressions for loops and conditionals cannot syntactically enforce whether or not these expressions are actually boolean-valued.

To address these (typing) concerns, we introduce a type system for IMP. As shown in Figure 1, our syntax for types consists of two basetypes; namely, bool and int. Because we need to give types to
variables (to type variable occurrences and variable assignment), we need to establish a (fixed) type for each variable in the program. The store typing \( A \) serves this purpose: it maps variables to types.

Based on these definitions, Figure 2 gives two typing judgements: Expression typing and command typing. The judgement forms are slightly different. Expressions yield a final value, so they are typed by some \( t \). By contrast, commands do not return a value (instead, they side-effect the store), so their typing is neither as an integer nor as a boolean, it is simply "okay." Unlike integers and booleans, "okay" is not a type, it is simply part of the judgement’s notation, to emphasize the fact that commands do not yield values.

Figures 3 and 4 give further syntax and rules that define the small-step operational semantics of IMP. These rules are like those given in class, except that they are the variant that use evaluation contexts.

When defining \( E \), we use standard left-to-right evaluation order for binary operations; since the binary operation itself does not influence this evaluation order, we define the syntax in terms of \( \bowtie \), which ranges over all binary operations of the language. As discussed in class, the “hole” case in both \( E \) and \( C \) is denoted by \( \Box \), and when we “fill” the (single) hole of a context with the appropriate missing part (either an expression or command), the filled context is itself a normal term (either an expression or command). As discussed in class, we denote a context \( C \) filled with a command \( c \) by \( C[c] \); similarly, \( C[e] \) and \( E[e] \) denote command and expression contexts that are filled by an expression \( e \).

Your task is to prove progress and preservation for this language.

**Definition 1** (Compatibility). \( A \sim \sigma \) (pronounced “Type environment \( A \) is compatible with store \( \sigma \)”) if:

- \( \text{dom}(A) = \text{dom}(\sigma) \), i.e., they talk about the same variables
- \( \forall x \in \text{dom}(\sigma) \) there exists a \( t \) such that \( A(x) = t \) and \( A \vdash \sigma(x) : t \). In other words, the type of \( x \) in \( A \) matches the type of the value stored in \( x \) in \( \sigma \).

The first part of the above definition may seem overly strong, but it will work out. Next, for convenience, we can put the above definition together with standard typing:

**Definition 2** (Configuration typing). Separately for expressions and commands:

- \( A \vdash (\sigma,c) \text{ ok} \) if \( A \sim \sigma \) and \( A \vdash c \text{ ok} \).

Now, prove the following two lemmas.

**Lemma 1** (Progress). For both expressions and commands stepping:

- If \( A \vdash (\sigma,e) : t \) then either \( e \) is a value \( v \) or there exist \( \sigma' \) and \( e' \) such that \( (\sigma,e) \rightarrow (\sigma',e') \).
- If \( A \vdash (\sigma,c) \text{ ok} \) then either \( c \) is skip, or there exist \( \sigma' \) and \( c' \) such that \( (\sigma,c) \rightarrow (\sigma',c') \).

**Lemma 2** (Preservation). For both expressions and commands stepping:

- If \( A \vdash (\sigma,e) : t \) and \( (\sigma,e) \rightarrow (\sigma',e') \) then \( A \vdash (\sigma',e') : t \)
- If \( A \vdash (\sigma,c) \text{ ok} \) and \( (\sigma,c) \rightarrow (\sigma',c') \) then \( A \vdash (\sigma',c') \text{ ok} \)
Boolean values

\[ b ::= \text{true} \mid \text{false} \]

Values

\[ v ::= b \mid n \]

Expressions

\[ e ::= x \mid v \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e_1 = e_2 \mid e_1 \leq e_2 \mid \neg e \mid e_1 \wedge e_2 \mid e_1 \lor e_2 \]

Commands

\[ c ::= \text{while } e \text{ do } c \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid x := e \mid c_1; c_2 \mid \text{skip} \]

Types

\[ t ::= \text{int} \mid \text{bool} \]

Store typing

\[ A ::= \emptyset \mid (A, x : t) \]

Figure 1: The IMP Language: Syntax of values, expressions, commands and types.

---

\[ A \vdash e : t \quad (\text{"Under store typing } A \text{, expression } e \text{ has type } t\text{"}) \]

\[ \begin{array}{ccc}
A \vdash b : \text{bool} & A \vdash n : \text{int} & A \vdash x : A(x) \\
A \vdash e_1 : \text{int} & A \vdash e_2 : \text{int} & A \vdash v : t \\
A \vdash e : \text{bool} & A \vdash e : \text{bool} & A \vdash e : \text{bool} \\
\end{array} \]

\[ \begin{array}{c}
\text{TE-Bool} \quad \text{TE-BoolOp} \\
\text{TE-Int} \quad \text{TE-Val} \quad \text{TE-Negate} \\
\text{TE-ArithOp} \quad \text{TE-BoolOp} \quad \text{TE-RelationOp} \\
\end{array} \]

\[ \begin{array}{c}
\circ_A \in \{+,-,\times\} \quad \circ_B \in \{\lor, \land\} \quad \circ_R \in \{\leq, =\} \\
A \vdash e_1 : \text{int} & A \vdash e_2 : \text{int} & A \vdash e_1 : \text{bool} \\
A \vdash e_1 \circ_A e_2 : \text{int} & A \vdash e_1 \circ_B e_2 : \text{bool} & A \vdash e_1 \circ_R e_2 : \text{bool} \\
\end{array} \]

---

\[ A \vdash c \text{ ok} \quad (\text{"Under store typing } A \text{, command } c \text{ is okay"}) \]

\[ \begin{array}{ccc}
A \vdash e : \text{bool} & A \vdash c \text{ ok} & A \vdash e : \text{bool} \\
A \vdash \text{while } e \text{ do } c \text{ ok} & A \vdash c_1 \text{ ok} & A \vdash c_2 \text{ ok} \\
A \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \text{ ok} & A \vdash c_1 \text{ ok} & A \vdash c_2 \text{ ok} \\
\end{array} \]

\[ \begin{array}{c}
\text{TC-While} \quad \text{TC-If} \\
\text{TC-Assign} \quad \text{TC-Sequence} \\
\text{TC-Skip} \\
\end{array} \]

\[ \begin{array}{c}
A \vdash x := e \text{ ok} \quad A \vdash c_1; c_2 \text{ ok} \quad A \vdash \text{skip ok} \\
\text{TC-Assign} \\
\text{TC-Sequence} \\
\text{TC-Skip} \\
\end{array} \]

Figure 2: The IMP Language: Typing judgements for expressions and commands.

---

Variable stores

\[ \sigma ::= \emptyset \mid \sigma[x \mapsto v] \]

Expression contexts

\[ \mathcal{E} ::= \mathcal{E} \circ e \mid v \circ \mathcal{E} \mid \neg \mathcal{E} \mid \Box \]

Command contexts

\[ \mathcal{C} ::= \text{if } \mathcal{E} \text{ then } c_1 \text{ else } c_2 \mid x := \mathcal{E} ; c ; \Box \]

Binary operations

\[ \circ ::= + \mid - \mid \times \mid \leq \mid \land \mid \lor \]

Figure 3: The IMP Language: Stores and evaluation contexts for expressions and commands.
Figure 4: The IMP Language: Small-step operational semantics, for expressions and commands.