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. It should have the type you specify in the empty type environment; i.e., it has type $\tau$ such that $\vdash e : \tau$.

(Hint: You can double-check your answers in OCaml. Extra credit: for any type that has no simply-typed lambda calculus term, give an OCaml term that does have the type without using the $\mathbf{:}$ operator to assign a type.)

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

2. Does the simply-typed lambda calculus with integers have a subject expansion property, meaning if $\Gamma \vdash e : \tau$ and $e' \rightarrow e$, does $\Gamma \vdash e' : \tau$? 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.

3. Recall the extended language of commands we saw in the last homework (see Figure 1). Define a type system for this language, defining the syntax of types $\tau$ and type environments $\Gamma$, and inference rules for the judgments $\Gamma \vdash e : \tau$ ("in environment $\Gamma$ expression $e$ has type $\tau$); $\Gamma \vdash s$ ("the types of variables in store $s$ have the types assigned by $\Gamma$"), and $\Gamma \vdash c$ ("command $c$ is well-typed in environment $\Gamma$"). Here is an example rule from the last judgment:

\[
\begin{array}{c}
\text{TIf} \\
\hline
\Gamma \vdash c_1 \\
\Gamma \vdash c_2 \\
\Gamma \vdash e : \mathbf{int} \\
\hline
\Gamma \vdash \mathbf{if}_{\mathbf{not0}} e \mathbf{then} c_1 \mathbf{else} c_2
\end{array}
\]

The goal of your type system is soundness, in other words: if $\Gamma \vdash s$ and $\Gamma \vdash c$ then either $(s, c) \rightarrow^* s'$ for some $s'$ or else $(s, c)$ diverges. Extra credit: prove this theorem.

variables $x, y, z \in V$
integers $i, j, k \in \mathbb{Z}$
lists $l, m ::= \mathbf{nil} \mid i, l$
expressions $e ::= x \mid i \mid e_1 + e_2 \mid e_1 \ast e_2 \mid e_1, e_2 \mid \mathbf{nil}$
values $v ::= i \mid l$
commands $c, d ::= \mathbf{skip} \mid x ::= e \mid \mathbf{if}_{\mathbf{not0}} e \mathbf{then} c_1 \mathbf{else} c_2 \mid \mathbf{while}_{\mathbf{not0}} e \mathbf{do} c \\
| c_1; c_2 \mid \mathbf{case} e \mathbf{of} \mathbf{nil} \Rightarrow c_1 \mathbf{or} (x :: y) \Rightarrow c_2$

Figure 1: Extended language of commands