Problem 1. \textit{Higher-Order Functions and Lambda Expressions.} Consider the following Haskell definitions of two functions:

\begin{verbatim}
foo :: [a -> b] -> [a] -> [b]
foo [] _ = []
foo _ [] = []
foo (f:fs) (x:xs) = (f x) : (foo fs xs)
\end{verbatim}

\begin{verbatim}
repeat :: a -> [a]
repeat x = x : (repeat x)
\end{verbatim}

For each of the following expressions, decide whether it type-checks or not. If it type-checks, give the final value to which the expression evaluates, if there is also a \texttt{show} function for it. If it does not type-check, give a precise reason.

\textbf{Hint:} Do parts (a), (b), (c), and (d) together (they depend on each other) and then parts (e), (f), and (g) together (they depend on each other).

(a) \((\text{repeat } (\lambda (x :: \text{Int}) \to x \times x))\)

\textit{Answer:}
\begin{verbatim}
(\text{repeat } (\lambda x \to x \times x)) :: [\text{Int} \to \text{Int}] -- type-checks
(\text{repeat } (\lambda x \to x \times x)) => -- no "show" function
-- for type [\text{Int} \to \text{Int}]
\end{verbatim}

(b) \((\text{foo } (\text{repeat } (\lambda (x :: \text{Int}) \to x \times x)))\)

\textit{Answer:}
\begin{verbatim}
(\text{foo } (\text{repeat } (\lambda x \to x \times x))) :: [\text{Int}] \to [\text{Int}] -- type-checks
(\text{foo } (\text{repeat } (\lambda x \to x \times x))) => -- no "show" function
-- for type [\text{Int}] \to [\text{Int}]
\end{verbatim}

(c) \((\text{foo } (\text{repeat } (\lambda (x :: \text{Int}) \to x \times x))) [1..4]\)

\textit{Answer:}
\begin{verbatim}
(\text{foo } (\text{repeat } (\lambda x \to x \times x))) [1..4] :: [\text{Int}] -- type-checks
(\text{foo } (\text{repeat } (\lambda x \to x \times x))) [1..4] => [1,4,9,16]
-- final value is a finite list of 4 elements
\end{verbatim}

(d) \((\text{map } (\text{foo } (\text{repeat } (\lambda (x :: \text{Int}) \to x \times x)))) (\text{repeat } [1..4])\)

\textit{Answer:}
map (foo (repeat (∖ x → x * x))) (repeat [1..4]) :: [[Int]] -- type-checks

map (foo (repeat (∖ x → x * x))) (repeat [1..4]) =>
[[1,4,9,16],[1,4,9,16],[1,4,9,16],... -- final value is an infinite list of 4-element finite lists

(e)  foo [\ f -> f 2, \ f -> f 2] [\ (x :: Int) -> x * x, \ (x :: Int) -> x * x * x]  
Answer:

\[ \begin{align*}
\text{foo [\ f -> f 2, \ f -> f 2]} & \ [\ x -> x*x, \ x -> x*x*x] :: [\text{Int}] \\
& \quad \text{-- type-checks} \\
\text{foo [\ f -> f 2, \ f -> f 2]} & \ [\ x -> x*x, \ x -> x*x*x] \Rightarrow [4,8] \\
& \quad \text{-- final value is finite list of 2 elements}
\end{align*} \]

(f)  foo (foo [\ f x -> f x,\ f x -> f x] [\ (x :: Int) -> x*x,\ (x :: Int) -> x*x*x]) [2,2]  
Answer:

\[ \begin{align*}
\text{foo(} & \text{foo[\ f x -> f x,\ f x -> f x]} [\ x -> x*x,\ x -> x*x*x]) [2,2] :: [\text{Int}] \\
& \quad \text{-- type-checks} \\
\text{foo(} & \text{foo[\ f x -> f x,\ f x -> f x]} [\ x -> x*x,\ x -> x*x*x]) [2,2] \Rightarrow [4,8] \\
& \quad \text{-- final value is finite list of 2 elements}
\end{align*} \]

(g)  foo [\ f -> f 2,\ f -> f 2]  
(\text{foo [\ f -> f 2,\ f -> f 2]} [\ (x :: Int) -> x * x, \ (x :: Int) -> x * x * x])  
Answer:

\[ \begin{align*}
\text{foo [\ f -> f 2,\ f -> f 2]} & \ (\text{foo [\ f -> f 2,\ f -> f 2]} [\ x -> x*x, \ x -> x*x*x]) :: \\
& \quad \text{-- does not type-check, because the type of the first} \\
& \quad \text{-- line is of the form [Int -> a] -> a while the type} \\
& \quad \text{-- of the second line is [Int]} \\
\text{foo [\ f -> f 2,\ f -> f 2]} & \ (\text{foo [\ f -> f 2,\ f -> f 2]} [\ x -> x*x, \ x -> x*x*x]) \Rightarrow \\
& \quad \text{-- no value returned}
\end{align*} \]
Problem 2. (*Call-by-Value and Call-by-Name*) Consider the following Mini-Haskell as well as Haskell expression \( M \):

\[
M = (\ x \to (\ y \to x * y)) \ (3 * 5) \ ((\ z \to z) \ 2)
\]

(a) The expression \( M \) is not fully parenthesized. Write the fully-parenthesized version of \( M \), i.e., insert in \( M \) all implicit parentheses:

**Answer:**

\[
(((\ x \to (\ y \to x * y)) \ (3 * 5)) \ ((\ z \to z) \ 2))
\]

(b) Carry out a *call-by-value* evaluation of \( M \), using the substitution model. You may use the fully-parenthesized version of \( M \) in part (a), if you prefer, but you do not have to:

**Answer:**

\[
\begin{align*}
(\ x \to (\ y \to x * y)) \ (3 * 5) \ ((\ z \to z) \ 2) &\Rightarrow \\
(\ y \to x * y) \ (3 * 5) \ ((\ z \to z) \ 2) &\Rightarrow \\
(\ y \to 15 * y) \ ((\ z \to z) \ 2) &\Rightarrow \\
(\ y \to 15 * y) \ 2 &\Rightarrow \\
15 * 2 &\Rightarrow \\
30 &
\end{align*}
\]

(c) Carry out a *call-by-name* evaluation of \( M \), using the substitution model. You may use the fully-parenthesized version of \( M \) in part (a), if you prefer, but you do not have to:

**Answer:**

\[
\begin{align*}
(\ x \to (\ y \to x * y)) \ (3 * 5) \ ((\ z \to z) \ 2) &\Rightarrow \\
(\ y \to (3 * 5) * y) \ ((\ z \to z) \ 2) &\Rightarrow \\
(3 * 5) * ((\ z \to z) \ 2) &\Rightarrow \\
15 * ((\ z \to z) \ 2) &\Rightarrow \\
15 * 2 &\Rightarrow \\
30 &
\end{align*}
\]

Consider another expression \( N \):

\[
N = (\ x \to \ y \to y)
\]

(d) Specify two expressions \( P \) and \( Q \) so that the *call-by-value* and *call-by-name* evaluations of the expression \((N \ P \ Q)\) (using the substitution model) are exactly the same.

**Answer:**

\[
P = 3 \\
Q = True
\]

(e) Can you specify an expression \( P \) so that the *call-by-name* evaluation of the expression \((N \ P \ 3)\) (using the substitution model) will not terminate? If yes, write such an expression \( P \). If no, explain in at most 2 lines.

**Answer:**

NO, it is not possible. While we can write an expression \( P \) whose evaluation will not terminate, the non-termination of \( P \) will not affect the final value of \((N \ P \ 3)\), which is always 3, because the binding ‘‘\( \ x \)’’ is dummy.
(f) Can you specify an expression \( P \) so that the call-by-value evaluation of the expression \( (N \; P \; 3) \) (using the substitution model) will not terminate? If yes, write such an expression \( P \). If no, explain in at most 2 lines.

\textit{Answer:}
\begin{center}
YES, it is possible. Such an expression is:
\[ P = (\text{let } z = 1 : z \; \text{in} \; z) \]
\end{center}

(g) Can you specify two expressions \( P \) and \( Q \) so that the call-by-value (or, if you prefer, call-by-name) evaluation of the expression \( (N \; P \; Q) \) (using the substitution model) will always terminate and return a Boolean value:

- True if and only if \( P \) and \( Q \) have the same type.
- False if and only if \( P \) and \( Q \) do not have the same type.

If yes, write such expression \( P \) and \( Q \). If no, explain in at most 2 lines.

\textit{Answer:}
\begin{center}
NO, it is not possible. No matter what the types of \( P \) and \( Q \) are, there is no possible communication between \( P \) and \( Q \), because the binding \(''\backslash \; x''\) is dummy.
\end{center}

\textbf{Problem 3. (Monomorphism versus Polymorphism)} Consider the following Haskell as well as Mini-Haskell expression \( M \):

\[
M = (\text{let } f = (\; \backslash \; x \rightarrow x) \; \text{in} \; (f \; f) \; 5)
\]

Assume the type of the constant 5 is Int.

(a) Ignore all typing issues in this question. Does the evaluation of \( M \) terminate in Haskell? If yes, write the final value which is returned. If no, explain in at most 2 lines.

\textit{Answer:}
\begin{center}
YES, the evaluation of \( M \) always terminates in Haskell, and the final returned value is 5.
\end{center}

(b) Ignore all typing issues in this question. Does the evaluation of \( M \) terminate in Mini-Haskell? If yes, write the final value which is returned. If no, explain in at most 2 lines.

\textit{Answer:}
\begin{center}
YES, the evaluation of \( M \) always terminates in Mini-Haskell, and the final returned value is 5.
\end{center}

(c) Is \( M \) monomorphically typable in Haskell? If yes, write down the final type assigned to \( M \). If no, explain in at most 2 lines.

\textit{Answer:}
\begin{center}
NO, it is not monomorphically typable, because of the self-application \((f \; f)\).
\end{center}
(d) Is $M$ polymorphically typable in Haskell? If yes, write down the final type assigned to $M$. If no, explain in at most 2 lines.

**Answer:**

YES, it is polymorphically typable and the final type assigned to $M$ is Int.

(e) Is it possible to transform $M$ into another Haskell expression $M'$ so that $M$ is polymorphically typable iff $M'$ is monomorphically typable? **No credit** without justification.

**Answer:**

YES, it is possible. For example, we can introduce a let-binding for every occurrence of ‘‘f’’ in the body of the let-expression. For the expression in question, this produces:

```haskell
let f1 = (\ x -> x) in
let f2 = (\ x -> x) in (f1 f2) 5
```

which is monomorphically typable iff $M$ is polymorphically typable.

(f) Consider the following expression $N$

$$N = (\ f -> (f f) 5) \ (\ x -> x)$$

Is $N$ typable in Haskell, whether polymorphically or monomorphically? **No credit** without justification.

**Answer:**

NO, it is not, because the type of a lambda-bound variable (here $f$) must be monomorphic. To type the self-application ($f f$), we need a different type for each of the two occurrences of $f$ -- which requires that the type for the binding occurrence of $f$, i.e. ‘‘$f$’’, must be polymorphic, and this is not permitted in Haskell.

(g) Consider the following expression $P$

$$P = (\ f -> f (f 5)) \ (\ x -> x)$$

Is $P$ typable in Haskell? **No credit** without justification.

**Answer:**

YES, it is typable, monomorphically. The required type for the lambda-bound ‘‘$f$’’ is ‘‘Int -> Int’’ and the required type for the lambda-bound ‘‘$x$’’ is ‘‘Int’’.