1. (Types) Consider the following functions (in ML syntax):

\[
\begin{align*}
\text{def } & I \ x = x \\
\text{def } & K \ x \ y = x \\
\text{def } & S \ f \ g \ x = f \ x \ (g \ x) \\
\text{def } & J = S \ (K \ I)
\end{align*}
\]

[Reminder: Function application in this language groups left-to-right, so that \( f \ x \ (g \ x) \) means “\( f \) applied to \( x \) and the result applied to the result of applying \( g \) to \( x \),” and \( \text{def } K \ x \ y = x \) means “define \( K \) so that applying it to a value \( x \) and then applying the result to a value \( y \) yields \( x \).”]

Find the types of each of these functions.

**Solution:**

\[
\begin{align*}
type(I) & = 'x \rightarrow 'x \\
type(K) & = 'x \rightarrow ('y \rightarrow 'x) \\
type(S) & = ('a \rightarrow 'b \rightarrow 'c) \rightarrow ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'c \\
type(J) & = ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'b
\end{align*}
\]

2. (Dynamic Scoping vs Lexical Scoping) Recall that a dynamically scoped variable is looked up in the call stack that is active at the moment when a name needs to be resolved and found in the most recent stack frame. By contrast, a lexically scoped variable is looked up according to the textual arrangement of their definitions.

Consider the following Java-variant, which uses dynamic scoping, code:

```java
class Performer {
    void a() { if (debug) { ... // Do extra debug printing } ... }
}

class Debugger {
    void debugPerformerA(Performer instance) {
        boolean debug = true;
        instance.a();
    }
}
```
When we call `debugPerformerA` method, does the program do extra debug printing and why? Write a counterpart in the (usual) lexically scoped Java and compare which one is more elegant.

**Solution:**

(a) Yes. Since under dynamic scoping, a() will find variable debug in its caller frame, where it’s set to be true.

(b) The simple idea is that we can create an instance variable as the debug flag, but the resulting program will be less elegant because it involves storing the previous debug flag and restoring it.

3. (Runtime Organization) Consider the following code

```python
def f0(g, x0):
    def f1(x1):
        def h():
            print x0, x1 # Point 1
            f0(h, 1)
        if x0 == 0:
            f1(3)
        else:
            g()
    f0(None, 0)
```

What is the output of the code? When execution reaches Point 1, what does the runtime stack look like? The specific layout is not important but you should label what information is in each frame and how are they linked.

**Solution:**

(a) Output is

```
0 3
```

(b) Runtime stack

```
h’s frame
--------
      SL ---> f1 frame
--------
      ra
--------
      DL ---> second f0 frame
--------
```
f0’s frame
--------
   ra
--------
   DL --> f1 frame
--------
f1’s frame --> contains arguments to f0: (ptr to h, SL to this frame), int 1
--------
   SL --> first f0 frame
--------
   DL --> first f0 frame
--------
   ra
--------
f0’s frame --> contains arguments to f1: int 3
--------
...