CSCI 3155: Principles of Programming Languages
Spring 2015 **Quiz 4**

Name: _____

1. **Typing**.

   (a) Recall the conditional expression $e_1 \,?\, e_2 : e_3$ from JAVASCRIPTY that evaluates e_1 to a Boolean and then continues with e_2 if that Boolean is **true** or otherwise e_3 if that Boolean is **false**. Recall that our type checking rule from Lab 4 for conditional expressions is as follows:

$$\textsc{TypeIf}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1 \,?\, e_2 : e_3 : \tau}$$

   Instead, suppose that we **replace** the above rule with the following rules:

$$\textsc{NewTypeIfTrue}\ \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{true} \,?\, e_2 : e_3 : \tau_2} \qquad \textsc{NewTypeIfFalse}\ \frac{\Gamma \vdash e_3 : \tau_3}{\Gamma \vdash \textbf{false} \,?\, e_2 : e_3 : \tau_3}$$

   i. 3 points Is this rule *unsound*? Unsound means that it allows expressions to be type-checked that would then get stuck during evaluation. Explain briefly. If you say it is unsound, give an expression that type checks but would get stuck during evaluation (using our standard small-step semantics).

   Solution: No, this rule is not unsound. We recall from the operational semantics for conditional expressions that

$$\textbf{true} \,?\, e_2 : e_3 \longrightarrow e_2\,,$$

   so the result is of the conditional is something of type e_2. The well-typedness of sub-expression e_3 is not necessary to check in NEWTYPEIFTRUE because it is never evaluated. The argument for NEWTYPEIFFALSE is similar because

$$\textbf{false} \,?\, e_2 : e_3 \longrightarrow e_3$$

   according to the operational semantics.

   ii. 4 points Compare this new set of rules (NEWTYPEIFTRUE and NEWTYPEIFFALSE) with the original set (TYPEIF). Are there expressions (with other operators) where the old set would allow but the new rules would not? If so, give an example expression and explain briefly. What about vice versa?

   Solution: Yes to both, there are expressions that are allowed by the old set but not the new set *and* vice versa.
   The expression $x \,?\, 0 : 1$ is allowed by the old set but not the new set. The new set is not terribly useful because it disallows most expression forms for the conditional test (i.e., anything other than **true** or **false**).
   The expression is **true** ? 0 : `"abc"` is allowed by the new set but not the old set. The old set conservative requires the same type for the "then-expression" and

the "else-expression" because it does not try to determine which way the conditional will go during evaluation. The new set drops this requirement because it can tell which way the conditional will go by restricting the form of the conditional test to the Boolean constants.

(b) Recall the (potentially) recursive function expression **function** $x(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ e$ from JAVASCRIPTY. Suppose that we replace the rule for type checking (potentially) recursive functions expressions from Lab 4 with the following one:

$$\frac{\Gamma[x_1 \mapsto \tau_1] \cdots [x_n \mapsto \tau_n] \vdash e : \tau \qquad \tau' = (x_1 : \tau_1, \ldots, x_n : \tau_n) \Rightarrow \tau}{\Gamma \vdash \quad \textbf{function}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ e \quad : \tau'}$$

i. [3 points] Is this rule *unsound*? Explain briefly.

Solution: Technically, both answers are possible, though the expected answer is that it is not unsound.
In defining PL semantics, a standard implicit assumption is that we read expressions up to renaming of bound variables (technical term: up to α-renaming). Or rather, we assume that the input expression uses distinct names for each binding site to avoid worrying about the scoping issue. Under this assumption, this rule is not unsound. All sub-expressions are type checked according to the type annotations on the declaration of function x. If x is used in e, then we have a type error because $x \notin \text{dom}(\Gamma)$.
If we don't make this assumption, the concern is that Γ already has a mapping for x that causes confusion and thus unsoundness in type checking the body e. Here's such an example expression that shows the unsoundness (i.e., type checks but causes a stuck error during evaluation):

```
const f = function (n: number): string { return "boo" };
(function f(n: number): number {
   return n === 0 ? 0 : (f(n - 1) + "hoo", 2)
 })(1)
```

ii. [3 points] Does this rule prohibit expressions that we would like to allow to be type-checked? Explain briefly.

Solution: Yes, this rule does not allow recursive function declarations to type check because the typing environment is not extended with x when checking the body expression e.
In summary, observe that there are two ways that a typing rule may not be what we want:

- *Unsoundness*: Allows an expression through the type checker that then during evaluation gets stuck. This situation is particularly bad because we have a type checker to specifically exclude such expressions and simplify our interpreter implementation.
- *Imprecision*: Does not allow an expression through the checker that would evaluate without error. We want to avoid imprecision where ever possible, but we know because of undecidability that there will always be some amount of imprecision. This is the inherent cost of using static checking.

2. **Recursion and Higher-Order Methods**. In this question, you will define an insertion sort function on integer lists. The main helper function for insertion sort is an `insert` function that inserts-in-order an element into a sorted list (i.e., it preserves sortedness). We give a definition of `insert` below:

```
def insert(n: Int, l: List[Int]): List[Int] = l match {
  case Nil => n :: Nil
  case h :: t => if (n <= h) n :: l else h :: insert(n, t)
}
```

(a) [4 points] *Insertion sort* is a simple quadratic sorting algorithm. It works by first recursively sorting the tail of the input list and then inserting-in-order the head element into the sorted tail.

Define an insertion `sort` function by completing the following template. Your function should be recursive. Do not call any library function. You will want to call the helper function `insert` defined above.

```
def sort(l: List[Int]): List[Int] = l match {
  case Nil =>



  case h :: t =>



}
```

Explain briefly in 1–2 sentences why your code implements insertion sort—use "inductive thinking."

Solution:

```
def sort(l: List[Int]): List[Int] = l match {
```

Name: _____

```
    case Nil => Nil
    case h :: t => insert(h, sort(t))
  }
```

(b) [3 points] Recall that the `foldRight` method implements a a generic reduction over lists. Specifically, l.`foldRight`$(z)(f)$ successively applies f to the elements of l from right-to-left and starting the accumulation with z:

`Nil.foldRight`$(z)(f)$	evaluates to	z
$(x_1$::x_2::$\ldots$::x_n::`Nil`$)$.`foldRight`$(z)(f)$	evaluates to	$f(x_1, f(x_2, \ldots f(x_n, z)\ldots))$

Re-define the `sort` function using `foldRight`. Your re-definition of `sort` should not be recursive itself. Hint: you will want to use the helper function `insert` again.

```
def sort(l: List[Int]): List[Int] =
```

Solution:

```
def sort(l: List[Int]): List[Int] =
  l.foldRight(Nil)(insert)
```