**CS558 Programming Languages – Fall 2023 – Study Questions Lecture 7a**

These questions are intended for self-study, to help review and deepen your understanding of the lecture. Sample answers are available. There is nothing to hand in.

1. Using the typing rules on slide 8, write typing derivations for the following expressions, or else explain briefly why it is impossible to produce such a tree. In all cases, assume that the initial typing environment is the empty environment $\emptyset$.

(a) `(+ (+ 1 2) 3)`

(b) `(+ x 1)`

(c) `(let x 0 (while (<= x 10) (:= x (+ x 1))))`

(d) `(let x (<= 0 1) (+ (if x 1 2) 3))`

(e) `(+ (if (<= 2 1) (<= 1 2) 3) 39)`

2. Recall from Lab 4 the expression form `(before` $e_1$ $e_2$`)` whose informal semantics are as follows: evaluate $e_1$, remember the result value $v$, evaluate $e_2$, throw away the result value, and then yield $v$ as the overall value of the expression. For example, the expression `(before a (:= a 42))` sets `a` to `42` and yields the old value of `a` as its result. Write down a static typing rule for `before` expressions in the style of slide 8.

3. Recall from slide 19 that C uses name equivalence for `struct` types, and this can be used to distinguish two structurally identical types in order to get finer-grained typechecking, as shown for `polar` and `rect` types on that slide.

Use the same technique to express the code on slide 18 so that it correctly distinguishes between Farenheit (`ftemp`) and Celsius (`ctemp`) temperatures, and permits the last assignment (converting a `ctemp` to an `ftemp`).

4. Consider the following Java class definitions.

```
class A {
  int x;
}

class B extends A {
  int y;
}
```

Each Java class definition defines a new type. `B` is declared as a sub-class of `A` (by the `extends` keyword). As a result, `B` *inherits* the fields of `A`, so every `B` value has an `x` field as well as a `y` field. Under Java's name-based typing rules, sub-classing implies subtyping, so `B<:A`.

(a) In the following function, which assignments (if any) are illegal under Java's static typing rules?

```
void foo() {
```

```
  A a = new B(); // #1
  B b = new B(); // #2
  a = b;         // #3
  b = a;         // #4
}
```

(b) A Java `interface` declarations defines a set of methods (functions). A `class` can be declared to *implement* an `interface` if the class definition defines all the methods listed in the interface with the same or compatible types. (The methods must also be declared `public`.) Consider the following interface and class definitions:

```
interface I1 {
    A f(A a);
}

interface I2 {
    B f(A a);
}

interface I3 {
    A f(B b);
}

interface I4 {
    B f(B b);
}

// A simple function to illustrate use of these interfaces:
void use_ifaces(I1 c1, I2 c2, I3 c3, I4 c4) {
    A a;
    B b;
    A a1 = c1.f(a);
    B b2 = c2.f(a);
    A a3 = c3.f(b);
    B b4 = c4.f(b);
}

class C1 {
    public A f(A a) { a.x = a.x + 1; return a;}
}

class C2 {
    public B f(A a) { B b = new B(); b.y = a.x; return b;}
}
```

```
class C3 {
    public A f(B b) { b.x = b.y; return b;}
}

class C4 {
    public B f(B b) { b.x = b.y; return b;}
}
```

Now consider the following structural subtyping rule for functions (an instance of which is illustrated on slide 14):

$$\frac{t_3 <: t_1 \qquad t_2 <: t_4}{t_1 \to t_2 <: t_3 \to t_4} \text{ (FunSub)}$$

This rule is said to be *covariant* on the result types (the subtyping relation goes in the same direction) but *contravariant* on the argument types (the subtyping relation goes in the opposite direction).

Pretend that Java uses this rule to determine when a member function in a class is a subtype of the declared type in an interface. (In reality, Java uses a somewhat less permissive rule, in which the argument types must be identical for subtyping to hold, but ignore that for this problem.) For each class, state which interfaces (if any) the class implements.