FLOLAC '07
Type Systems
Exercise 2

1. The canonical non-terminating computation, *(λx.xx) (λx.xx)*, was not expressible in the simply-typed λ-calculus. Neither was the self-application fragment, *self* $\equiv$ *λx.xx.* But *self* is indeed typable in the polymorphic lambda calculus. Please re-write *self* as a PLC expression.

2. Assume that we have

```
type t  ::=  Top                  --super type of all types
          |  Int                   --a primitive type
          |  t → t                --function types
          |  {l1: t1, …, ln:tn}   --record types
```

(a) How many different super types does `{a: Top, b: Top}` have?
(b) Can you find an infinite *descending* chain in the subtype relation using (1) as a basis—that is, an infinite sequence of types S_0, S_1, etc. such that each S_{i+1} is a subtype of S_i ?
(c) What about an infinite *ascending* chain?

3. Java array types are *covariant* with respect to the types of array elements (i.e., if B <: A, then B[ ] <: A[ ]). This can be useful for creating functions that operate on many types of arrays. For example, the following function takes in an array and swaps the first two elements in the array.

```
1:  public swapper (Object[] swappee){
2:      if (swappee.length > 1) {
3:          Object temp = swappee[0];
4:          swappee[0] = swappee[1];
5:          swappee[1] = temp;
6:      }
7:  }
```

This function can be used to swap the first two elements of an array of objects of any type. The function works as is and does not produce any type errors at compile-time or run-time.

(a) Suppose **a** is declared by **Shape [] a** to be an array of *shapes*, where **Shape** is some class. Explain why *covariance* (if B <: A, then B[ ] <: A[ ]) allows the type checker to accept the call **swapper(a)** at compile time.

(b) Suppose **Shape[] a** as in part (a). Explain why the call **swapper(a)** and execution of the body of **swapper** will not cause a type error or exception at run time.

(c) However, this decision of adopting covariant type for arrays is *not* always safe. Consider the following code fragment:

```
(1) String [] ss = "aTestString";
(2) Object [] os = ss; // covariant subtyping
(3) os[0] = new Integer(10);
(4) int i = ss[0].length();
```

Given covariant arrays, the above code fragment compiles OK. But it would lead to a runtime error. Please point out which line of code causes the error.

(d) To resolve the problem, Java uses run-time checks, as needed, to make sure that certain operations respect the Java type discipline at run time (Line 3 in the above code fragment). This also applies to the **swapper** function above. What run-time type checks occur in the compiled code for the **swapper** function and where? List the *line number(s)* of **swapper** and the check that occurs on that line.