CMSC330 Fall 2013 Practice Problems 6 Solutions

1. Programming languages
   a. Describe how functional programming may be used to simulate OOP.
      An object may be simulated as a tuple, where each element of the tuple is a closures representing a method for the object.
   b. Describe the difference between OCaml modules and Java classes.
      Both provide a public definition for a group of functions whose internal details are hidden, but Java classes can also instantiate objects and inherit attributes from other classes (not possible with OCaml modules).
   c. Describe the difference between strong and weak typing.
      Strong typing prevents types from being used interchangeably, weak typing allows types to be treated as other types through many implicit type conversions.
   d. Explain how call-by-name simplifies implementing lazy evaluation.
      Expressions to be evaluated lazily may be passed as arguments to functions, since function arguments are not evaluated until used.
   e. Describe the difference between an L-value and an R-value.
      L-values refer to the address of a symbol, R-values refer to the value for a symbol.
   f. What is an activation record (frame), and why is it usually allocated on a stack?
      An activation record contains state information for a function invocation. It is usually allocated on a stack so it can be easily freed upon function return by popping the stack.
   g. Describe the difference between ad-hoc and parametric polymorphism.
      Ad hoc polymorphism applies to code supporting a finite range of types whose combinations must be specified, parametric polymorphism applies to code written without mention to type that can transparently support an arbitrary number of types.

2. Function arguments
   For each code, explain whether *g* is an upward funarg.
   a. let f x = let g y = x + y in let app a b = a b in app g 1 ;;
      g is not an upwards funarg since it is a function parameter passed down to app, not returned by f
   b. let f x = let g y = x + y in g ;;
      g is an upwards funarg since it is a function return value for the 2nd let

   A funarg is simply a function argument where the function is either
   1. **Passed as a parameter to a function call**
   2. **Returned as the return value of a function call**

3. Static vs. Dynamic Scoping

Consider the following OCaml code.

```
let a = 1 ;;
let f = fun ( ) -> a ;;     // value of a determined here for static scoping
let a = 2 ;;
f ( );;                     // value of a determined here for dynamic scoping
```

a. What value is returned by the invocation of f( ) with static scoping? Explain.

1, since the binding for "a" in the function "f = fun ( ) -> a" refers to the closest lexical value of "a" at the point where the function is declared in the code (1st let a).

b. What value is returned by the invocation of f( ) with dynamic scoping? Explain.

2, since the binding for "a" in the function "f = fun ( ) -> a" refers to the closest value of "a" in the call stack at the point where the function is actually invoked (2nd let a).

Consider the following OCaml code.

```
let app f w = let x = 1 in f w ;;          // value of x determined here
                                           //   for dynamic scoping
let add x y = let incr z = z+x in app incr y;;  // value of x determined here
                                                //   for static scoping
(add 2 3) ;;
```

c. What is the order of invocation for the functions app, add, and incr when evaluating the expression (add 2 3)?

1) add, 2) app, 3) incr
incr is defined in add but not invoked until reaching the body of app (as f).

d. What value is returned by (add 2 3) with static scoping? Explain.

5, since the binding for *x* in the function *incr* refers to the closest lexical value of *x* (add *x*) at the point where the function is declared in the code.

e. What value is returned by (add 2 3) with dynamic scoping? Explain.

4, since the binding for *x* in the function *incr* refers to the closest value of *x* in the call stack (*let x = 1*) at the point where the function is actually invoked (by app f w … in f w).

4. Parameter passing

Consider the following C code.

```
int i = 2;
void foo(int f, int g) {
  f  = f-i;
  g = f;
}
int main( ) {
  int a[] = {2, 0, 1};
  foo(i, a[i]);
  printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

a. Give the output if C uses call-by-value

2 2 0 1, since the call to foo( ) creates 2 local variables f & g (initialized with the values of i & a[i]), and all changes to f & g do not affect i or a[i].

b. Give the output if C uses call-by-reference

0 2 0 0, since the call to foo( ) binds f to i & g to a[2], invoking foo( ) =

```
void foo(f → i, g → a[2]) {
  f  = f – i;                  // equivalent to i = i – i → i = 0
  g = f;                       // equivalent to a[2] = i → a[2] = 0
}
```

c. Give the output if C uses call-by-name

0 0 0 1, since the call to foo( ) replaces f with i & g with a[i], foo( ) =

```
void foo(f → i, g → a[i]) {
  f  = f – i;                  // equivalent to i = i – i → i = 0
  g = f;                       // equivalent to a[i] = i  → a[0] = 0
}
```

5. Lazy evaluation

Given the following OCaml code.

```
let doIf p x = if p then x else 0 ;;
let rec loop n = loop n ;;
doIf false (loop 0) ;;
```

a. What is the result of evaluating the doIf expression if OCaml uses call-by-value?

Infinite loop trying to evaluate loop 0 before its value is passed to doIf.

b. What is the result of evaluating the doIf expression if OCaml uses call-by-name?

0, since loop 0 is directly passed to doIf and is not evaluated if p is false.

c. Rewrite the code (using thunks) so that the result of evaluating the doIf expression is the same as if OCaml used call-by-name, even though OCaml uses call-by-value.

```
let doIf p x = if (p ( )) then (x ( )) else 0
let rec loop n = loop n
doIf (fun ( ) -> false) (fun ( ) -> (loop 0))
```

6. Garbage collection

Consider the following Java code.

```
Object a, b, c;
public foo( ) {
        a = new Object( );      // object 1
        b = new Object( );      // object 2
        c = new Object( );      // object 3
        a = b;
        b = c;
        c = a;
}
```

a. What object(s) are garbage when foo( ) returns? Explain why.

Object 1 is garbage there are no longer any references to it within the program. After foo( ) returns, a → object 2, b → object 3, c → object 2.

b. Describe the difference between mark-and-sweep & stop-and-copy.

Mark-and-sweep stops the program to determine what objects are still reachable. Stop-and-copy in addition will move reachable objects to new locations.

7. Polymorphism

Consider the following Java classes:

```
class A { public void a( ) { … } }
class B extends A { public void b( ) { … } }
class C extends B { public void c( ) { … }}
```

Explain why the following code is or is not legal

a. int count(Set<A> s) { … } … count(new TreeSet<A>( ));
   Legal. Actual parameter type (Set<A>) matches formal parameter type (Set<A>)
b. int count(Set<A> s) { … } … count(new TreeSet<B>( ));
   Illegal. Actual parameter type (Set<B>) is not a subclass of formal parameter type (Set<A>), even though B is a subclass of A.
c. int count(Set s) { … } … count(new TreeSet<A>( ));
   Legal. Type erasure will cause formal parameter type (TreeSet<A>) to become TreeSet, which matches actual parameter type (Set).
d. int count(Set<?> s) { … } … count(new TreeSet<A>( ));
   Legal. Actual parameter type (Set<A>) matches formal parameter type (Set<?>), since ? matches A.
e. int count(Set<? extends A> s) { … } … count(new TreeSet<B>());
   Legal. Actual parameter type (Set<B>) matches formal parameter type (Set<? extends A>), since "? extends A" can match A and its subclasses B & C (classes that extend A, including A)

f. int count(Set<? extends B> s) { … } … count(new TreeSet<A>());

Illegal. Actual parameter type (Set<A>) does not match formal parameter type (Set<? extends B>), since "? extends B" can match only B and its subclass C (classes that extend B, including B)

g. int count(Set<? extends B> s) { for (A x : s) x.a( ); … }

Legal. The actual parameter type (Set<? extends B>) indicates s contains elements of class B or its subclasses. So any element of s may be treated as an object of class B or its subclasses (e.g., C). The for loop treats elements of s as objects of class A, which is a superclass of B, and thus is legal (can use subclass in place of superclass).

h. int count(Set<? extends B> s) { for (C x : s) x.c( ); … }

Illegal. The actual parameter type (Set<? extends B>) indicates s contains elements of class B or its subclasses. So any element of s may be treated as an object of class B or its subclasses (e.g., C). The for loop treats elements of s as objects of class C, and is illegal since elements of s may be objects of class B (cannot use superclass in place of subclass).

i. int count(Set<? super B> s) { for (A x : s) x.a( ); … }

Illegal. The actual parameter type (Set<? super B>) indicates s contains elements of class B or its superclasses. So any element of s may be treated as an object of class B or its superclasses (e.g., A, Object). The for loop treats elements of s as objects of class A, and is illegal since elements of s may be objects of class Object (cannot use superclass in place of subclass).

j. int count(Set<? super B> s) { for (C x : s) x.c( ); … }

Illegal. The actual parameter type (Set<? super B>) indicates s contains elements of class B or its superclasses. So any element of s may be treated as an object of class B or its superclasses (e.g., A, Object). The for loop treats elements of s as objects of class C, which is not included and thus illegal.