Comprehensive Exam: Programming Languages Autumn 2007

1. (*15 points*) .... Short Answer

   Answer each question in a few words or phrases.

   (a) (*3 points*) Can a language that does not allow explicit deallocation and uses a correct implementation of garbage collection have dangling pointers? Justify your answer.

   Answer: This answer is no. The only time a piece of heap memory is freed by a correct garbage collector is when the program no longer can access it. Since dangling pointers occur when memory is freed when a program may still access it, there are no dangling pointers as a result of correct garbage collection.

   (b) (*3 points*) What is a closure and what problem does it solve?

   Answer: A closure is a pair consisting of a pointer to code and a pointer to an activation record, used to represent a function and its lexical environment. Closures are used to preserve static scope when a function is passed to another function, or returned from a function.

   (c) (*3 points*) Explain the difference between subtyping and inheritance (in at most two sentences).

   Answer: Subtyping is a relation between interfaces and inheritance is a relation between implementations.

   (d) (*3 points*) Assume that `Rectangle` is a subtype of `Shape`, written `Rectangle` <: `Shape`. Which of the following subtype relationships hold in principle?

      i. (`Shape` → `Rectangle`) <: (`Rectangle` → `Rectangle`)
      ii. (`Rectangle` → `Shape`) <: (`Rectangle` → `Rectangle`)

   Answer: The first subtype relationship is correct, because function types are contravariant in the argument type.

   (e) (*3 points*) Why do static fields of a Java class have to be initialized when the class is loaded? Why can't we initialize static fields when the program starts?

   Answer: Static fields are initialized when a class is loaded because this is the first point at which initialization is possible. It is important to initialize fields before they are accessed, and static fields may be used by static methods before any instances (objects) of the class are created.

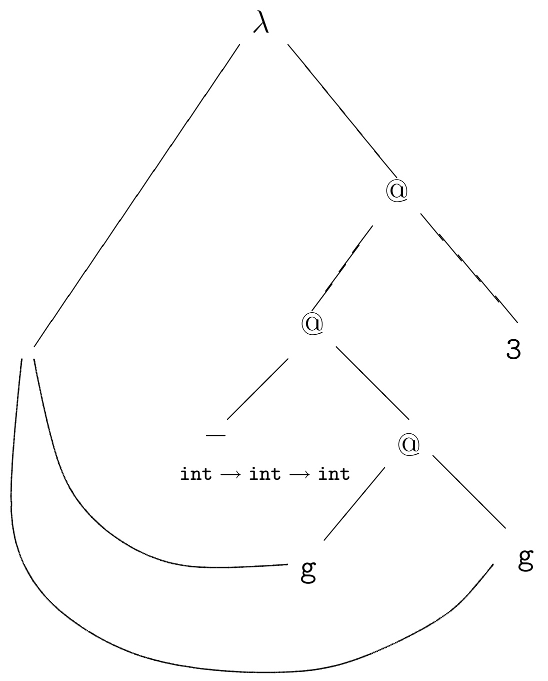
2. (*10 points*) .... Type Inference on Parse Graph

   Use the parse graph below to follow the steps of the ML type inference algorithm on the function declaration

```
fun f(g) = g(g) - 3;
```

   Write the type associated with each node of the graph, as the type inference algorithm proceeds from the bottom of the graph up towards the root. What is the output of the type checker?

   Answer: The function is not typeable in ML; the type checker reports *error.*



Answer: The type algorithm determines that the type a of g must satisfy some constraint of the form $a = a \to b$. This form of constraint cannot be solved by substituting type expressions for type variables.

3. (*11 points*) .... Parameter passing comparison

For the following Algol-like program, write the number printed by running the program under each of the listed parameter passing mechanisms.

In *pass-by-value-result*, also called call-by-value-result and copy-in/copy-out, parameters are passed by value, with an added twist. More specifically, suppose a function `f` with a pass-by-value-result parameter `u` is called with actual parameter `v`. The activation record for `f` will contain a location for formal parameter `u` that is initialized to the R-value of `v`. Within the body of `f`, the identifier `u` is treated as an assignable variable. On return from the call to `f`, the actual parameter `v` is assigned the R-value of `u`.

```
begin
   integer i;

   procedure pass ( x, y );
      integer x, y;  // types of the formal parameters
      begin
         x := x + 3;
         y := x + 5;
         x := y;
         i := i + 7
      end

   i := 1;
   pass (i, i);
   print i
end
```

(a) (*3 points*) pass-by-value

Answer: 8

(b) (*4 points*) pass-by-reference
Answer: 16

(c) (*4 points*) pass-by-value/result
Answer: 9

4. (*12 points*) .... Phantom Members

A C++ class may have virtual members that may be redefined in derived classes. However, there is no way to "undefine" a virtual (or non-virtual) member. Suppose we extend C++ by adding another kind of member, called a *phantom* member, that is treated as virtual, but only defined in derived classes if an explicit definition is given. In other words, a "phantom" function is not inherited unless its name is listed in the derived class. For example, if we have two classes

```
class A {
...
public:
   phantom void f(){...}
   ...
};
class B : public A {
...
public:
   ... /* no definition of f */
};
```

then `f` would appear in the `vtbl` for `A` objects and, if `x` is an `A` object, `x.f()` would be allowed. However, if `f` is not declared in `B`, then `f` might not need to appear in the `vtbl` for `B` objects and, if `x` is a `B` object, `x.f()` would not be allowed.

(a) (*8 points*) Are phantom members consistent with the design of C++, or is there some general property of the way the language is designed and implemented that would be destroyed? If so, explain what this property is, why it is important, and why it is destroyed.

Answer: If we add phantom functions to the language, then it will be possible to use public inheritance and *not* get a subtype. This is because we might choose to leave a phantom function undefined in a subclass. This breaks a property of C++that an initial segment of the `vtable` of a derived class matches the form of the `vtable` its base class. However, it might be possible for for C++ to allow phantom members and recognize that some derived classes do not result in subtypes.

(b) (*4 points*) JavaScript does allow a method of an object to be removed. Explain why this is consistent with the design goals and implementation mechanisms for JavaScript, referring to your answer to part (a) as appropriate.

Answer: JavaScript uses run-time type checking, so does not rely on static typing or subtyping for implementation efficiency or correctness.

5. (*12 points*) .... Race Conditions

The program on the following page contains two classes: `RaceInducer` creates a field `counter` that is the source of data races, and field `counter` is an object of class `DoubleCounter`, which supports methods `incrementBoth` and `getDifference`. For each of the following questions, justify your answer *briefly.*

(a) (*3 points*) If `incrementBoth` and `getDifference` were never allowed to execute at the same time, what would this program print?

Answer: The difference will always be 0, so a sequence of + signs will be printed, with a newline every 60 + signes.

(b) (*3 points*) In the program, `dif_value` in `run` is always either 0 or 1. Why is that? Why can't the difference exceed 1 or become negative?

Answer: Method `incrementBoth` is the only method that changes the values of `x` and `y`, and only one thread calls `incrementBoth` in this program. Therefore, every time that `x` is incremented, `y` is incremented next before `x` can be incremented again. The only reason the difference would be non-zero is if `getDifference` is allowed to execute *while* `incrementBoth` is running. Therefore, the difference is either 0 or 1.

(c) (*2 points*) One way to ensure that data races do not occur would be to insert synchronization primitives. For example, declaring

```
public synchronized int getDifference() {...}
public int incrementBoth() {...}
```

would prevent two threads from executing in method `getDifference` at the same time. Is this enough to ensure that `getDifference` always returns 0?

Answer: No. Only one thread will be executing `getDifference`, however, another thread can run while `incrementBoth` is running, leading to the same kind of behavior as before.

(d) (*2 points*) Is the following declaration

```
public int getDifference() {...}
public synchronized int incrementBoth() {...}
```

sufficient to ensure that `getDifference` always returns 0?

Answer: No. Again, only one thread will be executing `incrementBoth`, however, another thread can run `getDifference` at the same time. As a result, while the first thread is running, the second will get access to partially modified values, leading to the same exact problem as before.

(e) (*2 points*) If the following declaration is used,

```
public synchronized int getDifference() {...}
public synchronized int incrementBoth() {...}
```

what will the output be? Explain.

Answer: A bunch of + signs will be printed. This version places a lock on the only `DoubleCounter` object in the program ensures that only one thread will be running either of these methods at once. In other words, these two methods will be executed sequentially.

```
class RaceInducer extends Thread {
//  this object is shared between all instances of this class
    static DoubleCounter counter;
    static volatile boolean done = false;

    public void run() {
        try {
            for (int i = 0; i < 1000; i++) {
                if (i % 60 == 0) {
                    // insert line break
                    System.out.println();
                }
                int dif_value = counter.getDifference();
                // prints either a '+' or a '-'
                System.out.print("+-".charAt(dif_value));
                sleep(20); // suspends the current thread
            }
            done = true;
        } catch (InterruptedException e) {
            return;
        }
    }

    // entry point into the program
    public static void main(String[] x) {
        Thread ri = new RaceInducer();
        counter = new DoubleCounter();
        try {
            ri.start();     // starts a new thread and calls run()
            while (!done) {
                counter.incrementBoth();
                sleep(30);  // suspends the current thread
            }
            ri.join();
        } catch (InterruptedException e) {
            return;
        }
    }
}

/**
 * Shared data structure.
 * */
class DoubleCounter {
    protected int x = 0, y = 0;

    public int getDifference() {
        return x - y;
    }

    public void incrementBoth() throws InterruptedException {
        x++;
        Thread.sleep(9);
        y++;
    }
}
```