CS143 Final
Fall 2009

- Please read all instructions (including these) carefully.
- There are 4 questions on the exam, all with multiple parts. You have 2 hours to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: _____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

Problem	Max points	Points
1	20	
2	40	
3	40	
4	20	
TOTAL		

1. **Memory Management** (20 points)

- (5 points) You are using an application written in a language with automatic memory management, and you notice that the application stops responding to your input for a few seconds every once in a while. You suspect the garbage collector to be the source of this delay. From the list of automatic memory management strategies below, place a checkmark next to the strategies that could cause this behavior and justify your answer for each choice below.
  - Mark-and-sweep
  - Generational stop-and-copy
  - Reference counting

- (5 points) Your colleague designs and implements a garbage collection technique she calls "Ref-Count++" which extends reference counting to detect unreachable cyclic data structures. She is very proud of it and claims that programs managed by Ref-Count++ cannot have memory leaks. Assuming Ref-Count++ works as advertised, do you agree with her that programs managed by Ref-Count++ do not have memory leaks? Why or why not?

- (5 points) Your friend wants to mitigate the runtime overhead of automatic memory management by combining static and dynamic techniques. In particular, he would like his compiler to automatically insert as many delete calls as possible and have only the remaining objects collected at run-time by the garbage collector. After reading over your CS143 lecture notes, he decides liveness analysis is a great match for this problem. Specifically, he proposes to insert a call `delete v` as soon as variable `v` stops being live. Is this a correct memory management strategy? Explain your reasoning.

- (5 points) A colleague of yours who is an avid C++ advocate says that you can avoid memory leaks in C++ by allocating all objects on the stack, which you cannot do in Java. He also proposes that you can rewrite your Java program in C++ using only stack allocation and thereby avoid any memory leaks. In particular, he proposes that you replace any Java statement `Foo f = new Foo()` by the C++ code `Foo _f; Foo* f = &_f`, which makes `f` point to a stack-allocated `Foo` object instead of a heap allocated one. Is this a correct transformation? Justify your answer.

2. **Optimization and Dataflow Analysis** (40 points)

- (5 points) After profiling the programs generated by your Cool compiler, you notice that dynamic dispatch takes up a large chunk of running time. What is the minimum number of extra memory reads introduced by dynamic dispatch over static dispatch in Cool, assuming that dispatch tables are global and shared between different instances of a class? Justify your answer.

- (5 points) You realize that you can replace some of the dynamic dispatches by a static dispatch just by examining the inheritance tree of the entire Cool program. Given a method call `o.m(...)` where `o`'s static type is `C`, when can you safely replace this call with `o@C.m(...)` just by examining the inheritance tree and the set of methods defined in each class?

- Your project partner insists more can be done to address this performance bottleneck. In particular, she wants to design a dataflow analysis to detect which calls `o.m(...)` can be replaced by `o@C.m(...)`. For example, in the following program:

```
{ if ... then
    o <- new A
  else {
     x <- new A;
     o <- x
       }
  fi;
  o.m()
}
```

  it is safe to replace the dispatch call by `o@A.m()`. On the other hand, if we replaced one `new A` by `new B` for any type `B` distinct from `A` we could not do this transformation.

  The rest of this question walks you through designing such an analysis. To keep the problem simple, you may assume that methods take no arguments and dispatch expressions are always variables.

  – (2 points) List the dataflow values used by your analysis and their interpretation.

– (3 points) Draw the dataflow lattice (i.e., the ordering between these dataflow values).

– (2 points) Is your analysis forwards (i.e., information pushed from inputs to outputs) or backwards (i.e., from outputs to inputs)? Explain.

– (3 points) For each program point, describe how you initialize the dataflow value associated with each program variable.

– (15 points) Let $E_{in}(s, v)$ denote the dataflow value of variable `v` before executing statement s, and $E_{out}(s, v)$ denote the dataflow value of variable `v` after executing s. For each of the following statements s, give an equation defining the value of $E_{out}(s, v)$.

  * `v <- v2`

  * `v <- new C`

  * `v.m()`

– (5 points) Using the results of your dataflow analysis, explain how you determine which dynamic dispatches can safely be replaced with static dispatches.

3. **Semantic Analysis, Runtime Organization, and Code Generation** (40 points)
A *two-stack machine* has an alphabet of *symbols* and two stacks for storing symbols. The instructions of a two-stack machine are as follows:

```
   symbols{ X1, X2, ...}
   push1 X
   push2 X
   pop1
   pop2
   beq X L
L:
```

There is always exactly one `symbols{X1,X2,...}` instruction and it is always the first instruction of a program; this declares the set of symbols used in the program. The instruction `push1 X` pushes the symbol `X` on the first stack. The instruction `pop1` pops one symbol off of the first stack. The `push2 X` and `pop2` instructions work analogously with the second stack. The instruction `beq X L` tests the most recently popped symbol (from either stack); if that symbol is an `X`, the program jumps to label `L` and otherwise it falls through to the next instruction. An instruction `L:` does nothing except define a label in the program that can be the target of a branch.

There is a reserved symbol `BOS` (for "bottom of stack") that always exists but may not be declared and can only be referred to in branch instructions. If a stack is empty then a pop instruction returns `BOS`. Also, if a `beq X L` instruction is executed before any symbol has been popped from a stack, the branch to `L` is taken only if `X` is the symbol `BOS`.

As an aside, a two stack machine is equivalent to a Turing machine—it can encode any computable function. A one stack machine is strictly less powerful.

- (8 points) List any semantic checks that are needed to ensure that programs are well-formed. (Note there are no types in this language, and so no type checking. We are talking about other kinds of semantic checks.)

- (12 points) Describe the run-time organization of data you would use for this language. Include descriptions of which registers would be used for what purposes and any data/code regions you would use and how they would be managed. In both this and the next question, full credit solutions should be efficient in addition to being correct.

- (20 points) Write a code generator for this language. You may use any reasonable and clear pseudo-assembly language notation.

4. **Various** (20 points)

Array languages emphasize operations on entire vectors, rather than on individual data elements. Examples of such languages include APL, FP, FL, and more recently Map-Reduce and Hadoop. Consider the following grammar:

```
E -> zero(N)
   | inc
   | sum
   | map(E)
   | E o E
```

Expressions in this language are functions. Note that there are no variable names; *everything* is a function.

The function `inc` takes one integer argument and produces an integer result (the argument plus one). The other functions map vectors of integers to vectors of integers. For any vector argument, `zero(N)` returns a vector of length `N` of all 0's. The function `sum` takes an argument vector of integers and the result is another vector of integers where the ith element of the output vector is the sum of the first i elements of the input vector. The function `map(E)` is particularly interesting: here `E` is a function mapping integers to integers, and `map(E)` is the function that produces a new vector by applying `E` to each element of the input vector. The function `f o g` is function composition.

Consider the program

```
sum o sum o map(inc) o zero(4)
```

Running this program on input `<0>` (the input vector doesn't matter) the output is `< 1, 3, 6, 10>`, which is computed as follows:

```
(sum o sum o map(inc) o zero(4))(<0>) =
sum(sum(map(inc)(<0,0,0,0>))) =
sum(sum(<1,1,1,1>)) =
sum(<1,2,3,4>) =
<1,3,6,10>
```

- (5 points) Is the grammar above ambiguous? Why or why not?

- (5 points) Give an operational semantics rule for function composition `f o g`. For full credit, your rule should not only be correct but also be as simple as possible.

- (5 points) An important transformation that compilers for array languages can perform is

  ```
  map(f) o map(g) = map(f o g)
  ```

  Given one reason that a compiler might choose to replace the left-hand side by the right-hand side in a program.

- (5 points) Referring again to the transformation in the previous subproblem, now assume that the language includes exceptions. When a function raises an exception execution halts and the result is that exception (there is no facility for catching exceptions). Prove that if `f` and `g` can raise different exceptions, the transformation is incorrect.