Programming Language Qualifying Exam

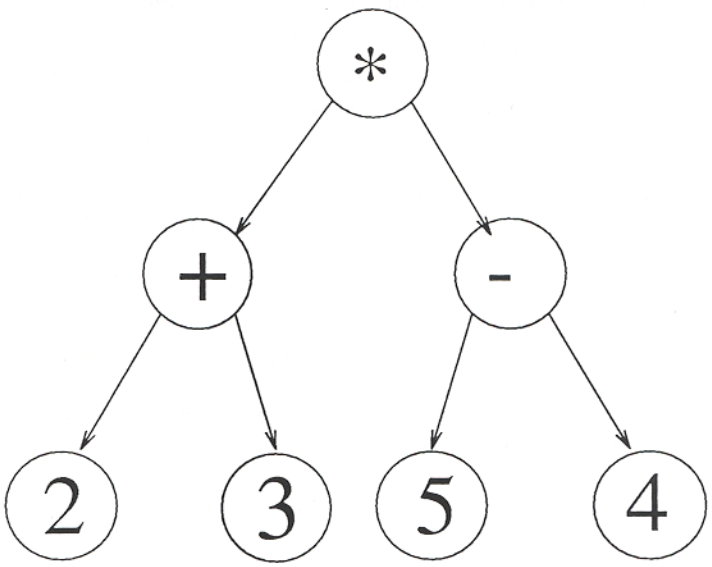
January 13, 1998

There are 6 problems and 5 pages to this exam. Please take a minute to make sure that you have all of the pages, and skim each question before beginning the exam. This is a closed book exam. Please write all of your answers in an exam booklet (or booklets). Do *not* put your name on the booklet but do put your id number.

We consider this to be a 90 minute exam. However, to avoid time pressure, we will allow $2\frac{1}{2}$ hours for you to complete the exam.

1. **Object-Oriented Programming (8 points):** The context for the following question is an object-oriented language like Java which has classes, objects, constructors and inheritance.

   Consider expressions like $(2+3)*(5-4)$ in which all operators are *binary* operators and all operands are integers. These expressions can be represented by binary trees like the following:



   In an object-oriented language, expression trees can be represented by objects which are instances of the following class:

```
class ExpTree {
   char operator; // assume binary operators are represented as chars
   int constant;
   ExpTree Left;
   ExpTree Right;
   .....
}
```

This class definition has the disadvantage that ExpTree objects have storage allocated in them for both operators and constants, even though a given ExpTree object cannot have valid data in both the `constant` field and the `operator/Left/Right` fields.

Design a class or classes for representing binary expression trees, without this waste of storage (that is, so that internal nodes do not have storage allocated for a constant, and leaf nodes do not have storage allocated for an operator and operands).

2. **Subtypes and Subclasses (12 points):**

   Informally, a type T is said to be a subtype of T' if whenever an object of type T' is expected, we can use an object of type T.

   Classes are collections of definitions used to construct objects. A class C is said to be a subclass of C' if C inherits from C' definitions of fields (instance variables) and methods for the objects it produces, possibly overriding or extending the definitions.

   In most typed object-oriented languages (e.g., Java and C++), if a class C produces objects of type T and class C' produces objects of type T', and C is a subclass of C', then T is a subtype of T'. Indeed in most of these languages, there is no distinction between the class that produced an object and the type of the object.

   (a) (6 points) In Java, if a class C defines a public field x with type τ, then a subclass of C must inherit the definition of x. In particular, it is not possible for a subclass to override the definition, assigning the same x a different type τ'. Why not? (Give an example demonstrating a problem.)

   (b) (6 points) Unlike C++, the designers of Java chose not to support multiple inheritance – that is in Java, a given class can be a direct subclass of at most one class, whereas in C++, a given class can directly inherit definitions from multiple classes. Give one reason why multiple inheritance is desirable and one reason why it is undesirable.

3. **Scope, Binding, and Evaluation (15 points):** Briefly (using at most a paragraph), describe the difference between each of the following pairs of terms.

   (a) (5 points) static *vs.* dynamic scope

   (b) (5 points) call-by-value *vs.* call-by-name evaluation

   (c) (5 points) call-by-name *vs.* call-by-need evaluation

4. **Control-Flow Graphs (20 points):** A *dominator* of a node b in a control flow graph is any node a such that all paths from $START$ to b pass through a. By definition, a node dominates itself.

   (a) (5 points) Write down the dominators of each node in the control flow graph of Figure 1 (see the next page).

   (b) (15 points) It is possible to formulate the computation of dominators as a dataflow problem. Write down a dataflow equation schema for this. Your answer must specify clearly whether the problem is a forward or backward flow problem, what the transfer function for a node is, and what the confluence operator is.

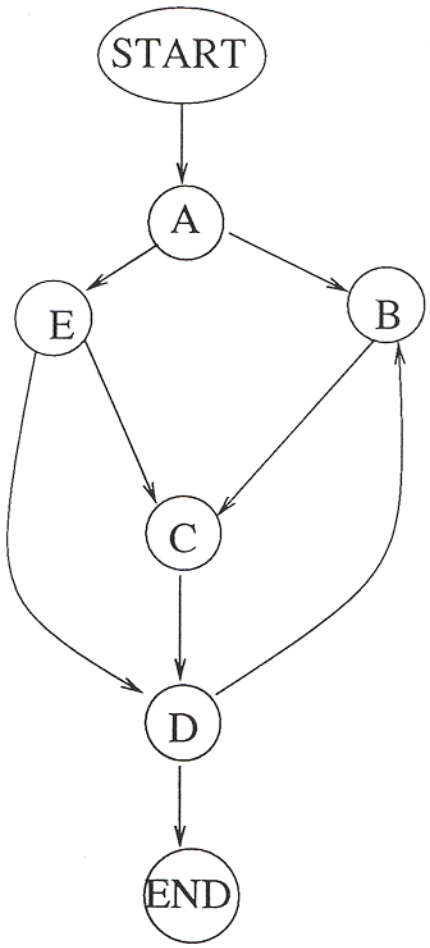


Figure 1: Control Flow Graph

5. **Combinators and the Lambda Calculus (25 points):**

In standard presentations of the (untyped) lambda calculus, terms (M) are either constants ($\mathbf{c}$), variables ($x, y, z, \ldots$), lambda-abstractions ($\lambda x.M$), or applications ($M_1\ M_2$). Computation is achieved by beta-reducing terms of the form $(\lambda x.M_1)M_2$ to $M_1[M_2/x]$ where the latter term is obtained by performing capture-avoiding substitution of M_2 for all of the free occurrences of the variable x within M_1. The formal treatment of variables, scope, and substitution is both tedious and error prone and has historically been the source of much frustration by language practitioners and theoriticians.

We can avoid all of these problems by using a variable-less representation or *combinator calculus* instead. In the combinator calculus there are three magic constants called combinators. The constants are S, K, and I. Terms are simply applications of constants. For example, $(\mathbf{SK})((\mathbf{KI})\mathbf{S})$ is a term in the combinator calculus. Computation is achieved by applying any of the following special rewriting rules:

$$\begin{array}{rcl} \mathbf{I}\,M & \longrightarrow & M \\ (\mathbf{K}\,M_1)\,M_2 & \longrightarrow & M_1 \\ ((\mathbf{S}\,M_1)\,M_2)\,M_3 & \longrightarrow & (M_1\,M_3)\,(M_2\,M_3) \end{array}$$

To perform a reduction step on a term, we find *any* subterm that matches the left-hand-side of a rule and rewrite that subterm using the pattern on the right-hand-side. If M reduces in one step to M', then we write $M \longrightarrow\!\!\!\!\!\rightarrow M'$.

The beauty of combinators is that any (closed) lambda-calculus term M can be compiled to a combinator term M' in such a way that reduction of M' mirrors reduction of M (in an appropriate technical sense) but without the need to worry about variables or substitution. Indeed, the combinator calculus is so simple that it is considered the ultimate "RISC" machine!

The translation from lambda to combinator terms is given by the following function $\mathcal{C}$ which is defined in terms of an auxiliary function $\mathcal{A}$ mapping a variable and a term to a term:

$$\begin{array}{rcl} \mathcal{C}[\![\mathbf{c}]\!] & = & \mathbf{c} \\ \mathcal{C}[\![x]\!] & = & x \\ \mathcal{C}[\![M_1\ M_2]\!] & = & (\mathcal{C}[\![M_1]\!])\,(\mathcal{C}[\![M_2]\!]) \\ \mathcal{C}[\![\lambda x.M]\!] & = & \mathcal{A}[\![x, \mathcal{C}[\![M]\!]]\!] \end{array}$$

$$\begin{array}{rcll} \mathcal{A}[\![x, \mathbf{c}]\!] & = & \mathbf{K}\,\mathbf{c} & \\ \mathcal{A}[\![x, x]\!] & = & \mathbf{I} & \\ \mathcal{A}[\![x, y]\!] & = & \mathbf{K}\,y & (y \neq x) \\ \mathcal{A}[\![x, (M_1\ M_2)]\!] & = & (\mathbf{S}\,(\mathcal{A}[\![x, M_1]\!]))\,(\mathcal{A}[\![x, M_2]\!]) & \end{array}$$

(a) (5 points) Let $M = (\lambda x.(\lambda y.(x\,y)))\,(\lambda z.z)$. Compute $\mathcal{C}[\![M]\!]$.

(b) (5 points) Two terms M_1 and M_2 are said to be alpha-equivalent and written $M_1 =_\alpha M_2$ if one can be obtained from the other by "systematically renaming" the variables in such a way that respects the scope of the variables.

In an implementation, we could test for alpha equivalence of M_1 and M_2 by checking to see if $\mathcal{C}[\![M_1]\!]$ and $\mathcal{C}[\![M_2]\!]$ produce the same combinator terms. Why would this be a bad idea in practice?

(c) (15 points) The following axioms and inference rule define a type system for the combinator calculus.

$$\text{(I)} \vdash \mathbf{I} : \tau \to \tau \qquad \text{(K)} \vdash \mathbf{K} : \tau_1 \to (\tau_2 \to \tau_1)$$

$$\text{(S)} \vdash \mathbf{S} : (\tau_1 \to (\tau_2 \to \tau_3)) \to ((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_3))$$

$$\text{(app)} \ \frac{\vdash M_1 : \tau_1 \to \tau_2 \qquad \vdash M_2 : \tau_1}{\vdash M_1\, M_2 : \tau_2}$$

Prove that if $\vdash M : \tau$ (i.e., M can be assigned the type τ using the inference rules above) and $M \longrightarrow\!\!\!\!\!\rightarrow M'$ then $\vdash M' : \tau$.

6. **Programming Methodology (20 points):**

This problem concerns the methodology for developing loops as described in *The Science of Programming*.

Given is an array $b[0{:}n-1]$ for some fixed $n \geq 0$, each element of which is colored either red, white, or blue. Develop an algorithm to permute the elements so that all the red elements are first and all the blue ones last. The color of an element may be tested with Boolean expressions `red`$(b[i])$, `white`$(b[i])$, `blue`$(b[i])$. The only way to permute the array elements is to swap two of them.

Your development should describe any loop invariants and how you found them. The more work you show, the easier it will be to give partial credit. Your grade to a large extent depends on suitable application of the methods for developing loops, and not only on the final algorithm.