Programming Languages (Honors)
G22.3110 Fall 1998

Final Exam

1. (a) Write a short program, using a Algol-like syntax (Pascal, Ada, whatever), whose result would differ depending on whether pass-by-value-result, pass-by-reference, or pass-by-name (Algol Copy Rule) is used.

(b) Explain the difference between lazy evaluation for a non-strict functional language and Algol’s pass-by-name.

2. Here is the code from my Scheme mini-interpreter that implements lambda expressions and function application.

   (define (eval exp env)
      (cond
         ...  
         ((eq? (car exp) 'lambda)
            (list 'closure exp env))
         (else
            (handle-call (map (lambda (sub-exp) (eval sub-exp env)) exp))))
   )

   (define (handle-call evald-exps)
      (let ((fn (car evald-exps))
             (args (cdr evald-exps)))
         (cond
            ((eq? (car fn) 'closure)
               (let ((formals (cadr (cadr fn)))
                     (body (cddr (cadr fn)))
                     (env (caddr fn)))
                 (handle-block body (append (bind formals args) env))))
            (else (error "Calling non-function")))
         )))

(a) How would this code be modified to support dynamic scoping?

(b) How would this code be modified to support lazy evaluation, assuming the language being interpreted was “pure” (i.e. no side-effects). What other changes would be needed to the interpreter (just give a general description).

3. We discussed in class how a computation in Prolog constituted a complete search of a search tree.

(a) There are two kinds of interior nodes in the search tree. What are they? Write a very simple Prolog program and give a query that would cause the search tree to exhibit both kinds of nodes. Draw the complete search tree (again, keep the program simple!).

(b) Is a Prolog computation a breadth-first or depth-first search of the tree? Is a computation in the idealized logic programming model and breadth-first or depth-first search? Explain both answers.
4. Consider the following C++ code:

```cpp
class EQ {
public:
    virtual int op==(EQ y) { return 1; }
private:
}

int f(EQ x, EQ y) {  
    if (x == y)       
        return 1;
    else             
        return 0;
}
```

(a) Suppose I have a class NUM derived from EQ. Can I redefine NUM's op== operator so that if I passed two objects of type NUM to f, f would call NUM's op==? Briefly explain your answer.

(b) At least one of the typed lambda-calculi covered in class has the power to address this problem. Name one such lambda calculus.

(c) In the lambda calculus you chose, above, write the code for f and describe why it is type-correct to apply f to two objects of type NUM.

(d) Suppose I wanted to restrict f so that it can only be called on two objects of exactly the same type. In the lambda calculus that you chose, can f be restricted this way? If the answer is no, explain how you would modify the type rules of that calculus to be able to enforce it. Be as specific as possible.

5. Save this one for last

This question deals with using SETL as a prototyping language for implementing the fixpoint operator fix (also known as the Y combinator). You remember that in the untyped lambda calculus, factorial can be defined as

\[ f \text{x}(f. \lambda x. \text{if } (= x 0) 1 (* x (f (- x 1)))) \]

Let's see if you can build this fixpoint operator in SETL, where functions are represented as maps over a subset of the natural numbers. For our purposes, assume that the set NAT is the set \{0..n\} for some finite, positive integer n.

(a) Write in SETL an expression (or a short piece of code) computing the set of complete functions, represented as SETL maps, of type NAT→NAT. A complete function is defined at all points in the domain and maps each element of the domain into a single element of the range. For example, if NAT = \{0,1\}, the set of complete functions of type NAT→NAT is

\{\{(0,0), (1,0)\}, \{(0,0), (1,1)\}, \{(0,1), (1,0)\}, \{(0,1), (1,1)\}\}

(b) Write a SETL procedure, feq(f1, f2) that returns true if f1 and f2 are equal and false otherwise, where f1 and f2 are functions of type NAT→NAT. Two functions are said to be equal if they define the same mapping from the domain to the range. Note that
SETL procedures and maps can be used (almost) interchangeably. Thus, \texttt{feq} should work independent of whether \texttt{f1} and \texttt{f2} are SETL procedures or SETL maps.

(c) We didn’t discuss it in class, but SETL has higher-order functions (like Scheme and ML). For example, here is a valid SETL program:

```setl
procedure foo(g);
    return lambda(x); g(x+1); end lambda;
end foo;
```

Notice that the type of \texttt{foo} is (\texttt{NAT->NAT}) $\rightarrow$ \texttt{NAT} $\rightarrow$ \texttt{NAT}.

Write the procedure \texttt{FIX} that takes any function \texttt{f} of type (\texttt{NAT->NAT}) $\rightarrow$ \texttt{NAT} and returns \texttt{f}'s fixpoint. That is, \texttt{FIX}(\texttt{f}) returns a function \texttt{g} such that \texttt{g} = \texttt{f}(\texttt{g}).

(d) Now write, using your \texttt{FIX} operator without explicit recursion, a SETL expression denoting a recursive function of type \texttt{NAT->NAT}. 