CSE583 Take Home Final Examination

Winter Quarter 2000

D. Notkin

From the time you first look at this exam, you have three (consecutive) hours to finish it. It is due by 11:59PM PST on Sunday March 12, 2000. There is no electronic turn-in! You must turn the exam in physically; get it to me (by snail mail, under my door, in my office mail box) in time.

You may consult any materials you want, except for other people. I recommend strongly that you not actually try to solve these problems using a computer, although you are not officially precluded from doing so.

Write (or type) legibly; if you use other paper for answers, make sure that they are stapled onto the exam and that the exam is marked clearly about where to find the answers. Be as clear and concise as possible to increase your chances of getting partial credit. There are 180 total points on the test.

Question (Max Points)	**Points**
1.a.i (15)	
1.a.ii (15)	
1.b (5)	
1.c (10)	
2.a.i (10)	
2.a.ii (15)	
2.a.iii (15)	
2.a.iv (5)	
3.a (30)	
3.b.i (5)	
3.b.ii (10)	
4.a (15)	
4.b (10)	
4.c (10)	
4.d (10)	
Total (180)	

NAME/STUDENT#: _____

1) Functional languages (25%)

a) Consider again the question about the type relationship, if any, that holds between two functions S → T and S' → T'. In contravariant typing,
S → T <= S' → T' (<= means "is a subtype of") if S' <= S and T <= T'.

Consider a different relation, equivariance, in which the arguments to the function must be contravariant (S' <= S) but the return types must be equal (T = T'). That is, S → T <= S' → T if S' <= S.

i) (15 points) Look again at the question from Assignment #2: We have an abstract type `Consumer`, parameterized by `T`, the type of object that the `Consumer` consumes. `Consumer` has a single operation `eat`, which takes a single argument of type `T` and doesn't return anything. We also have an abstract type `Producer`, again parameterized by `T`. `Producer` has a single operation `make`, which takes no arguments and returns an object of type `T`. Finally we have an abstract type `ProducerConsumer`, again parameterized by `T`, the type of object that the `ProducerConsumer` contains. `ProducerConsumer` has two operations: `eat` and `make`.

Using the equivariant rule, what is the subtype relation between the following pairs of types? Briefly explain your answer.

(1) `ProducerConsumer[Integer]` and `ProducerConsumer[Number]`

(2) `ProducerConsumer[Integer]` and `Consumer[Number]`

(3) `ProducerConsumer[Number]` and `Consumer[Integer]`

NAME/STUDENT#: _____

ii) (15 points) Discuss the consequence of using equivariant typing in a functional programming language with respect to expressiveness and type safety.

b) (5 points) Both Haskell and ML have variants of the `take` function, which accepts a list L and an integer N, returning the first N elements of the list L. Given that Haskell permits infinite lists but that ML does not, is the order of the two arguments material? Why or why not?

c) (10 points) In any functional-looking language, write a function that accepts a sorted list of elements and returns a list with all duplicate elements removed.

NAME/STUDENT#: _____

2) Object-oriented languages 25%

a) We've seen a number of different mechanisms for function invocation, including dynamic dispatch (all OO languages), multi-methods (in Cecil), and pattern matching (in ML, Haskell, and Prolog). An attempt to unify these mechanisms with a single, more general mechanism, can be seen in the following two examples (both of which use Cecil-like syntax for the method-binding aspects):

```
method zip(l1,l2)
   when l1 @ Nil or l2 @ Nil
{ return Nil; }

method zip(l1,l2)
   when l1 @ Cons and l2 @ Cons
{ return Cons(Pair(l1.head, l2.head),
              zip(l1.tail, l2.tail)); }
```

This takes a pair of lists and returns a list of pairs; lists are constructed recursively from either `Nil` types or `Cons` types. The first part of the function (with the "`@ Nil`") dispatches whenever either list is of type `Nil`; the second part dispatches when both are of type `Cons`.

A richer use is shown in this example:

```
predicate on_x_axis(p)
   when (p@CartesianPoint and test(p.y == 0)) or
        (p@PolarPoint and (test(p.theta = 0) or
                           (test(p.theta = pi)))

method draw(p) when on_x_axis(p) { … }
method draw(p) when p@Point { … }
```

This defines a boolean predicate `on_x_axis` that is one of the controls for invocation of one of the draw methods (the other draw method is invoked for other points).

i) (10 points) Discuss this more general method of dispatching in terms of efficiency.

NAME/STUDENT#: _____

ii) (15 points) Discuss this method in terms of type checking.

iii) (15 points) Discuss this method in terms of its ability to generalize the forms of dispatch listed above.

iv) (5 points) Discuss any other aspects of this method that are pertinent to language design.

NAME/STUDENT#: _____

3) Logic Programming/Constraint Logic Programming 25%

a) (30 points) Assume Prolog had only symbols and predicates, with absolutely no knowledge of numbers. That is, it would not have `is` or any arithmetic operators, and if it had symbols such as `1`, `2`, etc. then they would have no more meaning than `wuss` or `snork`.

Describe either (a) how you would realize basic arithmetic in this language (that is, show how you can construct representations of numbers and operations on those representations that capture the basics of arithmetic) or (b) why there is no way to realize this in this crippled form of Prolog.

NAME/STUDENT#: _____

b) Consider the following Prolog program:

```
member(K,node(K,_,_)).
member(K,node(N,S,_)) :- K < N, member(K,S).
member(K,node(N,_,T)) :- K > N, member(K,T).
```

i) (5 points) Could you swap the order of the second two rules and for all executions expect the same set of results to be returned (regardless of order of the results)?

ii) (10 points) Could you (in the original program) insert a cut in the second rule

```
member(K,node(N,S,_)) :- K < N, !,member(K,S).
```

and still compute the same set of results? If so, briefly explain. If not, briefly explain.

NAME/STUDENT#: _____

4) Miscellaneous 25%

a) (15 points) Compare and contrast the notion of variable in object-oriented, in functional, and in logic programming languages.

b) (10 points) Compare and contrast the notion of list in ML, Prolog, and Scheme.

c) (10 points) True or false and briefly justify your answer: Continuations are a straightforward way to implement tail recursion.

NAME/STUDENT#: _____

d) (10 points) T.R.G. Green argues that while structured programming helps write better programs because it helps organize information in an understandable way, there are other aspects of cognition that also come into play when designing programming languages. As partial justification of this belief, he cites a result of his own that showed large usability differences between two equally-well-structured; such effects can obviously not be explained by the principles of structured programming.

He defines the 'match-mismatch' law as saying that *every notation or information structure highlights some information at the cost of obscuring other information*. For example, a language in which strings aren't primitive (like C) is unlikely to be good for string processing because it exposes too much detail (the bytes) while obscuring higher-level information about strings (do they match? when is s1 > s2?).
Pick a language we discussed this quarter or another very common programming language. Using Green's match-mismatch law, evaluate three features of your selected language.

NAME/STUDENT#: _____