CSE 341 - Programming Languages
Final exam - Winter 2009

Your Name:

```
                                                    (for recording grades):
                                                    1. (max 10)

                                                    2. (max 5)
Total  (max 124):
                                                    3. (max 10)

                                                    4. (max 10)

                                                    5. (max 5)

                                                    6. (max 10)

                                                    7. (max 10)

                                                    8. (max 6)

                                                    9. (max 12)

                                                   10. (max 6)

                                                   11. (max 10)

                                                   12. (max 10)

                                                   13. (max 10)

                                                   14. (max 10)
```

Open book and notes. No laptop computers, PDAs, or similar devices. (Calculators are OK, although you won't need one.) Please answer the problems on the exam paper — if you need extra space use the back of a page.

1. (10 points) Write a Haskell function `count` that counts the number of occurrences of an item in a list. Here are some example expressions that use `count` and the result of evaluating them:

```
count 10 [10,20,30,10,10]  =>  3
count 5 []  =>  0
count 'a' "abcde"  => 1
```

2. (5 points) What is the most general possible type for your `count` function from Question 1?

3. (10 points) Is your `count` function from Question 1 tail-recursive? If it is, say so, and you're done with this question! Otherwise write a tail recursive version. You can use a helper function if needed.

4. (10 points) The Scheme metacircular interpreter implemented `if` as a special form, and `cond` as a derived expression. Suppose instead that `cond` was implemented as a special form. Write a Scheme function `if->conf` to implement `if` as a derived expression by rewriting it to a `cond`. You don't need to check for syntax errors. Hints: remember that for derived expressions, the Scheme interpreter takes a list representing the expression in one form (in this case as an `if`) and returns a new list that, when treated as code, evaluates to the same thing but using different constructs (in this case `cond`). (Recall how you implemented `let` for the assignment.) it's legal to have an `if` expression with just a true branch; in that case, if the condition is false, the value of the `if` expression is void. Conveniently, however, if you have a `cond` with no `else` part, and all the conditions are false, the value of the `cond` is void.

5. (5 points) The Scheme lecture notes included code to implement `my-delay` in Scheme as a function. However, unlike the built-in delay in Scheme, this `my-delay` function requires wrapping the expression to be delayed in a lambda. Here is the relevant code from the lecture notes, along with an example of delaying an expression:

```
(define-struct delay-holder (is-evaluated value))
(define (my-delay f)
  (make-delay-holder #f f))
(define d (my-delay (lambda () (+ 3 4))))
```

Write a Scheme macro `better-delay` that functions just like the built-in delay, so that the last line can be written as follows:

```
(define x (better-delay (+ 3 4)))
```

6. (10 points) Write a `fsequence` rule in CLP(R) that succeeds if its argument is a list with at least 3 numbers, such that each element starting with the third is the sum of the preceeding two numbers. For example, these goals should succeed:

```
fsequence([1,1,2,3,5]).
fsequence([4,6,10,16,26]).
```

and these should fail:

```
fsequence([1,1]).
fsequence([4,6,10,20]).
```

Notice that this rule succeeds on lists of at least 3 Fibonacci numbers, as well as on other sequences that obey the summation property.

7. (10 points) Write a `fibs` rule in CLP(R) that succeeds if its argument is a list of Fibonacci numbers. The list can have any number of elements, including none. You can use the `fsequence` rule from Question 6 as a helper. For example, here are the first several answers if you provide a variable as an argument:

```
1 ?- fibs(A).
A = []
*** Retry? y
A = [1]
*** Retry? y
A = [1, 1]
*** Retry? y
A = [1, 1, 2]
*** Retry? y
A = [1, 1, 2, 3]
*** Retry? y
A = [1, 1, 2, 3, 5]
*** Retry? y
A = [1, 1, 2, 3, 5, 8]
*** Retry? n
```

8. (6 points) Suppose that we have a version of the CLP(R) `append` rule with a cut:

```
append([],Ys,Ys) :- !.
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

What are all of the answers returned for the following goals? If there are an infinite number of answers, give the first three.

```
?- append([1,2,3],[10,11],As).
```

```
?- append(As,Bs,[1,2,3]).
```

```
?- append(As,Bs,Cs).
```

9. (12 points) Consider the following CLP(R) rule that takes three lists. It succeeds if every element of the third list is the sum of the corresponding elements of the first two lists.

```
add_lists([],[],[]).
add_lists([X|Xs],[Y|Ys],[Z|Zs]) :- X+Y=Z, add_lists(Xs,Ys,Zs).
```

   (a) Draw the simplified derivation tree for the goal `add_lists([2,4],[10,20],As).`

   (b) Draw the simplified derivation tree for the goal `add_lists([A,B],[B,10],[5,20]).`

   (c) Draw the simplified derivation tree for the goal `add_lists(As,As,As).` (This is an infinite tree; include at least 3 answers in the tree that you draw.)

   Continue on the back of this page as needed.

10. (6 points) Consider the following example in an Algol-like language.

```
begin
integer n;
procedure p(k: integer);
    begin
    n := n+10;
    k := k+n;
    n := n+1;
    print(n);
    print(k);
    end;
n := 4;
p(n);
print(n);
end;
```

(a) What is the output when k is passed by value?

(b) What is the output when k is passed by value result?

(c) What is the output when k is passed by reference?

11. (10 points) What are the differences among Ruby classes, Ruby mixins, and Java interfaces?

12. (10 points) Write a `sum` method for the Ruby `Enumerable` mixin. The sum of an empty collection should be zero. (This method will only work for collections of numbers; that's OK.)

13. (10 points) The following is a Ruby class definition for positive rational numbers (similar to the example in the handouts). Use the Comparable mixin to add methods for <, == ,>, and so forth. Write your answer by interspersing the new statements with the existing class definition. Hint: you only need to mix in Comparable and implement a <=> method. The <=> method should return -1, 0, or 1 if the receiver is less than, equal to, or greater than the argument respectively.

```
class PosRational
  attr_reader :num, :den


  def initialize(num,den=1)
    if num < 0 || den <= 0
      raise "PosRational received an inappropriate argument"
    end
    @num = num
    @den = den
  end









end
```

14. (10 points) True or false?

    (a) A Haskell expression of type `IO t` can never occur inside another expression of type `(Num t) => [t]` in a expression that typechecks correctly.

    (b) In Java, adding an upcast can never change whether or not a program compiles, but could change the behavior of a program that does compile without without the upcast.

    (c) In Java, `Point[]` is a subtype of `Object[]`.

    (d) In Java, `ArrayList<Point>` is a subtype of `ArrayList<Object>`.

    (e) In Java, `ArrayList<Point>` is a subtype of `ArrayList<?>`.

    (f) In Ruby, class `Object` is an instance of itself.

    (g) In Ruby, class `Object` is a subclass of itself.

    (h) In Ruby, class `Class` is an instance of itself.

    (i) In Ruby, class `Class` is a subclass of itself.

    (j) A Ruby class can have multiple superclasses, but only one mixin.