CSE 341 - Programming Languages
Final exam - Winter 2009 – Answer Key

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
```

```
count x [] = 0
count x (y:ys) | x==y = 1 + count x ys
               | otherwise = count x ys
```

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

```
count :: (Num t, Eq a) => a -> [a] -> t
```

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.

   It's not tail-recursive — here is a version that is.

```
count x ys = counthelper x ys 0

counthelper x [] n = n
counthelper x (y:ys) n | x==y = counthelper x ys (n+1)
                       | otherwise = counthelper x ys n
```

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.

```
(define (if->cond s)
  ;; if the length of s is 4, then there is an else part, otherwise not
  (if (= (length s) 4)
      (list 'cond (list (cadr s) (caddr s)) (list 'else (cadddr s)))
      (list 'cond (list (cadr s) (caddr s)))))
```

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)))
```

```
(define-syntax better-delay
  (syntax-rules ()
    ((better-delay e1)
     (make-delay-holder #f (lambda () e1)))))
```

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.

```
fsequence([A,B,A+B]).
fsequence([A,B,A+B|Cs]) :- fsequence([B,A+B|Cs]).
```

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
```

```
fibs([]).
fibs([1]).
fibs([1,1]).
fibs([1,1|Fs]) :- fsequence([1,1|Fs]).
```

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).
As = [1, 2, 3, 10, 11]


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


?- append(As,Bs,Cs).
Cs = Bs
As = []
```

(Note that each of these goals gives only one answer.)

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.)

Answers are in a separate file.

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? **15 18 15**

(b) What is the output when k is passed by value result? **15 18 18**

(c) What is the output when k is passed by reference? **29 29 29**

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

    A Ruby class, like classes in other object-oriented languages, holds the code that determines the behavior of an object. A class can be *instantiated* to make an instance of that class. Classes in Ruby are runtime objects that can receive messages, just like any other object in Ruby. A Ruby mixin is similar to a class in that it can carry code, but it can't be instantiated. Instead, its purpose is to be mixed in to some class to give it additional behavior. For example, the `Enumerable` mixin can be used to give "collection" behavior to some class, as long as that class implements the `each` method. In terms of Ruby implementation, classes are instances of the class `Class`. Mixins are instances of class `Module`; `Class` is a subclass of `Module`. A Java interface specifies the methods that an object should provide if it implements that interface, but doesn't hold any code. It is a compile-time construct that is used in declaring and statically checking types. It wouldn't make sense to have interfaces in Ruby, since Ruby is dynamically typed.

    Specific points that need to be noted for full credit on this question:

    (a) Ruby classes can be instantiated; mixins cannot.

    (b) Both Ruby classes and mixins specify code for their methods.

    (c) Java interfaces don't include code for their methods.

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.)

```
def sum
  total = 0
  each {|x| total=total+x}
  return total
end
```

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
  include Comparable
  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

  def <=> other
    return self.num*other.den <=> other.num*self.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. **true**

(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. **false** (It can both change whether or not a program compiles, and for a program that compiles can change its behavior. But threw out this question!)

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

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

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

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

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

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

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

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