1. (8 points) Write a Racket function `multicons` that takes an item `x`, a non-negative integer `n`, and a list `xs`; and returns a new list with `n` occurrences of `x` followed by `xs`. You don’t need to handle bad inputs. Examples:

```racket
(multicons 'z 3 '(a b c)) => (z z z a b c)
(multicons '(a b) 3 '(x y z)) => ((a b) (a b) (a b) x y z)
(multicons 'z 0 '(a b c)) => (a b c)
```

```racket
(define (multicons x n xs)
  (if (zero? n)
      xs
      (cons x (multicons x (- n 1) xs))))
```

Or here is a nice tail-recursive solution. (Both would receive full credit though.)

```racket
(define (multicons x n xs)
  (if (zero? n)
      xs
      (multicons x (- n 1) (cons x xs))))
```
2. (10 points) Suppose we have the following class in Ruby:

```ruby
class Toy

  def initialize(size, type)
    @size = size
    @type = type
  end

  # The goodness method returns a number indicating how good
  # the toy is. The default is that bigger is better!
  def goodness
    size
  end

end
```

(a) Add code to Toy to define public getters for `size` and `type` (but not setters), and to mix in the `Comparable` module. Write your extra code in the blank spaces in the `Toy` definition above. To compare two toys, compare their goodness. You should define in `Toy` the method needed by `Comparable`; this should then automatically let you compare two toys `t1` and `t2` using `t1>t2, t1<t2, t1==t2, and so on, without needing to define `<, >, and so on in Toy. Hint: `3<=>10` evaluates to `-1`.

```ruby
class Toy
  include Comparable
  attr_reader :size, :type

  def initialize(size, type)
    @size = size
    @type = type
  end

  # The goodness method returns a number indicating how good
  # the toy is. The default is that bigger is better!
  def goodness
    size
  end

  def <=> other
    goodness <=> other.goodness
  end
end
```

(b) Write a subclass of `Toy` called `StuffedAnimal`.
- `StuffedAnimal` has the same fields as `Toy` plus `numHugsGiven`.
- `numHugsGiven` should be 0 on initialization.
• The new initialize method for StuffedAnimal should still take 2 arguments: size and type. For full credit, when possible reuse relevant methods inherited from Toy.
• Redefine goodness for StuffedAnimal to be the product of its size and the number of hugs given.
• Add a public method hug that increments numHugsGiven by 1 when called.
• Add a public getter for numHugsGiven.

class StuffedAnimal < Toy
  attr_reader :numHugsGiven
  def initialize(size, type)
    super(size, type)
    @numHugsGiven = 0
  end
  def goodness
    size*numHugsGiven
  end
  def hug
    @numHugsGiven = numHugsGiven+1
  end
end
3. (10 points) Write a Haskell function `indices` that takes an item and a list of that same type of item, and returns a list of the positions of that item in the list. You can use a helper function if needed. Also give the most general type of the `indices` function. Examples:

```haskell
indices 'b' "ababb" => [1,3,4]
indices true [false,false,true] => [2]
indices 'x' "abc" => []
```

```haskell
indices x xs = helper x 0 xs

helper x n [] = []
helper x n (y:ys) =
    if x==y
      then n : helper x (n+1) ys
      else helper x (n+1) ys

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

Or here is a version that doesn’t use a helper function:

```haskell
indices x [] = []

indices x (y:ys) = if x==y then 0 : rest else rest
    where rest = map (+1) (indices x ys)
```
def test1(a,b)
    a["x"] = "squid"
    b["x"] = "clam"
end

def test2(a,b)
    a["x"] = "tuna"
    b = {"x"=>"starfish"}
end

a = Hash.new
test1(a,a)
puts a

test2(a,a)
puts a

b = Hash.new
puts b

c = Hash.new
puts c

test2(b,c)
puts b
puts c

{"x"=>"clam"}

{"x"=>"tuna"}

{"x"=>"squid"}

{"x"=>"clam"}

{"x"=>"tuna"}

{"x"=>"clam"}
5. (6 points) Suppose that Ruby passed parameters by reference. What would the output be in that case for the program in Question 4?

```ruby
{"x"=>"clam"}
{"x"=>"starfish"}
{"x"=>"squid"}
{"x"=>"clam"}
{"x"=>"tuna"}
{"x"=>"starfish"}
```
6. (10 points) Write a Prolog rule `index_of(X, Xs, N)` that finds the element at a given position in a list. You can assume that `N` is an integer in the goal. However, either `X` or `Xs` or both could be variables. Use `is` for arithmetic. Examples:

- `index_of(X, [a, b, c, d], 2)` should succeed with `X=c`
- `index_of(X, [a, b, c, d], 10)` should fail
- `index_of(X, [], 0)` should fail

```prolog
index_of(X, [X|_], 0).
index_of(X, [_|Xs], I) :- I > 0, I2 is I-1, index_of(X, Xs, I2).
```

7. (6 points) Using your rule from Question 6, what are all the answers returned for the following goals? If there are infinitely many, give the first three. Write `false` if the derivation fails. If your answer involves variables generated by Prolog, make up names like this: `_42` (the exact number you use in the name doesn’t matter).

(a) `index_of(b, [a, b, c, d], 3)` => `false`
(b) `index_of(a, Xs, 0)` => `Xs = [a|_]`
(c) `index_of(a, Xs, 2)` => `Xs = [_2, _3, a|_]`
8. (10 points) Rewrite your Prolog rule from Question 6 to use constraints on integers, using the clpfd library. Hint: to remind you of the syntax for constraints in clpfd, here are examples of constraints on $K$: $K > 5$, $K = J + 4$, $K >= 0$.

index_of(X, [X|_], 0).
index_of(X, [_|Xs], I) :- I > 0, J = I - 1, index_of(X, Xs, J).

9. (6 points) Using your improved rule from Question 8, what are all the answers returned for the following goals? If there are infinitely many, give the first three. Write false if the derivation fails. If your answer involves variables generated by Prolog, make up names like this: _42 (the exact number you use in the name doesn’t matter).

(a) index_of(b, [a, b, c, d, a, b, c, d], N) =>
   N = 1
   N = 5
(b) index_of(X, [a, b, c], N) =>
   X = a, N = 0
   X = b, N = 1
   X = c, N = 2
(c) index_of(a, Xs, N) =>
   Xs = [a|_1], N = 0
   Xs = [_, a|_3], N = 1
   Xs = [_, _, a|_6], N = 2
   ...
10. (10 points) Here are some groups of statements about Java types. Circle all statements that are correct as far as the Java compiler is concerned. In addition, write an E on the line next to each statement if that statement is correct as far as the Java compiler is concerned, but that could result in a runtime exception due to a type error.

For example, suppose that one group of statements is

Rectangle2D is a subtype of RectangularShape  
RectangularShape is a subtype of Rectangle2D  
Neither is a subtype of the other

You would circle “Rectangle2D is a subtype of RectangularShape” because that statement is correct as far as the Java compiler is concerned. You would not write an E next to it, since this could never result in a runtime exception due to a type error. You would not circle the other two statements.

Hint: note that any type T is a subtype of itself.

(a)  
Integer is a subtype of Object
Object is a subtype of Integer
Neither is a subtype of the other

(b)  
Integer[] is a subtype of Object[] E
Object[] is a subtype of Integer[]
Neither is a subtype of the other

(c)  
LinkedList<Integer> is a subtype of LinkedList<Object>
LinkedList<Object> is a subtype of LinkedList<Integer>
Neither is a subtype of the other

(d)  
LinkedList<?> is a subtype of LinkedList<?> extends RectangularShape
LinkedList<?> extends RectangularShape is a subtype of LinkedList<?>
Neither is a subtype of the other

(e)  
LinkedList<?> is a subtype of LinkedList<?> extends Object
LinkedList<?> extends Object is a subtype of LinkedList<?>
Neither is a subtype of the other
11. (10 points) True or false? Write T or F on the line in front of the question.

(a) F Racket’s eq? function could be added to OCTOPUS as a new primitive function.
   Object identity is not meaningful for ordinary Haskell data, so adding eq? to OCTOPUS would require changing how OCTOPUS data is represented. For example, (equal? (cons 1 '()) (cons 1 '())) evaluates to #t in both Racket and OCTOPUS, as it should. What about (eq? (cons 1 '()) (cons 1 '()))? In Racket this evaluates to #f — getting this to also evaluate to #f in OCTOPUS would require changing the representation of OctoValue.

(b) T Adding support for floating-point numbers to OCTOPUS would require changes to the lexer and/or parser, in addition to changes to the interpreter.

(c) F The class class in Ruby is a subclass of itself.

(d) T The class class in Ruby is an instance of itself.

(e) F Any two Haskell lists can be tested for equality, since the list type is in the Eq type class.
   (Consider a list of functions, for example [sin,cos]==[sin,cos]. Two lists can be tested for equality only if the elements are in the Eq type class.)

(f) T Any let* expression in Racket can be rewritten as a set of nested let expressions.

(g) F Adding a cut to a Prolog program may change the number of answers that are returned, but will never result in different answers.
   We ended up throwing out this question. The intended answer was F (consider removing the cut from the definition of not in the lecture notes, since with the cut you can get a fail and if no cut some answer) — but you could argue that is still just changing the number of answers.

(h) F Java methods can be contravariant in the return type.
   (But they can be covariant in the return type.)

(i) T Java methods can be overloaded based on the declared types of the method arguments.

(j) F In Ruby, a singleton class has only one superclass, but other classes may have multiple superclasses.

12. (12 points) Consider the following Ruby class definitions.

```ruby
class Book
  attr_reader :author, :title
  def initialize(author, title)
    @author = author
    @title = title
  end
  def description
    title + " by " + author + "."
  end
end

class Textbook < Book
  attr_reader :subject
  def initialize(author, title, subject)
    super(author, title)
    @subject = subject
  end
  def description
    return super + " A textbook about " + subject + "."
  end
end
```
Suppose we make three objects `b`, `t`, and `a` by evaluating these statements:

```ruby
b = Book.new("J.K. Rowling", "Harry Potter and the Deathly Hallows")
t = Textbook.new("James Stewart", "Calculus", "mathematics")
a = AnonymouslyWrittenBook.new("Haskell Good")
```

Then what is the result of evaluating each of these expressions? Hint: the `instance_variables` method returns an array of instance variable names, like this: `[:@x, :@y]`.

**b**
Harry Potter and the Deathly Hallows by J.K. Rowling.

**t**
Calculus by James Stewart. A textbook about mathematics.

**a**
Haskell Good by anonymous.

**b.instance_variables**
`[:@author, :@title]`

**t.instance_variables**
`[:@author, :@title, :@subject]`

**a.instance_variables**
`[:@title]`

Note that since it doesn’t call the inherited `initialize` method, `a` doesn’t have an `author` instance variable.