CSE 341: Homework 6

due 11/22/99

All functions you are asked to implement and turn in should be typed and tested with Miranda. With each function turn in the source code listing as well as sample output for a demonstrative set of tests. Do not forget to test *end cases*. Use the `turnin` program to turn in an electronic copy of your source code.

Part I: (Not to be turned in.)

Try these before section.

Suppose that the following Miranda script has been filed in:

```
cube x = x*x*x
my_map f [] = []
my_map f (x:xs) = f x : my_map f xs
my_append [] x = x
my_append (x:xs) ys = x : my_append xs ys
my_map2 f [] [] = []
my_map2 f (a:as) (b:bs) = f a b : my_map2 f as bs
rev f x y = f y x
alligator x y = 3+y
```

What is the result of evaluating the following Miranda expressions? (If there is a compile-time type error, or a run-time error, or a non-terminating computation, say so.) If the expression is followed by `::` then give the type, instead of the value.

```
cube 3
cube ::
my_map ::
my_map cube ::
my_map cube [1..]
my_append ::
my_map2 alligator [1..]  [10..]
my_map2 ::
my_map2 alligator ::
my_map2 (rev alligator) ::
alligator (1/0) (6/2)
alligator (6/2) (1/0)
alligator ::
(rev alligator) 10 20
rev ::
rev my_map2::
```

Part II: (Not to be turned in.)

These are some sample problems that we will be discussing in section.

1. Write a function `midpoint` that takes in two tuples containing two numbers each and computes and returns the midpoint. For example:

```
midpoint (2,0) (8,2)                  => (5.0,1.0)
midpoint (-5,1) (5,-1)                => (0.0,0.0)
```

2. Write a Miranda function `no_duplicates` that takes a list of numbers as an argument, and returns a new list which is like the argument but with duplicate elements dropped. The order of the elements in the returned list doesn't matter. For example:

```
no_duplicates [1, 3, 3, 8, 5, 3, 5]  => [1, 3, 8, 5]
no_duplicates [1, 3, 5]  => [1, 3, 5]
no_duplicates []  => []
```

Part III: (Due Monday, 11/22/99. Turn this in.)

Complete the following. You do not have to handle errors in input gracefully. **As usual, you must turn in output of your code on a demonstrative set of test cases.**

1. (5 points) Write a function `is_closer_than` that computes which point is closer to the origin, $(0, 0)$. It returns True if the first point is strictly closer to the origin than the second point. False otherwise. For example:

```
is_closer_than (2, 0) (8, 2)              => True
is_closer_than (-5, 1) (5, -1)            => False
```

2. (5 points) Write a *recursive* function `partition_points` takes as parameters a point, `x`, and a list of points, `lst`. It returns a tuple where the first item is a list of all the points in `lst` that are closer than `x` and the second item is a list with the rest of the points. For example:

```
partition_points (3, 0) [(1, 0), (3, 0), (4, 0)]
          => ([(1,0)], [(3,0) (4 0)])
```

3. (5 points) Write a recursive function `quicksort_points` that takes a list of points as its only parameter, and uses `partition_points` and the primitive function `++` (which appends two lists) to sort the points in increasing distance from the origin. For example:

```
quicksort_points [(3, 0), (2, 0), (-4, 0), (1, 0)]
          => [(1, 0), (2, 0), (3, 0), (-4, 0)]
```

4. (5 points) Write a function that, given a sorted list of unique characters, generates an infinite list that contains all possible strings generated with those characters in the following order. For example:

```
take 8 (gen_strings ['a','b'])
          => ["","a","b","aa","ab","ba","bb","aaa"]
```