Fall 2017
Programming Languages
Homework 3

- This homework includes an ML programming assignment and short answer questions. You should use Standard ML of New Jersey for the programming portion of the assignment. A link is available on NYU Classes.
- Due on Wednesday, November 22, 2017 at 5:00 PM Eastern Standard time. Submit a single ML solution file for the ML question as an attachment. Submit the short answers inline or as a PDF attachment.
- Late submissions will be accepted at a penalty of 15 points per day until Friday, November 24, 2017 at 5:00 PM, after which submissions will not be accepted.
- For the ML portion of the assignment, do not use imperative features such as assignment `:=`, references (keyword `ref`), or any mutable data structure, such as `Array`.
- You may use any published ML references in answering the questions. You may consult with other students to understand the ML language in general. All homework solutions including algorithmic details, comments, specific approaches used, actual ML code, etc., **must** be yours alone. Plagiarism of any kind will not be tolerated.
- Please see `http://www.smlnj.org/doc/errors.html` for information on common ML errors. Look in this document first to resolve any queries concerning errors before you ask someone else.
- There are 100 possible points. For the ML question, you will be graded based on program correctness, compliance with all requirements, use of comments, and programming style.

1. [15 points] **Garbage Collection 1**

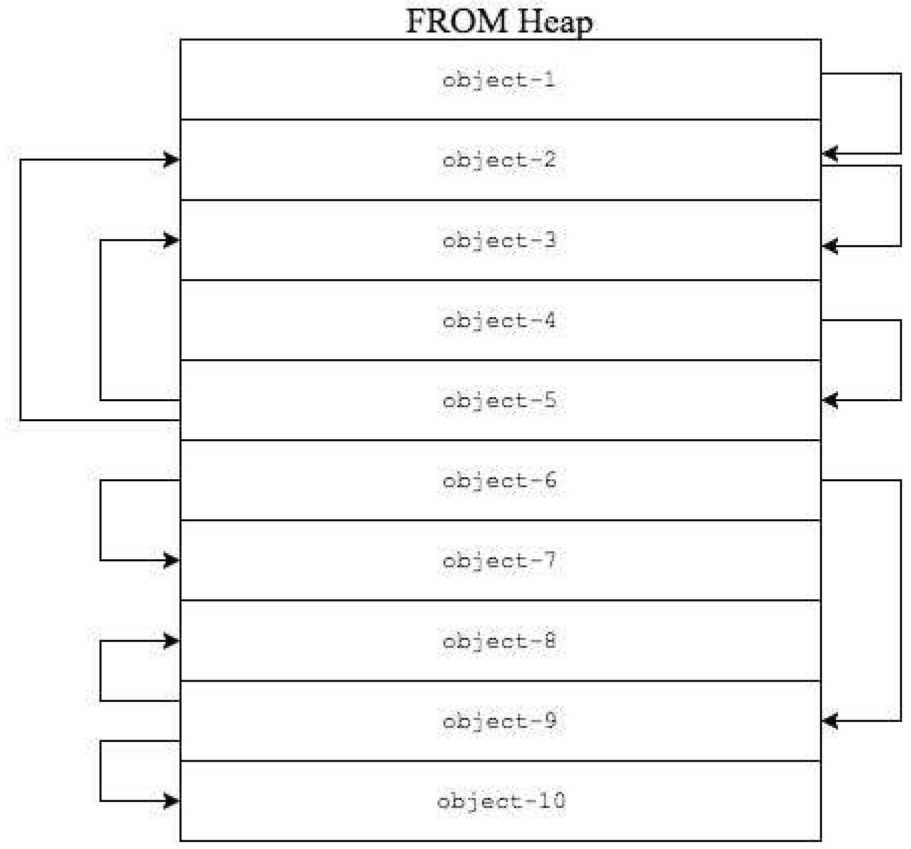
Consider the following pseudo-code:

```
  public class Program
  {
    // entry point
    public static void Main ()
    {
      Car carA = new Car(Color.Red, 200);
      Vehicle carB = new Car(Color.Blue, 600);
      Initialize(carB);
      Vehicle carC = carA;
      BeginEventLoop();
    }

     // initialize a car
   protected static void Initialize(Car c)
   {
      Carwash w = new Carwash();
      w.Wash(c);

      Garage g = new Garage();
      g.Store(c);
    }

    public static void BeginEventLoop() { ... }
 }

public class Vehicle { ... } // 20 bytes
public class Car extends Vehicle // 24 bytes total (including Vehicle's fields)
{
    private List<Wheel> wheels = new List<Wheel>();  // Assume 0 bytes

    Car(Color col, int horsePower) // constructor
      {
        for (int x=0; x < 4; x++)
          { wheels.Add(new Wheel()); }
      }
 }

 public class Carwash { ... }  // 32 bytes
 public class Garage  { ... }  // 32 bytes
 public class Wheel { ... } // 12 bytes
```

Assume the variables (e.g. `carA`, `carB`, etc.) are roots.

Draw a diagram showing the contents of FROM and TO space assuming copy collection fully runs just as `BeginEventLoop` executes. In your diagram, show where each variable is pointing at this moment in time. Also show the memory address of each object, assuming that FROM space begins at address 0 and TO space begins at 500. For objects with a non-null forwarding address, draw the forwarding address as an arrow pointing from the original object to the destination object. Assume that forwarding addresses occupy 8 bytes *in addition to* the size of each object itself.

2. [15 points] **Garbage Collection 2**



The above represents the FROM heap for a program that utilizes copy collection. Assume that the roots point to objects 1,5,6 and 9. Draw the FROM and TO space after the call to `traverse` for each of the roots, assuming they are processed in the order listed above. Also draw all non-null forwarding address pointers using arrows, similarly to the question above. To be clear, you should draw 4 heaps (each with FROM and TO space) in total.

3. [20 points] **Tail Recursion**

A *tail call* is a subprogram call that executes as the last step of a subprogram. A similar concept, *tail recursion*, occurs when a tail call in a subprogram t_n ultimately calls t_n again. This could mean that t_n calls t_n directly, or that t_n calls t_1 resulting in a sequence of tail calls $t_1, \ldots, t_n$.

A tail-recursive subprogram can be optimized to occupy no more than one activation record. This is accomplished by transforming the recursive calls into a loop such that the execution of each tail call would be transformed into one loop iteration. This optimization can be applied not just in functional languages, but imperative ones as well.

Many optimizing compilers apply the tail recursion optimization automatically. In fact, many of your recursive programs may have had the tail recursion optimization applied by the compiler without your knowledge. As an exercise, however, we will be doing this by hand to illustrate how the compiler would do it.

Interesting side note: the tail recursion optimization is the basis for the so-called *continuation passing style* (CPS) in functional languages, whereby functions never actually return. Instead, when their work is completed they invoke another function and pass a functional expression representing "the remainder of the program" (i.e., the continuation) to this function as an extra parameter.

Now consider the following C# code:

```
public static int f (int x)
{
    if (x == 0)
        return 0;

    return 1 + g(x - 1);
}

public static int g (int x)
{
    if (x == 0)
        return 0;

    return 2 + f(x - 1);
}
```

Methods `f` and `g` are not tail recursive because the final operation is not a tail call.

1. Modify the definitions of `f` and `g` such that they perform the same computation as above, but are tail recursive. Call the new methods `f_tc` and `g_tc` (where `tc` is short for "tail recursive."). That is, the final operation in `f_tc` should be the call to `g_tc` and the final operation in `g_tc` should be the call to `f_tc`.
2. Perform the tail recursion optimization manually: write a *single* method `fg_iter` that performs the same computation above, except using a loop instead of recursion. This method should accept three arguments: the two from above and a third argument specifying which of the two methods `f` or `g` is the first to execute. Your new implementation should contain no recursive calls and, thus, occupy only a single activation record.
3. Suppose you wish to know if the compiler you are using is performing the tail recursion optimization described above. Given your knowledge from above, write a routine that will test for the presence of the tail recursion optimization. The routine should demonstrate some sort of behavior that, when executed, will indicate that the routine has undergone the optimization or not. Write the routine and then explain the behavior you would be looking for to indicate that the tail recursion optimization is present (or not).

Although the examples above are written in C#, your solution can be written in either C# or pseudo-code. We will not be nitpicking over syntax, but make sure your solution makes sense. Write the answer on paper–do not attach source code files.

4. [50 points] **ML**

   Implement each of the functions described below, observing the following points while you do so:

   - You may freely copy and call any routines presented in the lecture slides.
   - Make an effort to avoid unnecessary coding by making your definitions as short and concise as possible. For example, a lot of ugly code can be avoided by using `foldl` or `foldr`.
   - Operator `div` is used for integer division, whereas `/` is used for real division.
   - You may find it necessary to typecast an int to a real. The syntax for doing so is `real(int_expr)`.
   - Make sure that your function's signature *exactly* matches the signature described for each function below.
   - You *will* encounter bizarre errors while you write your program and most of the time they will result from something quite simple. The first page of this assignment contains a link to a page which discusses the most common ML errors and an English translation of what each of them mean. Consult this before approaching anyone else. Google also exists.

   1. Write a function `avg_list : real list -> real` that averages a list of real numbers.
   2. Write a function `get_index : 'a list -> int -> 'a` which, given a list and index, retrieves the value of a zero-based index in the list. For example, `get_index [1,2,3] 1` evaluates to 2. Raise an exception if the index provided is not valid for the input list. To do this, declare `exception NoItem` at the top level. Then, from within your routine, write `raise NoItem`.
   3. Write a function `get_odd_midpoint : 'a list -> 'a` that evaluates to the midpoint of an odd-length list. For example, the midpoint of `[2, 7, 3, 12, 1]` is 3. You can assume that only odd-length lists will be supplied to the routine and that exception handling is not necessary here.
   4. Write a function `get_even_midpoint : real list -> real` that evaluates to the *average* of the two middle items in an even-numbered list. For example, the even midpoint of `[5.2, 2.3, 6.2, 9.3]` is the average of 2.3 and 6.2, or 4.25. You can assume that only even-length lists will be supplied to the routine and that exception handling is not necessary here. *Hint: "let" clauses exist.*

      Thought exercise (not to turn in): why are the signatures for `get_odd_midpoint` and `get_even_midpoint` different?
   5. Write a function `get_median : real list -> real` that evaluates to the median of the input list. (I recommend looking up the mathematical definition of median. You should find the lecture slides and the routines you wrote above helpful for solving this problem).
   6. Implement a function `listsum : int list -> int -> bool`: given an input integer list and an integer n, compute the sum of the numbers in the input list and evaluate to `true` if the sum is n, or `false` otherwise.
   7. Implement a function `isten : int list -> bool` which, given a list, determines if the sum equals 10. Write the function in terms of the function `listsum` above using partial application. See the lecture slides for ideas.
   8. Write a function `zip: 'a list * b list -> ('a * 'b) list` that takes a pair of lists (of equal length) and returns the equivalent list of pairs. Raise the exception `Mismatch` if the lengths don't match.
   9. Write a function `unzip : ('a * 'b) list -> 'a list * 'b list` that turns a list of pairs (such as generated with `zip` above) into a pair of lists.
   10. Write a function `scan_left : ('a -> 'b -> 'a)  -> 'a -> 'b list -> 'a list` that returns a list of each value taken by the accumulator during the processing of a fold.
       For example:
       `scan_left (fn x => fn y => x+y) 0 [1, 2, 3]` would return [0, 1, 3, 6].
       Hint: try starting with this curried definition of foldl:

```
 fun myfoldl F y [] = y
   | myfoldl F y (x::xs) = F (myfoldl F y xs) x;
```

11. Using `scan_left`, implement a function `fact_list : int -> int list` that takes an int n and returns $[1, 2!, ..., (n-1)!, n!]$. You may use the following function in your implementation:

```
fun countup n =
  let fun countup' 0 l = l
           | countup' i l = countup' (i - 1) (i::l)
        in
             countup' n []
       end
;
```

After you implement each function, check the type signature of the function and confirm that it matches what you see above. If the signature is not the same, it's automatically wrong. If you find yourself writing complicated or long-winded solutions, you are probably viewing the problem imperatively rather than functionally.