Assignment 7
Due: Sunday, May 24, 11:59PM

Problem 1: Lazy evaluation and functions

It is possible to evaluate function arguments at the time of the call (eager evaluation) or at the time they are used (lazy evaluation). Most programming languages (including ML) use eager evaluation, but we can simulate lazy evaluation in an eager language such as ML by using higher-order functions. Consider a sequence data structure that starts with a value and continues with a function (known as a thunk) to compute the rest of the sequence:

```
- datatype 'a Seq = Cons of 'a * (unit -> 'a Seq);

- fun head (Cons (x,_)) = x;
val head = fn : 'a Seq -> a

- fun tail (Cons (_ , xs)) = xs();
val tail = fn : 'a Seq -> 'a Seq
```

This lazy sequence data type provides a way to create infinite sequences, with each infinite sequence represented by a function that computes the next element in the sequence. For example, here is the sequence of infinitely many 1s:

```
- val ones = let fun f() = Cons(1,f) in f() end;
```

We can see how this works by defining a function that gets the nth element of a sequence and by looking at some elements of our infinite sequence:

```
- fun get(n,s) = if n= 0 then head s else get(n-1,tail s);
val get = fn : int * 'a Seq -> 'a

- get(0,ones);
val it = 1 : int

- get(5,ones);
val it = 1 : int

- get(245, ones);
val it = 1 : int
```

We can define the infinite sequence of all natural numbers by

```
- val natseq = let fun f(n)() = Cons(n,f(n+1)) in f(0) () end;
```

Using sequences, we can represent a function as a potentially infinite sequence of ordered pairs. Here are two examples, written as infinite lists instead of as ML code (note that ? is a negative sign in ML):

```
add1 = (0, 1) :: (~1, 0) :: (1, 2) :: (~2, ~1) :: (2, 3) :: ...
double = (0, 0) :: (~1, ~2) :: (1, 2) :: (~2, ~4) :: (2, 4) :: ...
```

Here is ML code that constructs the infinite sequences and tests this representation of functions by applying the sequences of ordered pairs to sample function arguments.

```
- fun make_ints(f) =  let
                  fun make_pos (n) = Cons( (n, f(n)), fn()=>make_pos(n+1))
                  fun make_neg (n) = Cons( (n, f(n)), fn()=>make_neg(n-1))
               in
               merge (make_pos (0), make_neg(~1))
               end;
 val make ints = fn : (int -> 'a) -> (int * 'a) Seq

- val add1 = make_ints (fn(x) => x + 1);
val add1 = Cons ((0,1),fn) : (int * int) Seq

- val double = make_ints (fn(x) => 2*x);
val double = Cons ((0,0),fn) : (int * int) Seq

- fun apply (Cons( (x1,fx1), xs) , x2) = if (x1 = x2) then fx1 else apply (xs(), x2);
val apply = fn : (‘a * ‘b) Seq * ‘a-> ‘b

- apply(add1, ~4);
val it = ~3 : int

- apply(double, 7);
val it = 14 : int
```

- Write merge in ML. Merge should take two sequences and return a sequence containing the values in the original sequences, as used in the `make_ints` function.
- Using the representation of functions as a potentially infinite sequence of ordered pairs, write compose in ML. Compose should take a function `f` and a function `g` and return a function `h` such that `h( x)` = `f ( g( x))`.

Problem 2: Lazy Lists Revisited

You have seen how to implement a lazy list in ML. In Haskell, a lazy list requires no special effort, since *all* data structures are lazy by default. In particular, the built-in list type is lazy.

1. Define in Haskell an infinite list called *code* that is simply a never-ending sequence of ones: $1, 1, 1, 1, 1, \ldots$;
2. Write a Haskell function *intList* n that will create a sequence of integers from n to infinity: $n, n+1, n+2, \ldots$ (You may **not** use the special built-in list syntax for this; build the list using only the cons operator *(:)*)
3. Write a Haskell function *takeN* that returns the first n elements from a list. (Do not use any standard functions for this.) For example,

   ```
   takeN 4 (intList 10)
   ```

   should evaluate to:

   ```
   [10, 11, 12, 13]
   ```

Problem 3: Implementation of Laziness

- The SML interpreters you have written before implement call-by-value parameter passing, in which variables are evaluated as soon as they are defined. This results in eager behavior, as you've seen in SML. To implement call-by-name, in which variables are not evaluated until they are needed (which results in "lazy" behavior) we need to modify our rules:

  1. For dynamic scope, we have:

```
         env(x) = e    env |- e --> v
(9a) -----------------------------------
        env |- x --> v

      env |- e1 --> (fn x => e3)  env, (x, e2) |- e3 --> v3
(11a) -----------------------------------------------------------
        env |- e1 e2 --> v3
```

  2. For static scope, we have:

```
     env(x) = (e, env1)   env1 |- e ---> v
(9b) ------------------------------------------
          env |- x --> v

(10a)  env |- (fn x => e) --> (fn x = > e, env)

      env |- e1 --> (fn x => e3, env1)  env1, (x, (e2, env)) |- e3 --> v3
(11b) -----------------------------------------------------------------------------
        env |- e1 e2 --> v3
```

  1. Write a new dynamically scoped interpreter

     `interp_lazy: (env * term) -> result`

     which implements call-by-name parameter passing, with dynamic scope.
  2. Write a new statically scoped interpreter

     `interp_lazy_static (env * term) -> result`

     which implements call-by-name parameter passing with static scope.

  You may need to modify your env datatype for one of the problems

- Provide an expression of type `term` which produces different results when evaluating it according to your dynamic lazy and static lazy interpreter, respectively.

Problem 4

Briefly discuss what you would change in your interpreters if the input to the interpreter had been type-checked before its invocation.