```
//Homework 3 - Dane E. Parchment Jr. 4925790

// Questions by Luis Averhoff

F# Homework 3
1.) Given vectors u = (u1, u2,..., un) and v = (v1, v2,..., vn), the inner product of u and v is defined to be u1*v1 + u2*v2
+ ... + un*vn. Write a curried F# function inner that takes two vectors represented as int lists and returns their inner
product:

let rec inner xs ys =
  match xs, ys with
  | [], [] -> 0
  | [], ys -> 0
  | xs, [] -> 0
  | x::xs, y::ys -> x * y + inner xs ys;;

 (*
    The head of the first list (xs) gets multiplied
    by the head of the second list (ys).
    We add the next call to this value and recursion
    takes care of everything.
    1 * 4 +
    2 * 4 +
    3 * 6 = 32
 *)

2.) Given an m-by-n matrix A and an n-by-p matrix B, the product of A and B is an m-by-p matrix whose entry in
position (i,j) is the inner product of row i of A with column j of B. Write an uncurried F# function to do matrix
multiplication.

let rec multiply (xs, ys) =
  match xs, ys with
  | [[]], [[]] -> []
  | _, [] -> []
  | [], _ -> []
  | x::xs, ys -> [List.map (inner x) ((transpose(ys)))] @ multiply (xs, ys);;

 (*
    We use combine, transpose and inner for this one.
    We have to perform the inner product of the the
    first matrix with the transposed version of the
```

```
    second matrix. The problem is that transpose
    takes in a list (ys) and inner takes in the value
    of whatever is in the list (x::xs).
    In order to combine functions that work with
    different types we have to use List.map.
    List.map (inner x) ((transpose(ys))) performs the
    inner product of the first list with the transposed
    version of the second list.
    We append the recursive call at the end with
    @ multiply (xs, ys) catch all values.
 *)
```

3.) Two powerful List functions provided by F# are List.fold and List.foldBack.
These are similar to List.reduce and List.reduceBack, but more general. Both take a binary function f, an initial value i, and a
list [x1;x2;x3;...;xn]. Each of these functions can be used to implement flatten, which "flattens" a list of lists:
Compare the
efficiency of flatten1 xs and flatten2 xs, both in terms of asymptotic time compexity and experimentally.
To make the analysis simpler, assume that xs is a list of the form [[1];[2];[3];...;[n]].

```
// The time complexity for flatten1 is O(n^2) and flatten2 is O(n).

// Questions by Dane E. Parchment Jr.
// Problem #4
// Analyze the below "twice" function and give a definition of the value of the function when
// it is of the form twice twice ... twice successor 0, with the twice function being called k times.
// ------------------------------------------------------------------------------------------------
let twice f = (fun x -> f (f x))
let successor n = n+1
// ------------------------------------------------------------------------------------------------
// Answer & Writeup
// When it is called:
(twice (twice (twice (twice successor)))) 0;;
// It returns: val it : int = 16
// When analyzing it, I called it in differing amounts and noticed a pattern:
// twice successor 0 => 1
// twice twice successor 0 => 2
// twice twice successor 0 => 4
// twice twice twice successor 0 => 16
// twice twice twice twice successor 0 => 65536
// Ok, so I am assumming that you can see the same pattern that I do right? This function seems to be
```

```
// replicating the powers of 2 via nested exponents of seemingly 2, an n-1 amount of times. In this case
// the n seems to represent what it is being exponented to (if exponented is a word, but I think you
// understand what I am trying to say).
//
// For example: 2^3 <==> 2^(2^2).
//
// If we follow the example above with the function calls above, it looks a bit like this:
//twice successor 0 = 2^1 = 2 = 2^1
//twice twice successor 0 = 2^2 = 2^2 = 4
//twice twice twice successor 0 = 2^3 = 2^(2^2) = 2^4 = 16
//twice twice twice twice successor 0 = 2^4 = 2^(2^(2^2)) = 2^(2^4) = 2^16 = 65536
//twice twice twice twice twice successor 0 = 2^5 = 2^(2^(2^(2^2))) = 2^(2^(2^4)) = 2^(2^16) = 2^65536 = stack
overflow!
//
// We can even recursively recreate the function by using **
let rec twiceRedone = function
 | 0 -> 1
 | n -> 2 ** twiceRedone(n - 1);;

// If we were to call it: twiceRedone 3;; <==> 16
//                        twiceRedone 4;; <==> 65536
// ------------------------------------------------------------------------------------------------
// BUGS AND PROBLEMS
// Since this was an analysis question I am certain that there are no bugs, unless I managed to get the
// answer wrong from the beginning!
// ------------------------------------------------------------------------------------------------

// Problem #5
// Map function on infinite streams
//
// Given the type:
type 'a stream = Cons of 'a * (unit -> 'a stream)
// Show how to define map f s on streams; this should give the stream formed by applying function f to each element of
stream s.
// ------------------------------------------------------------------------------------------------
let rec map f (Cons(x, fxs)) =
   Cons(f x, fun() -> map f (fxs()))
// ------------------------------------------------------------------------------------------------
// Writeup
// This one was pretty straightforward, we create a function called map, that accepts two parameters,
// f is the function that we wish to apply to the stream, and the second parameter is the stream
```

```
// itself as defined by the type (so it accepts a generic value and a function that itself returns the
// stream based on said generic value, if I correctly assummed how it worked). All that would be
// necessary then, is to apply the cons function that we defined as a type, as the main function since
// it will return the stream. Within cons our first parameter will be our function applied to the
// generic value x, followed by a function that applies our recursive map function, on f which is
// applied to the fxs function given by cons. This should allow us to iterate through each element
// within the stream (x), and apply a function to it (f), after which it will all be returned as a
// new stream (cons and fxs).
// ----------------------------------------------------------------------------------------------
// BUGS & PROBLEMS
// The code seems to be working as is, though if I misread or misinterpreted the code, then errors may
// occurr.
// ----------------------------------------------------------------------------------------------

// Problem 6
// In this problem, we begin our exploration of the use of F# for language-oriented programming.
// You will write an F# program to evaluate arithmetic expressions written in the language given
// by the following context-free grammar:
// ----------------------------------------------------------------------------------------------
 type Exp =
   Num of int
 | Neg of Exp
 | Sum of Exp * Exp
 | Diff of Exp * Exp
 | Prod of Exp * Exp
 | Quot of Exp * Exp
// ----------------------------------------------------------------------------------------------
 let rec evaluate = function
 | Num n -> Some n
 | Neg e -> match evaluate e with
          | Some x -> Some (-x)
          | _ -> None

  Sum (e1, e2) -> match (evaluate e1, evaluate e2) with
                | Some x, Some y -> Some (x + y)
                | _ -> None
  Diff (e1, e2) -> match (evaluate e1, evaluate e2) with
                | Some x, Some y -> Some (x - y)
                | _ -> None
  Prod (e1, e2) -> match (evaluate e1, evaluate e2) with
                | Some x, Some y -> Some (x * y)
```

```
            | _ -> None
  Quot (e1, e2) -> match (evaluate e1, evaluate e2) with
            | Some x, Some 0 -> None  // Can't divide by 0
            | Some x, Some y -> Some (x + y)
            | _ -> None // This may not be necessary, but just in case I decided to keep it
// Write Up
// This one required a little bit of thought at the beginning, but was relatively straightforward in
// its application after figuring out how the Num, and Neg worked. When understanding the types I looked
// at it like so:
//
// Num of int -> If given Num x we return the number x as an integer
// Neg of Exp -> If given an expression (I am assuming this can be the result of an operator) we
//            we return the negative version of it.
// Sum of Exp * Exp -> The * symbolizes that these will be uncurried (x, y) form, so we have to take
// two expressions and return the added result.
// The same as Sum is repeated for Diff, Prod, and Quot but for their respective operations.
//
// In terms of implementation I will use Sum as an example as the other 3 operations follow the exact
// same template. Anyway, what we do here is provide the Sum type with it's respective arguments. In
// this case 2 uncurried parameters. We then match them using match, and what we target is their
// evaluated values. If their is no values to match, in other words, if our uncurried parameter is empty
// we return None. Otherwise, we take read the uncurried parameter as Some x, and Some y, and return the
// operation on them as Some x + y. This is repeated for all of the other operators.
// ----------------------------------------------------------------------------------------------------
// BUGS & PROBLEMS
// I don't think there are any bugs per se, however for Quot it is possible that the last None statement
// is not necessary since we already check for whether Some y == 0, and that could be argued as
// being empty anyway. OTherwise, I don't think there will be any bugs.
// ----------------------------------------------------------------------------------------------------
```