Name: _____

ID : _____

CSE 130, Winter 2013: Final Examination

March 21st, 2013

- Do **not** start the exam until you are told to.
- This is a open-book, open-notes exam, but with no computational devices allowed (such as calculators/cellphones/laptops).
- Do **not** look at anyone else's exam. Do **not** talk to anyone but an exam proctor during the exam.
- Your answers should match the English description given in the problem, and also produce exactly the sample output given.
- Write your answers in the space provided.
- Wherever it gives a line limit for your answer, write no more than the specified number of lines. *The rest will be ignored.*
- Work out your solution in blank space or scratch paper, and only put your answer in the answer blank given.
- The points for each problem are a rough indicator of the difficulty of the problem.
- Good luck!

1.	15 Points	
2.	40 Points	
3.	10 Points	
4.	35 Points	
5.	30 Points	
TOTAL	130 Points	

1. [**15 points**] In this problem we will write an Ocaml function that takes a matrix, and sums all the integers in it. A matrix is a two dimentional array, which we will represent as a list of lists of integers. For example, the following is a 2×3 matrix (meaning the height of the matrix is 2 and the width is 3):

```
let m = [[ 1; 2; 3];
         [ 4; 5; 6]];;
```

You will use `fold` to write a function `sum_matrix:int list list -> int` which takes a matrix and sums up all the integers in the matrix. For example:

```
# sum_matrix( [[ 1; 2; 3 ]; [ 4; 5; 6 ]]);;
- : int = 21
# sum_matrix( [[ 1; 2 ]; [ 4; 5 ]]);;
- : int = 12
```

Recall that `fold` has type `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.

IMPORTANT: You are only allowed to use fold to manipulate lists. In particular, you are NOT allowed to match against a list, or compare a list to the empty list.

Write your `sum_matrix` below:

---

2. [**40 points**] **Prolog**

a. [**10 points**] Fill in the implementation of the predicate `remove_all(X,L1,L2)`, which holds when `L2` is the same as list `L1`, but with all occurance of `X` removed. For example:

```
1 ?- remove_all(3, [1,2,3,4,5], R).
R = [1, 2, 4, 5] .

2 ?- remove_all(10, [1,10,2,3,10,4,10,5], R).
R = [1, 2, 3, 4, 5] .
```

Fill in the skeleton below:

```
remove_all( _______ , _______ , _______ ).

remove_all( _______ , _______ , _______ ) :- ________________________________________

remove_all( _______ , _______ , _______ ) :- ________________________________________
```

b. [**10 points**] Fill in the implementation of the predicate `remove_first(X,L1,L2)`, which holds when `L2` is the same as list `L1`, but with the first occurance of `X` removed. For example:

```
1 ?- remove_first(3, [1,2,3,4,5], R).
R = [1, 2, 4, 5] .

2 ?- remove_first(10, [1,10,2,3,10,4,10,5], R).
R = [1, 2, 3, 10, 4, 10, 5] .
```

Write your implementation below (**hint**: the code for `remove_first` will look very similar to the code for `remove_all`):

```
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
```

c. [**10 points**] Fill in the implementation of the `prefix(A,B)` predicate below, which holds when `A` is a list prefix of `B`. For example:

```
1 ?- prefix([1,2,3], [1,2,3,4,5]).
true.

2 ?- prefix([1,3], [1,2,3,4,5]).
false.

3 ?- prefix(X, [1,2,3]).
X = [] ;
X = [1] ;
X = [1, 2] ;
X = [1, 2, 3] ;
false.
```

Fill in the skeleton below:

```
prefix( _________ , _________ ).


prefix( _________ , _________ ) :- ________________________________________
```

d. [**10 points**] Fill in the implementation of the `segment(A,B)` predicate below, which holds when `A` is a contiguous segment contained anywhere within list `B`. For example:

```
1 ?- segment([3,4], [1,2,3,4,5]).
true .

2 ?- segment([3,5], [1,2,3,4,5]).
false.

3 ?- segment([X,Y], [1,2,3,4]).
X = 1,
Y = 2 ;
X = 2,
Y = 3 ;
X = 3,
Y = 4 ;
false.

4 ?- segment([3,4,X], [1,2,3,4,5]).
X = 5 ;
false.
```

Fill in the skeleton below (**hint**: use `prefix` for the first case below).

```
segment( _________ , _________ ) :- ________________________________________


segment( _________ , _________ ) :- ________________________________________
```

3. [**10 points**] In this problem we will write a matrix transpose function in Python functions. A matrix is a two dimentional array, which we will represent as a list of lists of integers. For example, the following is a 2×3 matrix (meaning the height of the matrix is 2 and the width is 3):

```
A=[[ 1,  2, 3],
   [ 4,  5, 6]]
```

The *transpose* of a matrix `A` of dimensions $n \times m$ is a matrix `B` of dimensions $m \times n$ such that `A[i][j]` is equal to `B[j][i]` (for all valid indices `i` and `j` into matrix `A`). For example:

```
>>> transpose([[ 1,  2, 3],
               [ 4,  5, 6]])
  [[1, 4],
   [2, 5],
   [3, 6]]

>>> transpose([[ 1,  2],
               [ 3,  4]])
  [[1, 3],
   [2, 4]]
```

Fill in the implementation of `transpose` below:

```
def transpose(m):


   height = len(m)


   width = len(m[0])


   return [ [ ________ for ___ in ___________________ ] for ___ in ___________________ ]
```

4. [**35 points**] In this problem, you will implement Conway's game of life in Python. Here's an explanation of the game, adapted from Wikipedia.

The universe of the Game of Life is a two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, *alive* or *dead*. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The game starts with a given initial generation of dead and alive cells. At each step, the next generation is computed from the current generation by applying the above rules simultaneously to every cell in the generation – births and deaths occur simultaneously, so that the next generation is a pure function of the previous generation.

We will represent the grid of cells as a list of lists of integers, with 0 representing dead and 1 representing alive. For example, the following is the so-called "blinker" starting pattern:

```
g = [[ 0, 0, 0, 0, 0 ],
     [ 0, 0, 0, 0, 0 ],
     [ 0, 1, 1, 1, 0 ],
     [ 0, 0, 0, 0, 0 ],
     [ 0, 0, 0, 0, 0 ]]
```

We can refer to each pixel of the image by its horizontal (x) and vertical (y) coordinate. The top left corner is (0,0) and coordinates increase to the right and down. We can access coordinate (x,y) of the grid `g` by doing `g[y][x]`.

For convenience, here is a useful function for accessing elements of the grid, in such a way that out of bounds elements are considered dead:

```
def access(g, x, y):
   try: return g[y][x]
   except: return 0
```

a. [**10 points**] Write a function `count_live_neighbours(g, x, y)`, which counts the number of live neighbours of the

```
def count_live_neighbours(g, x, y):

   live = 0

   for x_delta in [ ________ , ________ , ________ ]:


      for y_delta in [ ________ , ________ , ________ ]:


         if ____________________________________________________________:


            ____________________________________________________________________________


   return live
```

b. [**15 points**] Now write the function `new_val(g, x, y)` which returns the new value at coordinate `x` and `y`, given the current grid `g`, according to the rules of the game of life (the four rules listed at the beginning of the problem).

```
def new_val(g, x, y):

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________

    ____________________________________________________________________
```

c. [**10 points**] Now fill in the core of the game of life, namely the `step` function below, which computes the next generation from the current generation `g`:

```
def step(g):

    height = len(g)

    width = len(g[0])


    return [ ____________________________________________________________________ ]
```

5. [**30 points**] In this question, you will write a decorator which "lifts" a function so that it operates on arrays of inputs.

a. [**10 points**] We start with lifting a function of one parameter. Fill in the implementation of the `lift_1` decorator below. The `lift_1` takes a function of one argument and returns a function which takes a single array of arguments, and applies the original function on each element of the array, to produce an array of results. For example:

```
>>> @lift_1                          >>> @lift_1
... def inc(x): return x+1           ... def sqr(x): return x*x

>>> inc([1,2,3])                     >>> sqr([1,2,3])
[2, 3, 4]                            [1, 4, 9]
```

Fill in the skeleton below:

```
def lift_1(f):


   def decorated(x):


      return [ ____________________________________________________________ ]


   return decorated
```

b. [**10 points**] We now move to two arguments. Fill in the implementation of the `lift_2` decorator, which takes a function of two arguments and returns a function which takes two arrays (of the same length) of arguments, and applies the original function on corresponding pairs of elements from the arrays, to produce an array of results. For example:

```
>>> @lift_2                          >>> @lift_2
... def plus(x,y): return x + y      ... def mul(x,y): return x * y

>>> plus([1,2,3,4], [4,5,6,7])      >>> mul([10,11], [2,3])
[5, 7, 9, 11]                        [20, 33]
```

Fill in the skeleton below:

```
def lift_2(f):


   def decorated(x,y):


      return [ ____________________________________________________________ ]


   return decorated
```

c. [**10 points**] Consider the following code:

```
@lift_2
@lift_2
def plus(x,y): return x + y
```

Give an illustrating example of how the above defined `plus` function operates on some arguments. Show the arguments, and the result.

Recalling the definition of matrix, what mathematical operation does the above `plus` function perform? Give your answer by adding only **THREE** words:

`The above plus function` _____ _____ _____

d. [**5 points**] **Extra-credit.** We now move on to functions of any number of arguments. Fill in the implementation of `lift` decorator below, which takes a function of n arguments, and returns a function which takes n arrays (of the same length), and applies the original function on the corresponding elements of the n arrays, to produce an array of results. For example, if we replace `lift_1` and `lift_2` with `lift` in all of the above examples, they would still produce the same results. As another example:

```
>>> @lift
... def avg(a,b,c,d): return (a+b+c+d)/4.0

>>> avg([1,2], [5,6], [10,12], [4,8])
[5.0, 7.0]
```

Fill in the skeleton below, **WITHOUT using any additional list comprehensions**. (**hint**: you may want to use a function previously defined in this final):

```
def lift(f):

   def decorated(*args):

      return [ ____________________________________________________________ ]

   return decorated
```