Name: ____________________

ID : ____________________

CSE 130, Winter 2012: Final Examination

March 20th, 2012

- 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.	55 Points	
2.	45 Points	
3.	15 Points	
4.	50 Points	
TOTAL	165 Points	

1. [**55 points**] In this question you are going to implement a dictionary which maps maps strings to integers, and is represented as an OCaml list of pairs. For example, here is a dictionary listing prices of items at your local sporting goods store:

```
let prices = [ ("Baseball Bat", 20); ("Soccer Ball", 10); ("Tennis Racket", 40) ]
```

The first element of each pair is the *key* and the second element is the *value.* There is at most one entry in the dictionary for a given key. Also, the list which represents the dictionary is *sorted* in increasing order of keys. You can use `=`, `<`, `<=`, `>` and `>=` for string comparison.

a. [**15 points**] Fill in the implementation of `find: (string*int) list -> string -> int`. Given a dictionary `d` and a key `k`, `find d k` returns the value for the give key, or raises `Not_found` if the key is not found (by using the command `raise Not_found`). Ideally, we would use a binary search, but since we're using a linked list (which does not support efficient direct access), you should instead do a linear search. However, for full credit, use the sortedness property to stop the search as early as possible.

```
let rec find d k =

    match d with

     | [] -> raise Not_found

     | (k',v') :: t ->  __________________________________________________________

   _______________________________________________________________________________

   _______________________________________________________________________________

   _______________________________________________________________________________
```

b. [**15 points**] Fill in the implementation of `add: (string*int) list -> string -> int -> (string*int) list`. Given a dictionary `d`, key `k` and value `v`, `add d k v` returns a new dictionary where the key `k` is bound to `v`. If the key already exists, the value for that key is updated. The input dictionary is sorted by keys in increasing order, and the returned dictionary should also be sorted in the same way. For example:

```
# add prices "Figure Skates" 100;;
- : (string * int) list =
[("Baseball Bat", 20); ("Figure Skates", 100); ("Soccer Ball", 10); ("Tennis Racket", 40)]

# add prices "Aikido Suit" 20;;
- : (string * int) list =
[("Aikido Suit", 20); ("Baseball Bat", 20); ("Soccer Ball", 10); ("Tennis Racket", 40)]

# add prices "Soccer Ball" 30;;
- : (string * int) list =
[("Baseball Bat", 20); ("Soccer Ball", 30); ("Tennis Racket", 40)]
```

Fill in the implementation of `add` below:

```
let rec add d k v =

    match d with

     | [] ->  ____________________________________________________________________

     | (k',v') :: t ->  __________________________________________________________

   _______________________________________________________________________________

   _______________________________________________________________________________

   _______________________________________________________________________________
```

c. [**5 points**] Use `map` to write a function `keys:(string*int) list -> string list`. Given a dictionary `d`, `keys d` should return the list of keys. For example, `keys prices` should return `["Baseball Bat";"Soccer Ball";"Tennis Racket"]`. Recall that `map` has type: `('a -> 'b) -> 'a list -> 'b list`.

```
let keys d = __________________________________________________________________

   _______________________________________________________________________________
```

d. [**5 points**] Use `map` to write a function `values:(string*int) list -> int list`. Given a dictionary `d`, `values d` should return the list of values. For example, `values prices` should return `[20,10,40]`.

```
let values d = __________________________________________________________________


   ______________________________________________________________________________
```

e. [**15 points**] Fill in the implementation of `key_of_max_val:(string*int)list -> string` by using `fold_left`. Given a dictionary `d`, `key_of_max_val d` returns the key of the maximum value in the dictionary (or raises `Not_found` if there are no entries in the dictionary). If multiple keys have the maximum value, then you may return either one. For example `key_of_max_val prices` would return `"Tennis Racket"`. Recall that the type of `fold_left` is `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.

```
let key_of_max_val d =

   (* definition of fold_fn *)

   let __________________________________________________________________________


   ______________________________________________________________________________


   ____________________________________________________________________________ in


   match d with

     | [] -> raise Not_found

     (* the call to fold should be: fold_left fold_fn base t *)


     | base::t ->  ______________________________________________________________


                   ______________________________________________________________
```

2. [**45 points**] In this problem we will write several Python functions to do basic manipulations of images. Images will be represented as lists of lists of integers, with each integer representing a pixel. For example, the following would be a simple image of a smiley face.

```
img1=[[ 11,  0, 12],
      [  0,  0,  0],
      [ 13,  0, 14],
      [ 15, 16, 17]]
```

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 an image `img` by doing `img[y][x]`.

a. [**10 points**] Fill in the body of the function `square_img`, which takes an image, and squares each integer in it. For example, `square(img1)` returns:

```
[[121,   0, 144],
 [  0,   0,   0],
 [169,   0, 196],
 [225, 256, 289]]
```

Fill the implementation of `square_img` below. Remember that you can only add code in the blank (and note that the expression being returned is a list comprehension, which has inside of it another list comprehension).

```
def square_img(img):



  return [ [ ________________________________________________________ ] for l in img ]
```

b. [**10 points**] Fill in the body of the function `crop_img`. The call `crop_img(img, x1, y1, x2, y2)` returns an image which only contains the pixels from `img` at coordinates `(x,y)`, where `x1 <= x < x2` and `y1 <= y < y2`. You can assume that all such coordinates exist in `img`. For example `crop_img(img1,0,1,2,4)` would return:

```
img1=[[  0, 0],
      [ 13, 0],
      [ 15, 16]]
```

Fill the implementation of `crop_img` below. Again, remember that you can only add code in the blank, and note that the expression being returned is a list compression (hint: you will need to use list slicing).

```
def crop_img(img,x1,y1,x2,y2):



  return [ ____________________________________________________________________________ ]
```

c. [**15 points**] To write our last image function, we will need a helper function called `zip`. Write the `zip` function below. Given lists `l1` and `l2`, `zip(l1,l2)` returns a list of pairs. The nth element of the returned list is a pair consisting of the nth element of `l1` and the nth element of `l2`. If one of the lists is smaller than the other, the returned list contains pairs only for indices that both lists have. For example, `zip([1,2,3], [4,5,6])` returns `[(1, 4), (2, 5), (3, 6)]`, and `zip([1,2,3], [4,5])` returns `[(1, 4), (2, 5)]`. You can `min` and `max` in your solution.

```
def zip(l1,l2):

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

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

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

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

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

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

d. [**10 points**] Fill the implementation of `add_imgs` below. Given two images `img1` and `img2` of the same size, `add_imgs(img1, img2)` returns an image where each pixel is the sum of the corresponding pixels from `img1` and `img2`. For example, `add_imgs(img1, img1)` returns:

```
[[22,  0, 24],
 [ 0,  0,  0],
 [26,  0, 28],
 [30, 32, 34]]
```

Fill the implementation of `add_imgs` below (hint: you will need another call to `zip` in the inner list comprehension).

```
def add_imgs(img1, img2):

  return [ [ ______________________________________ ] for (l1,l2) in zip(img1,img2)]
```

3. [**15 points**] In this question, you will write a decorator that transforms a mathematical function of one variable into its derivative. We will use a numerical approximation for computing the derivative. In particular, given a function f from reals to reals, and a small real number Δ, we define the "delta-derivative" f'_Δ as:

$$f'_\Delta(x) = (f(x + \Delta) - f(x))/\Delta$$

Write a decorator `derivative` which, given a parameter `delta`, converts a math function of **one** parameter into its "delta-derivative".

Because we compute the derivative using an approximation (the exact result is the limit as Δ goes to zero), and because Python floats are imprecise, your `derivative` decorator should round the result it returns to 2 decimals by using the `round` built-in function: the expression `round(f,d)` rounds the float `f` to `d` decimals.

Below are some sample outputs. Notice how the output produced by our "delta-derivative" coincides with what you learned in calculus class, namely: for $f(x) = 2x$, the derivative is $f'(x) = 2$; for $f(x) = x^2$, $f'(x) = 2x$; and for $f(x) = x^3$, $f'(x) = 2x^2$. Also note how a larger `delta` leads to a less precise answer.

```
>>> @derivative(0.0001)
... def double(x): return 2*x
...
>>> double(10.0)
2.00
>>> double(20.0)
2.00
>>> double(30.0)
2.00

>>> @derivative(0.0001)
... def square(x): return x*x
...
>>> square(10.0)
20.00
>>> square(20.0)
40.00
>>> square(30.0)
60.00

>>> @derivative(0.0001)
... def cube(x): return x*x*x
...
>>> cube(3.0)
27.00
>>> cube(4.0)
48.00
>>> cube(5.0)
75.00

>>> @derivative(0.1)
... def square(x): return x*x
...
>>> square(10.0)
20.10
>>> square(20.0)
40.10
>>> square(30.0)
60.10
```

Write your `derivative` decorator below:

Extra credit. For 5 points extra credit, write the `derivative` decorator using the class-based approach:

4. [**50 points**] In this question, you will implement integer sorting in Prolog in two different ways. First, we will use a simple approach, which is inefficient, and then we will implement a more efficient merge sort. Note that in Prolog, the less-than-or-equal operator is `=<`.

a. [**10 points**] Fill in the implementation of the `sorted` predicate below. The predicate `sorted(L)` should hold if `L` is sorted in increasing order. For example:

```
1 ?- sorted([]).
true.

2 ?- sorted([10]).
true.

3 ?- sorted([1,2,3,3,4]).
true.

4 ?- sorted([10,2,3,3,4]).
false.
```

Fill in the skeleton below (the first two lines are for base cases, and note that `[A,B|T]` matches a list where the first two elements are `A` and `B`, and where `T` is the rest of the list):

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


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


sorted([A,B|T]) :- ____________________________________________
```

b. [**10 points**] Fill in the implementation of `sort` below, using the `sorted` predicate from part (a). The predicate `sort(L1,L2)` holds if `L2` is a sorted version of `L1`. You can make use of the built-in predicate `permutation`, which takes two lists and returns true if the two lists contain the same elements, but possibly in a different order. Here are some examples of `permutation` and `sort`:

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

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

3 ?- sort([4,1,3,2], L).
L = [1, 2, 3, 4].
```

Fill the implementation of `sort` below.

```
sort(L1,L2) :- _____________________________________________________________
```

Extra credit. For 5 points of extra credit, you must do both of the following:

1. Add a cut in your solution for `sort` above, so that once Prolog finds the sorted list, it stops searching (which gives the output listed above for `sort([4,1,3,2], L)`, instead of giving one answer for `L` and waiting for you to type ";" or ".")
2. Explain below why your placement of cut is correct:

____

____

c. [**10 points**] In part (a) and (b) we implemented a simple `sort` predicate. We now start from scratch, and we will implement merge sort. First, we need to implement `split`. The predicate `split(L1,L2,L3)` holds if `L2` contains the even-indexed elements of `L1` (counting from 0) and `L3` contains the odd-indexed elements of `L1`. For example:

```
1 ?- split([0], X, Y).
X = [0],
Y = [].

2 ?- split([0,1,2], X, Y).
X = [0, 2],
Y = [1].

3 ?- split([0,1,2,3,4,5,6], X, Y).
X = [0, 2, 4, 6],
Y = [1, 3, 5].

4 ?- split(X, [0,1,2], [10,11,12]).
X = [0, 10, 1, 11, 2, 12].
```

Fill in the implementation of `split` below:

```
split([], [], []).


split([X], [X], []).


split([X | T], _________, _________) :- split(T, _________, _________).
```

d. [**10 points**] The next step is to implement `merge`, which takes two sorted lists and returns a sorted list containing all the elements of both input lists. Here is the OCaml implementation of `merge` that you need to match in Prolog:

```
let rec merge l1 l2 =
   match (l1, l2) with
   | ([], l) -> l
   | (l, []) -> l
   | (x::t1, y::t2) -> if x <= y then x::(merge t1 l2) else y::(merge l1 t2)
```

The `merge` predicate in Prolog takes three parameters: `merge(L1, L2, L3)` holds if `L3` is the result of merging the sorted lists `L1` and `L2` (meaning you can assume that `L1` and `L2` are sorted). For example:

```
1 ?- merge([1,10], [3,4,20], L).
L = [1, 3, 4, 10, 20] .

3 ?- merge([X,3], [2,4,Y], [1,2,3,4,6]).
X = 1,
Y = 6.
```

Fill in the implementation of `merge` below:

```
merge([],L,L).

merge(L,[],L).


merge( ________, ________, ________) :- ____________________________________________


merge( ________, ________, ________) :- ____________________________________________
```

e. [**10 points**] Finally, you will use `split` and `merge` to implement merge sort. The `merge_sort(L,S)` predicate holds if `S` is a sorted version of `L`. For example:

```
1 ?- merge_sort([20,3,6,2,7], X).
X = [2, 3, 6, 7, 20] .
```

Fill in the implementation of `merge_sort` below:

```
merge_sort(_____, _____).


merge_sort(_____, _____).


merge_sort(L,S) :- ____________________________________________________________


_______________________________________________________________________________


_______________________________________________________________________________
```