Name: _____

ID : _____

CSE 130, Fall 2013: Final Examination

December 9th, 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.
- 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.
- Good luck!

1.	20 Points	
2.	20 Points	
3.	25 Points	
4.	20 Points	
5.	30 Points	
TOTAL	115 Points	

1. [**20 points**] In this problem you will implement insertion sort in OCaml.

a. [**10 points**] First, you will implement insertion into a sorted list. Given a sorted list l and an integer i, (insert l i) returns a sorted list which contains all the elements of l, and in addition also contains the integer i (note that duplicates are allowed). For example:

```
# insert [] 10;;
- : int list = [10]
# insert [1;2;3;4] 3;;
- : int list = [1; 2; 3; 3; 4]
# insert [10;15;20;30] 40;;
- : int list = [10; 15; 20; 30; 40]
# insert [10;15;20;30] 5;;
- : int list = [5; 10; 15; 20; 30]
```

Fill in the code for insert below:

```
let rec insert l i =
```

---

b. [**10 points**] Now you will implement insertion sort using `fold_left`. Recall that the type of `fold_left` is given below:

```
fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Fill in the implementation below using `fold_left`:

```
let insertion_sort =

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

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

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

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

2. [**20 points**] Consider the following data type for representing arigthmetic expressions with variables:

```
type expr =
  | Var of string
  | Const of int
  | Plus of expr * expr
```

For example:

- `(Plus (Const 4, Const 7))` respresents `4+7`
- `(Plus (Const 4, Var "a"))` respresents `4+a`
- `(Plus (Var "a", Var "b"))` respresnts `a+b`
- `(Plus (Plus (Const 20, Const 10), Var "a"))` respresnts `(20+10)+a`

Write a function `simpl: expr -> expr` which simplifies additions of constants. In particular, given an expression `e`, `(simpl e)` returns a new expression which is equivalent to `e`, but in which all additions of constants have been computed. For example:

```
(* 4+7 simplifies to 11 *)
# simpl (Plus (Const 4, Const 7));;
- : expr = Const 11

(* (20+10)+a simplifies to 30+a *)
# simpl (Plus (Plus (Const 20, Const 10), Var "a"));;
- : expr = Plus (Const 30, Var "a")

(* (4+10)+7 simplifies to 21
   More details: 4+10 simplifies to constant 14, then 14+7 simplifies to 21 *)
# simpl (Plus (Plus (Const 4, Const 10), Const 7));;
- : expr = Const 21

(* 4+a simplifies to 4+a  -- nothing to simplify *)
# simpl (Plus (Const 4, Var "a"));;
- : expr = Plus (Const 4, Var "a")

(* a+b simplifies to a+b  -- nothing to simplify *)
# simpl (Plus (Var "a", Var "b"));;
- : expr = Plus (Var "a", Var "b")

(* (a+10)+7 simplifies to (a+10)+7 -- nothing to simplify because a+10 doesn't simplify *)
# simpl (Plus (Plus (Var "a", Const 10), Const 7));;
- : expr = Plus (Plus (Var "a", Const 10), Const 7)
```

Fill in the code below for simpl (note that in the code below, "_" is just the "else" case – there is nothing for you to fill in there).

```
let rec simpl e =

   match e with

   (* case for plus: *)

   | ________________________________________ ->

         let e1' = ______________________________________ in

         let e2' = ______________________________________ in

         match (e1',e2') with

         | ___________________________________ -> ________________________________

         | _ -> ________________________________________

   (* all other cases: *)

   | _ -> ________________________________
```

3. [**25 points**] You will use list comprehension to implement dictionaries. A dictionary here will be a list of pairs, where each pair contains a key and a value. Unlike in regular Python dictionaries, a given key can appear multiple times in the dictionary. **All operations in this question will be functional, meaning that the original dictionary is left unmodified, and a new dictionary is returned.**

a. [**5 points**] First, you will implement a lookup function. Given a dictionary d and key `k`, `lookup(d,k)` returns the list of all values associated with the given key. For example:

```
>>> d = [ ("a", 10), ("b", 20), ("c", 30), ("a", 40) ]
>>> lookup(d,"a")
[10, 40]
>>> lookup(d,"b")
[20]
>>> lookup(d,"c")
[30]
>>> lookup(d,"d")
[]
```

Fill in the implementation of lookup below:

```
def lookup(d,k):



   return [ ____________________________________________________________________ ]
```

b. [**5 points**] You will now implement the update operator. Given a dictionary `d`, a key `k` and a new value `v`, `update(d,k,v)` returns a new dictionary in which all occurrences of the key `k` have been updated to have the value `v`. Note: (1) if there are multiple occurrences of the key `k`, they are all updated (2) if there are no occurrences of key `k`, nothing is updated (3) the update function returns a new dictionary – it does not update the one that is passed in. For example:

```
>>> d = [ ("a", 10), ("b", 20), ("c", 30), ("a", 40) ]
>>> update(d, "a", "CSE130")
[('a', 'CSE130'), ('b', 20), ('c', 30), ('a', 'CSE130')]
>>> update(d, "b", "CSE130")
[('a', 10), ('b', 'CSE130'), ('c', 30), ('a', 40)]
>>> update(d, "d", "CSE130")
[('a', 10), ('b', 20), ('c', 30), ('a', 40)]
```

In your solution, you may find the following function useful:

```
def cond(b, t, f):
   if b: return t
   else: return f
```

Fill in the solution below:

```
def update(d,k,v):



   return [ ____________________________________________________________________ ]
```

c. [**5 points**] You will now implement deletion. Given a dictionary `d` and a key `k`, `delete(d,k,v)` returns a new dictionary in which all entries for the key `k` have been removed.
Fill in the implementation `delete` below:

```
def delete(d,k):


   return [ ____________________________________________________________________ ]
```

d. [**5 points**] You will now implement addition. Given a dictionary `d`, key `k` and value `v`, `add(d,k,v)` returns a new dictionary with the additional key-value pair at the end of the list representing the dictionary.
Fill in the implementation of `add` below:

```
def add(d,k,v):


    return ____________________________________________________________________
```

e. [**5 points**] You will now implement the `update` function from part b, but: **without** using list comprehension and **without** using the helper function `cond`. You can use other built-in functions if you want, but you don't need to.

```
def update(d,k,v):

   ____________________________________________________________________________

   ____________________________________________________________________________

   ____________________________________________________________________________

   ____________________________________________________________________________

   ____________________________________________________________________________
```

4. [**20 points**] In this question you will implement a decorator `in_range`, which you can assume will only be applied to functions that take integers and return integers. Given an integer `i` and a pair `range` of integers, the decorator `in_range(i, range)` adds the following behavior to the decorated function:

1. If `i == -1`, the decorated function should throw an exception if the return value is not in the given range.
2. If `i` is a valid index into the argument list, the decorated function should throw an exception if the `i`th argument is not in the given range.

Here are some examples. Note specifically the strings that are printed out in the exceptions – you need to replicated this behavior.

```
>>> @in_range(0, (0,10))
... @in_range(1, (-10,20))
... def plus(a,b): return a+b

>>> plus(10,-5)
5

>>> plus(11,3)
Exception: 0th arg 11 too big

>>> plus(-1,3)
Exception: 0th arg -1 too small

>>> plus(2,25)
Exception: 1th arg 25 too big

>>> plus(2,-13)
Exception: 1th arg -13 too small



>>> @in_range(-1, (5,10))
... def plus(a,b): return a+b

>>> plus(6,4)
10

>>> plus(3,2)
5

>>> plus(6,5)
Exception: Return value 11 too big

>>> plus(3,1)
Exception: Return value 4 too small
```

To raise an exception with message s, you should use the command `raise Exception(s)`. For example:

```
>>> raise Exception("Hello world!")
Exception: Hello world!
```

Write the implementation of `in_range` below (use `str(x)` to return the string representation of `x`):

5. [**30 points**] We are going to encode a graph over cities in Prolog. In particular, `link(a,b)` represents the fact that there is a path from city `a` to city `b`. For example:

```
link(san_diego, seattle).
link(seattle, dallas).
link(dallas, new_york).
link(new_york, chicago).
link(new_york, seattle).
link(chicago, boston).
link(boston, san_diego).
```

a. [**5 points**] First, write a predicate `path_2(A,B)` which holds if there is path of length two from `A` to `B`. The path is allowed to have duplicate cities. For example:

```
1 ?- path_2(new_york,B).
B = boston ;
B = dallas.

2 ?- path_2(A,dallas).
A = san_diego ;
A = new_york ;
false.
```

Write your implementation of `path_2` below:

---

b. [**5 points**] Write a predicate `path_3(A,B)` which holds if there is path of length three from `A` to `B`. The path is allowed to have duplicate cities. For example:

```
1 ?- path_3(A,B).
A = san_diego,
B = new_york ;
A = seattle,
B = chicago ;
A = B, B = seattle ;
A = dallas,
B = boston ;
A = B, B = dallas ;
A = new_york,
B = san_diego ;
A = B, B = new_york ;
A = chicago,
B = seattle ;
A = boston,
B = dallas.
```

Write your implementation of `path_3` below:

---

c. [**10 points**] Write a predicate `path_N(A,B,N)` which holds if there is a path of length `N` between `A` and `B`. The path is allowed to have duplicate cities, and you can assume that `N` is greater or equal to `1`. For example:

```
1 ?- path_N(new_york, B, 2).
B = boston ;
B = dallas ;
false.

2 ?- path_N(new_york, B, 3).
B = san_diego ;
B = new_york ;
false.

3 ?- path_N(A, san_diego, 5).
A = seattle ;
false.
```

Fill in the implementation of `path_N` below:

```
% case for N = 1

path_N(A,B,N) :- ____________________________________________________________________.


% case for N > 1

path_N(A,B,N) :- ____________________________________________________________________.
```

d. [**10 points**] Write a predicate `path(A,B)` which is true if there is a path from `A` to `B`, **without cycles**. You are allowed to use the built-in predicate `member(X,L)` which is true if `X` is an element of the list `L`. Fill in implementation of `path` below.

```
path(A, B) :- path_helper(A, B, ____________________________).


% In path_helper below, Seen is the cities we have see so far, so we
% can avoid cycles.

path_helper(A, B, Seen) :- link(A,B), not(member(B, Seen)).


path_helper(A, B, Seen) :-

    link(A,C),


    ------------------------------------------------------------ ,


    path_helper( ____________________  , ____________________  , ____________________ ).
```