Name: _____

ID : _____

CSE 130, Winter 2011: Final Examination

March 15, 2011

- 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 explanation and code. *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.	40 Points	
2.	35 Points	
3.	10 Points	
4.	25 Points	
5.	30 Points	
TOTAL	140 Points	

1. [**40 points**] A *dictionary* is a data structure that maps *(string) keys* to *values*. We will represent dictionaries using a polymorphic Ocaml datatype:

```
type 'a dict = Empty | Node of string * 'a * 'a dict * 'a dict
```

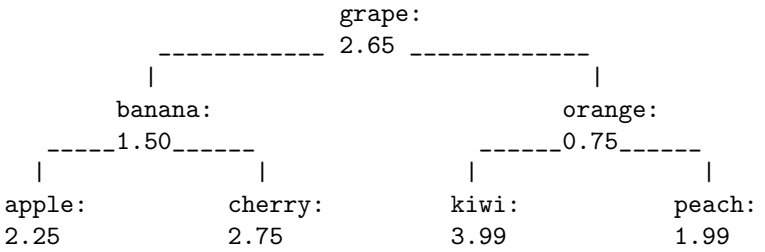
That is, a dictionary is represented as a tree, which is either empty, or a node that has:

1. a binding from a `string` key to an `'a` value,
2. a left sub-dictionary, and,
3. a right sub-dictionary.

For example, the dictionary:

```
apple : 2.25
banana: 1.50
cherry: 2.75
grape : 2.65
kiwi  : 3.99
orange: 0.75
peach : 1.99
```

that represents the prices (per pound) of various fruits, is represented by the tree:



which is encoded by the Ocaml value (of type `float dict`) bound to `fruitd`:

```
let fruitd =
   Node ("grape", 2.65,
         Node ("banana", 1.50,
               Node ("apple",  2.25, Empty, Empty),
               Node ("cherry", 2.75, Empty, Empty)),
         Node ("orange", 0.75,
               Node ("kiwi",   3.99, Empty, Empty),
               Node ("peach",  1.99, Empty, Empty)))
```

Notice that the tree is *Binary-Search-Ordered* meaning that for each node with a key `k`, the keys in the

- *left* subtree are (in alphabetical order) *less than* `k`,
- *right* subtree are (in alphabetical order) *greater than* `k`.

a. [**10 points**] Fill in the blanks below, to obtain an implementation of a function:

```
val find: 'a dict -> string -> 'a
```

such that:

```
find d k
```

returns the value associated with the key `k` in the dictionary `d`. Thus,

```
find fruitd "cherry"
```

should return `2.75`.

```
let rec find d k =
  match d with
  | Empty ->
      raise Not_found
  | Node (k', v', l, r) ->


      if k = k' then ____________________________________ else


      if k < k' then ____________________________________ else


      (* k > k' *)   ____________________________________
```

b. [**10 points**] Fill in the blanks below, to obtain an implementation of a function:

```
val add: 'a dict -> string -> 'a -> 'a dict
```

such that

```
add d k v
```

returns a new dictionary corresponding to *adding* the key-value pair `(k,v)` to the dictionary `d`. Thus, the expression:

```
let d0 = fruitd in
let d1 = add d0 "banana" 5.0 in
let d2 = add d1 "mango"  10.25 in
(find d2 "banana", find d2 "mango", find d2 "cherry")
```

should evaluate to: `(5.0, 10.25, 2.75)`.

```
let rec add d k v =
  match d with
  | Empty ->


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

  | Node (k', v', l, r) ->


      if k = k' then ____________________________________ else


      if k < k' then ____________________________________ else


      (* k > k' *)   ____________________________________
```

c. [**20 points**] Finally, fill in the blanks below, to obtain an implementation of a function:

```
val fold: (string -> 'a -> 'b -> 'b) -> 'b -> 'a dict -> 'b
```

such that

```
fold f b d
```

returns the result of "folding" the function `f` over the tree `d`, starting with the "base" value of `b` (analogous to how `List.fold_left f b xs` "folds" the function `f` over a list `xs` starting with the base value `b`). Your implementation should perform an *In-Order* traversal over the tree. That is, it should use the base value to recursively fold over the left subtree, then apply the result to the node's key-value binding, and then use the result to recursively fold over the right subtree. Thus, the expression:

```
fold (fun k v b -> b^","^k) [] fruitd
```

should concatenate the names of fruits in `fruitd` and return

```
",apple,banana,cherry,grape,kiwi,orange,peach",
```

and the expression:

```
fold (fun k v b -> b + v) 0 fruitd
```

should sum up the prices in the dictionary `fruitd` and return `15.88`.

```
let rec fold f b d =
  match d with
  | Empty ->

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

  | Node (k, v, l, r) ->

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

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

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

2. [**35 points**] In this problem, we will represent Python-style namespaces using Ocaml data structures. Consider the following datatype declaration:

```
type name_space = EmptyNameSpace
                | Info of (string * value) list * name_space

and  value      = Int of int
                | Method of (name_space -> int -> int)
```

A name space is either the empty name space, or it contains some information. The information it contains is a list of string-to-value bindings, along with a pointer to the parent name space. A value is either an int, or it is a method. A method takes a name space as the first parameter (the self pointer), and an additional integer, and returns an integer.

Suppose we had the following Python code:

```
class SimpleObj1:
   a = 0
   def f(self, i): return i+1

class SimpleObj2 (SimpleObj1):
   def g(self, i): return i+2

SimpleObj2()
```

The object created by the call to `SimpleObj2()` would be represented in our OCaml data structures as follows:

```
let method_f self i = i+1
let SimpleObj1 = Info([("a", Int(0)); ("f", Method(method_f))], EmptyNameSpace)

let method_g self i = i+2
let SimpleObj2 = Info([("g", Method(method_g))], SimpleObj1)
```

a. [**15 points**] Write an OCaml function `lookup: name_space -> string -> value` that takes a name space and a name, and searches the name space (and parent names spaces) for the given name. If a value is found, then the value should be returned. If no value is found, you should `raise NotFound`. For example, if you run `lookup SimpleObj2 "a"` you should get `Int(0)` back, and if you run `lookup SimpleObj2 "midori"` you will get an exception `NotFound`.
Write your `lookup` function below:

b. [**10 points**] We will now see how to use the lookup function. First, consider the following simple conversion functions:

```
exception TypeError
let to_int value =
  match value with
  | Int(i) -> i
  | _      -> raise TypeError

let to_method value =
  match value with
  | Method(m) -> m
  | _         -> raise TypeError
```

And consider the following Python code:

```
class SimpleObj3:
   a = 10;
   def f(self, i): return self.a + i

OBJ3 = SimpleObj3()
```

Fill in the OCaml code below so that the object created by `SimpleObj3()` above is represented in OCaml in the OBJ3 variable below (recall that `self` is a namespace!):

```
let method_f self i = (to_int (lookup _____________________________)) + i


let OBJ3 = Info([("a", Int(10)); ("f", Method(method_f))], EmptyNameSpace)
```

c. [**10 points**] Finally, we will write an OCaml function that performs dynamic dispatch. In particular, fill in the code below for the function `invoke_method: name_space -> string -> int -> int`, which takes as parameters a name space (in other words an object), a method name, an integer, and returns the result of applying that method name to the given object with the integer parameter:

```
let invoke_method self name i =

    (to_method (lookup _________________________)) ______________________
```

Now fill in the parameters to the `invoke_method` function below so that it performs the Python dispatch `OBJ3.f(3)`:

```
     invoke_method _________________________________
```

3. [10 points]

Write a decorator `print_first_k_args` that takes a parameter k, and decorates a function by printing, for each call to the function, the first k arguments (or all arguments if the function takes less than k arguments), aswell as the return value. For example:

```
def print_first_k_args(k):
   ...

@print_first_k_args(1)
def sum(a,b): return a + b

>>> sum(3,4)
Arg 1: 3
Return: 7
7

@print_first_k_args(2)
def sum(a,b): return a + b

>>> sum(3,4)
Arg 1: 3
Arg 2: 4
Return: 7
7

@print_first_k_args(3)
def sum(a,b): return a + b

>>> sum(3,4)
Arg 1: 3
Arg 2: 4
Return: 7
7

@print_first_k_args(1)
def fac(n):
    if n <= 0: return 1
    else: return n*fac(n-1)

>>> fac(3)
Arg 1: 3
Arg 1: 2
Arg 1: 1
Arg 1: 0
Return: 1
Return: 1
Return: 2
Return: 6
6
>>>
```

Write the `print_first_k_args` decroator below (hint: `str(x)` returns the string representation of `x`)

4. [**25 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 between 0 and 255. For example, the following would be a simple image of a smiley face.

```
img1=[[255,255,  0,  0,255,255],
      [255,255,  0,  0,255,255],
      [  0,  0,255,  0,  0,  0],
      [  0,  0,255,255,  0,  0],
      [255,  0,  0,  0,  0,255],
      [  0,255,255,255,255,  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 an image `img` by doing `img[y][x]`.

Your job will be to write several functions to create and manipulate such images. Except where specified otherwise, you may assume that the input to all functions is valid and well formed.

Hint: (for several parts) Accessing an index in a list beyond the bounds of the list raises an `IndexError`. It may be easier to just catch the exception than to check the bounds yourself.

a. [**5 points**] Fill in the body for the function `create_image` such that it returns a new image with width `w`, height `h`, and every pixel colored `c`. The rows of the image should *not* be aliased.
For example, `create_image(3,2,27)` should return `[[27, 27, 27], [27, 27, 27]]`.
Hint: This can be done elegantly in one line.

```
def create_image(w,h,c):
```

b. [**5 points**] Fill in the body for the function `well_formed`. This function should return `True` if the image passed in satisfies the following 2 criteria: All rows are the same length, and all color values are between 0 and 255 (both inclusive). If either or both fail to hold, the function should return `False`.

```
def well_formed(img):
```

c. [**5 points**] Fill in the body of the function `fill_rect`. This function should set all pixels with coordinates (`x`,`y`), where `x0` $\leq$ `x` $<$ `x1` and `y0` $\leq$ `y` $<$ `y1`, to color `c`. The coordinates (`x0`,`y0`) and (`x1`,`y1`) may lie outside the bounds of the image. Your `fill_rect` function should still set all of the pixels of the rectangle that do lie within the image to the color specified.

```
def fill_rect(img,x0,y0,x1,y1,c):
```

d. [**10 points**] Fill in the blanks below to get a function `fill_region` which behaves as follows. This function should start at (`x`,`y`), and find all *contiguous* pixels in `img` which can be reached from (`x`,`y`) by only moving horizontally or vertically one pixel at a time, which have the color `oldcolor`. For each such pixel, the function should *change* the pixel's color to `newcolor`. For example, running `fill_region(img,0,10,1,2)` on the image `img` which has the value shown on the left should result in changing `img` to the value shown on the right.

```
#before                              #after
img = [[ 0, 3, 0, 3, 0],             img = [[10, 3, 0, 3,10],
       [ 0, 3, 0, 3, 0],                    [10, 3, 0, 3,10],
       [ 0, 0, 3, 0, 0],                    [10,10, 3,10,10],
       [ 5, 0, 0, 0, 5],                    [ 5,10,10,10, 5],
       [ 0, 5, 5, 5, 0]]                    [ 0, 5, 5, 5, 0]]
```

Assume: Whenever `fill_region` is called, `oldcolor != newcolor` and `img[y][x] == oldcolor`.

```
def fill_region(img, oldcolor, newcolor, x, y , c):

  __________________________ = ____________________________

  for (x1, y1) in [ _______________, _______________, _____________, _______________]:

     try:

        if ______________________________ : ________________________________

     except:

        pass
```

5. [**30 points**] In this problem, you will write a SAT solver using Prolog. In particular, given a boolean formula, you will write Prolog code to find all possible ways of making the formula true. We encode boolean formulas in Prolog as follows:

kind of boolean formula	boolean formula	Prolog term
falsehood	*false*	`0`
truthness	*true*	`1`
Variable	A	`var(A)`
Negated Variable	$\neg A$	`not(var(A))`
Conjunction	$F1 \wedge F2 \wedge \ldots \wedge Fn$	`and([F1, F2, ..., Fn])`
Disjunction	$F1 \vee F2 \vee \ldots \vee Fn$	`or([F1, F2, ..., Fn])`

We assume that negation always appears right next to a variable (one can always push the negation to the inside through conjunctions and disjunctions to reach this form).

Here are some example formulas and the corresponding Prolog term:

boolean formula	Prolog term
$A \wedge B \wedge C$	`and([var(A),var(B),var(C)])`
$(A \vee B) \wedge (\neg A \vee C)$	`and([ or([var(A),var(B)]), or([not(var(A)),var(C)]) ])`

We encode a boolean variable being true or false by setting its value to 1 or 0, respectively.

Your first task will be to write a `sat` predicate that takes a formula as a parameter, and is true if the formula evaluates to true. Once you write this predicate, you should get the following behavior:

```
?- sat(or( [and([var(A),var(B)]), and([not(var(A)), not(var(B))])])).

A = 1,
B = 1 ;

A = 0,
B = 0 ;

No

?- sat(or( [var(A), not(var(A))])).
A = 1 ;

A = 0 ;

No

?- sat(or([var(A),var(B)])).
A = 1 ;

B = 1 ;

No
```

Note that, as shown in the last example above ($A \vee B$), your code for now does not need to generate *all* possible truth assignments that make the predicate true. We will worry about that later.

a. [**15 points**] Fill in the code below for the `sat` predicate:

```
sat(var(X)) :- X = 1.
sat(not(var(X))) :- X = 0.

sat(and([])).

%% Fill in the other case(s) for ``and'' here:


sat(and([X | Tail])) :- ______________________




sat(or([])) :- fail.

%% Fill in the other case(s) for ``or'' here:


sat(or([X | Tail])) :- ______________________


sat(or([_ | Tail])) :- ______________________
```

b. [**10 points**] Next, you will write a `bools` predicate that takes a list of Prolog variables, and iterates through all possible ways of assigning 0 and 1 to these variables. After you write this predicate, you should get the following behavior:

```
?- bools([A,B]).

A = 0,
B = 0 ;

A = 0,
B = 1 ;

A = 1,
B = 0 ;

A = 1,
B = 1 ;

No
```

Fill in the `bools` predicate below:

```
bool(X) :- X = 0.
bool(X) :- X = 1.
bools([]).
bools([X | Tail]) :- ____________________________
```

c. [**5 points**] Finally, you will put this all together in a predicate called `allsat`. This predicate takes a list of variables and a formula, and generates all truth assignments that make the formula true.

```
?- allsat([A,B], or([var(A),var(B)])).

A = 0,
B = 1 ;

A = 1,
B = 0 ;

A = 1,
B = 1 ;

A = 1,
B = 1 ;

No
```

Note that, as shown in the above example, it is perfectly fine if the output of your code repeats some truth assignments. Write the `allsat` predicate below: