CSE 130, Fall 2011
Instructor: Ranjit Jhala

Name/ID ____________________

Final Exam

Instructions: read these first!

Do not open the exam, turn it over, or look inside until you are told to begin.

Switch off cell phones and other potentially noisy devices.

Write your *full name* on the line at the top of this page. Do not separate pages.

You may refer to your single-page (double-sided) cheat sheet, but *no computational devices* (such as laptops, calculators, phones, friends, enemies, pets, lovers).

Read questions carefully. Show all work you can in the space provided.

Where limits are given, write no more than the amount specified.
The rest will be ignored.

Avoid seeing anyone else's work or allowing yours to be seen.

Do not communicate with anyone but an exam proctor.

If you have a question, raise your hand.

When time is up, stop writing.

The points for each part are rough indication of the time that part should take.

Run LaTeX again to produce the table

1. [**??** 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 with:

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

For example, consider the dictionary

fruit	price
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. This dictionary is represented by the tree (on the left) which in turn is represented by the Ocaml value (of type `float dict`) bound to `fruitd` on the right.

```
                 grape:
            _____2.65_____
            |            |
        banana:           orange:
     ___1.50___        ___0.75___
    |          |      |          |
apple:    cherry:  kiwi:       peach:
2.25      2.75     3.99        1.99
```

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

- keys in the *left* subtree are (lexicographically) *less than* `k`, and
- keys in the *right* subtree are (lexicographically) *greater than* `k`.

(a) [5 points] The function

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

should be defined so that `find d k` evaluates to the value associated with the key `k` in the dictionary `d`. For example `find fruitd "cherry"` evaluates to `2.75`.

Fill in the blanks below to implement `find` as described.

```
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] The function

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

should be defined so that `add d k v` gives a new dictionary that results from *adding* the key-value pair `(k,v)` to dictionary `d`. For example 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)`.

Fill in the blanks below to implement `add` as described.

```
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' *)   ____________________________________________
```

2. [**??** points] *"MapReduce is a software framework introduced by Google to support distributed computing on large data sets on clusters of computers. The framework is inspired by map and reduce functions commonly used in functional programming."* (From WikiPedia)

   This question will give you a flavor of what it is like to program in the MapReduce model, using a simple Ocaml implementation that runs on a single machine.

   (a) [5 points] Consider the function `expand` whose type is given at the bottom.

```
let rec expand f xs =
  match xs with
  | []       -> []
  | x::xs'   -> (f x) @ (expand f xs')

val expand : ('a -> 'b list) -> 'a list -> 'b list
```

   What does the following expression evaluate to:

```
let rec clone (x,n) = if n <= 0 then [] else x::(clone (x, n-1)) in
expand clone [("a",1); ("b", 2); ("c", 3)]
```

   Result:

   (b) [5 points] Consider the function `insert` whose type is given at the bottom.

```
let rec insert t (key,value) =
  match t with
  | []           -> [(key,[value])]
  | (k,vs)::t'   -> if k = key
                    then (k, value::vs)::t'
                    else (k, vs)::(insert t' (key,value))

val insert: ('a * 'b list) list -> ('a * 'b) -> ('a * 'b list) list
```

   What does the following expression evaluate to:

```
let t = [("judynails",[2]); ("larsumlaut",[2;2;9]); ("caseylynch",[3])] in
let kv= ("judynails",4) in
insert t kv
```

   Result:

   (c) [5 points] Consider the function `group` whose type is given at the bottom.

```
let group kvs = List.fold_left insert [] kvs

val group : ('a * 'b) list -> ('a * 'b list) list
```

   What does the following expression evaluate to:

```
let kvs = [("judynails", 3); ("larsumlaut", 8); ("caseylynch", 19);
           ("caseylynch", 12); ("larsumlaut", 7); ("judynails", 6)] in
group kvs
```

Result:

(d) [5 points] Consider the function `collapse` whose type is given at the bottom.

```
let collapse f gs = List.map (fun (k,v::vs) -> (k, List.fold_left f v vs)) gs

val collapse: ('a -> 'a -> 'a) -> ('b * 'a list) list -> ('b * 'a) list
```

What does the following expression evaluate to:

```
let gs = [("judynails",[9;3]); ("larsumlaut",[5;2;3]); ("caseylynch",[3;6])] in
collapse (fun x y -> x + y) gs
```

Result:

(e) [10 points] Finally, consider the function `map_reduce` whose type is given at the bottom.

```
let map_reduce xs mapper reducer =
  let kvs = expand mapper xs in
  let gs  = group kvs in
  let rs  = collapse reducer gs in
  rs

val map_reduce: 'a list -> ('a->('b *'c) list) -> ('c->'c->'c) -> ('b *'c) list
```

Intuitively, the `map_reduce` function takes the arguments:

- `xs`: A list of values of type `'a`, typically a list of documents,
- `mapper`: A function that maps a `'a` value to a list of key-value pairs `kvs` of type `'b * 'c list`,
- `reducer`: An accumulation function that takes a "current accumulation" value of type `'c` a "next value" of type `'c` and returns a new accumulated value of type `'c`.

The `map_reduce` function works as follows. First, it uses `mapper` to *expand* the list `xs` into a giant list of key-value pairs `kvs`. Second, it *groups* the key-value pairs using the keys to get a list `gs`. Third, it uses `reducer` to *reduces* the list of values in each group (indexed by `k`) using `reducer` into a single value that is indexed by `k`). In the real implementation, each of the three steps of `map_reduce` is carried out in parallel across several (thousands of!) machines. Assume that you are given

```
val words_of_document: document -> string list
val wwwdocs: document list
```

that is a function that returns a list of strings corresponding to the words in a given document, and the list of all documents on the Web. Suppose that you wish to compute the frequency with which different words appear in documents on the Web. That is you want to compute a list `[(w1,c1);(w2,c2);...;(wn,cn)]` where `ci` is the number of times the word `wi` appears in documents across the Web. Fill in the blanks below to show how `map_reduce`. can be used to compute the word frequency.

```
let wordcount =

    let fmap = ____________________________________________ in

    let fred = ____________________________________________ in
```

```
map_reduce wwwdocs fmap fred
```

3. [**??** 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) [10 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 ______________________
```

4. [**??** points] Consider the following Python class definition.

```
class Iter(object):
  def __init__(self, seed, ticker, value, finish):
    self.ticker = ticker
    self.finish = finish
    self.value  = value
    self.x      = seed

  def next(self):
    if self.finish(self.x): raise StopIteration
    v      = self.value(self.x)
    self.x = self.ticker(self.x)
    return v

  def __iter__(self):
    return self
```

Recall that

```
for x in y:
  c
```

is just a pretty way of writing

```
temp = y.__iter__()
while(True):
  try:
    x = temp.next()
  except StopIteration:
    break
  c
```

(a) [10 points] Fill in the blanks below, such that the resulting code executes without any errors. That is, such that *after the for-loop finishes*, the value of `z1` is `[0,1,2,3,4,5,6,7,8,9,10]`.

```
s1 = ________________________________________

def t1(x): return ____________________________

def v1(x): return ____________________________

def f1(x): return ____________________________

z1 = []
for x in Iter(s1,t1,v1,f1): z1.append(x)
assert(z1 == [0,1,2,3,4,5,6,7,8,9,10])
```

(b) [15 points] Fill in the blanks below, such that the resulting code executes without any errors. That is, such that *after the for-loop finishes*, the value of `z2` is `[1,1,2,3,5,8,13,21,34,55,89]`.

```
s2 = ______________________________________________________________

def t2(x): return ______________________________________________________

def v2(x): return ______________________________________________________

def f2(x): return ______________________________________________________

z2 = []
for x in Iter(s2,t2,v2,f2): z2.append(x)
assert(z2 == [1,1,2,3,5,8,13,21,34,55,89])
```

(c) [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), as well as the return value, as illustrated below (left). Write the `print_first_k_args` in the blanks below (hint: `str(x)` returns the string representation of `x`)

```
@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(1)
def fac(n):
    if n <= 1: return 1
    else: return n*fac(n-1)
>>> fac(3)
Arg 1: 3
Arg 1: 2
Arg 1: 1
Return: 1
Return: 2
Return: 6
6
```

5. [**??** points] For this problem, you will write Prolog code that checks whether a given ML expression is *well-scoped*, that is, that every variable that is used in the expression is *bound* in the expression. That is, your prolog code will check, just by looking at the code, not by running it, whether or not your nanoML implementation would have thrown a `Nano.MLFailure "Variable not bound: ..."` exception.

   First, we shall encode nanoML expressions as Prolog terms via the following grammar.

$$
\begin{array}{lll}
\textit{expr} & ::= & \mid \texttt{const}(\texttt{i}) \\
 & & \mid \texttt{var}(\texttt{x}) \\
 & & \mid \texttt{plus}(\textit{expr}, \textit{expr}) \\
 & & \mid \texttt{leq}(\textit{expr}, \textit{expr}) \\
 & & \mid \texttt{ite}(\textit{expr}, \textit{expr}) \\
 & & \mid \texttt{letin}(\texttt{var}(\texttt{x}), \textit{expr}, \textit{expr}) \\
 & & \mid \texttt{fun}(\texttt{var}(\texttt{x}), \textit{expr}) \\
 & & \mid \texttt{app}(\textit{expr}, \textit{expr})
\end{array}
$$

   The table below shows several examples of Ocaml expressions, the Prolog term encoding that expression.

ML Expression	Prolog Expression Term
`2`	`const(2)`
`x`	`var(x)`
`2 + 3`	`plus(const(2),const(3))`
`2 <= 3`	`leq(const(2),const(3))`
`fun x -> x <= 4`	`fun(var(x),leq(var(x),const(4)))`
`fun x -> fun y -> if x then y else 0`	`fun(var(x),fun(var(y), ite(var(x),var(y),const(0))))`
`let x = 10 in x`	`letin(var(x),const(10),var(x))`
`fun x -> let y = x in y + y`	`fun(var(x), letin(var(y),var(x) plus(var(y),var(y))))`

   (a) [10 points] Write a Prolog predicate `reads(E,X)` that is true if `X` is *read anywhere* inside the expression `E`. When you are done, you should get the following behavior:

```
?- reads(plus(const(2),const(3)), x).
   False.

?- reads(letin(var(x),const(1),var(a)), X).
   X = a
   True.

?- reads(fun(var(x),plus(var(a),var(b))), X).
   X = a;
   X = b;
   True.

?- reads(fun(var(b),plus(var(a),var(b))), X).
   X = a;
   X = b;
   True.
```

Write your solution by filling in the grid below. **Hint:** If you need an "Or", you may add extra rules where needed, (or better, just use the `;` operator.)

`reads(const(I), X) :-          0 = 1.       % i.e. false`
`reads(var(X), Y) :-`
`reads(plus(E1,E2), X) :-`
`reads(leq(E1,E2), X) :-`
`reads(ite(E1,E2,E3), X) :-`
`reads(letin(var(Y),E1,E2), X) :-`
`reads(fun(var(Y),E), X) :-`
`reads(app(E1,E2), X) :-`

(b) [15 points] Write a Prolog predicate `wellscoped(E)` that is true if `E` is *well-scoped*, that is, each variable that is read is previously bound. When you are done, you should get the following behavior:

```
?- wellscoped(plus(var(a),const(3))).
   False.

?- wellscoped(letin(var(a),const(1),plus(var(a),const(3)))).
   True.

?- wellscoped(fun(var(b),plus(var(a),var(b)))).
   False.

?- wellscoped(fun(var(b),fun(var(a), plus(var(a),var(b))))).
   True.

?- wellscoped(app(fun(var(a),plus(var(a),const(1))), var(a))).
```

```
   False.

?- wellscoped(app(fun(var(a),plus(var(a),const(1))),
              letin(var(a),const(1), var(a)))).
   True.
```

To define `wellscoped`, write a helper predicate `helper(E, Xs)` which is true if every variable that is *read* in `E` either occurs in `Xs` or occurs *bound inside* `E`. With this, you can define `wellscoped` as:

`wellscoped(E) :- helper(E, []).`

Write your definition for `helper` by filling in the grid below.

Hint: You *need not* use `reads`. You *may* use the built-in predicate `member(X,Ys)` which returns true if the atom `X` appears in the list `Ys`.

`helper(const(I), Xs) :-        0 = 0.  % i.e. true`
`helper(var(X), Xs) :-`
`helper(plus(E1,E2), Xs) :-`
`helper(leq(E1,E2), Xs) :-`
`helper(ite(E1,E2,E3), Xs) :-`
`helper(letin(var(Y),E1,E2), Xs) :-`
`helper(fun(var(Y),E), Xs) :-`
`helper(app(E1,E2), Xs) :-`