CSE 130, Spring 2012
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 any printed materials, but *no computational devices* (such as laptops, calculators, phones, iPads, 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]

   In each question below, we have given a Scala function with missing type information. Your job is to

   1. fill in appropriate types (so the function will be accepted by the typechecker),
   2. write down one set of suitable inputs (i.e. of the corresponding types),
   3. write down the output corresponding to the input.

   Hint: Recall that syntax for anonymous functions in Scala is `(x1,...,xn) => e` which is equivalent to Ocaml's `fun (x1,...,xn) -> e`

   (a) [2 points]

```
def plus(x: T1, y: T1): T2 = x + y

val out = plus(in1, in2)
```

   T1 = _____

   T2 = _____

   in1 = _____

   in2 = _____

   out = _____

   (b) [5 points]

```
def plussed(x: T1, y: T2): T3 = x + y(x)

val out = plussed(in1, in2)
```

   T1 = _____

   T2 = _____

   T3 = _____

in1 = _____

in2 = _____

out = _____

(c) [8 points]

```
def squash[A](xss: T1): T2 = {
  for ( xs <- xss
      ; x  <- xs )
  yield x
}

val out = squash(in)
```

T1 = _____

T2 = _____

in = _____

out = _____

(d) [10 points]

```
def reduce[A](xs: T1, f: T2): T3 = {
  var acc = xs(0)
  for (x <- xs.tail) {
    acc = f(acc, x)
  }
  acc
}

val out = reduce(in1, in2)
```

T1 = _____

`T2` = _____

`T3` = _____

`in1` = _____

`in2` = _____

`out` = _____

(e) [10 points]

```
def explode[A](xs: T1): T2 = {
  if (xs.isEmpty)
    List(List())
  else {
    for ( ys <- explode(xs.tail)
        ; z  <- List(List(), List(xs.head)) )
    yield (z ++ ys)
  }
}

val out = explode(in)
```

T1 = _____

T2 = _____

in = _____

out = _____

2. [**??** points]

"MapReduce is a software framework introduced by Google to support distributed computing on large data sets on clusters of computers." (From WikiPedia)

This question will give you a flavor of what it is like to program using the MapReduce model, using a simple Scala implementation.

(a) [7 points] Consider the function `expand` whose type is given at the bottom.

```
def expand[A, B](f: A => List[B], xs: Iterator[A]): Iterator[B] = {
  for ( x <- xs
      ; y <- f(x) )
  yield y
}
```

What is the value of `ans` below ?

```
val clone = (p => (0 until p._2).map(_ => p._1).toList)
val ans   = expand(clone, Iterator(("a", 1), ("b", 2), ("c", 3)))
```

Result:

(b) [8 points] Consider the function `insert`

```
def insert[K, V](table: Map[K, List[V]], key: K, v: V): Map[K, List[V]] = {
  if table.contains(key) {
    val vs = table(key)
    table += (key -> v::vs)
  } else {
    table += (key -> List(v))
  }
}
```

What is the value of `ans` below ?

```
val t   = Map( "judynails"  -> List(2)
             , "larsumlaut" -> List(2, 2, 9)
             , "caseylynch" -> List(3))

val ans = insert(t, "judynails", 4)
```

Result:

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

```
def group[K, V](kvs: Iterator[(K, V)]): Map[K, List[V]] = {
  var table: Map[K, List[V]] = Map()
  for ((k, v) <- kvs) {
    table = insert(table, k, v)
  }
  table
}
```

What is the value of `ans` below ?

```
val kvs = Iterator( ("judynails" , 3)
                  , ("larsumlaut", 8)
                  , ("caseylynch", 19)
                  , ("caseylynch", 12)
                  , ("larsumlaut", 7)
                  , ("judynails" , 6))

val ans = group(kvs)
```

Result:

(d) [10 points] Consider the function `collapse` whose type is given at the bottom.

```
def collapse[K, V](table: Map[K, List[V]], f: (V, V) => V): Map[K, V] = {
   table.mapValues(reduce(_, f))
}
```

Hint: The `reduce` function is from Question 1(d).

Hint: The method ‘mapValues‘ (for Scala `HashMaps`) behaves as follows:

```
scala> Map("one" -> 1, "two" -> 2).mapValues(_ + 100)
res: Map[String, Int] = Map("one" -> 101, "two" -> 102)
```

What is the value of `ans` below ?

```
let table = Map( "judynails"  -> List(9, 3)
               , "larsumlaut" -> List(5, 2, 3)
               , "caseylynch" -> List(3, 6)
               )
val ans   = collapse(table, (x, y) => x + y)
```

Result:

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

```
def mapReduce[E, K, V]( xs      : Iterator[E]
                      , mapper : E => List[(K, V)]
                      , reducer: (V, V) => V ) : Map[K, V] = {
  val kvs   = expand(mapper, xs)
  val table = group(kvs)
  val out   = collapse(table, reducer)
  out
}
```

Intuitively, the `mapReduce` function takes the arguments:

- `xs`: which is a collection of values of type `E`, e.g. a collection of documents,
- `mapper`: which is a function that maps each `E` value to a *list* of key-value pairs, `kvs` of type `List[K, V]`.
- `reducer`: An accumulation function that takes a "current accumulation" value of type `V` a "next value" of type `V` and returns a new accumulated value of type `V` (e.g. like `fold_left`).

First, the `mapper` function is used to `expand` the list `xs` into a giant collection of key-value pairs `kvs`. Second, the expanded set of key-value pairs is *grouped by* the key to get `table : Map[K, List[V]]` Third, the `reducer` is used to *reduce* the list of values in *each* group in the table, and the reduced table `out` is returned. In the real implementation, each of the three steps of `mapReduce` is carried out in parallel across several (thousands of!) machines.

Assume that you are *given* the following:

```
type Doc                                // Definition is unimportant
val wwwdocs: Iterator[Doc]              // The WWW as a Document collection
def docWords(d: Doc): List[String]
```

that is, a special type `Doc`, a collection of all WWW documents, and a function that returns a list of strings corresponding to the words in a given document. Your goal is to compute the **frequency** with which different words appear in documents on the Web. That is your goal is to compute a table `wordCount: Map[String, Int]` of the form

```
Map(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 `mapReduce`. can be used to compute the word frequency table `wordCount`:

```
val wordcount = {
  val fmap = _____________________________________________________
  val fred = _____________________________________________________
    mapReduce(wwwdocs, fmap, fred)
}
```

3. [**??** points] We will write several Scala functions to do simple manipulation of images represented by type

```
type Image = List[List[Int]]
```

i.e. lists of lists of integers, with each integer representing a pixel. For example, the following would be a simple image of a smiley face.

```
val img1 = List( List(11,  0, 12)
               , List( 0,  0,  0)
               , List(13,  0, 14)
               , List(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 `img: Image` as `img(y)(x)`

(a) [5 points] Fill in the body of the function `square`, which takes an image, and *squares* each integer in it. For example,

```
scala> square(img1)
res: Image = List( List(121,   0, 144)
                 , List(  0,   0,   0)
                 , List(169,   0, 196)
                 , List(225, 256, 289))
```

Fill in the blanks below to obtain an implementation of `square` .

```
def square(img: Image) : Image = {

  for ( ___ <- _____________________________________________ )

  yield _____________________________________________

}
```

(b) [10 points] Next, fill in the body of the function `crop`, such that `crop(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,

```
scala> crop(img1,0,1,2,4)
res: Image =  List( List(0,   0)
                  , List(13,  0)
                  , List(15, 16))
```

Fill in the blanks below to obtain an implementation of `crop` .

```
def crop(img: Image, x1:Int, y1: Int, x2:Int, y2: Int): Image = {

  for ( ___ <- _____________________________________________ )

  yield _____________________________________________

}
```

Hint: For a list `xs` the call `xs.slice(lo, hi)` returns the *sub-list* of the `lo, lo+1, ..., hi-1`-th elements of `xs` For example,

```
scala> List(0, 10, 20, 30, 40, 50, 60, 70).slice(2, 6)
res: List[Int] = List(20, 30, 40, 50)
```

(c) [10 points] Next, let us write a helper function `zip`. 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,

```
scala> zip(List(1,2,3), List(4,5,6))
res: List[Int] = List((1, 4), (2, 5), (3, 6))

scala> zip(List(1,2,3), List(4,5))
res: List[Int] = List((1, 4), (2, 5))
```

Fill in the blanks below to obtain an implementation of `zip` .

```
def zip[A](l1: List[A], l2: List[B]): List[(A, B)] = {

  ____________________________________________________________________

  ____________________________________________________________________

  ____________________________________________________________________

  ____________________________________________________________________

  ____________________________________________________________________

  ____________________________________________________________________

}
```

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

```
scala> add(img1, img1)
res: Img = List(List(22,  0, 24),
                List( 0,  0,  0),
                List(26,  0, 28),
                List(30, 32, 34))
```

Fill the implementation of `add_imgs` below.

Hint: You may need *another* call to `zip` ...

```
def add(img1: Image, img2: Image): Image = {

  for ((r1, r2) <- zip(img1, img2))

  yield ____________________________________________________

}
```

4. [**??** 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: In Prolog, the less-than-or-equal operator is =<.

   (a) [5 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:

```
?- sorted([]).
true.
?- sorted([10]).
true.
?- sorted([1,2,3,3,4]).
true.
?- 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):

```
% Base Case 1

sorted(_____________)  ____________________________________________

% Base Case 2

sorted(_____________)  ____________________________________________

% Inductive Case

sorted([X1, X2 | T])  ____________________________________________
```

   (b) [5 points] The predicate `sort(L1,L2)` holds if `L2` is a sorted version of `L1`. For example:

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

   Fill in the implementation of `sort` below, using the `sorted` predicate from part (a).

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

   Hint: You *may use* 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*. For example,

```
?- 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.
```

```
?- 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.
```

(c) [10 points] In the previous parts, we implemented a simple `sort` predicate, but with some cheating, as we used `permutation`. Now, we will implement mergesort *from scratch*.

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:

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

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

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

and, by the awesomeness of Prolog,

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

Fill in the implementation of `split` below:

```
% Base Case 1

split(_____, _____, _____)  ____________________________________________

% Base Case 2

split(_____, _____, _____)  ____________________________________________

% Inductive Case

split(_____, _____, _____)  ____________________________________________
```

(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 an OCaml implementation of `merge` that you need to match in Prolog:

```
let rec merge xs ys =
   match (xs, ys) with
   | ([], _)          -> ys
   | (_, [])          -> xs
   | (x::xs', y::ys') -> if x <= y
                         then x :: (merge xs' ys)
                         else y :: (merge xs ys')
```

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:

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

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

Fill in the implementation of `merge` below:

```
% Base Case 1

merge([], Ys, Res)          :-  ____________________________________________.

% Base Case 2

merge(Xs, [], Res)          :-  ____________________________________________.

% Inductive Case 1

merge(_____, _____, _____) :-  ____________________________________________.

% Inductive Case 2

merge(_____, _____, _____) :-  ____________________________________________.
```

(e) [10 points] Finally, you will use `split` and `merge` to implement merge sort. As a reminder, here is an Ocaml implementation:

```
let rec merge_sort xs = match xs with
  | []  -> []
  | [x] -> [x]
  | _   -> let ys, zs = split xs in
           merge (merge_sort ys) (merge_sort zs)
```

Define a predicate `merge_sort(L,S)` which holds if `S` is a sorted version of `L`. For example:

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

Fill in the implementation of `merge_sort` below:

```
% Base Case 1

merge_sort(_______, ______).

% Base Case 2

merge_sort(_______, ______).

% Inductive Case

merge_sort(L, S) :- ____________________________________

                    ____________________________________

                    ____________________________________

                    ____________________________________
```

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.

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

   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) :-`