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 \((x_1, \ldots, x_n) \Rightarrow e\) which is equivalent to Ocaml’s \(\text{fun } (x_1, \ldots, x_n) \rightarrow e\)

(a) [2 points]

\[
\text{def plus}(x: T1, y: T1): T2 = x + y
\]

\[
\text{val out} = \text{plus}(\text{in1}, \text{in2})
\]

T1 = ________________________________

T2 = ________________________________

in1 = ________________________________

in2 = ________________________________

out = ________________________________

(b) [5 points]

\[
\text{def plussed}(x: T1, y: T2): T3 = x + y(x)
\]

\[
\text{val out} = \text{plussed}(\text{in1}, \text{in2})
\]

T1 = ________________________________

T2 = ________________________________

T3 = ________________________________
\(\text{in1} = \) \\
\(\text{in2} = \) \\
\(\text{out} = \)

(c) \([8 \text{ points}]\)

\[
\text{def squash}[\text{A}](\text{xss}: \text{T1}): \text{T2} = \{ \\
\quad \text{for ( xs <- xss} \\
\quad \quad ; x <- xs ) \\
\quad \quad \text{yield x} \\
\quad \} \\
\]
\[
\text{val out} = \text{squash}(\text{in}) \\
\]
\(\text{T1} = \) \\
\(\text{T2} = \) \\
\(\text{in} = \) \\
\(\text{out} = \)

(d) \([10 \text{ points}]\)

\[
\text{def reduce}[\text{A}](\text{xs}: \text{T1}, f: \text{T2}): \text{T3} = \{ \\
\quad \text{var acc = xs(0)} \\
\quad \text{for (x <- xs.tail) } \{ \\
\quad \quad \text{acc = f(acc, x)} \\
\quad \quad \} \\
\quad \text{acc} \\
\} \\
\]
\[
\text{val out} = \text{reduce}(\text{in1, in2}) \\
\]
\(\text{T1} = \)
\[ T_2 = \]  

\[ T_3 = \]  

\[ in_1 = \]  

\[ in_2 = \]  

\[ out = \]
(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 \texttt{expand} whose type is given at the bottom.

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

What is the value of \texttt{ans} below?

\begin{verbatim}
val clone = (p => (0 until p._2).map(_ => p._1).toList)
val ans = expand(clone, Iterator(("a", 1), ("b", 2), ("c", 3)))
\end{verbatim}

\textbf{Result:}

(b) [8 points] Consider the function \texttt{insert}

\begin{verbatim}
    if table.contains(key) {
        val vs = table(key)
        table += (key -> v::vs)
    } else {
        table += (key -> List(v))
    }
}
\end{verbatim}

What is the value of \texttt{ans} below?

\begin{verbatim}
val t = Map( "judynails" -> List(2),
              "larsumlaut" -> List(2, 2, 9),
              "caseylynch" -> List(3))
val ans = insert(t, "judynails", 4)
\end{verbatim}

\textbf{Result:}
(c) [5 points] Consider the function `group` whose type is given at the bottom.

```scala
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?

```scala
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.

```scala
  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?

```scala
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.

```scala
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:

```scala
// Definition is unimportant
type Doc
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

\[
\text{Map}(w_1 \rightarrow c_1, w_2 \rightarrow c_2, \ldots, w_n \rightarrow c_n)
\]

where `c_i` is the number of times the word `w_i` 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`:

```scala
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

\[
\text{type Image} = \text{List[List[Int]]}
\]

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.

\[
\text{val img1} = \text{List( List(11, 0, 12)
, List( 0, 0, 0)
, List(13, 0, 14)
, List(15, 16, 17))}
\]

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 \(\text{img}\) as \(\text{img}(y)(x)\).

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

\[
\begin{align*}
\text{scala> square(img1)} \\
\text{res: Image = List( List(121, 0, 144)
, List( 0, 0, 0)
, List(169, 0, 196)
, List(225, 256, 289))}
\end{align*}
\]

Fill in the blanks below to obtain an implementation of \textit{square}.

\[
\text{def square(img: Image) : Image = }
\]

\[
\text{for ( ___ <- ________________________________ )}
\]

\[
\text{yield ________________________________}
\]

\}

(b) [10 points] Next, fill in the body of the function \textit{crop}, such that \textit{crop(img, x1, y1, x2, y2)} returns an image which only contains the pixels from \(\text{img}\) at coordinates \((x, y)\), where \(x1 \leq x < x2\) and \(y1 \leq y < y2\). (You can assume that all such coordinates exist in \(\text{img}\).) For example,

\[
\begin{align*}
\text{scala> crop(img1,0,1,2,4)} \\
\text{res: Image = List( List(0, 0)
, List(13, 0)
, List(15, 16))}
\end{align*}
\]

Fill in the blanks below to obtain an implementation of \textit{crop}.

\[
\text{def crop(img: Image, x1:Int, y1: Int, x2:Int, y2: Int): Image = }
\]

\[
\text{for ( ___ <- ________________________________ )}
\]

\[
\text{yield ________________________________}
\]

\}

**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
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 $n$th element of the returned list is a pair consisting of the $n$th element of $l1$ and the $n$th 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
scala> zip(List(1,2,3), List(4,5,6))
res: List[Int] = List((1, 4), (2, 5), (3, 6))
```

```scala
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 $zip$.

```scala
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
scala> add(img1, img2)
res: Img = List(List(22, 0, 24),
    List( 0, 0, 0),
    List(26, 0, 28),
    List(30, 32, 34))
```

Fill the implementation of $addimgs$ below. **Hint:** You may need another call to $zip$...

```scala
def add(img1: Image, img2: Image): Image = {
    _____________________________________________________
    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 \( \leq \).

(a) [5 points] Fill in the implementation of the \texttt{sorted} predicate below. The predicate \texttt{sorted(L)} should hold if \( L \) is sorted in increasing order. For example:

\[
\text{?- sorted([]).} \\
\text{true.} \\
\text{?- sorted([10]).} \\
\text{true.} \\
\text{?- sorted([1,2,3,3,4]).} \\
\text{true.} \\
\text{?- sorted([10,2,3,3,4]).} \\
\text{false.}
\]

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):

\[
\%
\text{Base Case 1}
\%
\text{sorted(____________) _________________________________________}
\%
\text{Base Case 2}
\%
\text{sorted(____________) _________________________________________}
\%
\text{Inductive Case}
\%
\text{sorted([X1, X2 | T]) _________________________________________}
\]

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

\[
\text{?- sort([4,1,3,2], L).} \\
L = [1, 2, 3, 4].
\]

Fill in the implementation of \texttt{sort} below, using the \texttt{sorted} predicate from part (a).

\[
\text{sort(L1, L2) :- _________________________________________________________}
\]

**Hint:** You may use the built-in predicate \texttt{permutation}, which takes two lists and returns true if the two lists contain the same elements, but possibly \textit{in a different order}. For example,

\[
\text{?- permutation([1,3,5], X).} \\
X = [1, 3, 5] ; \\
X = [1, 5, 3] ; \\
X = [3, 1, 5] ; \\
X = [3, 5, 1] ; \\
X = [5, 1, 3] ; \\
X = [5, 3, 1] ; \\
\text{false.}
\]
?- 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, 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:

```ocaml
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. [10 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.

\[ \text{expr ::= } \begin{array}{c} \text{const}(i) \\ \text{var}(x) \\ \text{plus}(\text{expr}, \text{expr}) \\ \text{leq}(\text{expr}, \text{expr}) \\ \text{ite}(\text{expr}, \text{expr}) \\ \text{letin}(\text{var}(x), \text{expr}, \text{expr}) \\ \text{fun}(\text{var}(x), \text{expr}) \\ \text{app}(\text{expr}, \text{expr}) \end{array} \]

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

<table>
<thead>
<tr>
<th>ML Expression</th>
<th>Prolog Expression Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>const(2)</td>
</tr>
<tr>
<td>x</td>
<td>var(x)</td>
</tr>
<tr>
<td>2 + 3</td>
<td>plus(const(2), const(3))</td>
</tr>
<tr>
<td>2 &lt;= 3</td>
<td>leq(const(2), const(3))</td>
</tr>
<tr>
<td>fun x -&gt; x &lt;= 4</td>
<td>fun(var(x), leq(var(x), const(4)))</td>
</tr>
<tr>
<td>fun x -&gt; fun y -&gt; if x then y else 0</td>
<td>fun(var(x), fun(var(y), ife(var(x), var(y), const(0))))</td>
</tr>
<tr>
<td>let x = 10 in x</td>
<td>letin(var(x), const(10), var(x))</td>
</tr>
<tr>
<td>fun x -&gt; let y = x in y + y</td>
<td>fun(var(x), letin(var(y), var(x), plus(var(y), var(y))))</td>
</tr>
</tbody>
</table>

(a) [10 points] Write a Prolog predicate \text{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:

?- \text{reads}((\text{const}(2), \text{const}(3)), x).
   False.

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

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

?- \text{reads}(\text{fun}(\text{var}(b), \text{plus}(\text{var}(a), \text{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:

```prolog
?- 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.

<table>
<thead>
<tr>
<th>helper(const(I), Xs) :-</th>
<th>0 = 0. % i.e. true</th>
</tr>
</thead>
<tbody>
<tr>
<td>helper(var(X), Xs) :-</td>
<td></td>
</tr>
<tr>
<td>helper(plus(E1,E2), Xs) :-</td>
<td></td>
</tr>
<tr>
<td>helper(leq(E1,E2), Xs) :-</td>
<td></td>
</tr>
<tr>
<td>helper(ite(E1,E2,E3), Xs) :-</td>
<td></td>
</tr>
<tr>
<td>helper(letin(var(Y),E1,E2), Xs) :-</td>
<td></td>
</tr>
<tr>
<td>helper(fun(var(Y),E), Xs) :-</td>
<td></td>
</tr>
<tr>
<td>helper(app(E1,E2), Xs) :-</td>
<td></td>
</tr>
</tbody>
</table>