
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 L ^A T _E X 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 $(x_1, \dots, x_n) \Rightarrow e$ 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 wwdocs: 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

$$\text{Map}(w_1 \rightarrow c_1, w_2 \rightarrow c_2, \dots, w_n \rightarrow c_n)$$

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`:

```
val wordcount = {
  val fmap = _____
  val fred = _____
  mapReduce(wwdocs, 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 $x_1 \leq x < x_2$ and $y_1 \leq y < y_2$. (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 `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> 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]

We say that one word is an *anagram* of another if *reordering* the letters of the first word results in the second word. Next, we will write a couple of Scala functions that determine if one word is a *anagram* of another.

Recall! In Scala, a `String` is in fact a *collection* of characters. Thus, we can do all the usual “collection-y” things to them, such as:

```
scala> val xs = "influx"
xs: java.lang.String = influx

scala> (xs(0), xs(1), xs(2))
res1: (Char, Char, Char) = (i,n,f)

scala> xs.head
res2: Char = i

scala> xs.tail
res2: String = nflux

scala> xs.length
res3: Int = 6

scala> xs.sorted
res4: String = filnux

scala> xs.isEmpty
res5: Boolean = false

scala> xs.reverse
res6: String = xulfni

scala> xs.slice(2, 5)
res7: String = flu
```

- (a) [5 points] Fill in the body of the function `isAnagram`, which takes two `Strings` and checks if the first one is an anagram of the second.

```
scala> isAnagram("influx", "flunxi")
res: Boolean = True

scala> isAnagram("influx", "flinux")
res: Boolean = True

scala> isAnagram("influx", "XULFNI") // uppercase is different char
res: Boolean = False

scala> isAnagram("influx", "linux") // missing 'f'
res: Boolean = False
```

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

```
def isAnagram(src: String, dst: String) : Boolean =
```

- (b) [7 points] Next, write a function that takes a word `w`, a character `c` and a position `i` and *splices* the character into the word at the given position. For example,

```
scala> spliceCharAt("influx", 'z', 0)
res: String = zinflux
```

```
scala> spliceCharAt("influx", 'z', 2)
res: inzflux
```

```
scala> spliceCharAt("influx", 'z', 6)
res: influxz
```

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

```
def spliceCharAt(w: String, c: Char, i: Int) : String=
```

- (c) [8 points] Finally, we will write a function that returns an iterator over *all* the possible anagrams (permutations) generatable from a given `String`. When you are done, you should see the following behavior:

```
scala> anagrams("dog").toList
res: List[String] = List(dog, odg, ogd, dgo, gdo, god)
```

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

```
def anagrams(w: String): Iterator[String] =
```

```
  if ( _____ ) {
```

```
  } else {
```

```
    for ( _____
          _____ )
```

```
  }
```

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.

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

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

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

- (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.)

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

(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`.

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