

CA215: Lab Exam 1: Sample Solutions

Important: these are just sample solutions, there are potentially other valid solutions for each question.

Question 1 [40 marks]

Write code which, given a pair of lists of Ints which are not necessarily of the same length, returns a corresponding pair of lists which are of the same length. The list lengths are made the same by padding the shorter list: insert 0s at the tail of the shorter list until it is the same length as the longer one. The top-level function must have the type declaration `match_lengths :: ([Int],[Int]) -> ([Int],[Int])`.

In the solution I give here, the `match_lengths` function simply decides which list to pad. If the lists are the same length then they are returned as-is – this includes the case where both lists are empty. If list1 is shorter then list1 is padded and list 2 is left as-is. If list2 is shorter then list2 is padded and list1 is left as-is. In both cases, the shorter list is padded by the difference in the lengths of the lists. The `pad` function actually carries out the padding by adding the specified number of 0s to the end of the list. Note that the `pad` function is recursive but the `match_lengths` function is not.

```
match_lengths :: ([Int],[Int]) -> ([Int],[Int])
match_lengths (list1,list2)
  | (length list1) == (length list2) = (list1,list2)
  | (length list1) < (length list2) = (pad ((length list2)-(length list1)) list1,list2)
  | otherwise = (list1,pad ((length list1)-(length list2)) list2)

pad :: Int -> [Int] -> [Int]
pad 0 list = list
pad x list = pad (x-1) (list ++ [0])
```

Question 2 [30 marks]

Write code which accepts an unsorted list of elements of type `a` which may contain duplicates and returns a sorted list (ascending order) of pairs `(a,Int)` where the first element in the pair occurred on the original list and the second element gives the number of times that element occurred on the original list. The top-level function must have the type declaration `tokens_to_types :: Ord a => [a] -> [(a,Int)]`.

This solution makes use of several functions which we looked at in class, in the lab exercises and in the revision exercises in order to put together the final solution. The first of these is the insertion sort:

```
sort :: Ord a => [a] -> [a]
sort [] = []
sort (h:t) = insert h (sort t)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (h:t)
  | x <= h = x:(h:t)
  | otherwise = h:(insert x t)
```

The second is a function to remove duplicates from a list, which itself makes use of a `find` function which says whether an element is on a list or not:

```
remove_duplicates :: Eq a => [a] -> [a]
remove_duplicates [] = []
remove_duplicates (h:t)
  | find h t = remove_duplicates t
  | otherwise = h : remove_duplicates t
```

```

find :: Eq a => a -> [a] -> Bool
find x [] = False
find x (h:t)
  | x==h = True
  | otherwise = find x t

```

The third is a function which counts the number of times a given element occurs on a list:

```

count :: Eq a => a -> [a] -> Int
count x [] = 0
count x (h:t)
  | x==h = 1 + count x t
  | otherwise = count x t

```

These functions are used together in order to solve the given problem. The top-level function `tokens_to_types` is not recursive, but uses local definitions to break down the problem. Firstly, it constructs a list of the unique elements occurring on the original list by removing the duplicates from it. Secondly, it sorts this list, and places the solution in locally-defined `nodups`. The recursive `get_counts` function then takes this `nodups` list and the original list as arguments and computes the number of occurrences of each `nodups` element on the original list, returning a list of paired `nodups` elements and counts.

```

tokens_to_types :: Ord a => [a] -> [(a,Int)]
tokens_to_types list = answer
  where
    nodups = sort (remove_duplicates list)
    answer = get_counts nodups list

get_counts :: Eq a => [a] -> [a] -> [(a,Int)]
get_counts [] dups = []
get_counts (h:t) list = (h,count h list): get_counts t list

```

Question 3 [30 marks]

Write code which returns an integer representing the number of seconds elapsed from a list of paired time types and values. The top-level function must have the type declaration `seconds_elapsed :: [(Time,Int)] -> Int`. Assume this data-type definition: `data Time = Seconds | Minutes | Hours`.

This solution is reasonably straightforward. There are 4 base cases, accounting for when the list is empty and when the list contains exactly 1 element where this 1 element could be of type `Seconds`, `Minutes` or `Hours`. The recursive case then computes the time accounted for by the head of the list and adds it to the time accounted for by the tail of the list. Note that the `seconds_elapsed` function is called twice in the right-hand side of the recursive case. The first of these calls will always match one of the base cases.

```

data Time = Seconds | Minutes | Hours

seconds_elapsed :: [(Time,Int)] -> Int
seconds_elapsed [] = 0
seconds_elapsed [(Seconds,s)] = s
seconds_elapsed [(Minutes,s)] = 60 * s
seconds_elapsed [(Hours,s)] = 60 * 60 * s
seconds_elapsed ( h:t ) = (seconds_elapsed [h]) + (seconds_elapsed t)

```