COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

Declarative Programming

Answers to workshop exercises set 4.

QUESTION 1
Write a Haskell version of the tree sort algorithm, which inserts all the to-be-sorted data items into a binary search tree, then performs an inorder traversal to extract the items in sorted order. Use simple structural induction where possible.

ANSWER
Here is a simple, direct expression of the algorithm:

```
>data Tree a = Empty | Node (Tree a) a (Tree a)

>treesort::  Ord a => [a] -> [a]
>treesort xs = tree_inorder (list_to_bst xs)

>list_to_bst:: Ord a => [a] -> Tree a
>list_to_bst [] = Empty
>list_to_bst (x:xs) = bst_insert x (list_to_bst xs)

>bst_insert:: Ord a => a  -> Tree a -> Tree a
>bst_insert i Empty = Node Empty i Empty
>bst_insert i (Node l v r)
>   | i <= v = (Node (bst_insert i l) v r)
>   | i  > v = (Node l v (bst_insert i r))

>tree_inorder:: Tree a -> [a]
>tree_inorder Empty = []
>tree_inorder (Node l v r) = tree_inorder l ++ [v] ++ tree_inorder r
```

If i == v, then bst_insert can insert i into either subtree; both will work.
It does have to insert i into a subtree, since sorting is usually required
to preserve duplicate occurrences of values.

QUESTION 2

Write a Haskell function to "transpose" a list of lists. You may assume that
all lists are non-empty, and that the inner lists are all the same length.
If you are given a list of N lists, each of length M, the result should be
a list of M lists, each of length N. For example,

```
transpose [[1,2],[4,4],[8,9]]
```

should return

```
[[1,4,8],[2,4,9]]
```

ANSWER
It is unclear what transpose [] should be. It would have to be a list of empty lists, but it could be of any length. In this version, we consider
transpose [] to be an error. We also ensure the inner lists are not empty
(using the nested pattern (x:xs1)). This allows us to have the invariant

```
transpose (transpose x) = x
```

The structural induction is therefore a bit different to typical cases -
we essentially do structural induction on *non-empty* lists. We use a higher order function, which makes the code shorter. (You could
write a specialised first-order function instead.)

```
>transpose :: [[a]] -> [[a]]
>transpose [] = error "transpose of zero-height matrix"
>transpose list@(xs:xss)
>  | len > 0   = transpose' len list
>  | otherwise = error "transpose of zero-width matrix"
>  where len = length xs

>transpose' len [] = replicate len []
>transpose' len (xs:xss)
>  | len == length xs = zipWith (:) xs (transpose' len xss)
>  | otherwise = error "transpose of non-rectangular matrix"
```

The transpose function ensures we have a non-empty list of non-empty lists, and calls transpose' to do the work, passing the length of the

first row, which will be the number of rows in the result.

zipWith (:) takes a list and a list of lists and conses elements of the first list onto the front of elements of the second. For example, zipWith (:) [3,4] [[1],[2]] = [[3,1],[4,2]]. It is defined in the prelude,
but here is a version which you could use also:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] [] = []
zipWith _ [] (_:_) = error "zipWith: list length mismatch"
zipWith _ (_:_) _ = error "zipWith: list length mismatch"
zipWith f (x:xs) (y:ys) = (f x y):(zipWith f xs ys)
```

When matching a call to transpose against the left hand sides of the equations
that define transpose, a call in which the argument list has length zero will match the head of the first equation, and a call in which the argument
list has length one will match the head of the second equation. Haskell will
try to match a call against the head of the third equation only if the matches
against the heads of the previous equations have failed, which in this case
means that the length of the argument list must be at least two. This is why
xss, which represents the tail of the argument list, will have a length of
one or more. This ensures that the recursive call in the third equation will never match the first equation, and thus will not throw that exception.

QUESTION 3
Write a Haskell function which takes a list of numbers and returns
a triple containing the length, the sum of the numbers, and the sum of the
squares of the numbers. Try coding this with (1) three separate traversals
of the list and (2) a single traversal of the list.

ANSWER
The three-pass version is simple. We use the length and sum functions from the Prelude (you can also easily define your own versions).

```
>stats1 ns =
>   (length ns,
>    sum ns,
>    sumsq ns
>   )

>sumsq [] = 0
>sumsq (n:ns) = n*n + sumsq ns
```

The single pass version uses pattern matching in a let expression to extract the three numbers from the result of the recursive call.

```
>stats2 [] = (0,0,0)
>stats2 (n:ns) =
>   let (l,s,sq) = stats2 ns
>   in (l+1, s+n, sq+n*n)
```