FUNC Practical after Lectures 1–3

♠ `take, drop` **(Grade I)** Define `take, drop :: Int -> [a] -> [a]`. The result of `take n xs` should be the first `n` items of the list `xs`, or the whole of `xs` if it has fewer than `n` items. The result of `drop n xs` should be *all but* the first `n` items of `xs`, or `[]` if `xs` has fewer than `n` items.
eg. `take 2 ["To", "be", "or", "not", "to", "be"]` ⇝ `["To", "be"]`
eg. `drop 4 ["To", "be", "or", "not", "to", "be"]` ⇝ `["to", "be"]`

♠ `positions` **(Grade II)** Define `positions :: Eq a => [a] -> a -> [Int]` so that `positions xs i` gives the list of numeric indices at which `i` occurs in `xs`.
eg. `positions ["To","be","or","not","to","be"] "be"` ⇝ `[1, 5]`

♠ `duplicates` **(Grade II)** Define `duplicates :: Eq a => [a] -> [a]` so that `duplicates xs` gives the list of items that occur more than once in `xs`. No item should occur in the *result* more than once.
eg. `duplicates ["To","be","or","not","to","be"]` ⇝ `["be"]`

♠ `sort` **(Grade III)** Define `sort :: Ord a => [a] -> [a]` so that `sort xs` gives a list containing the same multiset of items as `xs` but in non-decreasing order.
eg. `sort "quick brown fox"` ⇝ `" bcfiknooqruwx"`
Methods most suitable for destructive re-assignment in arrays may not be best for declarative list processing.

♠ `zipWith` **(Grade I)** Define `zipWith :: (a->b->c) -> [a] -> [b] -> [c]` so that `zipWith f xs ys` gives the list of results when `f` is applied to each item in `xs` and the item at the same position in `ys`. If `xs` and `ys` have different lengths, the result should be as long as the shorter argument.
eg. `zipWith (+) [1,1,2,3,5] [1,2,3,5]` ⇝ `[2,3,5,8]`

♠ `Show (Mat a)` & `transpose` **(Grade II)** Let an $m \times n$ *matrix* be a list of length n, in which each item is a list of length m. A simple representation type can be declared: `data Mat a = Mat [[a]]`. Define a suitable `Show (Mat a)` instance. Eg:

```
> Mat [[1,2,3],[4,5,6]]
 1 2 3
 4 5 6
```

Now define `transpose :: Mat a -> Mat a` to take an $m \times n$ matrix and give an $n \times m$ matrix, where the row and columns of the argument are the columns and rows of the result. Assume $m > 0$, $n > 0$.

```
> transpose (Mat [[1,2,3],[4,5,6]])
 1 4
 2 5
 3 6
```

♠ `Show (Tri a)`, `trol` & `tror` **(Grade II)** Let a *triangle of order n* be a list of length n whose i^{th} item is a list of length i. A simple representation type can be declared: `data Tri a = Tri [[a]]` Define a suitable `Show (Tri a)` instance. Eg.

```
> Tri [[0],[1,1],[1,2,1]]
   0
  1 1
 1 2 1
```

Define `trol :: Tri a -> Tri a` so that the result of `trol t` is a rotation of `t` 120 degrees anticlockwise. Eg:

```
> trol (Tri [[0],[1,1],[1,2,1]])
   1
  1 2
 0 1 1
```

Define also `tror :: Tri a -> Tri a`, the inverse of `trol`, to rotate triangles clockwise. Of course you could define `tror t = trol (trol t)` but is there a more direct and efficient solution?

♠ `sublists` **(Grade II)** A *sublist* of a list omits zero or more items. Define `sublists :: [a] -> [[a]]` so that `sublists xs` gives *all* sublists of `xs`.
eg. `sublists ["egg", "bacon"]` ⇝ `[[],["egg"],["bacon"],["egg", "bacon"]]`
Sublists need not be listed in the same order as in the example.

♠ `prefixes, suffixes` **(Grade II)** Define `prefixes, suffixes :: [a] -> [[a]]` to list all the non-empty prefixes (or suffixes) of a list argument `xs`, including `xs` if it is non-empty.
eg. `prefixes ["not","to","be"]` ⇝ `[["not"],["not","to"],["not","to","be"]]`
eg. `suffixes ["not","to","be"]` ⇝ `[["not","to","be"],["to","be"],["be"]]`

♠ `segments` **(Grade II)** A *segment* of a list `xs` is any non-empty list of items occurring consecutively in `xs`. Define `segments :: [a] -> [[a]]` to list *all* segments of its argument.
eg. `segments [1,2,3]` ⇝ `[[1],[2],[3],[1,2],[2,3],[1,2,3]]`
Segments need not be listed in the same order as in the example.

♠ `perms` **(Grade III)** Define `perms :: [a] -> [[a]]` so that `perms xs` lists all permutations of the items in `xs`.
eg. `perms [0,1,2]` ⇝ `[[0,1,2],[0,2,1],[1,0,2],[1,2,0],[2,0,1],[2,1,0]]`
Permutations need not be listed in the same order as in the example.

♠ `parts` **(Grade II)** A *partition* of a list `xs` is any list of segments which concatenate to form `xs`. Define `parts :: [a] -> [[[a]]]` to give all possible partitions of a given list.
eg. `parts [1..3]` ⇝ `[[[1],[2],[3]],[[1,2],[3]],[[1],[2,3]],[[1,2,3]]]`
Partitions need not be listed in the same order as in the example.

♠ `change` **(Grade II)** Define `change :: [Int] -> Int -> [[Int]]` so that, if `cs` represents a set of available coin values, and `m` a required amount of money, `change cs m` lists all the distinct bags of coins with total value `m`. Assume coin values in `cs` are positive, and `m` is non-negative. Bags can be represented as ordered lists, and may contain each coin value any number of times.
eg. `change [1,2,5] 6` $\leadsto$ `[[1,1,1,1,1,1],[1,1,1,1,2],[1,1,2,2],[1,5],[2,2,2]]`
Bags need not be listed in the same order as in the example.

♠ `Ktrain` **program (Grade II)** Implement in Haskell a program `Ktrain` to motivate and test trainee keyboard users. `Ktrain` takes as command-line argument the name of a text file, and presents each line in turn for the user to copy. After each line of input, `Ktrain` reports the total number of words typed so far, and the total number of errors — words missed or wrongly typed. At the end, `Ktrain` reports overall typing speed (in words per minute) and accuracy (word-count less error-count as a percentage of word-count). Here is a transcript of an example run:

```
$ Ktrain rhyme.txt
Half a pound of tuppenny rice;
Half a poind of tuppency rice;
6 words, 2 errors
half a pound of treacle;
haslf a pound fo ttracle;
11 words, 5 errors
that's the way the money goes:
that;s the way the money gos
17 words, 7 errors
pop goes the weasel!
peop goes the weasel!
21 words, 8 errors
speed 55wpm, accuracy 61%
```

As test files for copy-typing, try `Ktest0.txt` and `Ktest1.txt`.

Colin Runciman, February 2017