3 Recursive definitions on lists

Using pattern-matching on lists, give recursive definitions of:

1. a function $prod :: [Int] \to Int$ that calculates the product of a list of integers;

   In this document, we only provide one answer for each function. If you look at the solution code, you'll see a number of variants for most.

   $$\begin{aligned}&prod :: [Int] \to Int\\ &prod\;[\,] \quad\;\; = 1\\ &prod\;(x : xs) = x * prod\;xs\end{aligned}$$

   Notice that the base case (for the empty list) is 1, not 0 as for the function *sum*. The relationship between 1 and $*$ is that 1 is the unit of $*$, that is, $1 * x = x * 1 = x$ for any x. What happens if you make the base case 0?

2. a function $allTrue :: [Bool] \to Bool$ that determines whether every element of a list of booleans is true;

   $$\begin{aligned}&allTrue :: [Bool] \to Bool\\ &allTrue\;[\,] \quad\;\; = True\\ &allTrue\;(x : xs) = x \mathbin{\&\&} allTrue\;xs\end{aligned}$$

3. a function *allFalse* that similarly determines whether every element of a list of booleans is false;

   $$\begin{aligned}&allFalse :: [Bool] \to Bool\\ &allFalse\;[\,] \quad\;\; = True\\ &allFalse\;(x : xs) = not\;x \mathbin{\&\&} allFalse\;xs\end{aligned}$$

4. a function *decAll* :: [*Int*] → [*Int*] that decrements each integer element of a list by one;

```
decAll :: [Int] → [Int]
decAll [ ]      = [ ]
decAll (x : xs) = (x − 1) : decAll xs
```

5. a function *convertIntBool* :: [*Int*] → [*Bool*] that, given a list of integers, converts any zeros to *False*, and any other number to *True*;

```
convertIntBool :: [Int] → [Bool]
convertIntBool [ ]      = [ ]
convertIntBool (0 : xs) = False : convertIntBool xs
convertIntBool (_ : xs) = True : convertIntBool xs
```

6. a function *pairUp* :: [*Int*] → [*Char*] → [(*Int*, *Char*)] that pairs up corresponding elements of the two lists, stopping when either list runs out. For example:

```
pairUp [1,2,3] ['a','b','c'] = [(1,'a'),(2,'b'),(3,'c')]
pairUp [1,2]   ['a','b','c'] = [(1,'a'),(2,'b')]
pairUp [1,2,3] ['a','b']     = [(1,'a'),(2,'b')]
```

```
pairUp :: [Int] → [Char] → [(Int, Char)]
pairUp [ ]      _        = [ ]
pairUp _        [ ]      = [ ]
pairUp (x : xs) (y : ys) = (x, y) : pairUp xs ys
```

Note that this function has to analyse both arguments in parallel.

7. a function $takePrefix :: Int \to [a] \to [a]$ that returns the prefix of the specified length of the given list (or the whole list, if it is too short);

```
takePrefix :: Int → [a] → [a]
takePrefix 0 xs       = []
takePrefix n []       = []
takePrefix n (x : xs) = x : takePrefix (n − 1) xs
```

Like *pairUp*, this analyzes both arguments in parallel.

8. a function $dropPrefix :: Int \to [a] \to [a]$ that similarly drops such a prefix (or the whole list, if it is too short);

```
dropPrefix :: Int → [a] → [a]
dropPrefix 0 xs       = xs
dropPrefix n []       = []
dropPrefix n (x : xs) = dropPrefix (n − 1) xs
```

9. a function $member :: Eq\ a \Rightarrow [a] \to a \to Bool$ that determines whether a given list contains a specified element.

```
member :: Eq a ⇒ [a] → a → Bool
member [] y                = False
member (x : xs) y | x == y = True
member (_ : xs) y          = member xs y
```

10. a function $equals :: Eq\ a \Rightarrow [a] \to [a] \to Bool$ that determines whether two lists contain the same elements in the same order.

```
equals :: Eq a ⇒ [a] → [a] → Bool
equals []       []       = True
equals []       (y : ys) = False
equals (x : xs) []       = False
equals (x : xs) (y : ys) = (x == y) && equals xs ys
```

This is rather like *pairUp* in structure.

The definitions of the following functions deviate slightly from the usual pattern. You may find useful the function $error :: String \to a$, which takes a *String* as an error message and has whatever type you want. Give recursive definitions for:

11. a function $select :: [\,a\,] \to Int \to a$ that selects the element of the list at the given position;

    $$\begin{array}{l} select :: [\,a\,] \to Int \to a \\ select\ (x : xs)\ 0 = x \\ select\ (x : xs)\ n = select\ xs\ (n-1) \end{array}$$

    Notice that this function is undefined on the empty list (there is no equation for that case), and more generally when the requested index is beyond the end of the list.

12. a function $largest :: [\,Int\,] \to Int$ that calculates the largest value in a list of integers;

    $$\begin{array}{lcl} largest :: [\,Int\,] \to Int && \\ largest\ [\,x\,] & = & x \\ largest\ (x : xs) & = & max\ x\ (largest\ xs) \end{array}$$

    Notice that here, the base case is for singleton lists rather than empty lists; it isn't clear what to return as the largest element of the empty list. Therefore, the function is undefined for the empty list, as neither equation matches. The second equation is only applied when the first desn't match, so is applied only to lists of length two or more.

13. a function *smallest* that similarly calculates the smallest value in a list of integers;

Similarly,

$$\begin{array}{lcl} \mathit{smallest} :: [\,\mathit{Int}\,] \rightarrow \mathit{Int} & & \\ \mathit{smallest}\,[\,x\,] & = & x \\ \mathit{smallest}\,(x : \mathit{xs}) & = & \mathit{min}\;x\;(\mathit{smallest}\;\mathit{xs}) \end{array}$$

Some of your definitions probably have more general types than required; can you say which? (Hint: you will find model answers to all parts in the Haskell standard libraries; but they have been given different names here, so that you don't have name clashes.)

When we have covered the corresponding material in the lectures, you may want to return to consider which of these functions can be written more simply using *list comprehensions* or standard *higher-order operators* like *map* and *foldr*.