2 Lists and other algebraic datatypes

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

1. a function *prod* :: [*Int*] → *Int* that calculates the product of a list of integers;
2. a function *allTrue* :: [*Bool*] → *Bool* that determines whether every element of a list of booleans is true;
3. a function *allFalse* that similarly determines whether every element of a list of booleans is false;
4. a function *decAll* :: [*Int*] → [*Int*] that decrements each integer element of a list by one;
5. a function *convertIntBool* :: [*Int*] → [*Bool*] that, given a list of integers, converts any zeros to *False*, and any other number to *True*;
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')]
```

7. a function *takePrefix* :: *Int* → [*a*] → [*a*] that returns the prefix of the specified length of the given list (or the whole list, if it is too short);
8. a function *dropPrefix* :: *Int* → [*a*] → [*a*] that similarly drops such a prefix (or the whole list, if it is too short);
9. a function *member* :: *Eq a* ⇒ [*a*] → *a* → *Bool* that determines whether a given list contains a specified element.
10. a function *equals* :: *Eq a* ⇒ [*a*] → [*a*] → *Bool* that determines whether two lists contain the same elements in the same order.

(These questions mostly have answers already in the Prelude, but under a different name. I changed the name to avoid you having to deal with the clash. You could use `http://www.haskell.org/hoogle/` to find the standard names.)

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

Some of your definitions probably have more general types than required; can you say which?

Once 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*.

2.1 Binary trees

The datatype declaration

$$\mathbf{data}\ Tree\ a = Empty \mid Node\ (Tree\ a)\ a\ (Tree\ a)\ \mathbf{deriving}\ Show$$

defines binary trees. For example,

$$t :: Tree\ Int$$
$$t = Node\ (Node\ Empty\ 1\ Empty)\ 2\ (Node\ Empty\ 3\ Empty)$$

Use pattern-matching to give recursive definitions of:

14. a function $size :: Tree\ a \to Integer$ that calculates the number of elements in a tree;
15. a function $tree :: [a] \to Tree\ a$ that converts a list into a tree;
16. a function $memberT :: Eq\ a \Rightarrow a \to Tree\ a \to Bool$ that determines whether a given tree contains a specified element;

17. a function *searchTree* :: *Ord a* $\Rightarrow$ [*a*] $\rightarrow$ *Tree a* that converts a list into a search tree (a search tree is a tree in which, for a given node, all the values in the left subtree are smaller and all the values in the right subtree are larger);

18. a function *memberS* :: *Ord a* $\Rightarrow$ *a* $\rightarrow$ *Tree a* $\rightarrow$ *Bool* that determines whether a given search tree contains a specified element (this should be more efficient than your definition of *memberT*);

19. a function *inOrder*:: *Tree a* $\rightarrow$ [*a*] that produces the list of elements from an in-order tree traversal (an in-order tree traversal is one in which the left subtree is traversed, then the root node, and then the right subtree).