Exercises week 16

Exercise 16.1 Define an Haskell function `linear :: Int -> Tree Int` so that `linear` n produces a right-linear tree with n nodes. For instance, `linear 0` should produce `Lf`, and `linear 2` should produce `Br 2 Lf (Br 1 Lf Lf)`. The definition of `Tree` is given below.

```
data Tree a = Br a (Tree a) (Tree a) | Lf
```

Exercise 16.2 In the lecture, we have seen an Haskell function `preorder :: Tree a -> [a]` that returns a list of the node values in a tree, in *preorder* (root before left subtree before right subtree).

Now define a function `inorder` that returns the node values in *inorder* (left subtree before root before right subtree) and a function `postorder` that returns the node values in *postorder* (left subtree before right subtree before root):

```
inorder   : Tree a -> [a]
postorder : Tree a -> [a]
```

Thus if `t` is `Br 1 (Br 2 Lf Lf) (Br 3 Lf Lf)`, then `inorder t` is `[2, 1, 3]` and `postorder t` is `[2, 3, 1]`.

It should hold that `inorder (linear n)` is `[n, n-1, ..., 2, 1]` and `postorder (linear n)` is `[1, 2, ..., n-1, n]`, where `linear n` produces a right-linear tree as in Exercise 16.1.

Note that the postfix (or reverse Polish) representation of an expression is just a *postorder list of the nodes in the expression's abstract syntax tree*.

Finally, define a more efficient version of `inorder` that uses an auxiliary function `ino :: Tree a -> [a] -> [a]` with an accumulating parameter; and similarly for `postorder`.

Exercise 16.3 Extend the expression language `Expr` from `Intcomp1.hs` with multiple *sequential* let-bindings, such as this (in concrete syntax):

```
let x1 = 5+7   x2 = x1*2 in x1+x2 end
```

Next, revise the `eval` interpreter from `Intcomp1.hs` to work for the `Expr` language extended with multiple sequential let-bindings. To evaluate this, the right-hand side expression `5+7` must be evaluated and bound to `x1`, and then `x1*2` must be evaluated and bound to `x2`, after which the let-body `x1+x2` is evaluated.

Exercise 16.4 Revise the function `freevars :: Expr -> [String]` to work for the language as extended in Exercise 16.3. Note that the example expression in the beginning of Exercise 16.3 has no free variables, but `let x1 = x1+7 in x1+8 end` has the free variable `x1`, because the variable `x1` is bound only in the body (`x1+8`), not in the right-hand side (`x1+7`), of its own binding. (There *are* programming languages where a variable can be used in the right-hand side of its own binding, but this is not such a language.)

Exercise 16.5 Now modify the interpretation of the language from Exercise 16.3 so that multiple let-bindings are *simultaneous* rather than sequential. For instance,

```
let x1 = 5+7   x2 = x1*2 in x1+x2 end
```

should still have the abstract syntax

```
Let [("x1", ...); ("x2", ...)] (Prim "+" (Var "x1") (Var "x2"))
```

but now the interpretation is that all right-hand sides must be evaluated before any left-hand side variable gets bound to its right-hand side value. That is, in the above expression, the occurrence of `x1` in the right-hand side of `x2` has nothing to do with the `x1` of the first binding; it is a free variable.

Revise the `eval` interpreter to work for this version of the `Expr` language. The idea is that all the right-hand side expressions should be evaluated, after which all the variables are bound to those values simultaneously. Hence

```
let x = 11 in let x = 22  y = x+1 in x+y end end
```

should compute `12 + 22` because `x` in `x+1` is the outer `x` (and hence is 11), and `x` in `x+y` is the inner `x` (and hence is 22). In other words, in the let-binding

```
let x1 = e1  ...  xn = en in e end
```

the scope of the variables `x1 ... xn` should be `e`, not `e1 ... en`.