L28: Advanced functional programming

Exercise 0 (optional)

About this exercise

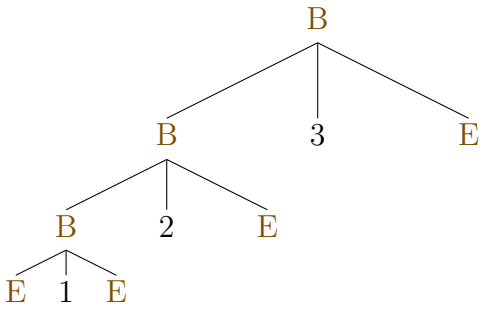
This exercise is offered as an aid to ensure that students start the course appropriately prepared. The exercise is **not assessed** and will have **no effect on your grade**.

If you have taken the courses Foundations of Computer Science (which introduces SML) and Types (which presents typed lambda calculi), or similar courses elsewhere, then it is likely that you will be adequately prepared.

If you have only limited experience with typed functional programming then you may find it helpful to work through one of the suggested OCaml introductions on the L28 course website before attempting this exercise. The course website also has suggestions for background reading for the more theoretical early lectures.

1 Trees, folds and maps

Many functional programs process tree-structured data such as the following value:



This value is built from two types of tree:

- Empty trees are denoted by E
- Branches are denoted by B and carry a label (such as `1`) and two child trees

(*a*) Define a type `tree` of trees with constructors for empty trees and for branches. Your trees should be able to hold any kind of data. For example, an `int tree` should hold `int` values, a `string tree` should hold `string values`, and so on.

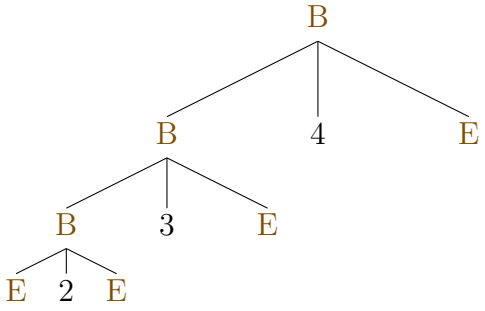
(*b*) The function `List.map` builds a list by applying a function to each element of an existing list. For example, the application of `List.map` to a three-element list behaves as follows:

```
  map f (a ::  b  ::  c  :: [])
↝
       f a :: f b :: f c :: []
```

Define an analogous function, `map_tree`, with the following type:

```
val map_tree : ('a -> 'b) -> 'a tree -> 'b tree
```

The call `map_tree f t` should build a tree with the same shape as `t` by applying `f` to each element of `t`. For example, applying `map_tree (fun x -> x + 1)` to the tree depicted above should produce the following tree:



(*c*) The function `List.fold_right` builds a value from a list as follows: it takes two arguments, `f` and `u`, and applies `f` each time it encounters a cons (`::`) constructor, and substitutes `u` for the nil (`[]`) constructor. For example, the application of `fold_right` to the `+` function, `0`, and a list of numbers behaves as follows:

```
List.fold_right (+) 0 (a :: b :: c :: [])
↝
                          a +  b +  c +  0
```

Define an analogous function, `fold_tree`, with the following type

```
val fold_tree : ('b -> 'a -> 'b -> b) -> 'b -> 'a tree -> 'b
```

The call `fold_tree f u t` should construct a value by applying `f` at each B node and substituting `u` for each E node. Then `fold_tree (+) 0` should produce `6` (i.e. `((0+1+0)+2+0)+3+0`) and `9` when applied to the two trees depicted above.

2 Type inference

(a) Here is a function `f` that accepts a function `g` and a pair `(x,y)` and applies `g` to the two elements of the pair:

```
let f g (x, y) = g x y
```

And here is the type that the OCaml compiler gives to `f`:

```
val f : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

Briefly outline the process by which the compiler produces this type from the definition of `f`.