Exercise 1  Traversing Trees  (3 Points)

The following code implements a breadth-first traversal of a tree, using the standard implementation of trees from Data.Tree:

```haskell
import Data.List
import Data.Monoid
import Data.Tree (Tree(..))

newtype BFSTree a = BFS (Tree a)

instance Foldable (BFSTree) where
  foldMap f (BFS tr) = go [tr]
  where
    go q = case q of
      [] -> mempty
      (Node x xs) : qs -> f x `mappend` go (qs ++ xs)
```

Hence one can run programs like

```haskell
foldr (:) [] (BFS some_tree)
```

to obtain a breadth-first unfolding of a tree `some_tree` into a list, instead of the default depth-first unfolding by

```haskell
foldr (:) [] some_tree
```

In contrast to the depth-first strategy, the breadth-first search strategy fully traverses even a tree with infinitely many nodes.

```haskell
t = Node "1(0)" [Node "2(1)" [Node "4(2)" [], Node "5(2)" []], Node "3(1)" []]
```

Then

```haskell
foldr (:) [] t == ["1(0)","2(1)","4(2)","5(2)","3(1)"]
foldr (:) [] (BFS t) == ["1(0)","2(1)","3(1)","4(2)","5(2)"]
```

(a) This implementation is based on using the list type `[BFSTree]` as a queue in which new trees are added at the back with the `qs ++ xs` command, which is highly inefficient, because it requires full traversal of the pending queue `qs` at every iteration. Reimplement `foldMap`, so that it behaves the same, but does not suffer from this issue. Compare the performance of both implementations by running tests on exponentially growing trees, e.g.

```haskell
expTree a b = Node (a, b) [expTree (a + 1) b, expTree a (b + 1)]
```

(b) Analogously, implement a strategy that zig-zags through the three in a breadth-first fashion, e.g.:

```haskell
foldr (:) [] (ZZS t) == ["1(0)","2(1)","3(1)","5(2)","4(2)"]
```
Exercise 2  PCF in Haskell  (4 Points)

PCF-Terms can be internally represented in Haskell in the following way.

--- Alias type for variable names as strings
type Name = String

--- Language Terms
data Expr = T | F |
| IfThenElse Expr Expr Expr |
| Nmb Int |
| Sum Expr Expr |
| Dif Expr Expr |
| Eq Expr Expr |
| Var Name |
| App Expr Expr |
| Lam Name Expr |
| Fix |
| deriving (Eq, Show) |

For example, we can define the following familiar terms:

omega = App Fix (Lam "x" (Var "x"))
church0 = Lam "z" (Lam "n" (Var "z"))
church1 = Lam "z" (Lam "n" (App (Var "n") (Var "z")))
church2 = Lam "z" (Lam "n" (App (Var "n") (App (Var "n") (Var "z"))))

(of course, nothing prevents one from forming nonsensical terms, which do not correspond to typable PCF-terms, like App (Nmb 0) (Nmb 0); there is a technique to faithfully represent typed languages within Haskell [1], but we do not have capacities to dwell on it).

Implement call-by-name small-step reduction as a Haskell function:

\[
\text{eval} :: \text{Expr} \rightarrow \text{Expr}
\]

so, that \( t' = \text{eval} t \) iff \( t \rightarrow_{\text{cbs}} t' \) for all correctly typed terms \( t, t' \) (implementing type checking is not required).

Exercise 3  Lazy Lists  (8 Points)

(a) Complete the PCF language, described at the lecture with

- type constructors for forming coproduct types \( A + B \) (corresponding to Haskell’s \textbf{Either} \( A \) \( B \)) and for forming lists \( A^* \) (corresponding to Haskell’s \([A]\)),
- with the corresponding constructors \texttt{inl}, \texttt{inr} (corresponding to \textbf{Left} and \textbf{Right}), \texttt{nil} and \texttt{cons} (corresponding to \([ \] \) and \( : \)),
- with the corresponding case-statements that pattern-match on the new constructors (take inspiration from PCF’s if-then-else and Haskell’s case operator).

More precisely, you need to: add the new rules for forming terms-in-contexts, to modify the definition of value and to add the new (big-step or small-step) call-by-name operational semantics rules.

\textbf{Hint:} Do the case of coproducts first and then switch to lists. In both cases, success depends on understanding the relevant notion of value correctly; recall that values are irreducible closed
terms, w.r.t. the evaluations strategy of interest — it is thus advisable to fully describe what a value is in each case and then produce the corresponding rules of the operational semantics.

(b) Extend the interpreter from Exercise 1 accordingly.

(c) How the notion of value would change if we considered the call-by-value semantics?

(d) Translate the following Haskell program
\[
\text{fib} = 1 : 1 : [a + b \mid (a, b) \leftarrow \text{zip} \text{ fib} (\text{tail fib})]
\]
to the above described extended PCF. **Hint:** For a start, desugar the above Haskell program suitably, to make the syntax closer to that of PCF, e.g. \([f \ a \ b \mid (a, b) \leftarrow xs]\) can be replaced with \([f (\text{fst} x) (\text{snd} x) \mid x \leftarrow xs]\), and the latter can be modelled by the \text{map} function. You then need to implement the relevant auxiliary functions (\text{zip, map, ...}) in PCF and define \text{fib} using them.

(e) Analogously, translate the program \((\text{fib !! 4})\) to a PCF program \(p\) and prove that \(p \downarrow 5\), alternatively that \(p \rightarrow^* 5\) in the small-step semantics style. It is also OK to submit a program that generates the requisite transition (chain), e.g. by running the interpreter from part (b).

**Exercise 4  curry and uncurry  (5 Points)**

Given two complete partial orders \(A\) and \(B\), let \(A \rightarrow_c B\) be the space of continuous functions from \(A\) to \(B\), more precisely

\[A \rightarrow_c B = \{f : A \rightarrow B \mid A \rightarrow B, \text{ such that } f \text{ is continuous}\}.
\]

Note that \(A \rightarrow_c B\) is a partial order under the pointwise extension from \(B\), i.e. \(f \sqsubseteq g\) for \(f, g : A \rightarrow_c B\) if \(f(x) \sqsubseteq g(x)\) for all \(x \in A\).

(a) Show that \(A \rightarrow_c B\) are complete partial orders, if \(A\) and \(B\) are so;

(b) Given complete partial order \(A\), \(B\) and \(C\), show that \text{curry}: \((C \times B \rightarrow_c A) \rightarrow (C \rightarrow_c (A \rightarrow_c B))\) and \text{uncurry}: \((C \rightarrow_c (A \rightarrow_c B)) \rightarrow (C \times B \rightarrow_c A)\) are monotone and continuous where

\[
\text{(curry } f)(x)(y) = f(x, y) \quad \text{ and } \quad \text{(uncurry } f)(x, y) = f(x)(y)
\]

(you are encouraged to experiment with \text{ghci}, in which these functions are available under the same name). **Hint:** Use the fact from the lecture that \(A \times B\) is a complete partial order whenever \(A\) and \(B\) are, where \((a, b) \sqsubseteq (a', b')\) means by definition that \(a \sqsubseteq a'\) in \(A\) and \(b \sqsubseteq b'\) in \(B\);

**References**