Exercise sheet 10— Solutions

1 `map` via `traverse`

```
def mapViaTraverse[F[_],A,B](fa: F[A])(f: A => B)(using Traverse[F]): F[B] = ???
```

As mentioned in the lecture, `traverse` is equivalent to `map` followed by `sequence`. `traverse` expects a function, that returns an Applicative, so we need to modify our function passed to map accordingly.

As `map`'s signature does not require an Applicative, we can choose an arbitrary one, as long as we can get the value out of it afterwards (as `traverse` gives us the resulted wrapped inside the Applicative).

For this use case, `Id` is an obvious choice:

```
fa.traverse(f(_).pure[Id])
```

(The notation `x.pure[F]` is a `cats` shorthand for `Applicative[F].pure(x)`)

This way, `traverse` gives us an `Id[F[B]]`, which is identical to `F[B]`.

But every other `Applicative` mentioned in the lecture is also usable, for example `List`:

```
fa.traverse(f(_).pure[List]).head
```

2 Traverse instance for trees

2.1 Implementation

```
given Traverse[Tree] with
  import fp06.given
  import fp06.Tree.*

  def traverse[G[_],A,B](fa: Tree[A])(f: A => G[B])(using Applicative[G]): G[Tree[B]] =
    fa match
      case Leaf(a)             => f(a).map(Leaf(_))
      case Branch(left, right) => left.traverse(f).map2(right.traverse(f))(Branch(_, _))

  def foldLeft[A, B](fa: Tree[A], b: B)(f: (B, A) => B): B =
    summon[Foldable[Tree]].foldLeft(fa, b)(f)

  def foldRight[A, B](fa: Tree[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B] =
    summon[Foldable[Tree]].foldRight(fa, lb)(f)
```

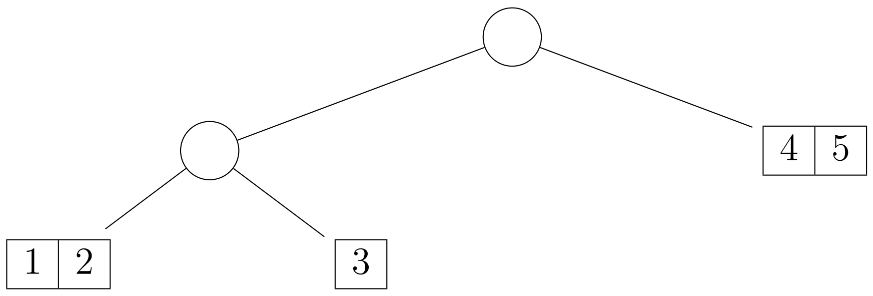
In our `traverse` implementation, we first look at the passed tree. If it is a `Leaf`, we got a value which we can call `f` with. We get a `G[B]` back, but need a `G[Tree[B]]`, so we wrap the value *in the Applicative* into a `Leaf` using `map`.

If we have a `Branch` instead, we call `traverse` recursively for both subtrees, which gives us two `G[Tree[B]]`. We can combine those into a `Branch` inside `G` again using `map2`.

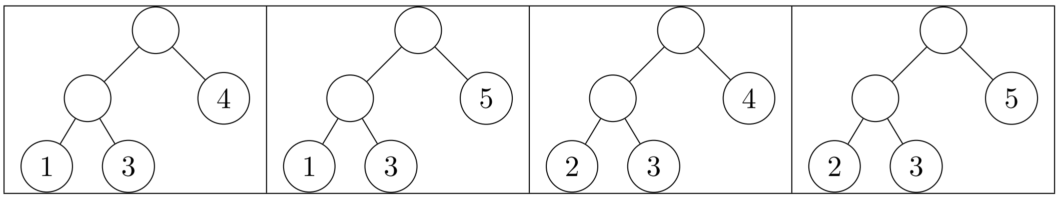
As said, we can reuse the implementations of the foldable instance from the sixt exercise sheet. We import the givens from the `fp06` package and use `summon[Foldable[Tree]]` to get the instance, on which we can then call `foldLeft` and `foldRight` respectively. Of course copying the code would also be possible.

2.2 Behaviour of sequence

Let's first think abount the behaviour of `sequence` for a `Tree[List[Int]]`, i.e. using the `sequence` method of the List traverse instance. We have a tree, which has Lists in its leaves:

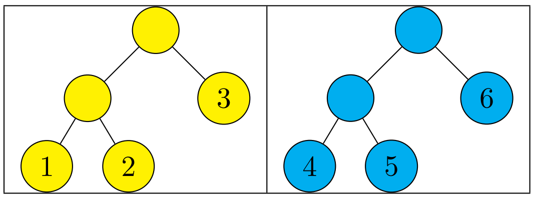


We flip that using `sequence`: `(x: Tree[List[Int]]).sequence` → `List[Tree[Int]]`

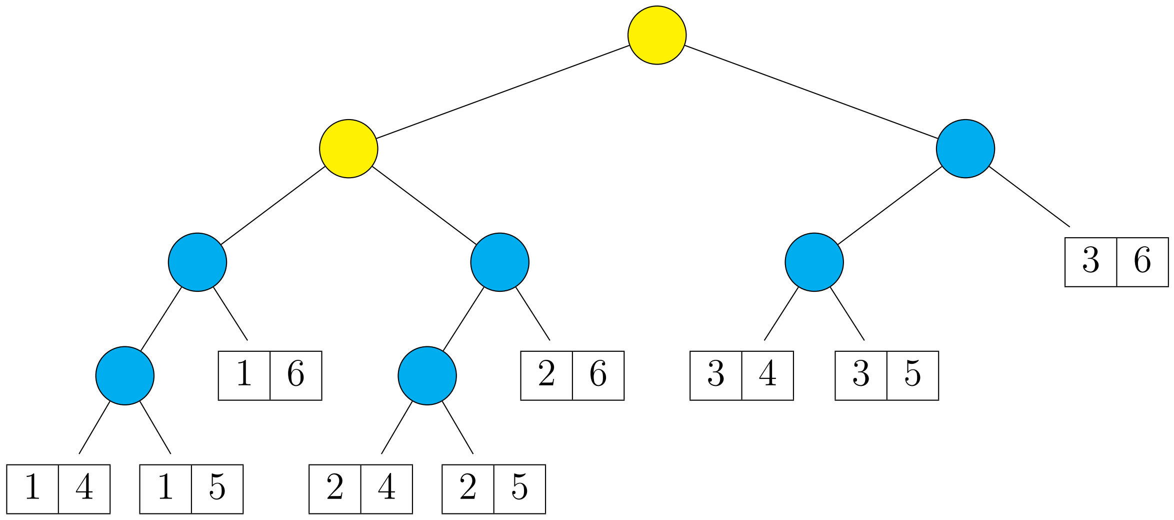


So we get a list of trees, which have the same stucture as the original tree. For all leaves with multiple elements we get a tree each with every possible combination with the other leaves (so in total as many trees as the product of all list lengths, here $2 \cdot 1 \cdot 2 = 4$).

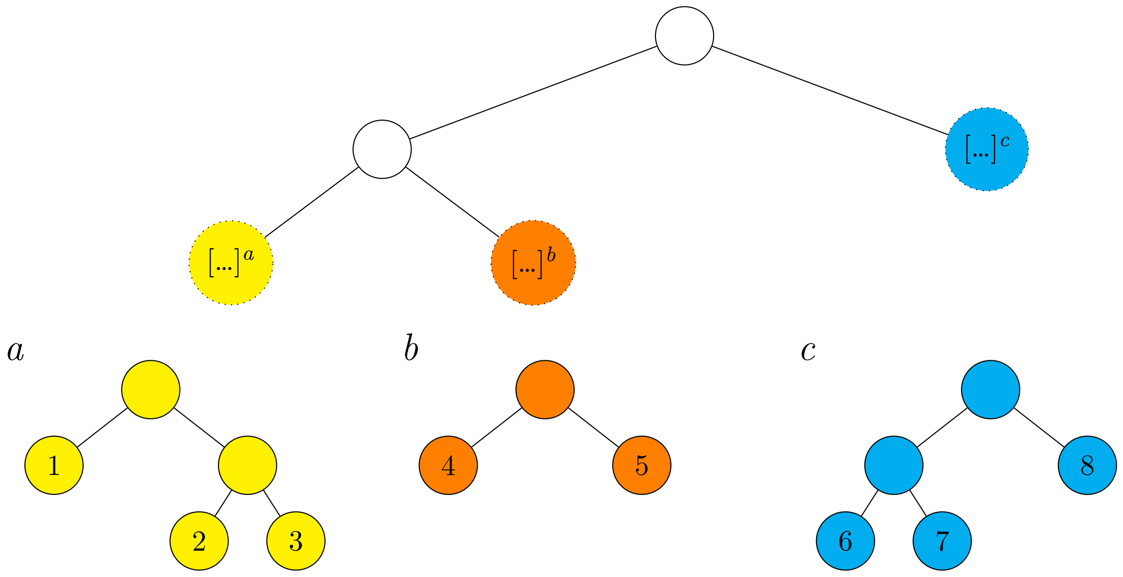
What about the other direction, i.e. `(x: List[Tree[Int]]).sequence` → `Tree[List[Int]]`? Here an example, two trees in a list:



As we want to have a single tree in the end, `sequence` has to combine them. This happens by replacing the leaves of the first tree with the structure of the second tree. The leaves then contain a list with the value from the replaced leaf in the first tree and the value at that position in the second tree, so basically the path through the tree. This results in the following larger tree:

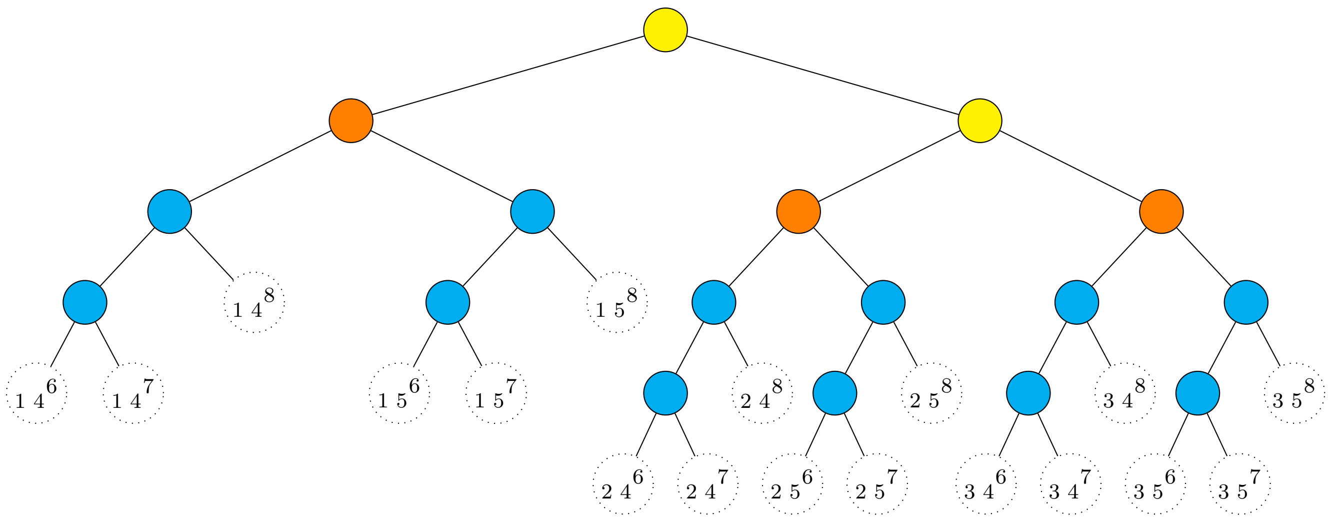


Something similar happens, if we flip a nested tree with `sequence` umkehren. In the following image we see a tree (white nodes), which has a tree with integers in every leaf (colored nodes), so the type is `Tree[Tree[Int]]`:



Similar to `sequence` on `List[Tree[Int]]`, the stucture of the first (i.e. here the leftmost) tree can be found at the root, the structure of the second one in place of the first tree's leaves, the third tree's structure in place of the second's leaves and so on.

The leaves of the last tree then conain a tree *as their element*, which has the stucture of the previously outer tree. This tree's elements are the numbers which were leaves in the colored trees on the path to this leaf:



3 Accumulating with State

3.1 reverse

As a reminder, the implementation of `mapAccum`:

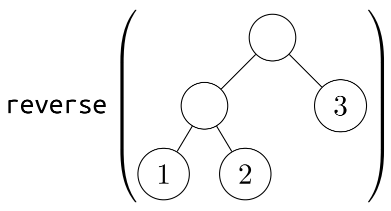
```scala
def mapAccum[F[_]:Traverse,S,A,B](fa: F[A], s: S)(f: (S,A) => (S,B)): (S,F[B]) =
  fa.traverse(a => State(s => f(s, a))).run(s).value
```

As lists already implement `reverse` and we can convert every `Traverse` into a list, we use this to set our starting value for the accumulation: the list of elements in reverse order.

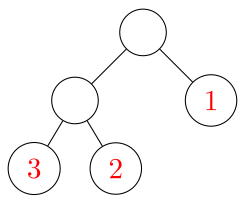
In the function, which we pass to `mapAccum`, we ignore the current element of our traversable functor, we only need the traversable for keeping its structure. We get the values from the reversed list. From that list we return the head as value and the tail as new state. We therefore use the reversed list as a stack, from which we take elements one by one and put them in the position that `fa` determines:

```scala
def reverse[F[_],A](fa: F[A])(using Traverse[F]): F[A] =
  mapAccum(fa, fa.toList.reverse)((l, _) => (l.tail, l.head))._2
```

Here a step-by-step example with a tree. On the left the call, in the table the single steps, on the right the result:



A	S	B
–	$\boxed{3}\boxed{2}\boxed{1}$	–
①	$\boxed{2}\boxed{1}$	3
②	$\boxed{1}$	2
③	∅	1



At the beginning of the call, our state is the list of all leaf values in reversed order. In the forst step the `A` passed to the function is the value from the node on the outer left. This value is ignored, but instead the first value from the stack (S) is used. We continue this, until we have traversed all leaves.

3.2 foldLeft via mapAccum

```scala
def foldLeftViaMapAccum[F[_]:Traverse,A,B](fa: F[A], z: B)(f: (B, A) => B): B =
  mapAccum(fa, z)((s, a) => (f(s, a), ()))._1
```

Similar to the `toList` implementation in the lecture, we don't need the value in `F` that `mapAccum` produces in the end, but our state `S`. Therefore we always return unit, as we did in `toList`.

We set the starting value for our state to the value `z` passed to `foldLeft`. In the anonymous function we then set this to the result of the function `f` passed to `foldLeft`.