6 Type classes

Exercise 6.1 (Warm-up: type classes, `BinaryTree.lhs`). Recall the definition of binary trees.

```
data Tree elem = Empty | Node (Tree elem) elem (Tree elem)
    deriving (Eq, Ord)
```

The `deriving` clause automagically generates instance declarations for the type classes `Eq` and `Ord`. To appreciate this convenience, remove the `deriving` clause and program the instances by hand.

```
instance (Eq elem) ⇒ Eq (Tree elem) where ...
instance (Ord elem) ⇒ Ord (Tree elem) where ...
```

Are these instance declarations actually useful? Can you think of a use-case where you would need to compare two binary trees? *Warning:* note that the method name for ordering is `<=` (two ASCII symbols) and not `≤` (one Unicode character).

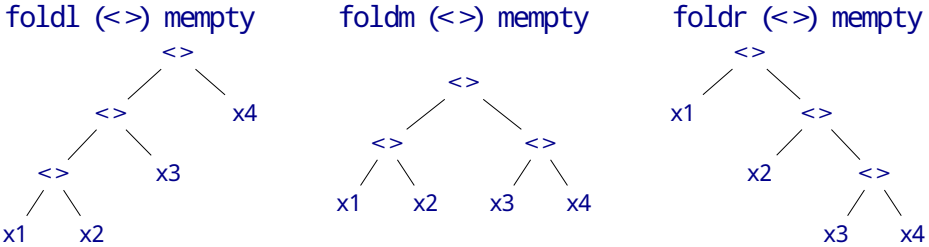
Exercise 6.2 (Warm-up: map-reduce, `MapReduce.lhs`).

1. How many ways are there to turn the type `Bool` into a monoid? (There are sixteen *candidates* as there are sixteen functions of type `Bool → Bool → Bool`.) Define all of them using `newtype` definitions—think hard about telling names.

2. For each of the Boolean monoids, what is the meaning of `reduce`? Are any of these functions predefined (under a different name)?

Exercise 6.3 (Programming: map-reduce, `MapReduce.lhs`). The type of lists forms a monoid: `mempty` is the empty list `[]`, and `<>` is list concatenation `++`. The type of *ordered* lists also forms a monoid: what are `mempty` and `<>`? Provide a suitable instance declaration. Can you use the monoid to implement a sorting function? Exercise 3.3 is also potentially useful.

Exercise 6.4 (Programming: map-reduce, `MapReduce.lhs`). In the lectures we have implemented `reduce` in terms of a higher-order function: `reduce = foldr (<>) mempty`. Of course, this is a rather arbitrary choice: `reduce = foldl (<>) mempty` works equally well. Since the operation is associative, it does not matter how nested applications of '`<>`' are parenthesized. The overall result is bound to be the same. However, there is possibly a big difference in running time. For many applications, a balanced "expression tree" is actually preferable, see for example Exercise 6.3. In particular, as a balanced tree can in principle be evaluated in parallel!



Define a higher-order function

```
foldm :: (a → a → a) → a → ([a] → a)
```

that constructs and evaluates a balanced expression tree, see also Exercise 4.3.

1. The types of `foldl`, `foldm`, and `foldr` are quite different. How come?

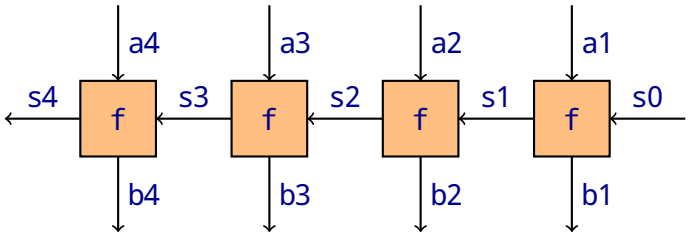
2. Implement `foldm` using a *top-down* approach: Split the input list into two halves, evaluate each halve separately, and finally combine the results using the monoid operation (*divide and conquer*).

3. Implement `foldm` using a *bottom-up* approach: Traverse the input list combining two adjacent elements e.g. `[x1, x2, x3, x4]` becomes `[x1 <> x2, x3 <> x4]`. Repeat the transformation until the list is a singleton list.

(In general, the task of finding a good expression tree is an optimization problem, which requires some knowledge of the run-time behaviour of the monoid operation e.g. matrix chain multiplication.)

Exercise 6.5 (Programming and hardware design, `MapReduce.lhs`). The art of map-reduce is to find a suitable monoid for the problem at hand. Framing a task as a composition of `map` and `reduce` then opens the door to a massively parallel implementation. The exploitation of parallelism is useful on a very large scale (the internet), but also on a very small scale (electronic circuits). This exercise looks into the latter continuing Exercise 5.6.

Reconsider the ripple carry adder:



At first sight, there seems to be little opportunity for parallelization. The computation of the carry is inherently sequential: it ripples through the gates from right to left (hence the name of the circuit). The leftmost full adder can only operate once all the others have finished their work. Perhaps surprisingly, the computation of the carry can be improved. (For simplicity, we only consider computing the final carry. To speed up hardware addition, all the intermediate carries are needed too. But this is mostly a matter of using a scan instead of a fold—a scan in hardware is called a *parallel prefix circuit*. The resulting adder is called *carry look-ahead adder*.)

Now, the first step is to concentrate on the carries, ignoring the sum bit (you will see in a moment why the function is called `kpg`):

```
kpg' :: (Bit, Bit) → (Carry → Carry)
kpg' s  = \ c → snd (fullAdder (s, c))
```

For each pair of summand bits `(s1, s2)`, the partial application `kpg (s1, s2)` is a function that takes an input carry to an output carry. However, function composition is associative: invoking map-reduce

```
reduce ∘ map kpg :: [(Bit, Bit)] → (Carry → Carry)
```

we can map the given list of summand bits to a function that in turn maps the initial to the final carry. Problem solved!

Well, not quite: how can we possibly implement higher-order functions such as composition in hardware? (Even in Haskell, we haven't gained anything: the nested compositions can be evaluated in parallel, but each composition will create a new function at run-time, a so-called *closure*. When the resulting function is then applied to an initial carry, the evaluation will proceed purely sequential.)

How to make progress? An important observation is that there are not that many total functions of type `Carry → Carry = Bit → Bit`. Four, to be precise. If we partially evaluate `kpg (s1, s2)`, we see that actually only three out of four occur:

```
kpg :: (Bit, Bit) → (Carry → Carry)
kpg (O,  O  )  =  \ _c  → O  -- kill
kpg (O,  I  )  =  \ c   → c  -- propagate
kpg (I,  O  )  =  \ c   → c  -- propagate
kpg (I,  I  )  =  \ _c  → I  -- generate
```

If both summand bits are `O`, the output carry is always `O`—the input carry is killed. Dually, if both summand bits are `I`, the output carry is always `I`—the output carry is generated. In the other two cases, the carry is propagated. The idea is to work with *representation* of functions instead of actual functions.

1. The three functions `const O`, `id`, and `const I` can be represented using a simple enumeration type (we keep the traditional names):

   ```
   data KPG  =  K | P | G
   ```

   Turn `KPG` into a monoid.

2. Test your implementation exhaustively. To this end define a function

   ```
   apply :: KPG → (Carry → Carry)
   ```

   that maps syntax to semantics. We expect the following properties to hold:

   ```
   apply mempty        =  id
   apply (a <> b) =  apply a ∘ apply b
   ```

   That is, `mempty` represents the identity and `<>` corresponds to function composition. (Mathematically speaking, `apply` is a *monoid homomorphism* from the `KPG` monoid to a transformation monoid.)

3. *Optional:* you may argue that using an enumeration type is still cheating. After all, we want to implement addition in silicon. Agreed. Represent `KPG` using two bits

   ```
   newtype KPG'  =  KPG (Bit, Bit)
   ```

   and adapt the monoid instance to the new representation. Again, test the implementation exhaustively.

Exercise 6.6 (Worked example: digital sorting, `DigitalSorting.lhs`). Both *comparison-based* sorting and searching are subject to lower bounds:

- sorting requires $\Omega(n \log n)$ comparisons, and
- searching for a key requires $\Omega(\log n)$ comparisons,

where n is the number of keys in the input. The purpose of this exercise is to show that we can do a lot better: we can sort in *linear* time if we employ the structure of search keys!

In the comparison-based approach to sorting the ordering function is treated as a *black box*:

```
sortBy :: (a → a → Ordering) → [a] → [a]
```

The higher-order function `sortBy` can only call its first argument, but not inspect it: the ordering is treated as an oracle. As a benefit of the abstraction, there is one *generic* sorting function for all types.

By contrast, in the key-based approach to sorting (a.k.a. digital sorting) we provide a tailor-made sorting function for each type of interest.

```
class Rank key where
  sort  ::  [(key, val)] → [val]
  rank  ::  [(key, val)] → [[val]]

  sort  =  concat ∘ rank
```

Actually, two methods are provided. Both take as input a so-called association list, a list of key-value pairs. The values are treated as *satellite data*: the methods are *parametric* in the value type `val`. The sorting function, `sort`, outputs the value components according to the given order on `key` without, however, returning the key components. The ranking function, `rank`, additionally groups the value components into non-empty runs of values associated with equal keys. The following examples illustrate their use, assuming suitable instances of `Rank` for characters and lists.

```
 ⋙ sort [("ab", 1), ("ba", 2), ("aba", 3), ("ba", 4)]
[1,3,2,4]
 ⋙ rank [("ab", 1), ("ba", 2), ("aba", 3), ("ba", 4)]
[[1],[3],[2,4]]
```

In the last example, we have three runs as there are three different strings in the input. The last run, `[2,4]`, contains the values associated with the key that occurs twice i.e. `"ba"`.

It may seem rather strange that the keys are dropped in the process. Observe, however, that there is actually no loss of generality: if we wish to retain the keys, we simply add them to the value component.

```
 ⋙ kvs = [("ab", 1), ("ba", 2), ("aba", 3), ("ba", 4)]
 ⋙ rank [ (k, (k, v)) | (k, v) ← kvs]
[[("ab",1)],[("aba",3)],[("ba",2),("ba",4)]]
```

We can use the generic comparison-based sorter to give an executable, but inefficient specification of the key-based sorter:

```
sort' :: (Ord key) ⇒ [(key, val)] → [val]
sort' kvs  =  map snd (sortBy (\ kv1 kv2 → compare (fst kv1) (fst kv2)) kvs)
```

The concrete ordering is provided by the type class `Ord`. While the ordering is known—there is at most one `Ord` instance for each type—the generic sorter does not and cannot make use of it. To see how we can take advantage of a white-box approach, consider the second simplest type, the one-element type `()`. Its comparison function is defined:

```
instance Ord () where
    compare () ()  =  EQ
```

Since the type contains only one element, we need not inspect the keys at all! Sorting is just a matter of extracting the value components.

```
instance Rank () where
    sort kvs   =  map snd kvs
    rank kvs   =  [ map snd kvs | not (null kvs) ]
```

Now, for each instance of `Ord`, we aim to provide a corresponding, tailor-made instance of `Rank`.

1. Also specify the ranking function in terms of `compare`, using `sortBy` and `groupBy`. The executable specifications of `sort` and `rank` are actually quite useful: they can be used as (default) implementations if we need to quickly provide an instance of `Rank`.

2. Conversely, can you define `compare` in terms of `sort` or `rank`?

3. Products are ordered using the so-called lexicographic ordering:

   ```
   instance (Ord elem1, Ord elem2) ⇒ Ord (elem1, elem2) where
       compare (a1, a2) (b1, b2)  =  compare a1 b1 ▷ compare a2 b2
   ```

   Only if the first components are equal, the second components are taken into account.

   ```
   (▷) :: Ordering → Ordering → Ordering
   LT  ▷ _ord  =  LT
   EQ  ▷ ord   =  ord
   GT  ▷ _ord  =  GT
   ```

   Define a corresponding instance of `Rank` i.e.

   ```
   instance (Rank key1, Rank key2) ⇒ Rank (key1, key2) where ...
   ```

   Hint: you only have to provide an implementation `rank`. The type of this instance is

   ```
   rank :: (Rank key1, Rank key2) ⇒ [(key1, key2), val] → [[val]]
   ```

   The context indicates that instances of rank for both `key1` and `key2` are available. The first step is to reorganize the input list such that the rank for `key1` can be used (the function `assoc((x,y),z) = (x,(y,z))` might be useful for that purpose). Then you should use the rank for `key2` to process the result list further.

4. The ordering on sums is given by:

   ```
   instance (Ord elem1, Ord elem2) ⇒ Ord (Either elem1 elem2) where
       compare (Left   a1)  (Left   a2)  =  compare a1 a2
       compare (Left   a1)  (Right  b2)  =  LT
       compare (Right  b1)  (Left   a2)  =  GT
       compare (Right  b1)  (Right  b2)  =  compare b1 b2
   ```

   Elements of the form `Left a` are strictly smaller than elements of the form `Right b`. If the "tags" are equal, their arguments determine the ordering. Again, define a corresponding instance of `Rank` i.e.

```
instance (Rank key1, Rank key2) ⇒ Rank (Either key1 key2) where ...
```

5. Strings and, more generally, lists are also ordered using the lexicographic ordering (guess where the name comes from). Recall that a list is *either* empty, or a *pair* consisting of an element and a list. If we capture this description as a type,

```
type List elem  =  Either () (elem, [elem])

toList :: [elem] → List elem
toList []        =  Left ()
toList (a : as)  =  Right (a, as)
```

then we can let the compiler (!) generate the code for ordering:

```
instance (Ord elem) ⇒ Ord [elem] where
    compare as bs  =  compare (toList as) (toList bs)
```

Driven by the type, the compiler automagically combines the orderings for `Either`, `()`, and `(elem, [elem])`. If we inline the various definitions by hand, we obtain the equivalent instance:

```
instance (Ord elem) ⇒ Ord [elem] where
    compare []        []        =  EQ
    compare []        (_ : _ )  =  LT
    compare (_ : _ )  []        =  GT
    compare (a : as)  (b : bs)  =  compare a b ▷ compare as bs
```

Use the same approach to define a corresponding instance of `Rank`:

```
instance (Rank key) ⇒ Rank [key] where ...
```

6. Re-implement (a variant of) the function `repeatedSegments` from the lectures, see "Case study: DNA analysis" §3.3.

```
repeatedSegments :: (Rank key) ⇒ Int → [key] → [[Integer]]
```

The call `repeatedSegments m` takes a list of keys as an input and returns the *positions* of repeated segments of length `m` e.g.

```
 ⋙ dna
ATGTAAAGGGTCCAATGA
 ⋙ repeatedSegments 3 dna
[[0,14]]
```

The segment `ATG` of length `3` occurs at positions `0` and `14`. To be able to test your implementation also provide:

```
instance Rank Base where ...
```

Hint: instances of `Rank` for small enumeration types typically use a technique called *sorting by distribution* also known as bucket sort.

Hints to practitioners 6. In Hint 5 we have touched upon GHC's language extensions. GHC is especially well-known for its many type- and class-system extensions. A useful one is

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

Recall that you can only provide one class instance per type. However, sometimes there is more than one candidate. For example, there are at least four possible instances for `Monoid Int`. This is a typical use-case for `newtype`: we introduce a new type for each instance i.e.

```
newtype Additive = Sum { fromSum :: Int }
    deriving (Eq, Ord, Show)
```

instance Monoid Additive where ...

As the keyword suggests, a `newtype` declaration introduces a new type, which is incompatible with all other existing types. *Slogan:* all data- and newtypes are born unequal. In particular, we cannot say `4711 + Sum 0815` as `Int` and `Sum` are different types. This is usually a welcome feature as the main purpose of a newtype is to erect an abstraction barrier—the other use-case for `newtype` is to define abstract datatypes. However, we also cannot say `Sum 4711 + Sum 0815` as `Sum` has not inherited any instances from `Int`. This is where the language pragma `GeneralizedNewtypeDeriving` comes into play. It allows us to write

```
newtype Additive = Sum { fromSum :: Int }
    deriving (Eq, Ord, Show, Num)
```

freeing us from the arduous task of providing an instance of `Num` by hand. Now, `Sum 4711 + Sum 0815` evaluates to `Sum 5526`. (The expression `4711 + Sum 0815` actually also type-checks as integer literals are overloaded i.e. `4711` is shorthand for `Sum 4711`.)