6 Type classes

Exercise 6.1 (*Warm-up*: Type class instances (worked example), `Pronounce.hs`).
This exercise is a step-by-step example illustrating a simple use of type classes; the amount of code you will have to write is minimal.

1. In Week 1, we implemented a `say :: Integer → String` function. Now, having this function only work on `Integer` is a bit restrictive, so suppose we want to also have this function available for data types `Int`, `Char`, `Float`, `Double`, etc. We are going to create a type class for this, and have already added an instance for `Char`:

   ```
   class Pronounceable a where
     pronounce :: a → String

   instance Pronounceable Char where
     pronounce c = unwords ["the","character", "'"++[c]++"'"]
   ```

   Put your solution of `Say.hs` (or the solution provided to you) in the same directory as `Pronounce.hs`, and load `Pronounce.hs` in GHCi;

2. Create instances of the `Pronounceable` class for the type `Integer` and `Int`, in which the `pronounce` function will produce the same result as the `say` function from Exercise 1.6.

   (Reminder: you can use the `toInteger` function to convert a `Int` to `Integer`)

3. Create an instance of `Pronounceable` for `Double` which uses the `pronounce` instance for `Integer` to put floating point numbers in words, rounded to the first decimal:

   ```
   pronounce 23.0 ⟹ "twenty three point zero"
   pronounce 37.5 ⟹ "thirty seven point five"
   pronounce 3.14 ⟹ "three point one"
   ```

   (You can use the `round` and `truncate` functions to convert real numbers to integers.)

4. We have provided an instance of `Pronounceable` which can be used to pronounce lists of `Pronounceable` things; add an instance that work on tuples `(a,b)` when both `a` and `b` are `Pronounceable`.

Exercise 6.2 (*Warm-up:*, Monoids, `Monoids.hs`). A type `a` can be an instance of `Monoid` if we can find proper definitions for the monoid operator and an identity element:

```
(<>) :: a → a → a
mempty :: a
```

Which should satisfy the following equalities (or 'laws') for all objects `x`, `y`, `z`:

```
left-identity:  mempty <> x = x
right-identity: x <> mempty = x
associativity:  x <> (y <> z) = (x <> y) <> z
```

For each of the following functions, investigate if they can be used as a monoid operation `<>`. If not, explain why not; otherwise, give the corresponding value that can be used as `mempty`.

1. The list 'cons' operator, `(:) :: a → [a] → [a]`
2. The boolean operator `(||) :: Bool → Bool → Bool`
3. The function `mod :: (Integral a) ⇒ a → a → a`.
4. The function `max :: (Ord a) ⇒ a → a → a`.
5. The partially applied function `zipWith (+)`, which has the type `(Num a) ⇒ [a] → [a] → [a]`
6. Function composition, `(.) :: (b → c) → (a → b) → (a → c)`
7. The 'list mingling' operator `++/` from Exercise 3.4:

   ```
   (++/) :: [a] → [a] → [a]
   []     ++/ ys = ys
   (x:xs) ++/ ys = x:(ys ++/ xs)
   ```

Exercise 6.3 (*Warm-up*: Alternative list monoids, `OrdList.hs`).

1. The type of lists, `[a]`, forms a monoid with `mempty = []`, and `(<>) = (++)`. The type of *ordered* lists (i.e. whose elements are sorted) also forms a monoid: what would be useful choices for `mempty` and `(<>)`?
2. Declare a distinct type for ordered lists, `newtype OrdList a = ...`. The intent is that programs *never* produce a value of `OrdList a` where the list contained in it is not ordered. (See Hint 2 on how to turn this into a strong guarantee in real-world code.)
3. Having a distinct type allows creating an instance of `Monoid` for *ordered lists*, which has the proper versions of `<>` and `mempty`: Create instances of `Semigroup` and `Monoid` to define them:

   ```
   instance Semigroup (OrdList a) where
     ...
   instance Monoid (OrdList a) where
     ...
   ```

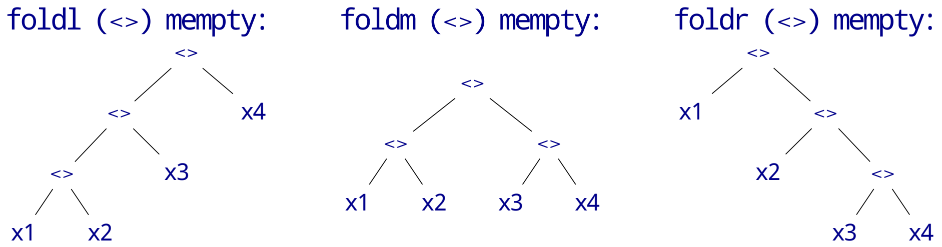
   `Semigroup` is a super-class for `Monoid` and actually declares the associative operator `<>`, which `Monoid` inherits and extends with the identity `mempty`. For background on the distinction `Semigroup`/`Monoid`, see Hint 1.

Exercise 6.4 (*Warm-up*: Execution strategies, `MapReduce.hs`). In the lecture we have defined `reduce` as the `Monoid`-concatenation function `mconcat`, which in turn is defined as `foldr (<>) mempty`.

But we could have chosen differently. Due to the monoid laws, the result does not depend on how nested applications of '`<>`' are parenthesized. So in theory, `reduce = foldl (<>) mempty` works as well. However, there can be a vast difference in execution time. For many applications (especially if we can use parallel computation), a balanced "expression tree" is preferable. To achieve this, we are going to define a higher-order function:

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

that constructs and evaluates a balanced expression tree like so: for the list `[x1,x2,x3,x4]`,



1. Implement `foldm` using a *top-down* approach: Split the input list into two halves, evaluate each half separately, and finally combine the results using the monoid operation (*divide and conquer*), in the same vein as Exercise 4.5.2.

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

3. To compare evaluation strategies, use the provided `test` function, which reduces pseudo-random elements using a monoid and folding function of your choice. E.g., `test Sum foldl 50000` will sum 50000 integers; `test (\x→[x]) foldr 50000` will create a list of 50000 integers; and `test (\x→OrdList[x]) foldm 50000` will perform a merge sort (using Exercise 6.3)

   What is the best folding strategy for each of these three monoids (`Sum`, lists, ordered lists)? (Use the GHCi command `:set +s` to get timing info, and you can use `bench` instead of `test` to suppress output for bigger benchmarks; also see Hint 3 for a word of caution.)

Exercise 6.5 (*Mandatory:* Monoids, `BoolMonoids.hs`). Even though we mostly think of `&&` and `||`, there are sixteen functions of type `Bool → Bool → Bool`. Only 8 of these are *associative*: for example, the NAND function (`nand x y = not (x && y)`) is not associative. Furthermore, not all associative functions have an identity: for example, the function `const x y = x` will never satisfy the right-identity law.

But the `&&` operator, for instance, is associative, and has `True` as a left- and right-identity, so `Bool` can be turned into a monoid using `&&`.

1. There are exactly four functions that can be used to make `Bool` an instance of `Monoid`; find all of them, show that they satisfy the 'monoid laws', and properly define distinct `Semigroup` and `Monoid` instances using `newtype` declarations. Come up with good names.

   (Again, see Hint 1 and Exercise 6.3 if the presence of `Semigroup` confuses you.)

2. For each of the Boolean monoids, what is the meaning of `mconcat`?

Exercise 6.6 (*Mandatory:*, Function specialization, `DigitalSorting.hs`). Most sorting algorithms that you know will be based on a *comparison function* between objects, and the better ones will have a computational complexity of $O(n \log n)$ comparisons, with n the number of keys to be sorted, and you may even have been told that that is the best we can do.

But this is only partly true. *Comparison-based sorting* treats objects as black boxes: the only source of information being a function that compares black boxes, two at a time. After all, a sorting algorithm must be *generic*: it has to work equally well on numbers, strings, playing cards...

But if you are sorting a deck of cards, you will probably find that you are not using an approach solely based on comparing two cards at a time. Maybe you first sort all the suits in separate piles, and then sort each pile in turn, i.e. something like:

```
data Suit  = Clubs | Diamonds | Hearts | Spades      deriving (Eq,Ord)
data Value = Ace | Numeral Int | Jack | Queen | King deriving (Eq,Ord)
data Card  = Card { suit :: Suit, value :: Value }   deriving (Eq,Ord)

sortCards :: [Card] → [Card]
sortCards = concat . map (sortOn value) . groupBy (\x y→suit x == suit y) . sortOn suit
```

And you can do this since playing cards are *not* black boxes. In this exercise we will explore how *type classes* can be used to create a similar *generic* sorting algorithm, that nonetheless can achieve better complexity by inspecting the structure of objects.

To do this, we will define a function that takes an *association list* of keys and values `[(key,a)]`, and sorts the values based on the provided keys in a way that allows further processing. The trick that achieves this can already be glimpsed in `sortCards`: the composition of `groupBy` and `sortOn` takes a list of cards, and produces a sorted *list-of-lists of cards*, where all cards that have the same suit are collected together. We will generalize this 'card trick' to the concept of a *ranking function*, which computes a list that has values with identical keys grouped together, and has those groups sorted with respect to each other:

```
class Rankable key where
  rank :: [(key,a)] → [[a]]
```

For example, if the types `Suit` and `Value` are instances of `Rankable`, ranking a hand of cards based on their face value or suit (assuming the order ♣ < ♦ < ♥ < ♠), should look like this:

```
  ⋙ rank [ (value card, card) | card ← hand ]
[[♠A],[♠3,♣3],[♥7],[♥Q]]
  ⋙ rank [ (suit card, card)  | card ← hand ]
[[♣3],[♥Q,♥7],[♠3,♠A]]
```

Although the keys are not returned by `rank`, this doesn't mean that you necessary have to lose them (see below). Of course, to turn `rank` into a useful sorting function similar to `sortOn` requires a bit of pre- and post-processing:

```
digitalSortOn :: (Rankable key) ⇒ (a → key) → [a] → [a]
digitalSortOn f = concat . rank . map (\x→(f x, x))
```

If the values are keys themselves, we can dispense with the higher-order function:

```
digitalSort :: (Rankable a) ⇒ [a] → [a]
digitalSort = digitalSortOn id
```

By creating instances of the `Rankable` class for types, we gain the ability to sort based on that type. While we also get that ability from the `Ord` class, that class only gives us *comparison-based sorting*, whereas `Rankable` instances allows more efficient strategies.

1. Before creating instances of `Rankable`, first create a 'reference' ranking function based on the `Ord` class, which will also serve as a fallback algorithm. So, write a function:

   ```
   genericRank :: (Ord key) ⇒ [(key,a)] → [[a]]
   ```

   which ranks values based on the provided key using a comparison-based approach. To be more precise, the function `genericRank` should produce a list of lists such that:

   - values end up in the same list *if and only if* they had identical keys; and
   - the lists themselves are presented in order, based on the key their values had.

   You can use the definitions in `Deck.hs` (such as `shuffledDeck`) to test your function on the playing card example.

2. Create instances of `Rankable` for the types `Int`, `Integer`, and `Char` which use `genericRank`.

3. Create an instance of `Rankable` for the type `Bool`. Since there only two possible keys (`False` and `True`), a ranking can be produced by going over the list and collecting all values associated with the key `False`, and all the values associated with the key `True`. If this is done in one pass it is called a *distribution sort* or *bucket sort*.

4. When tuples `(a,b)` are compared using operations from the `Ord` class, a *lexicographical* ordering is used: the first element takes precedence over the second, which only is taken into account when the first elements are equivalent.

   Create an instance of `Rankable` for tuples `(a,b)` which ranks them in lexicographical order, so start with:

   ```
   instance (Rankable key1, Rankable key2) ⇒ Rankable (key1, key2) where
     rank = ...
   ```

   As a reminder, the type of `rank` for this instance will be:

   ```
     rank :: (Rankable key1, Rankable key2) ⇒ [((key1,key2),a)] → [[a]]
   ```

   Tip: a helper function `assoc :: ((k1,k2),a) → (k1,(k2,a))` will be useful. You can tell if your instance is correct by comparing its output with `genericRank`.

5. Create an instance of `Rankable` for `Maybe key`. As a reminder, the rank function will have the signature:

   ```
     rank :: (Rankable key) ⇒ [(Maybe key,a)] → [[a]]
   ```

   Elements that do not have a key (i.e., their key is `Nothing`) should be ranked before elements that do, that are then ranked based on the value inside the `Maybe key`.

6. Like tuples, `String`s (and more generally, lists) should also be ranked by lexicographical order.

   Create an instance of `Rankable` for lists (i.e. `[key]`). The Haskell module `Data.List` contains a function `uncons :: [a] → Maybe (a, [a])` which can be useful.

7. Define a function `rankWithKey :: (Rankable key) ⇒ [(key,a)] → [[(key,a)]]` which gives the same ranking as `rank`, but doesn't discard keys.

8. *Optional:* Another common type is `Either a b`, which can hold values of type `a` and `b` using the constructors `Left` or `Right`. Create an instance of `Rankable` for `Either a b`.

9. *Optional:* In `Deck.hs`, create instances of `Rankable` for types `Suit`, `Value`, and `Card`; in the first two cases, a bucket sort is logical. Yes, this is overkill for sorting cards! But perhaps you can think of other data that consists of (strings of) four elements.

You may be wondering if it is possible to always use the `genericRank` function if there is no *explicit* instance provided of `Rankable`. The answer is: *only by turning on several GHC extensions*, and you probably shouldn't. See Hint 4.

Exercise 6.7 (*Extra*: Derived instances of standard type classes, `MyList.hs`).
In Haskell, if you write your own algebraic data type, you can get a lot of operations such as comparisons and a pretty printer for free:

```
data MyList a = a :# MyList a | Null
  deriving (Eq,Ord,Show)
```

However, Sometimes these 'free operations' may not be what you want them to be.

1. First of all, write functions `fromList :: [a] → MyList a` and `toList :: MyList a → [a]` which convert between `MyList` and Haskell lists. We will need these for this exercise.

2. Since we derived `Ord`, we get comparisons for free. For example, we can ask to compute `fromList [1,2,3] ≤ fromList [4,5,6]`, which will be `True`. However, the ordering that `≤` gives has something odd. Find two lists `x` and `y` such that `x ≤ y`, but for which it does not hold that `fromList x ≤ fromList y`. What do you think causes this? *(Note: compare the definition of `MyList a` with the one for `List a` on the slides of week 4)*

3. If you type `fromList [1,2,3]` in GHCi, it outputs the raw `MyList` syntax at you. This suffices at first, but it's not very nice; after all for normal lists we get the much nicer `show [1,2,3] ⟹ "[1,2,3]"`.

   Create an instance of the `Show` type class for `MyList a` so that:

   ```
   show(fromList [1,2,3]) ⟹ "fromList [1,2,3]"
   ```

   Obviously, this only works if the type `a` is also an instance of `Show`.

Hints to practitioners 1. Haskell is a language that is still evolving; and that means that things change over time. Originally, the `Monoid` class defined both a monoid operation `mappend` and identity element `mempty`. At some point, it was evidently discovered that it is also useful to generalize over data types that do have an associative operation (like `Monoid`), but don't have identity; thus `Semigroup` was born.

Since 2018, the logical choice was made that `Semigroup` should become a super-class of `Monoid`, since every monoid is also a semigroup. This does have the downside that we have to create instances of `Semigroup` every time we want to create an instance of `Monoid`.

Hints to practitioners 2. In Haskell, information hiding is done through using *abstract data types*. An example is an associative map `Map k v`: its internal representation is hidden (probably a balanced binary search tree!), and you can only manipulate it through the provided functions in the `Data.Map` module.

Information can be hidden by controlling what is *exported* from a module. If you just write:

```
module OrderedList where
import Data.List

newtype OrdList a = OList [a]

fromList :: (Ord a) ⇒ [a] → OrdList a
fromList xs = OList (sort xs)

addElem :: (Ord a) ⇒ a → OrdList a → OrdList a
addElem x (OList xs) = OList (insert x xs)
```

Then every module that import `OrderedList` can easily create values of `OrdList a` that are *not* ordered lists. This is because everything defined in `OrderedList` is exported by default. We can restrict this by carefully exporting only the functions that establish or maintain the desired guarantee, by changing the first line to:

```
module OrderedList (OrdList, fromList, addElem) where
```

This exports the type `OrdList`, and the safe functions `fromList` and `addElem`, but *not* the data constructor `OList`. So any other module that imports `OrderedList` has no choice but to create values of `OrdList a` through the provided functions. This achieves a similar result as making class members private in C++. Also see: https://wiki.haskell.org/Abstract_data_type.

Hints to practitioners 3. Testing Haskell programs for efficiency presents two challenges.

First of all, lazy evaluation means that you have to make sure that the function you want to benchmark is actually evaluated; otherwise it will never be run, making the test pointless. The precise effects of laziness (and how to occasionally enforce strict evaluation) will be the topic of a future lecture.

Second, GHCi is an *interpreter*, and doesn't optimize your code; to get efficient code requires *compiling* your code using `ghc` with optimizations turned on. It is quite possible that a program that eats all your available memory resources in GHCi is extremely efficient when compiled with `ghc -O3`.

So, while `:set +s` is nice, it is not a very reliable benchmark.

Hints to practitioners 4. You may guess that a 'default instance' of `Rankable` can be defined as follows.

```
instance (Ord a) ⇒ Rankable a where
  rank = genericRank
```

And if we ask GHC nicely enough, it will in fact allow this. But it does require turning on several extensions, some of which introduce problems of their own.

1. In standard Haskell, type instances can only be defined for a type definition. We can create an instance `Rankable (Tree a)`, but not `Rankable a`, and neither an instance `Rankable (Tree Char)`. However, this restriction can be lifted by enabling the `FlexibleInstances` language extension, by putting the following *directive* at the start of your file:

   ```
   {-# LANGUAGE FlexibleInstances #-}
   ```

   This extension is very common, and not a cause for concern.

2. If we add an instance of `Rankable a`, that means that for many types, there are now two instances available. For example, for `Maybe a` Haskell can choose the specialized instance, or the 'default' one. To state explicitly that this is intentional, the 'default' implementation has to be marked as OVERLAPPABLE, using a *pragma*:

   ```
   instance {-# OVERLAPPABLE #-} (Ord a) ⇒ Rankable a where
     rank = genericRank
   ```

   This starts being a bit uncomfortable: essentially we are introducing an ambiguity in our program and trusting that GHC will always resolve it in a proper way.

3. GHCi will also complain that the extension `UndecidableInstances` needs to be enabled. This is because this 'catch all' instance can lead to unintended loops. For example, suppose someone else comes along and also adds:

   ```
   instance (Eq a, Rankable a) ⇒ Ord a where
     x ≤ y = rank [(x,True),(y,False)] /= [[False],[True]]
   ```

   Now, you are always one innocuous mistake away from ending up in a infinite loop. Since in this case, if a type is neither an explicit instance of `Ord` or `Rankable`, GHC will happily try to compute a comparison on that type by endlessly cycling through the two instances above. I.e. the comparison will call `rank`, which uses comparisons, that will call `rank`, which uses comparisons... If you are really unlucky, GHC itself might even end up in an infinite loop during *type checking*.

   So, this extension can be quite dangerous, and should be use very thoughtfully—not just because GHC tells you to.