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:* Standard type classes, `Nat.hs`).
In mathematics (to be precise, in the *Peano axioms*) the natural numbers are defined as follows:

- The number 0 is a natural number.
- Every natural number has a successor that is a natural number.
- Distinct numbers have distinct successors, 0 is never a successor.

In Haskell, we can capture this using an algebraic data type:

```haskell
data Nat = O | S Nat
```

The value `O` corresponds to the number 0, and the constructor function `S` corresponds to the increment function. So for example, the number 3 can be represented as `S (S (S O))`.

1. Create the overloaded function `fromNat :: (Num t) ⇒ Nat → t` to convert `Nat` to a non-negative number, and a function `toNat :: (Ord t, Num t) ⇒ t → Nat` to convert it back again. *(Tip: if this sounds scary, recall Exercise 2.3.)*

   You may assume that the input to `toNat` is a natural number.

2. Provide instances for the type classes `Eq` and `Ord` for `Nat`, so we can compare `Nat` objects. **Do not use `deriving` at this point**: this *is* an exercise!

   Recall that for the `Eq` class instance, you need to give a definition of the `(==)` operator, and for `Ord` you need to give a definition of `(<=)`.

3. In Haskell, there is the `Enum` class[1]. If we make `Nat` an instance of `Enum`, we can use this type in list comprehensions, and use expressions such as `[O, S (S O) ..]`. Create this instance, giving definitions for four functions declared in the `Enum` class:

   - `succ :: Nat → Nat`, giving the successor of an object (3 is the successor of 2).
   - `pred :: Nat → Nat`, giving the predecessor of an object, if it exists.
   - `toEnum :: Int → Nat`, which converts an `Int` to a `Nat`.
   - `fromEnum :: Nat → Int`, which converts a `Nat` to an `Int`.

   (Do not worry about the possibility that `Int` may overflow.)

4. If you comment out your instances of `Eq` and `Ord`, and instead write:

   ```haskell
   data Nat = O | S Nat
     deriving (Show,Eq,Ord)
   ```

   Does the functionality of the `(==)` and `(<=)` operators change? If your instances were correct, it should not—do some manual testing! How do you think the `derived` instances work?

5. Change the data type definition to:

   ```haskell
   data Nat = S Nat | O
     deriving (Show,Eq,Ord)
   ```

[1] Documented in https://hackage.haskell.org/package/base-4.18.0.0/docs/GHC-Enum.html#t:Enum

Does this change the functionality of the `derived` `(==)` and `(<=)` operators? Do some manual tests and explain any differences you find.

Exercise 6.3 (*Warm up:* Working with functors, `FMapExpr.hs`).
While the *name* 'functor' sounds very abstract and mathematical, it essentially just embodies an operation that you have been using since Exercise 1.5: namely, that of `map` to 'lift' an operation to a different type. Except that it is now called `fmap`. So, whenever you see `Functor`, think: 'the type class that allows you to use `fmap`'.

Now, consider the following expressions:

```
fmap (\x→x+1) [1,2,3]

fmap ("dr." ++) (Just "Sjaak")

fmap toLower "Marc Schoolderman"

fmap (fmap ("dr." ++)) [Nothing, Just "Marc", Just "Twan"]
```

For each of these expression:

- Describe what they compute.
- Determine the `Functor` instance used for each `fmap` occurrence.
- Determine the *type* of each `fmap` occurrence *(Note: this will be completely determined by your answer to the previous point.)*

Check your answers using GHCi! (See Hint 5 on checking your answer for the last two questions.)

Exercise 6.4 (*Warm-up*: Implementing your own functor, `TreeMap.hs`).
In Exercise 4.3, we introduced binary trees as follows:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

and we said: "Like `[a]`, the type of a list, `Tree a` is a polymorphic type: it stores elements of type `a`. Thus, we can have a tree *of* strings, a tree *of* integers, a tree *of* lists of things, and even trees *of* trees ...". I.e., `[]` and `Tree` are of the same *kind*.

Of course, we have `map` on lists, and so it is not unreasonable to also want that operation for *binary search trees* as well (it was already defined for `Btree`, the type of *leaf trees*, in the lecture). We are going to make a new instance of `Functor` for this.

1. Create an instance of `Functor` for the kind of `Tree`. You can define `fmap` directly using the recursive design pattern for `Tree`. *(In particular, you do not have to define a function* `mapTree` *first!)* What is the type of `fmap` here?

   Remember the boiler plate for writing instances:

   ```
   instance Functor Tree where
     -- fmap :: ???
     fmap f Leaf          = ...
     fmap f (Node x lt rt) = ...
   ```

2. Test your `fmap` instance on some example trees. For example `fmap (+1) (fromAscList [1,2,3])` should produce the same tree as `fromAscList [2,3,4]`.

3. `fmap` applied to a *binary search tree* (as defined in Exercise 4.3) is not guaranteed to result in a binary search tree. Try to find a binary search tree `tree` and lambda-expression so that the result of `fmap (\x→...) tree` is no longer a binary search tree. What additional requirement should hold for a function `f` to make sure that `fmap f` *does* preserve the requirements for binary search trees? Discuss whether you think the `Functor` instance for `Tree` is a good idea.

Exercise 6.5 (*Mandatory*: Truthy values, `Truthy.hs`). Some people believe that `0` means the same thing as `False`, and that any non-zero integer means the same thing as `True`. Sometimes these people even design programming languages.

However, Haskell is a strictly typed language, and the only things that can be `True` or `False` are booleans. So you can not write `if x - 1 then ...` when you mean `if x - 1 == 0 then ...`.

To accommodate people with a yearning for untyped programming, we can, however, define a typeclass for values that can (un)reasonably be considered as 'truthy'. Then we can write `if truthy (x - 1) then ...`

1. Define a typeclass `Truthy` with a single function `truthy` that converts its argument to a boolean. The intent is that the function `truthy` returns `True` if its argument is *truthy*, and `False` if the argument is *falsy* (meaning not-*truthy*).

2. Define an instance of `Truthy` for `Bool` (in the obvious way), and for `Integer` (where 0 is *falsy* and everything else is *truthy*).

3. Define a new datatype `Nope` with a single constructor of the same name, that is always *falsy*.

4. Make an instance of `Truthy` for pairs, so that a pair is *falsy* iff all its elements are *falsy*.

5. Define operators `(&&&)` and `(|||)` that behave like *and* and *or* for truthy types. For instance

   ```
   > 1 + 1 &&& 0
   False
   > Nope ||| 42
   True
   ```

   Note: you should include the following infix declarations, otherwise the first line would be interpreted as `1 + (1 &&& 0)`:

   ```
   infixr 3 &&&
   infixr 2 |||
   ```

6. Make a function `ifThenElse` such that `ifThenElse x y z == y` if `x` is *truthy*, and `ifThenElse x y z == z` if `x` is *falsy*.

Exercise 6.6 (*Mandatory:* Json encoding and decoding `Json.hs`). Json is a file format that is commonly used to encode arbitrary objects. It is based on Javascript syntax, but it can be used from any other language as well.

In this exercise we will implement a type class to encode Haskell values into Json, and another type class for decoding Json back to Haskell values. For example, we could encode a Haskell value like

```
Person {name="Twan", age=38, knowsFP=True}
```

as a json file

```
{
  "name": "Twan",
  "age": 38.0,
  "knowsFP": true
}
```

Instead of directly generating text, we will use a data type for `Json` values as an intermediate step:

```
data Json
  = JSNull                    -- null
  | JSFalse                   -- false
  | JSTrue                    -- true
  | JSNumber Double           -- numbers, 123.456
  | JSString String           -- strings, "hello"
  | JSList [Json]             -- lists,   [x,y,..]
  | JSObject [(String,Json)]  -- objects, {"k":v, "k2":v2, ..}
```

Note that this data type can encode booleans, numbers, strings, lists, and objects (which are represented as key/value pairs).

1. Create a class `ToJson` with a function `toJson`. The intention is that `toJson` takes a value of some type, and produces a `Json` representation of that value.

   For example, the unit value `()` could be represented as `JSNull`

   ```
   instance ToJson () where
     toJson () = JSNull
   ```

2. Create instances of `ToJson` for `Bool`, `Double`, and `String`.

3. Create an instance of `ToJson` for lists.

4. Create an instance of `ToJson` for pairs. You should encode pairs as Json lists with two elements.

5. Create an instance of `ToJson` for the data type.

   ```
   data Person = Person
     { name :: String
     , age :: Double
     , knowsFP :: Bool
     }
   ```

6. Uncomment `testToJson` at the bottom of the file, and use it to test your implementation.

In addition to encoding Haskell values into `Json`, we can do the opposite, and decode `Json` values into Haskell values of a specific type. In contrast to what we did above, decoding can fail. We can use `Maybe` to represent this possible failure.

7. Create a class `FromJson` for types that can be decoded from Json.

   Here is an example instance for this class:

   ```
   instance FromJson () where
     fromJson JSNull = Just ()
     fromJson _      = Nothing
   ```

8. Create instances of `FromJson` for `Bool`, `Double`, and `String`.

9. Create an instance of `FromJson` for pairs.

10. Create an instance of `FromJson` for `Person`.

11. Uncomment `testFromJson`, and use it to test your implementation.

12. *Optional:* Create an instance of `FromJson` for `Int`.

13. *Optional:* Create an instance of `FromJson` for lists.

Exercise 6.7 (*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. 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]]
```

Note that the item groups themselves do *not* have to be internally sorted by `rank`. Although the keys are not returned by `rank`, this doesn't mean that you necessary have to lose them (see below).

Of course, when we can *rank* items, we can also *sort* them. To turn `rank` into a useful sorting function similar to `sortOn` only 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
```

So, 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 allow 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 $O(n \log n)$ 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 are only two possible keys (`False` and `True`), a ranking can be produced in $O(n)$ time instead of $O(n \log n)$ by going over the list and collecting all values associated with the key `False`, and all the values associated with the key `True`. This 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 have the key `Nothing` should be ranked before elements that have a key `Just ...`, and the latter should be ranked based on the key inside the `Just` constructor.

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 key1 key2`.

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 6.8 (*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.

Hints to practitioners 5. As an example, in the first expression of Exercise 6.3:

```
fmap (\x→x+1) [1,2,3]
```

The `Functor` instance used is that for lists (`[]`), since the second argument to `fmap` is a list. And so, looking at the type of `fmap`:

```
(a → b) → f a → f b
```

and using `f = []`, we get:

```
(a → b) → [a] → [b]
```

(we can in fact also write `(a → b) → [] a → [] b`.)

We can check that this is correct by using a type annotation:

```
⋙ (fmap :: (a → b) → [a] → [b]) (\x→x+1) [1,2,3]
[2,3,4]
```

Of course, in this case `fmap` is simply the same as plain old `map`.