7 Loose ends

Exercise 7.1 (*Warm-up:* Emptiness, `Empty.hs`).
There are various ways to see whether a list is empty in Haskell:

- `length list == 0`, i.e. testing whether its length is 0.
- `list == []`, i.e. comparing it with 'the' empty list.
- `null list`, i.e. using the standard function `null :: [a] → Bool`.
- Use a well-aimed pattern match in a function definition or via `case` expressions, e.g.

```
case list of []     → something
             (x:xs) → something else
```

Two of the above methods are totally fine, where it depends on the situation which one is more convenient. But the other two are objectively worse. Which ones should you avoid, and why? Use what you know about types and how functions are evaluated in Haskell.

Exercise 7.2 (*Warm-up:* Type derivation, `PolyTypes.hs`). Derive the *most general type* of the following polymorphic functions. Also include any type class information, where needed.
Do this manually, using the type inference method shown in the lecture.

```
justs xs           = [ x | Just x ← xs, x /= ' ' ]

orderPairs xs      = map (\(x,y)→(min x y, max x y)) xs

unmaybe (Just x)   = x
unmaybe Nothing    = Nothing

accumulate f st    = let (x,st') = f st in x : accumulate f st'
```

Check your answers using GHCi! If you had a different answer, explain the difference.

Exercise 7.3 (*Warm-up:* Type instances, `FindDefs.hs`). Give **non-trivial** function definitions that match the following types:

```
mapFilter :: (a → Maybe b) → [a] → [b]

lift      :: (a → b → Maybe c) → (Maybe a → Maybe b → Maybe c)

compute   :: (Monoid n) ⇒ (a → n) → [a] → n

fuse      :: (a → b → c) → (a → b) → a → c
```

For the purpose of this exercise, a *trivial* function is one that always returns the same result no matter the input, that does not terminate, or that produces a run time error when evaluated.

Huffman encoding

The next few exercises below revolve around the implementation of a Huffman coding program, which provides a mechanism for compressing data. The purpose of the exercises is to allow you to apply what you have learned in “Functional Programming” so far to a slightly larger task. (However, even though the exercises share a common theme, they can be implemented independent of each other.)

The conventional representation of a binary data (or ASCII text) uses elements of a fixed size: e.g., 8-bit *bytes*, 64-bit integers, etc. A document containing 1024 ASCII characters will take up 8192 bits, and 1024 int64’s will take up 65536 bits.

The idea behind Huffman coding is to use the fact that some data points appear more frequently than others. Therefore, Huffman encoding moves away from a fixed-length encoding to a variable-length encoding in which the frequently used elements have a smaller bit encoding than rarer ones. As an example, consider the text

```
hello world
```

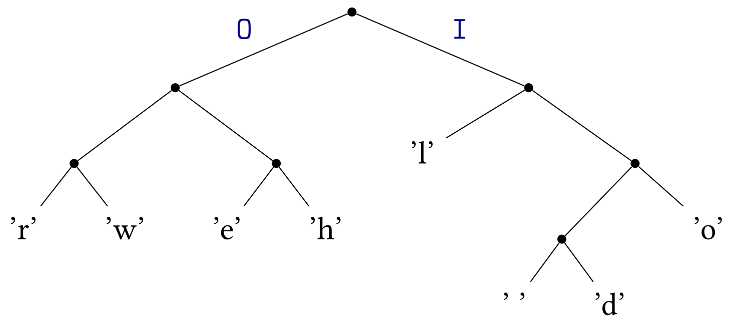
and note that the letter `l` occurs thrice. The table shown below gives a possible Huffman encoding for the eight letters:

`'r'`	`OOO`	`'e'`	`OIO`	`'l'`	`IO`	`'d'`	`IIOI`
`'w'`	`OOI`	`'h'`	`OII`	`' '`	`IIOO`	`'o'`	`III`

The more frequent a character, the shorter the code. Using this encoding the ASCII string above is Huffman encoded to the following data:

```
OIIOIOIOIOIIIIIOOOOIIIIOOOIOIIOI
```

It is important that the codes are chosen in such a way that an encoded file gives a unique decoding that is the same as the original one. The reason why the codes shown above are decipherable is that *no code is a prefix of any other code.* This property is easy to verify if the code table is converted to a code tree:



Each code corresponds to a path in the tree e.g. starting at the root the code `OII` guides us to `'h'` i.e. we walk left, right, and right again. Given a code tree and an encoded document, the original text can be decoded.

For a slightly larger example, consider the text shown in Table 1. The document is 487 characters long, but contains only 30 different characters (including the newline character `'\n'`). The shortest Huffman codes for this document are 3 bits long (for `' '` and `'e'`); the longest codes comprise 9 bits (for `'C'`, `'A'`, `'q'`, `'F'`, `'x'`, `'w'`). The Huffman-encoded document has a total

```
An alternative functional style of programming is founded on the
use of combining forms for creating programs. Functional programs
deal with structured data, are often nonrepetitive and nonrecursive,
are hierarchically constructed, do not name their arguments, and do
not require the complex machinery of procedure declarations to become
generally applicable. Combining forms can use high level programs to build
still higher level ones in a style not possible in conventional languages.
```

Table 1: Excerpt of *"Can programming be liberated from the von Neumann style?"*, by John Backus (1978)

length of 2096 bits. As the original text is $8 \times 487 = 3896$ bits, we obtain a compression factor of almost 2. (Ignoring the fact that we also need to store the code tree along the compressed data.)

Strange results are obtained if the encoded document contains just one character, repeated many times—for which the Huffman tree consists only of a single leaf, and the path describing it is empty. Therefore, we assume that the to-be-encoded text contains at least *two* different characters.

Exercise 7.4 (*Mandatory:* constructing a frequency table, `Huffman.hs`). Define a function

```
frequencies :: (Ord a) ⇒ [a] → [(a,Int)]
```

that constructs a frequency table, a list of frequency-character pairs, that records the number of occurrences of each element in a list. E.g.,

```
⋙ frequencies "hello world"
[(' ',1),('d',1),('e',1),('h',1),('l',3),('o',2),('r',1),('w',1)]
```

Tip: go ahead and re-use *anything you want* from Exercise 5.8. It is not cheating, software engineering is often software *re*-engineering!

Exercise 7.5 (*Mandatory*: Constructing a Huffman tree, `Huffman.hs`). A Huffman tree or simply a code tree is an instance of a *leaf tree*, in which all data is held in the leaves instead of at the branches. Such a tree can be defined by the datatype declaration: (*different than Exercise 4.3!*)

```
data Btree elem = Tip elem | Bin (Btree elem) (Btree elem)
```

The algorithm for constructing a Huffman tree works as follows:

- Convert a list of frequency-character pairs into a list of frequency-tree pairs, mapping each character to a `Tip`;
- Sort the list of frequency-tree pairs on the *frequency* part of the pair, so that less frequent characters will be at the front of the sorted list;
- Take the first two pairs off the list, add the frequencies and combine the trees to form a `Bin` branch; insert these as a pair into the remaining list of pairs in such a way that the resulting list is still sorted on the frequency part;
- Repeat the previous step until a singleton list remains, which contains the Huffman tree for the character-frequency pairs.

Implement the above algorithm in a function

```
huffman :: [(a,Int)] → Btree a
```

that constructs a code tree from a frequency table. For example,

```
 ⋙ huffman (frequencies "hello world")
Bin (Bin (Bin (Tip 'r') (Tip 'w'))
         (Bin (Tip 'e') (Tip 'h')))
    (Bin (Tip 'l')
         (Bin (Bin (Tip ' ') (Tip 'd'))
              (Tip 'o')))
```

yields the Huffman tree shown on page 2. You can use the provided putTree function to inspect the tree graphically. Some words of advice:

- How to maintain the ordering of the list of frequency-tree pairs is up to you. In Data.List, there are useful higher order functions for working with ordered lists using user-provided comparison; but you can also choose to represent frequency-tree pairs using a custom data type introduced via data/newtype, that you overload the Ord type class for. Something like:

  ```
  data WithFreq a = WithFreq { freq :: Int, value :: a } deriving Show
  instance Ord (WithFreq a) where
    (≤) x y = freq x ≤ freq y
  ```

  will allow the use of ordinary sort, etc. This will also require you to provide an instance for the super-class of Ord: Eq, which defines the (==) operator. Use the approach you prefer.

- To test your function, we provided a set of test data that tests the intermediate stages for constructing the Huffman tree for "hello world". See the function testHuffman.

- *(Extra, optional)* The description of the algorithm above involves the use of an ordered list to essentially implement a *priority queue*: the central step of the algorithm involves extracting two pairs with minimum frequency from this queue and inserting a freshly created pair.

  Specialized data types exists that make this easy and efficient exist in the pqueue library, that **you are allowed to install** for this exercise if you want. This is a more adventurous option, but not necessarily more difficult.

  For the documentation of this library see: https://hackage.haskell.org/package/pqueue. Using either Data.PQueue.Prio.Min or Data.PQueue.Min can be a good choice.

Exercise 7.6 (*Mandatory:* Encoding ASCII text, Huffman.hs). It is now possible to Huffman-encode a text document consisting of a list of characters. Write a function

```
data Bit = O | I
encode :: (Ord a) ⇒ Btree a → [a] → [Bit]
```

that, given a code tree, converts the list of characters into a sequence of bits representing the Huffman coding of the document. For example,

```
 ⋙ ct = huffman (frequencies "hello world")
 ⋙ encode ct "hello world"
[O,I,I,O,I,O,I,O,I,O,I,I,I,I,I,O,O,O,O,I,I,I,I,O,O,O,I,O,I,I,O,I]
```

Again, some tips:

- It is useful to first implement a function such as

  ```
  codes :: Btree a → [(a, [Bit])]
  ```

  that creates a *association list* between characters and code sequences from the code tree. This simplifies the task of mapping each character to its corresponding Huffman code. E.g.,

  ```
   ⋙ codes ct
  [('r',[O,O,O]),('w',[O,O,I]),('e',[O,I,O]),('h',[O,I,I]),
     ('l',[I,O]),(' ',[I,I,O,O]),('d',[I,I,O,I]),('o',[I,I,I])]
  ```

  As a neat trick, since we know that codes with a shorter code sequences are more frequent, sorting this association list in order of the length of the code sequence will give a little efficiency boost.

- An association list, as shown above, is a very crude implementation of a an *associative map*! So, another (and slightly better) solution is to use a `Map a [Bit]` instead of the association list `[(a, [Bit])]`, which allows using the lookup routines from `Data.Map`: https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html. Remember that `Data.Map` should be imported using the `qualified` keyword:

  ```
  import qualified Data.Map as M
  ```

- Testing this function is a bit hard until you also do the next exercise. We provided a correct Huffman tree for "hello world", so being able to reproduce the above result is a good sign. It is easier to test your `codes` function—create a few small Huffman trees and check if you get a code table that you expect.

Exercise 7.7 (*Mandatory*: Decoding a Huffman binary, `Huffman.hs`). Finally, write a function

```
decode :: Btree a → [Bit] → [a]
```

that, given a code tree, converts a Huffman-encoded document back to the original list of characters. For example,

```
 ⋙ ct = huffman (frequencies "hello world")
 ⋙ encode ct "hello world"
[O,I,I,O,I,O,I,O,I,O,I,I,I,I,I,O,O,O,O,I,I,I,I,O,O,O,I,O,I,I,O,I]
 ⋙ decode ct it
"hello world"
```

Test your function on appropriate test data. In general, decoding is the inverse of encoding: `decode ct . encode ct = id`. Use this relationship to test your functions thoroughly. Start with the small samples we provided. For a realistic test, you can load external files as a `String` in GHCi using this syntax:

```
   ⋙ str ← readFile "FILENAME.TXT"
```

I.e. this reads the contents of the named file into the `String` variable `str`. (Yes, that's the left-arrow, "`<-`", not an equals-sign.)

Exercise 7.8 (*Extra:* Generic programming, `DigitalSorting.hs`).
In Exercise 6.6, you had to implement a generic ranking algorithm by writing the following type class, and providing specialized instances for `Bool`, `Maybe`, tuples, and lists:

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

By using *generic programming*, we can in fact get a specialized instance of the `Rankable` type class for most types, by simply implementing it for some primitive types, `Either`, tuples, and the *unit type*, `()`. Having those, the only thing needed to make an instance of `Rankable` is to convert a type to the *generic representation*, which is a purely mechanical process as shown in the lecture.

1. Due to Exercise 6.6, we already have instances for primitive types `Int`, `Integer`, `Char`, and tuples. If you haven't done so already, complete it by adding an instance for `Either key1 key2`:

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

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

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

2. Create an instance for the *unit type*, `()`. Note that since there is only one possible key of that type—which predictably is denoted as `()`—the ranking this produces is going to be very unimpressive: `rank [((), "a"), ((), "b")] ⟹ [["a", "b"]]`. The type of `rank` for this instance will be:

   ```
   rank :: [((),a)] → [[a]]
   ```

   Tip: it is up to you to decide if `rank []` should produce `[]` or `[[]]`. More generally, you will have to decide if you want empty lists to appear in a ranking. You should be consistent: if your instance of `Either` doesn't produce empty lists, than neither do so here. If you *do* sometimes produce empty lists, then you should also make sure that your ranking functions (which can call each other recursively) will work with them.

3. Using the techniques shown in the lecture, create a type `MAYBE a` that is the generic type representation of `Maybe a`, but expressed **only** using `(,)`, `Either` and the unit type `()`. Also write a function `toMAYBE :: Maybe a → MAYBE a` that converts values. Replace your instance of `Maybe a` from Exercise 6.6, part 5, by this:

   ```
   instance (Rankable key) ⇒ Rankable (Maybe key) where
     rank = rank . map (\(k,v)→(toMAYBE k,v))
   ```

4. Create a type `LIST a` that is a generic representation for lists `[a]`. Replace your instance of `Rankable` from Exercise 6.6, part 6 with an instance that uses it.

5. *Optional* You can also replace your instance of `Bool` by converting it to a generic representation `BOOL`. If you had a particular efficient implementation of bucket sort in Exercise 6.6, can you achieve the same efficiency for `BOOL`?

Exercise 7.9 (*Extra:* Type errors, `TypeErrors.hs`).
We promised in week 1 that you would see many type errors. Were we right or were we right? Even though they can be *really* cryptic, maybe you are actually getting a bit handy at recognizing what mistakes trigger them. To test your trouble-shooting skill, here is this strange exercise in analysing code that is "almost" correct, or in other words, *wrong*!

Explain why GHC complains about these definitions (use your knowledge of the type system). What might the original programmer have intended to write?

```
-- error!
maxlist [] = 0
maxlist xs = foldl max xs

-- error!
prepend xs [] = xs
prepend xs ys = xs : ys

-- error!
double f = f f
```

Usually, a good approach in trouble-shooting type errors is trying to declare a type for the expression or definition that you suspect may be involved in the error. E.g., the definition `maxlist` seems to suggest the type `[Int] → Int`, or, perhaps the more general `(Num a, Ord a) ⇒ [a] → a`.

While type errors can be terribly frustrating, the vast majority catch serious mistakes *early* that would be really hard to debug *later*. Hopefully you will in fact have observed that once you get rid of all type errors, your solutions are usually also correct. Sadly, this is not *always* true, but it is a reason why computer scientists like type systems so much! (I can assure you that none of my Haskell programs would ever work properly if GHC didn't type check them.)

If you like this exercise, maybe you should consider a job as teaching assistant!