Prof. Dr. Ralf Hinze
M.Sc. Sebastian Schweizer

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Functional Programming: Exercise 6

Sheet published: Wednesday, May 22nd
Submission deadline: Wednesday, June 5th, 12:00 noon

Note regarding the public holiday On May 30th, there is a public holiday. We won't have an exercise session on that day. This sheet therefore is for two weeks, the submission deadline is on Wednesday, June 5th, 12:00 noon and it will be discussed the day after. Since you have two weeks for this sheet, it is a bit longer than usual. Start early in order to not run out of time at the end.

Huffman coding The exercises below evolve around the implementation of a Huffman coding program, which provides a mechanism for compressing ASCII text. The purpose of the exercises is to allow you to apply what you have learned in "Functional Programming" to a slightly larger task. (However, even though the exercises share a common theme, they are fairly independent e.g. you can attempt the last exercise without having solved the earlier ones.)

The conventional representation of a textual document on a computer uses the ASCII character encoding scheme. The scheme uses seven or eight bits to represent a single character; a document containing 1024 characters will therefore occupy one kilobyte. The idea behind Huffman coding is to use the fact that some characters appear more frequently in a document. Therefore, Huffman encoding moves away from the fixed-length encoding of ASCII to a variable-length encoding in which the frequently used characters have a smaller bit encoding than rarer ones. As an example, consider the text

```
hello world
```

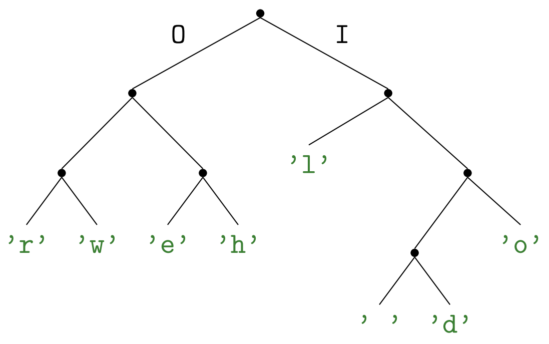
and note that the letter `l` occurs thrice. The table shown below gives a possible Huffman encoding for the eight letters:

'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:

```
OIIOIOIOIOIIIIOOOOIIIIOOOIOIIOI
```

It is important that the codes are chosen in such a way that an encoded document gives a unique decoding that is the same as the original document. 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 following text excerpt of “Why Functional Programming Matters” by J. Hughes:

```
As software becomes more and more complex, it
is  more  and  more important to structure it
well.  Well-structured  software  is  easy to
write,   easy   to   debug,  and  provides  a
collection  of modules that can be re-used to
reduce future programming costs. Conventional
languages place a conceptual limit on the way
problems   can   be   modularised.  Functional
languages  push  those  limits  back. In this
paper we show that two features of functional
languages    in    particular,    higher-order
functions and lazy evaluation, can contribute
greatly  to  modularity.  Since modularity is
the key to successful programming, functional
languages  are  vitally important to the real
world.
```

The document is 696 characters long, but contains only 35 different characters (including the newline character `'\n'`). The shortest Huffman code for this document is 3 bits long (for `' '`); the longest codes comprise 10 bits (for `'A'`, `'C'`, `'F'`, `'I'`, `'S'`, `'W'`, `'x'`, and `'z'`). The Huffman encoded document has a total length of 3019 bits. As the original text is $8 * 696 = 5568$ bits long, we obtain a compression factor of almost 2. (Of course, this is not quite true as in reality 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 6.1 (Constructing a frequency table)**.** Define a function

```
frequencies :: (Ord char) ⇒ [char] → [With Int char]
```

that constructs a frequency table (a list of frequency-character pairs) that records the number of occurrences of each character within a piece of text. For example,

```
⟫⟫ frequencies "hello world"
[1 :– ' ', 1 :– 'd', 1 :– 'e', 1 :– 'h', 3 :– 'l', 2 :– 'o', 1 :– 'r', 1 :– 'w']
```

The datatype *With a b* is similar to the pair type (a, b), but with a twist: when comparing two pairs only the first component is taken into account:

```
infix 1 :–
data With a b = a :– b
instance (Eq a) ⇒ Eq (With a b) where
  (a :– _) == (b :– _) = a == b
instance (Ord a) ⇒ Ord (With a b) where
  (a :– _) ⩽ (b :– _)   = a ⩽ b
```

We use *With a b* instead of (a, b) as this slightly simplifies the next step.

Exercise 6.2 (Constructing a Huffman tree)**.** A Huffman tree or simply a code tree is an instance of a *leaf tree*, a tree in which all data is held in the leaves. Such a tree can be defined by the datatype declaration

```
data Tree elem = Leaf elem | Tree elem :^: Tree elem
```

The algorithm for constructing a Huffman tree works as follows:

- sort the list of frequencies on the *frequency* part of the pair—i.e. less frequent characters will be at the front of the sorted list;
- convert the list of frequency-character pairs into a list of frequency-tree pairs, by mapping each character to a leaf;
- take the first two pairs off the list, add the frequencies and combine the trees to form a branch; insert this 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.

(Do you see why the special pair type *With Int char* is useful?)

For example, for the sorted list of frequencies

```
[1 :– ' ', 1 :– 'd', 1 :– 'e', 1 :– 'h', 1 :– 'r', 1 :– 'w', 2 :– 'o', 3 :– 'l']
```

the algorithm takes the following steps (*L* is shorthand for *Leaf*):

```
[1 :– L ' ', 1 :– L 'd', 1 :– L 'e', 1 :– L 'h', 1 :– L 'r', 1 :– L 'w', 2 :– L 'o', 3 :– L 'l']
[1 :– L 'e', 1 :– L 'h', 1 :– L 'r', 1 :– L 'w', 2 :– (L ' ' :^: L 'd'), 2 :– L 'o', 3 :– L 'l']
[1 :– L 'r', 1 :– L 'w', 2 :– (L 'e' :^: L 'h'), 2 :– (L ' ' :^: L 'd'), 2 :– L 'o', 3 :– L 'l']
[2 :– (L 'r' :^: L 'w'), 2 :– (L 'e' :^: L 'h'), 2 :– (L ' ' :^: L 'd'), 2 :– L 'o', 3 :– L 'l']
[2 :– (L ' ' :^: L 'd'), 2 :– L 'o', 3 :– L 'l', 4 :– ((L 'r' :^: L 'w') :^: (L 'e' :^: L 'h'))]
[3 :– L 'l', 4 :– ((L ' ' :^: L 'd') :^: L 'o'), 4 :– ((L 'r' :^: L 'w') :^: (L 'e' :^: L 'h'))]
[4 :– ((L 'r' :^: L 'w') :^: (L 'e' :^: L 'h')), 7 :– (L 'l' :^: ((L ' ' :^: L 'd') :^: L 'o'))]
[11 :– (((L 'r' :^: L 'w') :^: (L 'e' :^: L 'h')) :^: (L 'l' :^: ((L ' ' :^: L 'd') :^: L 'o')))]
```

First, the characters that occur only once are combined. The most frequent character, `'l'`, is considered only in the second, but last step, which is why it ends up high in the tree. The final tree is the Haskell rendering of the code tree shown in the introduction above.

a) Write a function

```
huffman :: [With Int char] → Tree char
```

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

```
⟫ huffman (frequencies "hello world")
((Leaf 'r' :^: Leaf 'w') :^: (Leaf 'e' :^: Leaf 'h'))
   :^: (Leaf 'l' :^: ((Leaf ' ' :^: Leaf 'd') :^: Leaf 'o'))
```

yields the Huffman tree shown above.

b) Apply the algorithm to the relative frequencies of letters in the English language, see for example `https://en.wikipedia.org/wiki/Letter_frequency`.

c) *Optional:* lists are the functional programmers favourite data structure but they are not always appropriate. The algorithm above uses an ordered list to maintain frequency-tree pairs. A moment's reflection reveals that the ordered list really serves as a *priority queue*, where priorities are given by frequencies. The central step of the algorithm involves extracting two pairs with minimum frequency and inserting a freshly created pair. Both of these operations are well supported by priority queues. If you feel energetic, implement a priority queue and use the implementation to replace the type of ordered lists.

Exercise 6.3 (Encoding ASCII text)**.** It is now possible to Huffman encode a document represented as a list of characters.

a) Write a function

```
data Bit = O | I
encode :: (Eq char) ⇒ Tree char → [char] → [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]
```

Test your function on appropriate test data, cutting and pasting the example evaluations into your file. *Hint:* it is useful to first implement a function

```
codes :: Tree char → [(char, [Bit])]
```

that creates a code table (a character-code list) from the given code tree. This greatly simplifies the task of mapping each character to its corresponding Huffman code. For example,

```
⟫ 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])]
```

b) *Optional:* lists are the functional programmers favourite data structure but they are not always appropriate. The function *codes* should really yield a *finite map* (aka dictionary or look-up table), which maps characters to codes. An association list (a list of key-value pairs) is a simple implementation of a finite map. Better implementations include balanced search trees or tries. If you feel energetic, implement a finite map and use the implementation to replace the type of association lists.

Exercise 6.4 (Decoding a Huffman binary)**.** Finally, write a function

$$decode :: Tree\ char \to [Bit] \to [char]$$

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"
```

Again, test your function on appropriate test data, cutting and pasting the example evaluations into your file. In general, decoding is the inverse of encoding: $decode\ ct \circ encode\ ct = id$. Use this relationship to test your program more thoroughly.