Functional Programming 2

Practicals

Ralf Hinze

0 Getting started

Like in "Functional Programming 1", we will be using GHCi for the practicals (https://www.haskell.org/ghc/). To run GHCi, simply open a terminal window and type 'ghci'. One typically uses a text editor to write or edit a Haskell script, saves that to disk, and loads it into GHCi. To load a script, it is helpful if you run GHCi from the directory containing the script. You can simply give the name of the script file as a parameter to the command ghci. Or, within GHCi, you can type ':load' (or just ':l') followed by the name of the script to load, and ':reload' (or just ':r') with no parameter to reload the file previously loaded.

There are practicals for each of the lectures; most of the practicals are programming exercises, but some can also be solved using pencil and paper. For some of the exercises there are skeletons of a solution to save you from having to type in what is provided to start with. The header of each exercise provides some guidance, e.g. "**Exercise 1.1** (Warm-up: constructing a frequency table, Huffman.lhs)" indicates that this is a warm-up exercise and that there is a source file available, called Huffman.lhs. The skeleton files, the slides of the lectures, and these instructions can be obtained via one of the following commands (you need to have Git installed, see https://git-scm.com/):

```
git clone git@gitlab.science.ru.nl:ralf/FP2.git
git clone https://gitlab.science.ru.nl/ralf/FP2.git
```

The Git repository will be updated on a regular basis. To obtain the latest updates, simply type `git pull` in the directory FP2. If you encounter any problems, please see the teaching assistants.

We will not introduce Haskell's module system in the lectures. However, for solving the exercises some basic knowledge is needed. Appendix A provides an overview of the most essential features.

Haskell and GHC fully support unicode, see www.unicode.org and https://wiki.haskell.org/Unicode-symbols. Both the lectures and the practicals make fairly intensive use of this feature e.g. we usually write '→' instead of '->'. Unicode support in the source code is turned on using a language pragma:

```
{-# LANGUAGE UnicodeSyntax #-}
module Database
where
import Unicode
```

The module *Unicode.lhs*, which can be found in the repository, defines a few obvious unicode bindings e.g. '∧' for conjunction '&&', '⩽' for ordering '<=', and '∘' for function composition '.'. Of course, the use of Unicode is strictly optional.

Have fun! Ralf Hinze

1 Recap: Functional Programming 1

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 1" 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
```

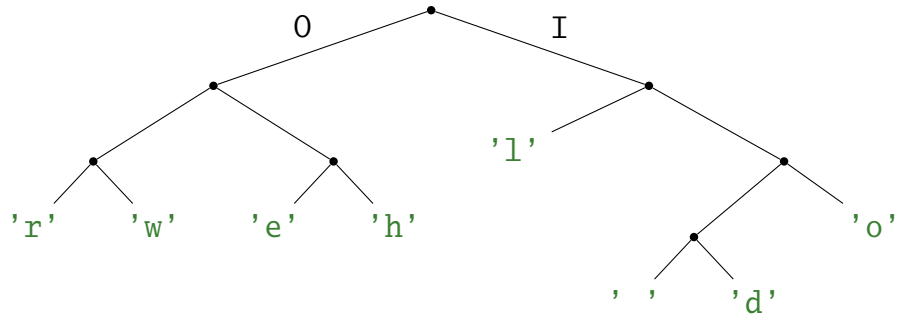
and note that the letter `l` occurs thrice. The table shown below gives a possible Huffman encoding for the eight letters:

`'r'`	`000`	`'e'`	`010`	`'l'`	`10`	`'d'`	`1101`
`'w'`	`001`	`'h'`	`011`	`' '`	`1100`	`'o'`	`111`

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

```
01101010101111100001111000101101
```

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:



```
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.
```

Table 1: Excerpt of “Why Functional Programming Matters” by J. Hughes

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 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 $8 \cdot 696 = 5568$ 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 1.1 (Warm-up: constructing a frequency table, `Huffman.lhs`)**.** Define a function

$$frequencies :: (Ord\ char) \Rightarrow [char] \rightarrow [With\ Int\ char]$$

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 1.2 (Constructing a Huffman tree, `Huffman.lhs`)**.** 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.

1. 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.

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

3. *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 1.3 (Encoding ASCII text, `Huffman.lhs`)**.** It is now possible to Huffman encode a document represented as a list of characters.

1. 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])]
   ```

2. *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 1.4 (Decoding a Huffman binary, `Huffman.lhs`)**.** Finally, write a function

```
decode :: Tree char → [Bit] → [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.

2 Lazy evaluation

Exercise 2.1 (Pencil and paper)**.** Recall the functional implementation of Insertion Sort from the very first lecture of "Functional Programming 1".

```
insertionSort :: (Ord a) ⇒ [a] → [a]
insertionSort [ ]       = [ ]
insertionSort (x : xs) = insert x (insertionSort xs)

insert :: (Ord a) ⇒ a → [a] → [a]
insert a [ ]      = [a]
insert a (b : xs)
  | a ⩽ b       = a : b : xs
  | otherwise = b : insert a xs
```

Harry Hacker proposes to implement the function *minimum* in terms of *insertionSort*:

```
minimum :: (Ord a) ⇒ [a] → a
minimum = head ∘ insertionSort
```

1. Evaluate the expression *minimum* $[2, 7, 1, 9, 6, 5]$ by hand—twice, first using the applicative-order evaluation strategy and then using the normal-order strategy. (There is no need to meticulously list all the steps; only show the major ones.)

2. What's the running-time of *minimum* in a strict language, one that uses applicative-order evaluation? What's the running-time in a lazy language such as Haskell?

Exercise 2.2 (Dynamic programming, `Chain.lhs`)**.** Recall that an operation, say, '$\cdot$' is associative iff $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. Associativity allows us to chain operations without introducing ambiguity e.g. $a \cdot b \cdot c \cdot d$. There are 5 different ways to parenthesize the expression, but thanks to associativity all of them are equal. However, while the final results are the same, the computations may differ widely in terms of costs: space and time consumption.

For example, recall that the running-time of list concatenation $+\!\!+$ is proportional to the length of its first argument. Thus, if a_i is a list of length i, then the computation on the left is almost twice as expensive as the one on the right:

$$\underset{1+3+6=10}{((a_1 +\!\!+ a_2) +\!\!+ a_3) +\!\!+ a_4} \qquad \underset{1+2+3=6}{a_1 +\!\!+ (a_2 +\!\!+ (a_3 +\!\!+ a_4))}$$

Another example is afforded by matrix multiplication.[1] The cost of multiplying an $i \times j$ matrix by an $j \times k$ matrix is roughly $i * j * k$. Say $A_{i,j}$ is a matrix of dimension $i \times j$, then there are two ways to compute the chain $A_{1,2} \times A_{2,3} \times A_{3,4}$:

$$\begin{array}{cc} (A_{1,2} \times A_{2,3}) \times A_{3,4} & A_{1,2} \times (A_{2,3} \times A_{3,4}) \\ 1 * 2 * 3 + 1 * 3 * 4 = 18 & 1 * 2 * 4 + 2 * 3 * 4 = 32 \end{array}$$

Clearly, the arrangement of the left is preferable.

How to compute the arrangement with minimum cost? To start with, we define a cost function, an *abstract* version of matrix multiplication that, given the dimensions of the argument matrices, computes the cost of the multiplication and the dimension of the resulting matrix (the type *With* was introduced in Exercise 1.1):

```
type Cost = Integer
type Dim = (Integer, Integer)
(×) :: Dim → Dim → With Cost Dim
(i, j) × (j', k) | j == j' = (i * j * k) :- (i, k)
```

We additionally "lift" the abstract operation to arguments annotated with costs, accumulating the various costs:

```
(⟨×⟩) :: With Cost Dim → With Cost Dim → With Cost Dim
(c1 :- d1) ⟨×⟩ (c2 :- d2) = (c1 + c + c2) :- d where c :- d = d1 × d2
```

To determine the minimum cost for chaining a sequence of matrices, represented by a non-empty list of dimensions, we take a brute-force approach:

```
minCost :: [Dim] → With Cost Dim
minCost [a] = 0 :- a
minCost as  = minimum [minCost bs ⟨×⟩ minCost cs | (bs, cs) ← split as]
```

where the helper function *split* returns all possible ways to partition the non-empty argument list into two non-empty lists:

```
split :: [a] → [([a], [a])]
split [a]     = []
split (a : as) = ([a], as) : [(a : bs, cs) | (bs, cs) ← split as]
```

[1] Strictly speaking, matrices do not form a monoid as matrix multiplication is partial: it is only defined if the dimensions of the argument matrices match. We gloss over this detail. One can turn matrix multiplication into a total operation if the dimension is integrated into the matrix type. Matrices then form what is known as a category, a typed version of monads.

To illustrate, here are some example calls:

```
⟩⟩⟩⟩ minCost [(i, i + 1) | i ← [1 .. 3]]
18 :– (1, 4)
⟩⟩⟩⟩ minCost [(i, i + 1) | i ← [1 .. 9]]
328 :– (1, 10)
⟩⟩⟩⟩ minCost [(10, 30), (30, 5), (5, 60)]
4500 :– (10, 60)
```

1. Abstract away from the specifics of matrix multiplication to solve the problem of optimal chains for arbitrary associative operations.

   ```
   minimumCost :: (size → size → With Cost size)
                  → [size] → With Cost size
   ```

   The generic version is parametrized by a cost function, an abstract version of the associative operation, that computes the cost for some notion of size. Test your implementation by instantiating the generic optimizer to the problem of matrix chain multiplication.

2. Define cost functions for other associative operations:

   (a) list concatenation: the size is given by the length of the list, the cost is proportional to the size of the first argument;

   (b) addition of infinite precision integers: the size is given by the number of digits, the cost is proportional to the maximum of the argument sizes. (Why?)

3. As it stands the generic optimizer is pretty useless: we only get to know the minimum cost, but not the associated expression tree. Augment the optimizer to also return the optimal expression tree (the type *Tree* was defined in Exercise 1.2):

   ```
   optimalChain :: (size → size → With Cost size)
                   → [size] → With Cost (With size (Tree size))
   ```

   For example, applied to the cost function for matrix multiplication, we obtain

   ```
   ⟩⟩⟩⟩ optimalChain (×) [(10, 30), (30, 5), (5, 60)]
   (4500 :– (10, 60)) :– ((Leaf (10, 30) :^: Leaf (30, 5)) :^: Leaf (5, 60))
   ```

   The expression tree shows that the cheapest evaluation first multiplies the matrices of dimensions $(10, 30)$ and $(30, 5)$ and then combines the result with the third matrix of dimension $(5, 60)$.

Hint: try to re-use as much as possible from the previous code. Perhaps you can define *optimalChain* in terms of *minimumCost*?

4. Test your implementation using the cost function for list concatenation. What do you observe? Why is $+\!\!+$ declared to be right associative: **infixr** $5\,+\!\!+$?

5. You may have noticed that the brute-force algorithm is rather slooow if applied to larger argument lists. The reason is simple: the number of possible arrangements grows exponentially. (You may remember that the number of binary trees of size n is given by the Catalan numbers.) There are, however, only a quadratic number of different sub-trees as a sub-tree corresponds to a segment of the to-be-multiplied list of elements. Use memoization to improve the running-time of the brute-force algorithm. *Hint:* a segment of a list can be represented by two integers, the index of the first and the index of the last element.

Exercise 2.3 (Infinite data structures, `Stream.lhs`)**.** *Note:* many of the definitions below clash with definitions in the standard prelude. The skeleton file for this exercise uses a *hiding* clause to avoid these clashes, see also Section A.3.

Haskell's list datatype comprises both finite and infinite lists. The type of streams defined below only contains infinite sequences.

```
data Stream elem = Cons {head :: elem, tail :: Stream elem}
infixr 5 ≺
(≺)  :: elem → Stream elem → Stream elem
a ≺ s = Cons a s
```

As a simple example, the sequence of natural numbers is given by *from* 0 where *from* is defined:

```
from :: Integer → Stream Integer
from n = n ≺ from (n + 1)
```

1. Define functions

```
repeat :: a → Stream a
map    :: (a → b) → (Stream a → Stream b)
zip    :: (a → b → c) → (Stream a → Stream b → Stream c)
```

that lift elements, unary operations, and binary operations pointwise to streams. For example,

```
⟩⟩⟩⟩ repeat 1
⟨ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...⟩
⟩⟩⟩⟩ map (2∗) (from 0)
⟨ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, ...⟩
⟩⟩⟩⟩ zip (∗) (from 0) (from 1)
⟨ 0, 2, 6, 12, 20, 30, 42, 56, 72, 90, 110, 132, 156, 182, 210, 240, ...⟩
```

2. Make *Stream elem* an instance of the *Num* type class. The general idea is that the methods are lifted pointwise to streams. This instance allows us to define the natural numbers and the Fibonacci numbers as follows:

```
nat, fib :: Stream Integer
nat = 0 ≺ nat + 1
fib  = 0 ≺ 1 ≺ fib + tail fib
```

3. Define a function

```
take :: Integer → Stream elem → [elem]
```

that allows us to inspect a finite portion of a stream: *take n s* returns the first *n* elements of *s*. Use the function to turn *Stream elem* into an instance of *Show*.

4. The function *diff* computes the difference of a stream.

```
diff :: (Num elem) ⇒ Stream elem → Stream elem
diff s = tail s − s
```

Here are some examples calls:

```
⟩⟩⟩⟩ diff fib
⟨ 1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...⟩
⟩⟩⟩⟩ (nat − 2) ∗ (nat + 3)
⟨− 6, −4, 0, 6, 14, 24, 36, 50, 66, 84, 104, 126, 150, 176, 204, 234, ...⟩
⟩⟩⟩⟩ diff it
⟨ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, ...⟩
⟩⟩⟩⟩ diff it
⟨ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...⟩
```

The second difference of $(nat - 2) * (nat + 3)$ is a constant stream, as the original stream is a polynomial of degree 2.

Finite difference has a right-inverse: anti-difference or summation. Derive its definition from the specification $diff\ (sum\ s) = s$ additionally setting $head\ (sum\ s) = 0$. Here are some examples calls:

```
⟩⟩⟩⟩ sum (2 * nat + 1)
⟨ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, ...⟩
⟩⟩⟩⟩ sum (3 * nat ^ 2 + 3 * nat + 1)
⟨ 0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375, ...⟩
⟩⟩⟩⟩ sum fib
⟨ 0, 0, 1, 2, 4, 7, 12, 20, 33, 54, 88, 143, 232, 376, 609, 986, ...⟩
```

Can you express the resulting streams without summation e.g. $sum\ (2 * nat + 1) = nat\ \hat{}\ 2$?

Exercise 2.4 (Worked example: minimax algorithm, `Minimax.lhs`)**.** The purpose of this exercise is to explore the AI behind "strategic games", where a human plays against a computer. For simplicity, we confine ourselves to impartial two-player games where players take alternate moves and the same moves are available to both players. As an example, consider the following game:

A position of the game is given by two positive integers, say, $m, n > 0$. A move consists of picking a number, say, m and dividing it into two positive natural numbers $i, j > 0$ such that $i + j = m$. If a player cannot make a move, i.e. $m = n = 1$, they lose. Here is an example run of the game:

initial position $5, 7$
Player A selects 7 and returns $3, 4$
Player B selects 4 and returns $1, 3$
Player A selects 3 and returns $1, 2$
Player B selects 2 and returns $1, 1$
Player A loses

The position $5, 7$ is actually a losing position: no matter what Player A's first move, they will always lose, unless Player B makes a mistake. This can be seen by inspecting the entire game tree spawned by the initial position, see Figure 1.

1. Implement a function

```
type Position = (Integer, Integer)
moves :: Position → [Position]
```

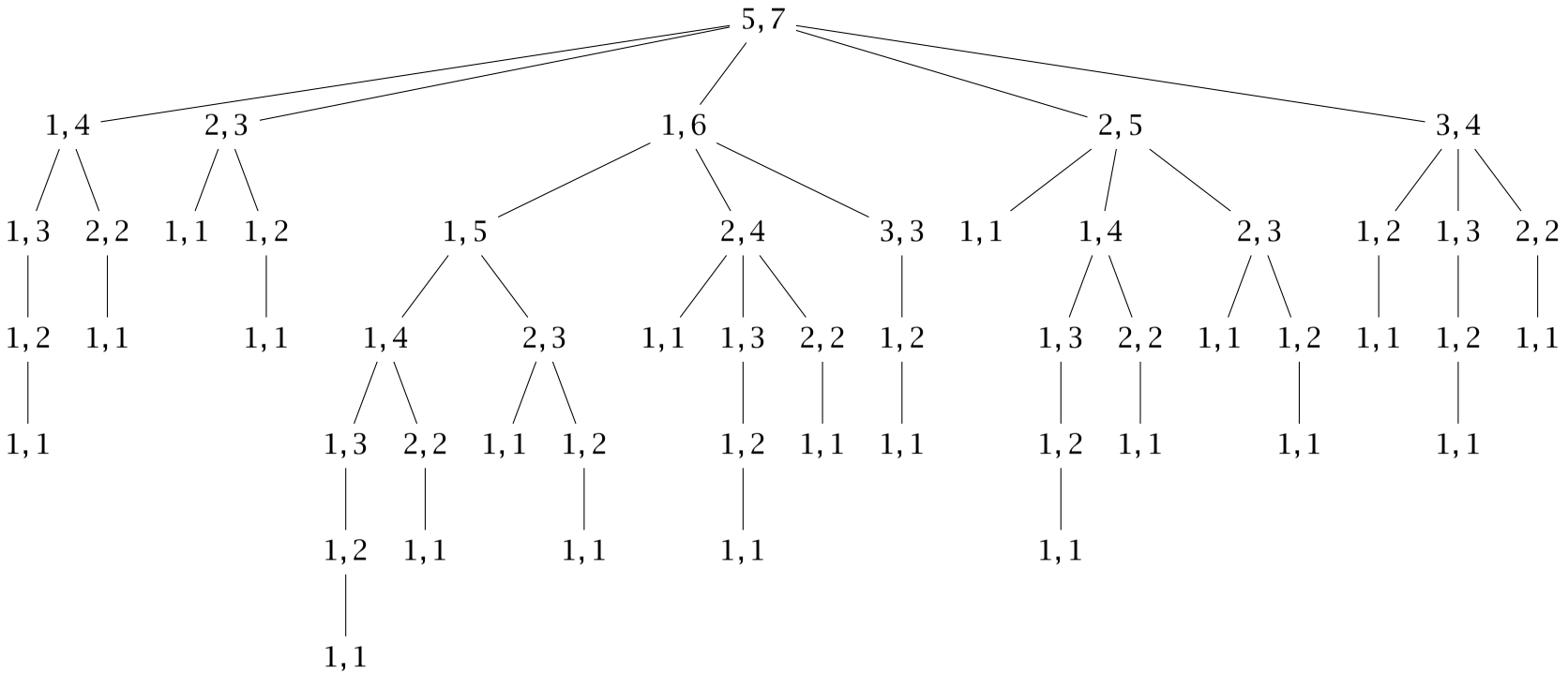


Figure 1: Game tree spawed by the initial position $5, 7$ (pick'n'divide game)

that returns a list of all possible moves from a given position e.g. $moves\ (1,1) = [\,]$ and $moves\ (5,7) = [\,(1,4),(2,3),(1,6),(2,5),(3,4)\,]$.

2. A game tree is an instance of a *multiway tree*, which is defined by the following datatype declaration:

   $$\textbf{data}\ Tree\ elem = Node\ elem\ [\,Tree\ elem\,]$$

   Define a function

   $$gametree :: (position \to [\,position\,]) \to (position \to Tree\ position)$$

   that constructs the entire game tree for a given initial position e.g. $gametree\ moves\ (5,7)$ should yield the tree shown in Figure 1. (Perhaps, you want to write a function that displays a game tree in a human-readable way.)

3. Implement a function

   $$size :: Tree\ elem \to Integer$$

   that computes the size of a multiway tree. As usual, the size corresponds to the number of elements contained in the tree.

4. Define functions

   $$\begin{aligned} winning &:: Tree\ position \to Bool \\ losing\ \ &:: Tree\ position \to Bool \end{aligned}$$

   that determine whether the root of a game tree is labelled with a winning or a losing position. A position is a winning position iff *some* successor position is a losing position. Dually, a position is a losing position iff *all* successor positions are winning positions. By that token, a final position is a losing position.

   Apply both *size* and *winning* to some largish game tree and measure the running times (using e.g. `set +s` within GHCi). What do you observe?

5. *Optional:* show that you lose if you don't win and vice versa:

   $$\begin{aligned} not\ (winning\ gametree) &= losing\ gametree \\ not\ (losing\ gametree) &= winning\ gametree \end{aligned}$$

6. *Optional:* in case you need some food for thought here are some additional games worth exploring.

mark game A position is given by a positive integer $n > 0$; the final position is 1; a move from a non-final position consists of replacing n by either $n - 1$ or by $n \,/\, 2$ provided n is even.

down-mark game The set-up is identical to the mark game, except that n is replaced by either $n - 1$ or by $\lfloor n \,/\, 2 \rfloor$.

up-mark game Again, the set-up is identical to the mark game, except that n is replaced by either $n - 1$ or by $\lceil n \,/\, 2 \rceil$.

Exploring an entire game tree is usually not an option, in particular, if the tree is wide and deep. To illustrate, if each position gives rise to exactly two moves, then a game tree of depth 500 has $2^{500} \approx 3 \cdot 10^{150}$ leaves. For comparison, the number of atoms in the observable universe is estimated at 10^{78}–10^{82}. Because of the exponential growth game trees are only evaluated up to some given depth e.g.

```
evaluate :: Integer → Position → Value
evaluate depth = maximize static ∘ prune depth ∘ gametree moves
```

7. Define a function

   ```
   prune :: Integer → Tree elem → Tree elem
   ```

   that prunes a multiway tree to a given depth i.e. the depth of *prune depth gametree* is at most *depth*.

8. Implement a function

   ```
   type Value = Int   — [−100..100]
   static :: Position → Value
   ```

   that statically evaluates a given position, returning some integer in the range $[-100\,..\,100]$. In general, the higher the value, the higher we *estimate* our chances of winning. Thus, -100 indicates a definite losing position, 100 a definite winning position. The other values reflect the uncertainty in our judgement.

   (If you have toyed a bit with the pick'n'divide game, you may be able to come up with a static evaluation that precisely predicts the outcome. In that case, you may want to consider one of the other games suggested above.)

9. Define functions

   ```
   maximize :: (position → Value) → (Tree position → Value)
   minimize :: (position → Value) → (Tree position → Value)
   ```

that evaluate a game tree, based on the static evaluation function provided. The function *maximize* works as follows: if the node has no sub-trees (as a result of pruning or because it is a final position), then the static evaluation is used; otherwise, *minimize* is applied to each of the sub-trees and the maximal value of these is returned. Dually, *minimize* returns the negation of the static evaluation for leaf nodes; otherwise, it applies *maximize* to each of the subtrees, returning the minimal resulting value.

10. *Optional:* show that

$$\textit{negate}\ (\textit{maximize static gametree}) = \textit{minimize static gametree}$$
$$\textit{negate}\ (\textit{minimize static gametree}) = \textit{maximize static gametree}$$

11. *Optional:* re-implement *maximize* and *minimize* using *alpha-beta pruning*.

3 Imperative programming

Exercise 3.1 (Warm-up: word count, `WordCount.lhs`)**.** Implement a simplified version of the UNIX command `wc`, see Figure 2. For example, applied to the current snapshot of the practicals, the program displays.

```
ralf@melian ~/Teaching/FP2/practicals $ wc *.lhs
  356  2006 13078 FP1.lhs
   61   403  2873 Introduction.lhs
  291  1295  9235 IO.lhs
  419  2537 14580 Lazy.lhs
  136  1081  5663 Modules.lhs
  117   355  2627 Monads.lhs
  118   208  3254 practicals.lhs
  127   414  2713 Types.lhs
   58    94   978 Unfolds.lhs
 1683  8393 55001 total
```

For accessing the program's command line arguments, the standard library *System.Environment* provides the command *getArgs*::*IO* [*String*]. As an example, the program

```
module Main where
import System.Environment

main :: IO ()
main = do args ← getArgs
          putStrLn (unwords args)
```

echoes the command line arguments to the standard output. Notice that a stand-alone Haskell program must contain a *Main* module that contains a definition of *main*::*IO* (). To compile the program, use GHC's built-in make facility, `ghc --make`, e.g.

```
ralf@melian $ ghc --make Echo.lhs
[1 of 1] Compiling Main             ( Echo.lhs, Echo.o )
Linking Echo ...
ralf@melian $ ./Echo hello world
hello world
```

Exercise 3.2 (Mastermind, `Mastermind.lhs`)**.** *Mastermind* is a game for two players: the *code-maker* and the *code-breaker*. The goal of the code-breaker is to guess the code word designed by the code-maker. In more detail:

```
WC(1)                    User Commands                    WC(1)

NAME
       wc - print newline, word, and byte counts for each file

SYNOPSIS
       wc [FILE]...

DESCRIPTION
       Print newline, word, and byte counts for each FILE, and
       a total line if more than one FILE is specified. A word
       is a non-zero-length sequence of  characters  delimited
       by white space.
```

Figure 2: An excerpt of the man page for `wc` (slightly simplified).

- The *code-maker* thinks of a code word that consists of a sequence of *four* codes, where a single code is chosen from a total of *eight* colours: *white*, *silver*, *green*, *red*, *orange*, *pink*, *yellow*, and *blue*. There are no restrictions on the code word: it may consist of four identical colours, but also of four different colours. Thus, in total there are $8^4 = 4096$ possible combinations to choose from.
- The *code-breaker* tries to guess the code word in as few turns as possible. If they need more than *twelve* guesses, they lose. A turn consists of making a guess, suggesting a code word i.e. a sequence of four colours. The *code-maker* responds by providing two hints:
  - the number of correct colours in the right positions;
  - the number of colours placed in the wrong position.

Write a console I/O program that implements the game *Mastermind*, where the user takes the role of a *code-breaker* and the computer is the *code-maker*. Try to minimize the I/O part of your program. Furthermore, design the code in such a way that it abstracts away from the various constant values mentioned above: it should work for any code length, for any number of colours, and for any maximum number of tries.

The standard library *System.Random* provides an extensive infrastructure for pseudo-random number generation. As an example use case, the

computation *dice* delivers a random integer between 1 and 6.

```
dice :: IO Int
dice = getStdRandom (randomR (1, 6))
roll :: IO Int
roll = do a ← dice
          b ← dice
          return (a + b)
```

Like *dice*, the computation *roll* is not pure: the answer may be different for each invocation:

```
⟩⟩⟩⟩ roll
5
⟩⟩⟩⟩ roll
12
```

Recall that for expressions of type *IO*, the Haskell interpreter first performs the computation and then prints the resulting value.

Exercise 3.3 (Worked example: one-time pad encryption, `OTP.lhs`)**.** The purpose of this exercise is to develop a program that encrypts and decrypts text files using the one-time pad method[2] (OTP for short).

OTP is an encryption method that provides, at least theoretically, perfect security and is thus unbreakable. To encode a message, it is XOR-ed it with a stream of *truly* random numbers. Decryption works in the same way, requiring an exact copy of the original random number stream. In theory, this method is unbreakable. In reality, there are several problems:

- Generating random numbers is far from trivial. Most random number generators found in programming languages are not truly random.
- Every random stream may be used only once, and has to be destroyed afterwards. Destroying data on a public computer is difficult.
- The transmitter and receiver must have an exact copy of the random stream. Therefore, the system is vulnerable to damage, theft, and copying.

[2] Source: http://en.wikipedia.org/wiki/One-time_pad

Despite these drawbacks, the OTP method has been used, for example, by the KGB, the main security agency of the former Soviet Union. They used truly random streams printed in a very small font, so that it could be easily hidden, see Figure 3.

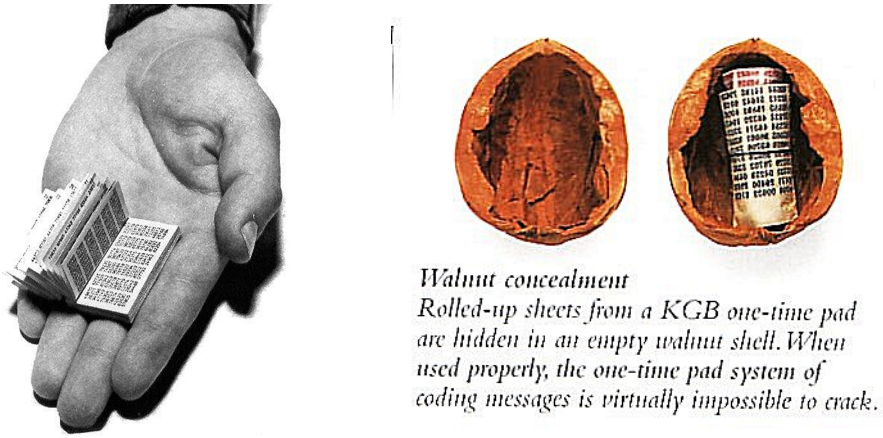


Figure 3: Truly random stream sources for OTP encryption, used in espionage. Sources: `http://www.ranum.com/security/computer_security/papers/otp-faq/` (left image); `http://home.egge.net/~savory/chiffre9.htm` (right image).

Haskell's global random number generator is initialized automatically in some system-dependent fashion, for example, by using the time of day, or Linux's kernel random number generator. To get *reproducible* behaviour (for en- and decryption) initialize it with some random seed:

```
main = do setStdGen (mkStdGen 4711)
```

Of course, you should keep the seed value secret, top secret. Use the pseudo-random number generator, see Exercise 3.2, to generate a sequence of pseudo random numbers $r_0, r_1, \ldots$.

Encryption and decryption then works as follows. For simplicity and to avoid bit fiddling, we use a Caesar cipher instead of XOR. Let A be the input text file that should be encrypted/decrypted to the output text file B. Read the ASCII characters $a_0, a_1, \ldots, a_n$ from A, and write the characters $b_0, b_1, \ldots, b_n$ to B.

$$b_i = \begin{cases} a_i & \textbf{if } a_i < 32 \\ ((a_i - 32 \oplus r_i) \bmod (128 - 32)) + 32 & \textbf{if } a_i \geqslant 32 \end{cases}$$

The case distinction ensures that only printable ASCII characters ($32 \leqslant a_i < 128$) are encrypted, whereas non-printable ones ($0 \leqslant a_i < 32$) are simply copied. When *encrypting*, $\oplus$ is *addition* ($+$). When *decrypting*, $\oplus$ is *subtraction* ($-$).

Write a console I/O program that accepts the input `encrypt` *A B*, where *A* and *B* are paths to text files. It encrypts the contents of *A* and writes the result to *B*. The input `decrypt` *A B* instructs the program to decrypt *A*, writing the result to *B*.

Test your program on some important input file *A*, for example, your bachelor thesis, that you first encrypt and then decrypt. Verify that the decrypted file is identical to *A*. (If you feel adventurous, delete the input file after encryption and only retain the decrypted variant.)

Exercise 3.4 (Singly-linked lists, `LinkedList.lhs`)**.**
Consider the implementation of singly-linked lists shown in the lectures.

```
type ListRef elem = IORef (List elem)
data List elem =  Nil | Cons elem (ListRef elem)
```

1. Define constructor functions:

   ```
   nil  :: IO (ListRef elem)
   cons :: elem → ListRef elem → IO (ListRef elem)
   ```

   The operation *nil* creates an empty linked list; *cons* attaches an element to the front of a given linked list.

2. Implement conversion functions

   ```
   fromList :: [elem] → IO (ListRef elem)
   toList   :: ListRef elem → IO [elem]
   ```

   that convert between Haskell's standard lists and singly-linked lists.

3. Define an internal iterator for singly-linked lists:

   ```
   foreach :: ListRef a → (a → IO b) → IO (ListRef b)
   ```

   The call *foreach list action* applies *action* to each element of the singly-linked list *list*, returning a singly-linked list collecting the

results. For example,

```
⟩⟩⟩⟩ as ← fromList [0..3]
⟩⟩⟩⟩ bs ← foreach as (\n → do print n; return (n + 1))
0
1
2
3
⟩⟩⟩⟩ print (toList bs)
[1,2,3,4]
```

Can you modify the implementation so that the elements of the input list are overwritten? Do you have to change *foreach*'s type signature?

4 Applicative functors and monads

Exercise 4.1 (Warm-up: non-determinism and lists, `Evaluator.lhs`)**.** Recall the expression datatype shown in the lectures. Extend the type and its evaluator by non-deterministic choice.

```
data Expr
  = Lit Integer        — a literal
  | Expr :+: Expr      — addition
  | Expr :*: Expr      — multiplication
  | Div Expr Expr      — integer division
  | Expr :?: Expr      — non-deterministic choice
```

The expression *e1*:?:*e2* either evaluates to *e1* or to *e2*, non-deterministically. For example,

```
toss :: Expr
toss = Lit 0 :?: Lit 1
```

evaluates either to 0 or to 1. To illustrate, here are some example evaluations involving *toss*.

```
⟩⟩⟩⟩ evalN toss
[0,1]
⟩⟩⟩⟩ evalN (toss :+: Lit 2 :*: toss)
[0,2,1,3]
⟩⟩⟩⟩ evalN (toss :+: Lit 2 :*: (toss :+: Lit 2 :*: (toss :+: Lit 2 :*: toss)))
[0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]
```

As you can see, the evaluator returns a list of all possible results.

```
evalN :: Expr → [Integer]
```

Haskell's list datatype is already an instance of *Functor*, *Applicative*, and *Monad*. So, all you have to do is to extend the interpreter in, say, applicative style by adding one equation for :?:.

Exercise 4.2 (Worked example: environments, `Evaluator.lhs`)**.**

1. Augment the expression datatype by variables.

```
data Expr
  = Lit Integer        — a literal
  | Expr :+: Expr      — addition
  | Expr :*: Expr      — multiplication
  | Div Expr Expr      — integer division
  | Var String         — a variable
```

The values of variables are provided by an environment, a mapping from variable names to values.

```
evalR :: Expr → [(String, Integer)] → Integer
```

For simplicity, the environment is implemented by a list of name-value pairs, a so-called association list. (The standard prelude offers a function `lookup::(Eq key) ⇒ key → [(key, val)] → Maybe val` for looking up a value by key.) For example,

```
⟩⟩⟩⟩ evalR (Var "a" :+: Lit 1) [("a",4711),("b",0815)]
4712
⟩⟩⟩⟩ evalR (Var "a" :*: Var "b") [("a",4711),("b",0815)]
3839465
⟩⟩⟩⟩ evalR (Var "a" :*: Var "c") [("a",4711),("b",0815)]
0
```

If a variable is not defined in the environment, it evaluates to 0.

Like in the previous exercise, extend the original interpreter in applicative style by adding one equation, this time for *Var s*. What's actually the applicative functor, the *f* in *Expr → f Integer*? (You will notice that this type is already an instance of *Functor*, *Applicative*, and *Monad*.)

2. *Optional:* change the interpreter to support *both* extensions, non-deterministic choice and variables.

Exercise 4.3 (Programming, `Generate.lhs`)**.** For purposes of testing it is often useful to enumerate the elements of a datatype. If the datatype is finite and small, we may want to enumerate all of its elements. For other datatypes, we could choose to enumerate all elements of some specified size or up to some given size. Generating elements can be accomplished in a systematic manner using the applicative instance of Haskell's list datatype. For example,

```
bools :: [Bool]
bools = pure False ++ pure True

maybes :: [elem] → [Maybe elem]
maybes elems = pure Nothing ++ (pure Just ⟨*⟩ elems)
```

Each definition is closely modelled after the corresponding datatype definition. Do you see how?

1. Apply the idea to generate cards, elements of type *Card*.

```
data Suit   = Spades | Hearts | Diamonds | Clubs
data Rank = Faceless Integer | Jack | Queen | King
data Card = Card Rank Suit | Joker
```

2. Adapt the idea to recursive datatypes such as lists or trees.

```
data Tree elem = Empty | Node (Tree elem) elem (Tree elem)
```

   In case you need some inspiration, here are some example calls:

```
⟩⟩⟩⟩ lists bools 1
[[False], [True]]
⟩⟩⟩⟩ lists bools 2
[[False, False], [False, True], [True, False], [True, True]]
⟩⟩⟩⟩ trees (lists bools 2) 1
[Node Empty [False, False] Empty, Node Empty [False, True] Empty,
 Node Empty [True, False] Empty, Node Empty [True, True] Empty]
⟩⟩⟩⟩ trees (lists bools 2) 2
[Node Empty [False, False] (Node Empty [False, False] Empty),
 Node Empty [False, False] (Node Empty [False, True] Empty),
 Node Empty [False, False] (Node Empty [True, False] Empty),
 Node Empty [False, False] (Node Empty [True, True] Empty),
 ...
 Node (Node Empty [True, True] Empty) [True, False] Empty,
 Node (Node Empty [True, True] Empty) [True, True] Empty]
```

Exercise 4.4 (Pencil and paper: applicative functor)**.**

1. Recall the functor laws:

$$fmap\ id = id \tag{1a}$$
$$fmap\ (f \circ g) = fmap\ f \circ fmap\ g \tag{1b}$$

   Show that the applicative functor laws, repeated for reference below, imply the functor laws if we assume $fmap\ f\ m = pure\ f \langle * \rangle\ m$.

$$pure\ id \langle * \rangle\ v = v \tag{2a}$$
$$pure\ (\circ) \langle * \rangle\ u \langle * \rangle\ v \langle * \rangle\ w = u \langle * \rangle\ (v \langle * \rangle\ w) \tag{2b}$$
$$pure\ f \langle * \rangle\ pure\ x = pure\ (f\ x) \tag{2c}$$
$$u \langle * \rangle\ pure\ x = pure\ (\backslash f \to f\ x) \langle * \rangle\ u \tag{2d}$$

2. Show that the interchange law (2d) is equivalent to the following property:

$$pure\ f \langle * \rangle u \langle * \rangle pure\ x = pure\ (flip\ f) \langle * \rangle pure\ x \langle * \rangle u \tag{3}$$

   where *flip* is defined $flip\ f\ x\ y = f\ y\ x$.

3. *Optional:* show that any expression built from *pure* and $\langle * \rangle$ can be transformed to a normal form in which a single pure function is applied to impure arguments:

$$pure\ f \langle * \rangle u_1 \langle * \rangle \ldots \langle * \rangle u_n$$

5 Type and class system extensions

Exercise 5.1 (Warm-up: random-access lists, `RandomAccessList.lhs`)**.** In the lectures we have discussed the concept of a *numerical representation*, a container type that is modelled after some number type. The intriguing idea is to use number-theoretic operations as blueprints for operations on container types. Haskell's list datatype, for example, can be seen as being modelled after the unary numbers (aka Peano numbers) with concatenation corresponding to addition.

```
data Nat      = Z    | S           Nat
data List elem = Zero | Succ elem (List elem)
```

A second example is afforded by random-access lists which are modelled after the binary numbers with consing corresponding to increment and indexing to comparison (roughly speaking).

```
data Bin
  = N
  | O Bin
  | I Bin
data Sequ elem
  = Nil
  | OCons      (Sequ (Pair elem))
  | ICons elem (Sequ (Pair elem))
type Pair elem = (elem, elem)
```

(Note that the constructor names are different from the ones used in the lectures in order to avoid name clashes and to further emphasize the correspondence between number systems and container types.)

1. Define functions that convert between unary and binary numbers:

   ```
   unary  :: Bin  → Nat
   binary :: Nat → Bin
   ```

2. Define functions that convert between standard lists and random-access lists:

   ```
   toList   :: Sequ elem → List elem
   fromList :: List elem → Sequ elem
   ```

Use the implementations of the first part as blueprints for the operations above. *Reminder:* since *Sequ elem* is a *nested datatype*, functions on random-access lists are likely to use *polymorphic recursion*. So you should always provide explicit type signatures—type inference in the presence of polymorphic recursion is not feasible.

Exercise 5.2 (Programming, `RandomAccessList12.lhs`)**.**

There are only 10 types of people in the world:
those who understand binary, and those who don't.

Binary numbers are an example of a *positional number system* where a number is represented by a sequence of weighted digits.

$$(d_0 \ldots d_{n-1})_2 = \sum_{i=0}^{n-1} d_i \times 2^i \qquad \text{with } d_i \in \{0, 1\}$$

For binary numbers it is customary to use the digits 0 and 1. However, this choice is by no means cast in stone. A viable alternative is to use the digits 1 and 2. Here is how to count in the $\{1, 2\}$-binary number system:

$$()_2,\ (1)_2,\ (2)_2,\ (11)_2,\ (21)_2,\ (12)_2,\ (22)_2,\ (111)_2,\ (211)_2,\ (121)_2,\ \ldots$$

The number zero is represented by an empty sequence. Incrementing a sequence of twos gives a sequence of ones (cascading carry).

Replay the development of the lectures using this variant of the binary numbers i.e. implement random-access lists based on the $\{1, 2\}$-binary number system. Your implementation should support at least the following interface.

```
data Sequ elem
nil  :: Sequ elem
cons :: elem → Sequ elem → Sequ elem
head :: Sequ elem → elem
tail :: Sequ elem → Sequ elem
(!)  :: Sequ elem → Int → elem
```

Why is it advantageous to use a *zeroless* numerical representation? *Hint:* what's the running time of *s* ! *i*? Does the running time depend on the index *i* or on the size of *s*?

Optional: the $\{1, 2\}$-binary number system is irredundant: every natural number has a unique representation in this system. If we allow

to use the three digits 0, 1, and 2, then we obtain a *redundant* number system as e.g. $(011)_2 = 6 = (22)_2$. Is there any advantage in basing a container type on a redundant number system? *Hint:* think hard about the interplay of *cons* and *tail*.

Exercise 5.3 (Worked example: digital searching, sequel of Exercise 6.6 of FP1, `DigitalSearching.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. However, often comparison is *not* a constant-time operation, consider e.g. comparing two strings. The purpose of this exercise is to show that we we can search in linear time if we employ the structure of search keys, linear in the size of the search key that is. In other words, the entire search takes as long as a single comparison!

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

```
type Map key val

empty :: (Ord key) ⇒ Map key val
insert  :: (Ord key) ⇒ key → (Maybe val → val) → Map key val → Map key val
lookup :: (Ord key) ⇒ key → Map key val → Maybe val
```

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 functions can only use it but not inspect it: the ordering is treated as an oracle.

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

```
data family Map key :: ∗ → ∗

class (Ord key) ⇒ Key key where
  empty :: Map key val
  insert  :: key → (Maybe val → val) → Map key val → Map key val
  lookup :: key → Map key val → Maybe val
```

The type *Map key val* is the type of *finite maps* from keys of type *key* to values of type *val*—finite maps are also known as dictionaries, look-up tables, association lists etc. (As an aside, *Map key* is a family of

datatypes, rather than types as we want to introduce a tailor-made data structure for each key type.)

The method *empty* creates an empty map. Loosely speaking, *insert* adds a key-value pair to a given map. To cater for the possibility that the map already contains an entry for the key, we provide *insert* with an update function of type *Maybe val → val*. This function is called with *Just old* where *old* is the current value associated with the key, and *Nothing* if no such entry exists. (The name *insert* is actually a bit of a misnomer, *update* may be more appropriate.)

To see how we can take advantage of a *white-box* approach to searching, 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. The corresponding finite map is either empty or a singleton that contains the value associated with `()`.

```
data instance Map () val = Empty | Single val
instance Key () where
  empty = Empty
  insert () f (Empty)    = Single (f Nothing)
  insert () f (Single v) = Single (f (Just v))
  lookup () (Empty)    = Nothing
  lookup () (Single v) = Just v
```

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

1. 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. Define a corresponding instance of *Key* i.e.

   ```
   data instance Map (Either key1 key2) = ..
   instance (Key key1, Key key2) ⇒ Key (Either key1 key2) where ..
   ```

2. 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
```

   Again, define a corresponding instance of *Key* i.e.

```
data instance Map (key1, key2) = ..
instance (Key key1, Key key2) ⇒ Key (key1, key2) where ..
```

3. 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 *Key*:

```
data instance Map [key] val = ..
instance (Key key) ⇒ Key [key] where ..
```

What about other datatypes such as binary trees?

Hint: the type of digital search trees, *Map key val*, takes two type arguments: *key* and *val*. These are treated quite differently: *Map* is defined by case analysis on the type of keys, but it is parametric in the type of values. In other words, *Map key* is a functor. The system behind the case analysis can be seen more clearly if we write *Map K V* as an exponential i.e. V^K. The definition of *Map K V* is based on the *laws of exponents*:

$$V^{A+B} = V^A \times V^B$$
$$V^{A\times B} = (V^B)^A$$

A map for a sum type is a pair of maps, one for keys of the form *Left a* and one for elements of the form *Right b*. A map for a product type is given by a nested map: to look up (a, b) we first look up a, which yields a finite map, in which we look up b. (This approach is reminiscent of the treatment of two-dimensional arrays in C or Java: they are simulated by nested arrays.)

Exercise 5.4 (Programming, `Printf.lhs`)**.** Fancy some type hacking? This exercise discusses an alternative implementation of C's *printf* that is both type-safe and extensible. The basic idea is the same as for the implementation given in the lectures—the format directives are elements of singleton types so that the type of *printf* can *depend* on the type of the format directive—but the details differ. To illustrate, given these format directives

```
data D = D
data F = F
data S = S

infixr 4 &
(&) :: a → b → (a, b)
a & b = (a, b)
```

the type and class magic allows us to invoke *printf* with a variable

number of arguments:

```
⟩⟩⟩⟩ printf ("I am " & D & " years old.") 51
"I am 51 years old."
⟩⟩⟩⟩ printf ("I am " & D & " " & S & " old.") 1 "year"
"I am 1 year old."
⟩⟩⟩⟩ fmt = "Color " & S & ", Number " & D & ", Float " & F
⟩⟩⟩⟩ :type fmt
fmt :: (String, (S, (String, (D, (String, F)))))
⟩⟩⟩⟩ printf fmt "purple" 4711 3.1415
"Color purple, Number 4711, Float 3.1415"
```

Turning to the implementation, ideally we would like to define a type family and a type class,

```
type family Arg dir res :: ∗
class Format dir where
  printf :: dir → Arg dir String
```

with instances

```
type instance Arg D res = Int → res
instance Format D where
  printf D = \i → show i
...
```

Sadly, this approach only works for primitive directives such as *D* or *F*, but not for compound ones such as *D* & *F*. (Have a go. If you can make it work, I'd really like to know!)

One way out of this dilemma is to define *printf* in terms of a more complicated function.

```
printf :: (Format dir) ⇒ dir → Arg dir String
printf dir = format dir id ""
class Format dir where
  format :: dir → (String → a) → String → Arg dir a
instance Format D where
  format D cont out = \i → cont (out ++ show i)
```

The helper function *format* takes two additional arguments in addition to the format directive: a so-called continuation and an accumulating string. The instance declaration shows how to use the arguments: the

textual representation of *i* is appended to the accumulator *out*, the result of which is then passed to the continuation *cont*.

Fill in the missing bits and pieces i.e. define type instances and class instances for *F*, *S*, *String*, and (*dir1*, *dir2*). *Hint:* do use the skeleton file provided as it includes pragmas to enable the necessary type and class system extensions.

6 Duality: folds and unfolds

Exercise 6.1 (Warm-up: programming, `Foldr.lhs`)**.** In the lectures we have re-defined *foldr* in order to exhibit the duality between folds and unfolds more clearly.

```
data List elem list = Nil | Cons elem list

fold :: (List elem ans → ans) → ([elem] → ans)
fold alg = consume
  where consume []       = alg Nil
        consume (x : xs) = alg (Cons x (consume xs))
```

But, are the two definitions, *foldr* and *fold*, actually equivalent?

1. Define *foldr* in terms of *fold*.
2. Conversely, define *fold* in terms of *foldr*.
3. *Optional:* are your definitions specific to *foldr* and *fold*, or would your approach work for any functions of types $(elem \to ans \to ans) \to ans \to ([elem] \to ans)$ and $(List\ elem\ ans \to ans) \to ([elem] \to ans)$?

   Put differently, establish the following isomorphism:

   $$(elem \to ans \to ans) \to ans \to ([elem] \to ans) \cong (List\ elem\ ans \to ans) \to ([elem] \to ans)$$

   Try to transform the first into the second type by applying general laws such as currying. *Hint:* note that $List\ a\ b \cong 1 + a \times b$.

Exercise 6.2 (Programming, `Minimax2.lhs`)**.** Reconsider the *multiway trees* of Exercise 2.4.

```
data Tree elem = Node elem [Tree elem]
```

1. Introduce a base functor for *Tree* and provide suitable class instances of *Functor* and *Base*.
2. Re-define the functions

   ```
   size, depth :: Tree elem → Integer
   ```

   using *fold*. The function *size* computes the size of a multiway tree i.e. the number elements of type *elem*; the function *depth* computes the length of the longest path from the root to a leaf.

3. Re-implement the function

$$gametree :: (position \to [position]) \to (position \to Tree\ position)$$

that constructs a game tree in terms of *fold*.

4. Re-implement the function

$$winning :: Tree\ position \to Bool$$

using *fold*. The function determines whether the root of a game tree is labelled with a winning position.

Exercise 6.3 (Worked example: algorithmic duality, `Sorting.lhs`)**.**

⟨...⟩ *we can see that sorting is worth of serious study, as a practical matter. Even if sorting were almost useless, there would be plenty of rewarding reasons for studying it anyway! The ingenious algorithms that have been discovered show that sorting is an extremely interesting topic to explore in its own right. Many fascinating unsolved problems remain in this area, as well as quite a few solved ones. From a broader perspective we will find also that sorting algorithms make a valuable* case study *of how to attack computer programming problems in general.*

The Art of Computer Programming, Volume 3—Donald E. Knuth

We have used sorting as a running example to illustrate the idea of algorithmic duality. This exercise continues the journey looking at a family of sorting algorithms that are based on search trees, either implicitly or explicitly. Like Heap Sort and Mingle Sort, these algorithms work in two phases:

- first phase: create a search tree from an unordered list;
- second phase: flatten the search tree to an ordered list.

In order to be able to apply the machinery of higher-order operators such as folds and unfolds, we assume the following definition of binary trees and its associated base functor.

```
data Tree elem = Empty | Node (Tree elem) elem (Tree elem)
data TREE elem tree = EMPTY | NODE tree elem tree
```

1. Define functions that create a search tree.

$$\begin{aligned}&grow1, grow2 :: (Ord\ elem) \Rightarrow [elem] \to Tree\ elem\\&grow1 = unfold\ (para\ (fmap\ (id \triangledown inn) \circ sprout))\\&grow2 = fold\quad (apo\quad (sprout \circ fmap\ (id \vartriangle out)))\end{aligned}$$

   More precisely, define the *algorithmic core* of these functions:

$$sprout :: (Ord\ a) \Rightarrow List\ a\ (x \times TREE\ a\ x) \to TREE\ a\ (x + List\ a\ x)$$

   Be careful to establish and to maintain the search tree property. Is the definition of *sprout* cast in stone, or are there any algorithmic choices? (Recall that there are at least four ways to define the counterpart of *sprout* for constructing heaps—two bad ones and two good ones.)

   Try to describe the workings of *grow1* (which focusses on the input) and *grow2* (which focusses on the output) in your own words. *Hint:* if you have not yet developed an intuitive understanding of the higher-order operators, it might be wortwhile to transform applications of *fold* and *unfold* etc into recursive definitions.

2. Define functions that flatten a search tree.

$$\begin{aligned}&flatten1, flatten2 :: (Ord\ elem) \Rightarrow Tree\ elem \to [elem]\\&flatten1 = fold\quad (apo\quad (wither \circ fmap\ (id \vartriangle out)))\\&flatten2 = unfold\ (para\ (fmap\ (id \triangledown inn) \circ wither))\end{aligned}$$

   Again, you only have to define the *algorithmic core* of these functions:

$$wither :: (Ord\ a) \Rightarrow TREE\ a\ (x \times List\ a\ x) \to List\ a\ (x + TREE\ a\ x)$$

   Be careful to preserve the relative order of elements so that a search tree is flattened to an ordered list (inorder traversal).

   Try to describe the workings of *flatten1* (which focusses on the input) and *flatten2* (which focusses on the output) in your own words.

3. Combine the phases to form sorting algorithms—there are four combinations altogether. Do you recognize any of the algorithms? One of them implements a version of Quick Sort, where the search tree structure is made explicit.

4. *Optional:* can you eliminate the intermediate data structure, the search tree? Generically?

Exercise 6.4 (Pencil and paper: uniqueness and fusion)**.** In the lectures we have introduced a *generic* definition of fold that works for arbitrary recursive datatypes.

$$\begin{aligned} &fold :: (Base\ f) \Rightarrow (f\ a \to a) \to (Rec\ f \to a) \\ &fold\ a = a \circ fmap\ (fold\ a) \circ out \end{aligned}$$

The generic definition enjoys generic properties!

First of all, folds enjoy the following *uniqueness property*:

$$f = fold\ a \iff f \circ inn = a \circ fmap\ f \tag{4}$$

The law states that *fold* is the unique solution of its defining equation.

1. Show that the uniqueness property implies the *computation law*:

   $$fold\ a \circ inn = a \circ fmap\ (fold\ a) \tag{5}$$

2. Show that the uniqueness property implies the *reflection law*:

   $$id = fold\ inn \tag{6}$$

   Note that $inn :: f\ (Ref\ f) \to Rec\ f$ is an algebra.

3. Finally, show that the uniqueness property implies the *fusion law*:

   $$h \circ fold\ a = fold\ b \impliedby h \circ a = b \circ fmap\ h \tag{7}$$

   The fusion law states a condition for fusing a function with a fold to form another fold.

4. Use the properties above to show that *inn* and $fold\ (fmap\ inn)$ are inverses of each other:

   $$\begin{aligned} &fold\ (fmap\ inn) \circ inn = id \\ &inn \circ fold\ (fmap\ inn) = id \end{aligned}$$

   In other words, $fold\ (fmap\ inn) = out$.

5. *Optional:* can you dualize the laws to unfolds?

A Modules

Haskell has a relatively simple module system which allows programmers to create and import modules, where a *module* is simply a collection of related types and functions.

A.1 Declaring modules

Most projects begin with something like the following as the first line of code:

```
module Main
where
```

This declares that the current file defines functions to be held in the *Main* module. Apart from the *Main* module, it is recommended that you name your file to match the module name. So, for example, suppose you were defining a number of protocols to handle various mailing protocols, such as POP3 or IMAP. It would be sensible to hold these in separate modules, perhaps named *Network.Mail.POP3* and *Network.Mail.IMAP*, which would be held in separate files. Thus, the POP3 module would have the following line near the top of its source file.

```
module Network.Mail.POP3
where
```

This module would normally be held in a file named

```
src/Network/Mail/POP3.hs .
```

Note that while modules may form a hierarchy, this is a relatively loose notion, and imposes nothing on the structure of your code.

By default, all of the types and functions defined in a module are exported. However, you might want certain types or functions to remain private to the module itself, and remain inaccessible to the outside world. To achieve this, the module system allows you to explicitly declare which functions are to be exported: everything else remains private. So, for example, if you had defined the type *POP3* and functions *send*::*POP3* → *IO* () and *receive*::*IO POP3* within your module, then these could be exported explicitly by listing them in the module declaration:

```
module Network.Mail.POP3 (POP3 (..), send, receive)
```

Note that for the type *POP3* we have written *POP3* (..). This declares that not only do we want to export the *type* called *POP3*, but we also want to export all of its constructors too.

A.2 Importing modules

The *Prelude* is a module which is always implicitly imported, since it contains the definitions of all kinds of useful functions such as $map :: (a \to b) \to (f\ a \to f\ b)$. Thus, all of its functions are in scope by default. To use the types and functions found in other modules, they must be imported explicitly. One useful module is the *Data.Maybe* module, which contains useful utility functions:

$$\begin{aligned} &maybe &&:: b \to (a \to b) \to Maybe\ a \to b \\ &catMaybes &&:: [Maybe\ a] \to [a] \end{aligned}$$

Importing all of the functions from *Data.Maybe* into a particular module is done by adding the following line below the module declaration, which imports every entity exported by *Data.Maybe*

```
import Data.Maybe
```

It is generally accepted as good style to restrict the imports to only those you intend to use: this makes it easier for others to understand where some of the unusual definitions you might be importing come from. To do this, simply list the imports explicitly, and only those types and functions will be imported:

```
import Data.Maybe (maybe, catMaybes)
```

This imports *maybe* and *catMaybes* in addition to any other imports expressed in other lines.

A.3 Qualifying and hiding imports

Sometimes, importing modules might result in conflicts with functions that have already been defined. For example, one useful module is *Data.Map*. The base datatype that is provided is *Map* which efficiently stores values indexed by some key. There are a number of other useful functions defined in this module:

$$\begin{aligned} &empty &&:: Map\ k\ v \\ &insert &&:: (Ord\ k) \Rightarrow k \to v \to Map\ k\ v \to Map\ k\ v \\ &update &&:: (Ord\ k) \Rightarrow k \to Map\ k\ v \to Maybe\ v \end{aligned}$$

It might be tempting to import *Map* and these auxiliary functions as follows:

```
import Data.Map (Map (..), empty, insert, lookup)
```

However, there is a catch here! The *lookup* function is initially always implicitly in scope, since the *Prelude* defines its own version. There are a number of ways to resolve this. Perhaps the most common solution is to qualify the import, which means that the use of imports from *Data.Map* must be prefixed by the module name. Thus, we would write the following instead as the import statement:

```
import qualified Data.Map
```

To actually use the functions and types from *Data.Map*, this prefix would have to be written explicitly. For example, to use *lookup*, we would actually have to write *Data.Map.lookup* instead.

These long names can become somewhat tedious to use, and so the qualified import is usually given as something different:

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

This brings all of the functionality of *Data.Map* to be used by prefixing with *M* rather than *Data.Map*, thus allowing you to use *M.lookup* instead.

Another solution to module clashes is to hide the functions that are already in scope within the module by using the *hiding* keyword:

```
import Prelude hiding (lookup)
```

This will override the *Prelude* import so that the definition of *lookup* is excluded.