Functional Programming 1

Practicals

Ralf Hinze

0 Getting started

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: definitions, `Database.lhs`)" indicates that this is a warm-up exercise training primarily the concept of "definitions" and that there is a source file available, called `Database.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/FP1.git
git clone https://gitlab.science.ru.nl/ralf/FP1.git
```

The Git repository will be updated on a regular basis. To obtain the latest updates, simply type `git pull` in the directory `FP1`. 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. '$\wedge$' for conjunction '&&', '$\leqslant$' for ordering '<=', and '$\circ$' for function composition '.'. Of course, the use of Unicode is strictly optional.

Have fun! Ralf Hinze

1 Programming with expressions and values

Exercise 1.1 (Warm-up: definitions, `Database.lhs`). Consider the following definitions, which introduce a type of persons and some sample data.

```
type Person = (Name, Age, FavouriteCourse)

type Name            = String
type Age             = Integer
type FavouriteCourse = String

frits, peter, ralf :: Person
frits = ("Frits", 33, "Algorithms and Data Structures")
peter = ("Peter", 57, "Imperative Programming")
ralf  = ("Ralf",  33, "Functional Programming")

students :: [Person]
students = [frits, peter, ralf]
```

1. Add your own data and/or invent some additional entries. In particular, add yourself to the list of students.

2. The function *age* defined below extracts the age from a person, e.g. *age ralf* ⟹ 33. (In case you wonder why some variables have a leading underscore see Hint 1.)

   ```
   age :: Person → Age
   age (_n, a, _c) = a
   ```

   Define functions

   ```
   name            :: Person → Name
   favouriteCourse :: Person → FavouriteCourse
   ```

   that extract name and favourite course, respectively.

3. Define a function *showPerson* :: *Person* → *String* that returns a string representation of a person. You may find the predefined operator ++ useful, which concatenates two strings e.g. `"hello, "` ++ `"world\n"` ⟹ `"hello, world\n"`. *Hint:* *show* converts a value to a string e.g. *show* 4711 = `"4711"`.

4. Define a function *twins* :: *Person* → *Person* → *Bool* that checks whether two persons are twins. (For lack of data, we agree that two persons are twins if they are of the same age.)

5. Define a function *increaseAge* :: *Person* → *Person* which increases the age of a given person by one e.g.

```
⟫⟫ increaseAge ralf
("Ralf",34,"Functional Programming")
```

6. The function *map* takes a function and a list and applies the function to each element of the list e.g.

```
⟫⟫ map age students
[33,57,33]
⟫⟫ map (\p → (age p, name p)) students
[(33,"Frits"),(57,"Peter"),(33,"Ralf")]
```

The function *filter* applied to a predicate and a list returns the list of those elements that satisfy the predicate e.g.

```
⟫⟫ filter (\p → age p > 50) students
[("Peter",57,"Imperative Programming")]
⟫⟫ map (\p → (age p, name p)) (filter (\p → age p > 50) students)
[(57,"Peter")]
```

Create expressions to solve the following tasks: a) increment the age of all students by two; b) promote all of the students (attach `"dr "` to their name); c) find all students named Frits; d) find all students whose favourite course is Functional Programming; e) find all students who are in their twenties; f) find all students whose favourite course is Functional Programming and who are in their twenties; g) find all students whose favourite course is Imperative Programming or who are in their twenties.

Exercise 1.2 (Pencil and paper: evaluation). 1. Recall the implementation of Insertion Sort from §0.4 (listed below, with some minor modifications).

```
insertionSort :: [Integer] → [Integer]
insertionSort []       = []
insertionSort (x : xs) = insert x (insertionSort xs)

insert :: Integer → [Integer] → [Integer]
insert a [] = a : []
insert a (b : xs)
  | a ⩽ b = a : b : xs
  | a > b = b : insert a xs
```

The function *insert* takes an element and an ordered list and inserts the element at the appropriate position e.g.

$$\begin{aligned}
& \mathit{insert}\ 7\ (2 : (9 : [\,]))\\
\Longrightarrow\quad & \{\ \text{definition of } \mathit{insert} \text{ and } 7 > 2\ \}\\
& 2 : (\mathit{insert}\ 7\ (9 : [\,]))\\
\Longrightarrow\quad & \{\ \text{definition of } \mathit{insert} \text{ and } 7 \leqslant 9\ \}\\
& 2 : (7 : (9 : [\,]))
\end{aligned}$$

Recall that Haskell has a very simple computational model: an expression is evaluated by repeatedly replacing equals by equals. Evaluate the expression *insertionSort* $(7 : (9 : (2 : [\,])))$—by hand, using the format above. (We have not yet discussed lists in any depth, but I hope you will be able to solve the exercise anyway. The point is that evaluation is a purely mechanical process—this is why a computer is able to perform the task.)

2. The function twice applies its first argument twice to its second argument.

```
twice f x = f (f x)
```

(Like *map* and *filter*, it is an example of a higher-order function as it takes a function as an argument.) Evaluate *twice* $(+1)\ 0$ and *twice twice* $(*2)\ 1$ by hand. Use the computer to evaluate

```
⟫⟫ twice ("|"++) ""
⟫⟫ twice twice ("|"++) ""
⟫⟫ twice twice twice ("|"++) ""
⟫⟫ twice twice twice twice ("|"++) ""
⟫⟫ twice (twice twice) ("|"++) ""
⟫⟫ twice twice (twice twice) ("|"++) ""
⟫⟫ twice (twice (twice twice)) ("|"++) ""
```

Is there any rhyme or rhythm? Can you identify any pattern?

Exercise 1.3 (Pencil and paper: λ-expressions). 1. An alternative definition of *twice* builds on λ-expressions.

```
twice = \f → \x → f (f x)
```

Re-evaluate $\mathit{twice}\ (+1)\ 0$ and $\mathit{twice}\ \mathit{twice}\ (*2)\ 1$ using this definition. You need to repeatedly apply the evaluation rule for λ-expressions (historically known as the β-rule).

$$(\backslash x \to \mathit{body})\ \mathit{arg} \Longrightarrow \mathit{body}\ \{x := \mathit{arg}\}$$

A function applied to an argument reduces to the body of the function where every occurrence of the formal parameter is replaced by the actual parameter e.g. $(\backslash x \to x + x)\ 47 \Longrightarrow x + x\ \{x := 47\} \Longrightarrow 47 + 47 \Longrightarrow 94$.

2. It is perhaps slightly worrying that you can apply a function to itself (as in $\mathit{twice}\ \mathit{twice}\ (*2)\ 1 = ((\mathit{twice}\ \mathit{twice})\ (*2))\ 1$). Can you guess the type of twice?

Exercise 1.4 (Worked example: prefix and infix notation).

1. Haskell features both alphabetic identifiers, written *prefix* e.g. $\mathit{sin}\ \mathit{pi}$, and symbolic identifiers, written *infix* e.g. $2 + 7$. The use of infix notation for addition is traditional. (The symbol “+” is a simplification of “et”, Latin for “and”.) Most programming languages (with the notable exception of LISP) have adopted infix notation. But is this actually a wise thing to do? What are the advantages and disadvantages of infix over prefix (or postfix) notation. Discuss!

2. Infix notation is inherently ambiguous: $x \otimes y \otimes z$. What does this mean: $(x \otimes y) \otimes z$ or $x \otimes (y \otimes z)$? To disambiguate without parentheses, operators may *associate* to the left or to the right. Subtraction associates to the left: $5 - 4 - 2 = (5 - 4) - 2$. Why? Concatenation of strings associates to the right: $\texttt{"F"} +\!\!+ \texttt{"P"} +\!\!+ \texttt{"1"} = \texttt{"F"} +\!\!+ (\texttt{"P"} +\!\!+ \texttt{"1"})$. Why? Haskell allows the programmer to specify the *association* of an operator using a *fixity declaration*:

   infixl $-$
   infixr $+\!\!+$

   Function application can be seen as an operator (“the space operator”) and associates to the left: $f\ a\ b$ means $(f\ a)\ b$. (On the other hand, the “function type” operator associates to the right: $\mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}$ means $\mathit{Integer} \to (\mathit{Integer} \to \mathit{Integer})$.) Haskell also features an explicit operator for function application, which associates to the right: $f \$ g \$ a$ means $f \$ (g \$ a) = f\ (g\ a)$. Can you foresee possible use-cases?

3. The operator

   $$\textbf{infixl} \otimes$$
   $$a \otimes b = 2 * a + b$$

   can be used to capture binary numbers e.g. $1 \otimes 0 \otimes 1 \otimes 1 \Longrightarrow 11$ and $(1 \otimes 1 \otimes 0) + 4711 \Longrightarrow 4717$. The fixity declaration determines that $\otimes$ associates to the left. Why this choice? What happens if we declare $\textbf{infixr}\otimes$?

4. Association does not help when operators are mixed: $x \oplus y \otimes z$. What does this mean: $(x \oplus y) \otimes z$ or $x \oplus (y \otimes z)$? To disambiguate without parentheses, there is a notion of *precedence* (or binding power), eg $*$ has higher precedence (binds more tightly) than $+$.

   $$\textbf{infixl}\ 7 *$$
   $$\textbf{infixl}\ 6 +$$

   The precedence level ranges between 0 and 9. Function application ("the space operator") has the highest precedence (ie 10), so $\mathit{square}\ 3 + 4 = (\mathit{square}\ 3) + 4$. Find out about the precedence levels of the various operators and *fully* parenthesize the expression below.

   $$f\, x \geqslant 0\ \&\&\ a\ ||\ g\, x\, y * 7 + 10 == b - 5$$

Exercise 1.5 (Programming). Define the string

$$\mathit{thisOldMan} :: \mathit{String}$$

that produces the following poem (if you type $\mathit{putStr}\ \mathit{thisOldMan}$).

This old man, he played one,
He played knick-knack on my thumb;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played two,
He played knick-knack on my shoe;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played three,
He played knick-knack on my knee;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played four,
He played knick-knack on my door;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played five,
He played knick-knack on my hive;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played six,
He played knick-knack on my sticks;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played seven,
He played knick-knack up in heaven;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played eight,
He played knick-knack on my gate;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played nine,
He played knick-knack on my spine;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

This old man, he played ten,

He played knick-knack once again;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home.

Try to make the program as short as possible by capturing recurring patterns. Define a suitable function for each of those patterns.

Exercise 1.6 (Programming, `Shapes.lhs`). The datatype *Shape* defined below captures simple geometric shapes: circles, squares, and rectangles.

```
data Shape
   = Circle Double                 -- radius
   | Square Double                 -- length
   | Rectangle Double Double       -- length and width
  deriving (Show)
```

Examples of concrete shapes include *Circle* (1 / 3), *Circle* 2.1, *Square pi*, and *Rectangle* 2.0 4.0.

The function *showShape* illustrates how to define a function that consumes a shape. A shape is one of three things. Correspondingly, *showShape* consists of three equation, one for each kind of shape.

```
showShape :: Shape -> String
showShape (Circle r)        = "circle of radius " ++ show r
showShape (Square l)        = "square of length " ++ show l
showShape (Rectangle l w) = "rectangle of length " ++ show l
                                ++ " and width " ++ show w
```

Use the same definitional scheme to implement the functions

```
area        :: Shape -> Double
perimeter   :: Shape -> Double
center      :: Shape -> (Double, Double)    -- x- and y-coordinates
boundingBox :: Shape -> (Double, Double)    -- width and height
```

(The names are hopefully self-explanatory.)

Hints to practitioners 1. Functional programming folklore has it that a functional program is correct once it has passed the type-checker. Sadly, this is not quite true. Anyway, the general message is to exploit the compiler for *static* debugging: compile often, compile soon. (To trigger a re-compilation after an edit, simply type `:reload` or `:r` in GHCi.)

We can also instruct the compiler to perform additional sanity checks by passing the option `-Wall` to GHCi e.g. call `ghci -Wall` (turn all warnings on). The compiler then checks, for example, whether the variables introduced on the left-hand side of an equation are actually used on the right-hand side. Thus, the definition $k\ x\ y = x$ will provoke the warning "Defined but not used: y". Variables with a leading underscore are not reported, so changing the definition to $k\ x\ \_y = x$ suppresses the warning.

2 Types and polymorphism

Exercise 2.1 (Warm-up: programming).

1. How many total functions are there that take one Boolean as an input and return one Boolean? Or put differently, how many functions are there of type $Bool \to Bool$? Define all of them. Think of sensible names.

2. How many total functions are there that take two Booleans as an input and return one Boolean? Or put differently, how many functions are there of type $(Bool, Bool) \to Bool$? Define at least four. Try to vary the definitional style by using different features of Haskell, e.g. predefined operators such as $||$ and $\&\&$, conditional expressions (**if** .. **then** .. **else** ..), guards, and pattern matching.

3. What about functions of type $Bool \to Bool \to Bool$?

Exercise 2.2 (Programming, `Char.lhs`). Haskell's *String*s are really lists of characters i.e. **type** $String = [Char]$. Thus, quite conveniently, all of the list operations are applicable to strings, as well: for example, *map toLower* `"Ralf"` $\Longrightarrow$ `"ralf"`. (Recall that *map* takes a function and a list and applies the function to each element of the list.)

1. Define an equality test for strings that, unlike $==$, disregards case, e.g. `"Ralf"` $==$ `"raLF"` $\Longrightarrow$ *False* but *equal* `"Ralf"` `"raLF"` $\Longrightarrow$ *True*.

2. Define predicates

   $$\begin{aligned} &isNumeral :: String \to Bool \\ &isBlank \quad\;\; :: String \to Bool \end{aligned}$$

   that test whether a string consists solely of digits or white space. You may find the predefined function $and :: [Bool] \to Bool$ useful which conjoins a list of Booleans e.g. $and\ [1 > 2, 2 < 3] \Longrightarrow False$ and $and\ [1 < 2, 2 < 3] \Longrightarrow True$. You also may want to import *Data.Char*, see Appendix A for details.

3. Define functions

   $$\begin{aligned} &fromDigit :: Char \to Int \\ &toDigit \quad\;\; :: Int \to Char \end{aligned}$$

   that convert a digit into an integer and vice versa, e.g. *fromDigit* `'7'` $\Longrightarrow$ 7 and *toDigit* $8 \Longrightarrow$ `'8'`.

4. Implement the Caesar cipher $shift :: Int \to Char \to Char$ e.g. $shift\ 3$ maps `'A'` to `'D'`, `'B'` to `'E'`, ..., `'Y'` to `'B'`, and `'Z'` to `'C'`. Try to decode the following message (*map* is your friend).

```
msg = "MHILY LZA ZBHL XBPZXBL MVYABUHL HWWPBZ JSHBKPBZ "
   ++ "JHLJBZ KPJABT HYJUBT LZA ULBAYVU"
```

Exercise 2.3 (Programming). Explore the difference between machine-integers of type *Int* and mathematical integers of type *Integer*. Fire up GHCi and type:

```
⟫⟫  product [1 .. 10] :: Int
⟫⟫  product [1 .. 20] :: Int
⟫⟫  product [1 .. 21] :: Int
⟫⟫  product [1 .. 65] :: Int
⟫⟫  product [1 .. 66] :: Int
```

The expression $product\ [1\,..\,n]$ calculates the product of the numbers from 1 up to n, aka the factorial of n. The type annotation $::Int$ instructs the compiler to perform the multiplications using machine-integers. Repeat the exercise using the type annotation $::Integer$. What do you observe? Can you explain the differences? On my machine the expression $product\ [1\,..\,66] :: Int$ yields 0. Why? (Something to keep in mind. Especially, if you plan to work in finance!)

Exercise 2.4 (Programming).

1. Define a function

   $$swap :: (Int, Int) \to (Int, Int)$$

   that swaps the two components of a pair. Define two other functions of this type (be inventive).

2. What happens if we change the type to

   $$swap :: (a, b) \to (b, a)$$

   Is your original definition of *swap* still valid? What about the other two functions that you have implemented?

3. What's the difference between the type $(Int, (Char, Bool))$ and the type $(Int, Char, Bool)$? Can you define a function that converts one "data format" into the other?

Exercise 2.5 (Warm-up: static typing).

1. Which of the following expressions are well-formed and well-typed? Assume that the identifier b has type $Bool$.

   $(+4)$
   div
   $div\ 7$
   $(div\ 7)\ 4$
   $div\ (7\ 4)$
   $7\ \text{`}div\text{`}\ 4$
   $+\ 3\ 7$
   $(+)\ 3\ 7$
   $(b, \texttt{'b'}, \texttt{"b"})$
   $(abs, \texttt{'abs'}, \texttt{"abs"})$
   $abs \circ negate$
   $(*3) \circ (+3)$

   If you get stuck, try to evaluate the expressions and/or see Hint 2. (As an aside, if you prefer ASCII over Unicode: in ASCII function composition is simply a full stop i.e. "`.`").

2. What about these?

   $(abs\circ) \circ (\circ negate)$
   $(div\circ) \circ (\circ mod)$

   (They are more tricky—don't spend too much time on this.)

3. Try to infer the types of the following definitions.

   $i\ x = x$
   $k\ (x, y) = x$
   $b\ (x, y, z) = (x\ z)\ y$
   $c\ (x, y, z) = x\ (y\ z)$
   $s\ (x, y, z) = (x\ z)\ (y\ z)$

   If you get stuck see Hint 2. Are any of these functions predefined (perhaps under a different name)? Again, see Hint 2.

Exercise 2.6 (Worked example: polymorphism). The purpose of this exercise is to explore the concept of *parametric polymorphism*. (The findings are not specific to Haskell or functional programming. Many statically typed object-oriented languages feature parametric polymorphism under the name of *generics*.)

1. Define total functions of the following types:

   (a) $Int \to Int$

   (b) $a \to a$

   (c) $(Int, Int) \to Int$

   (d) $(a, a) \to a$

   (e) $(a, b) \to a$

   How many total functions are there of type $Int \to Int$? By contrast, how many total functions are there of type $a \to a$?

2. Define total functions of the following types:

   (a) $(a, a) \to (a, a)$

   (b) $(a, b) \to (b, a)$

   (c) $(a \to b) \to a \to b$

   (d) $(a, x) \to a$

   (e) $(x \to a \to b, a, x) \to b$

   (f) $(a \to b, x \to a, x) \to b$

   (g) $(x \to a \to b, x \to a, x) \to b$

   Have you worked on a similar exercise before? Perhaps in a different context? *Hint:* read "$\to$" as logical implication and "," as logical conjunction.

3. Define total functions of the following types:

   (a) $Int \to (Int \to Int)$

   (b) $(Int \to Int) \to Int$

   (c) $a \to (a \to a)$

   (d) $(a \to a) \to a$

   How many total functions are there of type $(Int \to Int) \to Int$? By contrast, how many total functions are there of type $(a \to a) \to a$?

Hints to practitioners 2. GHCi features a number of commands that are useful during program development: e.g. `:type` ⟨expr⟩ or just `:t` ⟨expr⟩

shows the type of an expression; `:info` ⟨name⟩ or just `:i` ⟨name⟩ displays information about the given name e.g.

$\rangle\rangle\rangle\rangle$ $:\mathit{info}\ \mathit{map}$
$\mathit{map} :: (a \rightarrow b) \rightarrow [\,a\,] \rightarrow [\,b\,]$ — Defined in ‘GHC.Base’
$\rangle\rangle\rangle\rangle$ $:\mathit{type}\ \mathit{map} \circ \mathit{map}$
$\mathit{map} \circ \mathit{map} :: (a \rightarrow b) \rightarrow [\,[\,a\,]\,] \rightarrow [\,[\,b\,]\,]$

This is particularly useful if your program does not typecheck. (Or, if you are too lazy to type in signatures.)

More detailed information about the standard libraries is available online: `https://www.haskell.org/hoogle/`. Hoogle is quite nifty: it not only allows you to search the standard libraries by function name, but also by type! For example, if you enter `[a] -> [a]` into the search field, Hoogle will display all list transformers.

```
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam
nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam
erat, sed diam voluptua. At vero eos et accusam et justo duo dolores
et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est
Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur
sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et
dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam
et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea
takimata sanctus est Lorem ipsum dolor sit amet.
```

Table 1: Example text.

3 Lists

Exercise 3.1 (Programming: transformational programming, `WordList.lhs`). Write a *non-recursive* program that computes the word list of a given text, ordered by frequency of occurrence.

```
type Word = String
wordList :: String → [(Word, Int)]
```

For clarity, *wordList* includes the frequencies in the result, e.g.

```
⟫⟫ wordList lorem
[("accusam",2),("aliquyam",2),("at",2),("clita",2),("consetetur",2),
("dolore",2),("dolores",2),("duo",2),("ea",2),("eirmod",2),("elitr",2),
("eos",2),("erat",2),("est",2),("gubergren",2),("invidunt",2),("justo",2),
("kasd",2),("labore",2),("magna",2),("no",2),("nonumy",2),("rebum",2),
("sadipscing",2),("sanctus",2),("sea",2),("stet",2),("takimata",2),
("tempor",2),("ut",2),("vero",2),("voluptua",2),("amet",4),("diam",4),
("dolor",4),("ipsum",4),("lorem",4),("sed",4),("sit",4),("et",8)]
```

where *lorem* :: *String* is bound to the text displayed in Table 1.

You may find the following library functions useful (in alphabetical order): *filter*, *group*, *head*, *length*, *map*, *sort*, *sortOn*, *words*. To be able to use (some of) them you need to import *Data.List*, see Appendix A for details. (As an aside, *sort* and *sortOn* implement stable sorting algorithms. Why is this a welcome feature for this particular application?) Can you format the output so that one entry is shown per line?

Exercise 3.2 (Warm-up: list design pattern). Using the *list design pattern* discussed in the lectures, give recursive definitions of

1. a function $\mathit{allTrue} :: [\mathit{Bool}] \to \mathit{Bool}$ that determines whether every element of a list of Booleans is true;
2. a function *allFalse* that similarly determines whether every element of a list of Booleans is false;
3. a function $\mathit{member} :: (\mathit{Eq}\ a) \Rightarrow a \to [a] \to \mathit{Bool}$ that determines whether a specified element is contained in a given list;
4. a function $\mathit{smallest} :: [\mathit{Int}] \to \mathit{Int}$ that calculates the smallest value in a list of integers;
5. a function *largest* that similarly calculates the largest value in a list of integers.

The purpose of this exercise is to train the list design pattern for defining list consumers. However, once we have covered the corresponding material in the lectures, you may want to return to consider which of these functions can be written more simply using standard *higher-order functions* like *map* and *foldr*.

Exercise 3.3 (Programming). A *run* is a non-empty, non-decreasing sequence of elements. Use the *list design pattern* to define a function

$$\mathit{runs} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [[a]]$$

that returns a list of runs such that $\mathit{concat} \circ \mathit{runs} = \mathit{id}$, e.g.

```
⟫⟫ runs "hello, world!\n"
["h","ello",",","  w","or","l","d","!","\n"]
⟫⟫ concat it
"hello, world!\n"
```

Partitioning a list into a list of runs is a useful pre-processing step prior to sorting a list. Do you see why?

Exercise 3.4 (Programming, `DNA.lhs`). Recall the representation of bases and DNA strands introduced in the lectures.

```
data Base = A | C | G | T
  deriving (Eq, Ord, Show)
type DNA     = [Base]
type Segment = [Base]
```

1. Define a function $\mathit{contains} :: \mathit{Segment} \rightarrow \mathit{DNA} \rightarrow \mathit{Bool}$ that checks whether a specified DNA segment is contained in a DNA strand. Can you modify the definition so that a list of positions of occurrences is returned instead?

2. Define a function $\mathit{longestOnlyAs} :: \mathit{DNA} \rightarrow \mathit{Integer}$ that computes (the length of) the longest segment that contains only the base A.

3. Define a function $\mathit{longestAtMostTenAs} :: \mathit{DNA} \rightarrow \mathit{Integer}$ that computes (the length of) the longest segment that contains at most ten occurrences of the base A. (This is more challenging. Don't spend too much time on this part.)

Exercise 3.5 (Worked example: testing, `QuickTest.lhs`).

Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.
The Humble Programmer by Edsger W. Dijkstra

Beware of bugs in the above code;
I have only proved it correct, not tried it.
Donald Knuth

Harry Hacker has implemented a very nifty, highly efficient sorting function. At least, that's what he thinks. We are probably well advised to test his program thoroughly before using it in production code. The purpose of this exercise is to develop a little library for testing programs. The library can be seen as a tiny *domain-specific language* (DSL) for testing embedded in Haskell. (More on DSLs and EDSLs in §5.)

The principal idea is to take a *type-driven* approach to testing. To scrutinize a function f of type, say, $A \rightarrow B \rightarrow C$ we need probes for inputs of type A, probes for inputs of type B, and the to-be-verified property for results of type C. For simplicity, probes are just lists of inputs and a property is just a Boolean function.

$$\begin{aligned}&\textbf{type}\ \mathit{Probes}\ a &&= [a]\\ &\textbf{type}\ \mathit{Property}\ a &&= a \rightarrow \mathit{Bool}\end{aligned}$$

Given $\mathit{pa} :: \mathit{Probes}\ A$, $\mathit{pb} :: \mathit{Probes}\ B$, and $\mathit{pc} :: \mathit{Property}\ C$, the expression $\mathit{pa} \text{-->} \mathit{pb} \text{-->} \mathit{pc}$ is then a test procedure for functions of type $A \rightarrow B \rightarrow C$. Applied to f i.e. $(\mathit{pa} \text{-->} \mathit{pb} \text{-->} \mathit{pc})\ f$ it exercises its argument with all possible combinations of probes, checking each of the resulting outputs. For example, Harry's sorting function is exercised by

```
(permutations [0 .. 9] --> ordered) niftySort
```

Here *permutations* generates all permutations of its input list and *ordered* checks whether a list is ordered. Of course, Harry could easily defeat the test by defining *niftySort xs* = [ ]. A better approach is to check his implementation against a *trusted* sorting algorithm. This is accomplished using a variant of -->:

```
(permutations [0 .. 9] ==> \inp res → trustedSort inp == res) niftySort
```

Here, *inp* is bound to the original input and *res* is the result of Harry's program.

Turning to the implementation, --> and ==> can be conveniently implemented using list comprehensions.

```
infixr 1 -->, ==>
(-->) :: Probes a → Property b → Property (a → b)
(==>) :: Probes a → (a → Property b) → Property (a → b)
probes --> prop = \f → and [prop (f x) | x ← probes]
probes ==> prop = \f → and [prop x (f x) | x ← probes]
```

1. Define the predicate

   ```
   ordered :: (Ord a) ⇒ Property [a]
   ```

   that checks whether a list is ordered i.e. the sequence of elements is non-decreasing.

2. Apply the list design pattern to define the generator

   ```
   permutations :: [a] → Probes [a]
   ```

   that produces the list of all permutations of its input list. (How many permutations of a list of length n are there?)

3. Use the combinators to define a testing procedure for the function *runs* :: (*Ord a*) ⇒ [*a*] → [[*a*]] of Exercise 3.3.

4. Harry Hacker has translated a function that calculates the *integer square root* from C to Haskell.

   ```
   isqrt :: Integer → Integer
   isqrt n = loop 0 3 1
     where loop i k s | s ⩽ n      = loop (i + 1) (k + 2) (s + k)
                      | otherwise = i
   ```

It is not immediately obvious that this definition is correct. Define a testing procedure $isIntegerSqrt :: Property\ (Integer \to Integer)$ to exercise the program. Can you actually figure out how it works?

5. Define a combinator

```
infixr 4 ⊗
(⊗) :: Probes a → Probes b → Probes (a, b)
```

that takes probes for type a, probes for type b, and generates probes for type (a, b) by combining the input data in all possible ways e.g.

```
⟫⟫  bools = [False, True]
⟫⟫  bools ⊗ bools
[(False, False), (False, True), (True, False), (True, True)]
⟫⟫  bools ⊗ bools ⊗ bools
[(False, (False, False)), (False, (False, True)), (False, (True, False)), (False, (True, True)),
  (True, (False, False)), (True, (False, True)), (True, (True, False)), (True, (True, True))]
⟫⟫  chars = "Ralf"
⟫⟫  bools ⊗ chars
[(False, 'R'), (False, 'a'), (False, 'l'), (False, 'f'), (True, 'R'),
  (True, 'a'), (True, 'l'), (True, 'f')]
```

If as contains m elements, and bs contains n elements, then $as \otimes bs$ contains ...

Exercise 3.6 (Algorithmics: greedy algorithms, `Format.lhs`). An important problem in text processing is to format text into lines of some fixed width, ensuring as many words as possible on each line. If we assume that adjacent words are separated by one space, a list of words ws will fit in a line of width n if $length\ (unwords\ ws) \leqslant n$. Define a function

```
format :: Int → [Word] → [[Word]]
```

that given a maximal line width and a list of words returns a list of fitting lines so that $concat \circ format\ n = id$. Is this always possible? (As an aside, a function f with the property $concat \circ f = id$ computes a *partition* of its input. Have you seen this property before?)

For example, to format the text shown in Table 1 to a line width of 40 we type:

```
⟫⟫  putStr $ unlines $ map unwords $ format 40 $ words lorem
```

The resulting output is shown below

```
Lorem ipsum dolor
sit amet, consetetur sadipscing elitr,
sed diam nonumy eirmod tempor invidunt
ut labore et dolore magna aliquyam
erat, sed diam voluptua. At vero eos
et accusam et justo duo dolores et ea
rebum. Stet clita kasd gubergren, no sea
takimata sanctus est Lorem ipsum dolor
sit amet. Lorem ipsum dolor sit amet,
consetetur sadipscing elitr, sed diam
nonumy eirmod tempor invidunt ut labore
et dolore magna aliquyam erat, sed
diam voluptua. At vero eos et accusam
et justo duo dolores et ea rebum. Stet
clita kasd gubergren, no sea takimata
sanctus est Lorem ipsum dolor sit amet.
```

Apply the list design pattern to define *format*. You may want to take a greedy approach that makes locally optimal choices. For simplicity, we are content if the first line, but only the first line, wastes a lot of space. Once we have introduced the function *foldl* in the lectures, you may want to revisit this simplifying, but slightly odd assumption: usually, waste is permitted only in last line. (In general, text formatting is an optimization problem for some suitable measure of "badness" of formatting.)

Hints to practitioners 3. Do not confuse [*Base*] with the singleton list [*base*]. The first is a type; the second is a value, a shorthand for *base*:[ ]. Identifiers for values and identifiers for types live in different name spaces. Hence it is actually possible to use the same name for a type variable and for a value variable e.g.

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

The occurrence of *a* on the first line is a type variable; the occurrence of *a* on the second line is a value variable of type *a* :: *a*. If you think that this is confusing, simply use different names e.g.

```
insert :: (Ord elem) ⇒ elem → [elem] → [elem]
```

However, keep this “feature” in the back of your head, if you read other people’s code. There is a, perhaps, unfortunate tendency to use the same names both for types and elements of types.

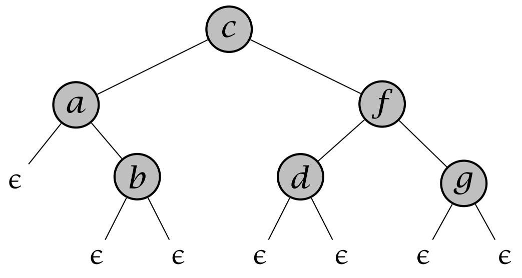


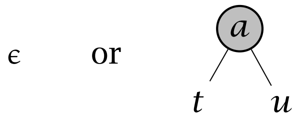
Figure 1: A binary tree of size 6.

4 Algebraic datatypes

Exercise 4.1 (Warm-up: data structures, `BinaryTree.lhs`). Binary trees are probably the second most popular data structure after lists and arrays. Binary trees are everywhere: expression trees, pedigrees, and tournament trees are examples of binary trees; binary trees can be used to implement sets, finite maps, and priority queues.

A *binary tree* is either

- empty, or
- a node that consists of a left tree, an element, and a right tree.



The recursive definition introduces *binary* trees as each node features exactly *two sub-trees*.

In computer science, trees typically grow downwards; that is, trees are drawn with the root at the top. The terminology surrounding trees is partly inspired by biological trees (root, leaf etc) and partly by pedigrees (child, parent etc). Consider the binary tree shown in Figure 1.

- The node c is the *root*; the empty sub-trees are also known as *leaves*. (A tree with n nodes has $n + 1$ leaves. Would you agree?)
- All nodes, with the exception of the root, have a *parent*: d's parent is f; f's parent is c; c has no parent as it is the root.
- Nodes may or may not have children: f has *children* d and g; d has no children; a has one child.

- The nodes a and f are *siblings*; d and g are *siblings*.

The recursive definition of trees can be easily transliterated into a Haskell datatype definition.

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

Like the type of lists, *Tree elem* is a *container type*: a binary tree contains elements of type *elem*. Thus, we have a tree *of* strings, a tree *of* lists *of* characters, or a tree *of* integers, etc.

1. Capture the binary tree shown in Figure 1 as a Haskell expression of type *Tree Char*.

2. Conversely, picture the Haskell expressions below.

```
Node Empty 4711 (Node Empty 0815 (Node Empty 42 Empty))
Node (Node (Node Empty "Frits" Empty) "Peter" Empty) "Ralf" Empty
Node (Node Empty 'a' Empty) 'k' (Node Empty 'z' Empty)
```

3. We have emphasized in the lectures that every datatype comes with a pattern of definition. Write down the "tree design pattern". Apply the design pattern to define a function *size* :: *Tree elem* → *Int* that calculates the number of elements contained in a given tree. The size of the tree shown in Figure 1 is 6.

4. Define functions *minHeight*, *maxHeight* :: *Tree elem* → *Int* that calculate the length of the shortest and the length of the longest path from the root to a leaf. The minimum height of our running example in Figure 1 is 2; the maximum height is 3.

5. What is the relation between the size, the minimal, and the maximal height of a tree?

6. Define a function *member* :: (*Eq elem*) ⇒ *elem* → *Tree elem* → *Bool* that determines whether a specified element is contained in a given binary tree.

Exercise 4.2 (Programming, `BinaryTree.lhs`).

1. Define functions *preorder*, *inorder*, *postorder* :: *Tree elem* → [*elem*] that return the elements contained in a tree in pre-, in-, and post-order, respectively e.g.

   ```
   ⟫⟫ preorder abcdfg
   "cabfdg"
   ⟫⟫ inorder abcdfg
   "abcdfg"
   ⟫⟫ postorder abcdfg
   "badgfc"
   ```

   where *abcdfg* represents the tree shown in Figure 1 (see Exercise 4.1.1). What is the running time of your programs?

2. Define a function *layout* :: (*Show elem*) ⇒ *Tree elem* → *String* that turns a tree into a string, showing one element per line and emphasizing the structure through indentation e.g. *putStr* (*layout abcdfg*) produces (turn your head by 90° to the left):

   ```
       / 'a'
           \ 'b'
   - 'c'
           / 'd'
       \ 'f'
           \ 'g'
   ```

3. *Optional:* Define a function that generates suitable LaTeX-code for typesetting binary trees. (There are a variety of packages for typesetting trees that you may want to use as a starting point.)

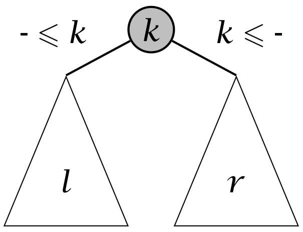
Exercise 4.3 (Programming, `BinaryTree.lhs`). 1. Define a function *build* :: [*elem*] → *Tree elem* that constructs a binary tree from a given list of elements such that *inorder* ∘ *build* = *id*. The shape of the tree does not matter. (Apply the list design pattern.)

2. Define a function *balanced* :: [*elem*] → *Tree elem* that constructs a balanced tree from a given list of elements, i.e. for each node the size of the left and the size of the right sub-tree should differ by at most one. The order of elements, however, does not matter. (Again, apply the list design pattern.)

3. Harry Hacker claims that his function *create* :: *Int* → *Tree* () can construct a tree of size *n* in logarithmic time i.e. *create n* takes $\Theta(\log n)$ steps. Genius or quacksalver?

Exercise 4.4 (Programming: data structures, `BinarySearchTree.lhs`).
A *binary search tree* is a binary tree such that

- the left sub-tree of every node only contains elements less than or equal to the element in the node;
- the right sub-tree of every node only contains elements greater than or equal to the element in the node.



In other words, an inorder traversal of the tree yields a non-decreasing sequence of elements. For example, *registry* defined below is a binary search tree.

```
registry :: Tree String
registry = Node (Node (Node Empty "Frits" Empty) "Peter" Empty) "Ralf" Empty
```

1. Define a function *member* :: (*Ord elem*) ⇒ *elem* → *Tree elem* → *Bool* that determines whether a specified element is contained in a given binary search tree. What's the difference to Exercise 4.1.6?

2. Define a function *insert* :: (*Ord elem*) ⇒ *elem* → *Tree elem* → *Tree elem* that inserts an element into a search tree e.g.

   ```
   ⟫⟩ insert "Nienke" registry
   Node (Node (Node Empty "Frits" (Node Empty "Nienke" Empty))
     "Peter" Empty) "Ralf" Empty
   ```

   (Just in case you have implemented binary search trees in an imperative or object-oriented language: is there a fundamental difference between the implementations?)

3. Define a function *delete* :: (*Ord elem*) ⇒ *elem* → *Tree elem* → *Tree elem* that removes an element from a binary search tree e.g.

   ```
   ⟫⟫ delete "Frits" registry
   Node (Node Empty "Peter" Empty) "Ralf" Empty
   ```

   If the element is not contained in the tree, the input tree is simply returned unchanged.

4. Use the library of Exercise 3.5 to test your code. In particular, define a function *isSearchTree* :: (*Ord elem*) ⇒ *Tree elem* → *Bool* that checks whether a given binary tree satisfies the search tree property. The most difficult part is to define a function that generates binary *search* trees. One approach is to program a function *trees* :: [*elem*] → *Probes* (*Tree elem*) that generates *all* trees whose inorder traversal yields the given list of elements: *and* [*inorder t* == *xs* | *t* ← *trees xs*]. Applied to an ordered list, *tree* will then generate search trees. (For the mathematically inclined: which sequence does [*length* (*trees* [1 . . *i*]) | *i* ← [0 . .]] generate? If you are clueless, `oeis.org` is your friend.)

Exercise 4.5 (Worked example: red-black trees, `RedBlackTree.lhs`). The running time of *member*, *insert*, and *delete* is bounded by the height of the binary search tree. Sadly, in the worst case the height is proportional to the size. To improve linear to logarithmic running time, a multitude of balancing schemes have been proposed: 2-3 trees, AA-trees, 2-3-4 trees, red-black trees, AVL-trees, 1-2 brother trees, B-trees, (a, b)-trees, size-balanced trees, splay trees, etc. This exercise discusses a fairly popular balancing scheme: red-black trees.

A red-black tree is a binary tree whose nodes are coloured either red or black.

```
data RedBlackTree elem
  = Leaf
  | Red   (RedBlackTree elem) elem (RedBlackTree elem)
  | Black (RedBlackTree elem) elem (RedBlackTree elem)
 deriving (Show)
```

Elements of this type are required to satisfy:

Red-condition: Each red node has a black parent.

Black-condition: Each path from the root to a leaf contains exactly the same number of black nodes—this number is called the tree's *black height*.

The conditions ensure that the height of a red-black tree of size n is bounded by $\Theta(\log n)$. Do you see why? Note that the red-condition implies that the root of a red-black tree is black.

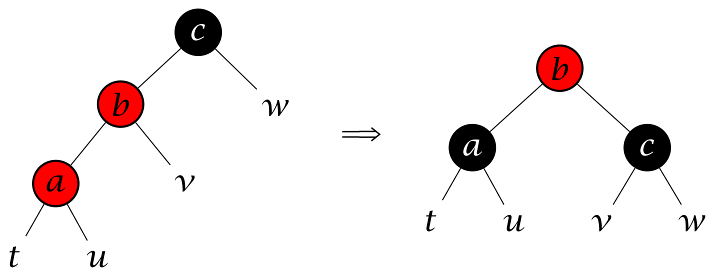
1. Adapt the membership test to red-black trees.

$$member :: (Ord\ elem) \Rightarrow elem \to RedBlackTree\ elem \to Bool$$

2. Adapt the insertion algorithm to red-black trees. (Do not worry about red- and black-conditions initially.)

$$insert :: (Ord\ elem) \Rightarrow elem \to RedBlackTree\ elem \to RedBlackTree\ elem$$

Let's agree that we always create a red node for the to-be-inserted element. Consequently, the black-condition stays intact. Only the red-condition is possibly violated. The idea is to repair violations through *local* transformations along the search path. The diagram below shows a tree shape that violates the red-condition and illustrates how to repair the defect.



There are three other illegal tree shapes that can occur after inserting a red node. Which ones?

Now introduce a *smart constructor*

$$black :: RedBlackTree\ elem \to elem \to RedBlackTree\ elem \to RedBlackTree\ elem$$

that implements these local transformations, and replace the actual constructor *Black* by the smart constructor in the code for *insert*. What happens if a red node percolates to the top?

3. Again, test the code using the library of Exercise 3.5. Define a function *isRedBlackTree* :: *RedBlackTree elem* → *Bool* that checks whether a tree is indeed a proper red-black tree. To exercise *insert* you also need to write a generator for red-black trees. Can you adopt the function *trees* of Exercise 4.4.4 to this setting?

Exercise 4.6 (Programming and mathematics, `Calculus.lhs`).
Lisa Lista's younger brother just went through differential calculus at high school. She decides to implement derivatives in Haskell to be able to easily double-check his homework solutions. To this end she introduces a datatype of primitive functions and a datatype of compound functions:

```
data Primitive
  = Sin    -- trigonometric: sine
  | Exp    -- exponential
  deriving (Show)
infixl 6 :+:
data Function
  = Const Rational              -- constant function
  | Id                          -- identity
  | Prim Primitive              -- primitive function
  | Function :+: Function       -- addition of functions
  deriving (Show)
```

The idea is that each element of *Primitive* and *Function* *represents* a function over the reals e.g. *Id* represents $\lambda\ x \to x$, *Const r* represents $\lambda\ x \to r$, *Prim Sin* :+: *Const* 2 :+: *Id* represents $\lambda\ x \to sin\ x + 2 + x$. In general, if *e1* represents *f1* and *e2* represents *f2*, then *e1*:+:*e2* represents the function $\lambda\ x \to f1\ x + f2\ x$.

1. Add a few more bells and whistles: more primitives, multiplication of functions (:∗:), composition of functions (:∘:), powers (:^:) etc.

2. Define a function *apply* :: *Function* → (*Double* → *Double*) that applies the representation of a function to a given value. In a sense, *apply* maps syntax to semantics: the representation of a function is mapped to the actual function.

3. Define a function *derive* :: *Function* → *Function* that computes the derivative of a function.

4. After Lisa has captured the rules of derivatives as a Haskell function (see Part 3), she tests the implementation on a few simple examples. The initial results are not too encouraging:

```
⟩⟩⟩⟩  derive (Const 1 :+: Const 2 :∗: Id)
Const (0 % 1) :+: (Const (0 % 1) :∗: Id :+: Const (2 % 1) :∗: Const (1 % 1))
⟩⟩⟩⟩  derive (Id :◦: Id :◦: Id)
Const (1 % 1) :◦: (Id :◦: Id) :∗: (Const (1 % 1) :◦: Id :∗: Const (1 % 1))
```

   Implement a function *simplify* :: *Function* → *Function* that simplifies the representation of a function using the laws of algebra. (This is a lot harder than it sounds!) *Hint:* use smart constructors.

Hints to practitioners 4. Inventing names is hard. In Haskell, we introduce names for values including functions, types and type variables, datatypes and their constructors, type classes and their methods (see §6), and modules (see Appendix A). As a rule of thumb, the wider the scope of an entity, the more care should be exercised in choosing a suitable identifier.

For example, the argument of a function only scopes over the function body i.e. the right-hand side of an equation in definitional style.

```
swap :: (a, b) → (b, a)
swap (x, y) = (y, x)
```

Hence it is perfectly fine to use short names such as *x* and *y*, which are, of course, not very telling. Or would you prefer the definition below?

```
swap (firstComponent, secondComponent)
  = (secondComponent, firstComponent)
```

Likewise, the type variables *a* and *b* only scope over the signature of *swap*. Again, it acceptable to use short names. On the other hand, the signature of *swap* may appear in the interface documentation of a module. But would you prefer the signature below?

```
swap :: (typeOfFirstComponent, typeOfSecondComponent)
          → (typeOfSecondComponent, typeOfFirstComponent)
```

By contrast, the identifier *swap* scopes over the entire module where its definition resides. (And beyond, if *swap* is exported by the module.) Hence the name should be chosen with care. (Are you happy with *swap* or would you prefer *swapTheComponentsOfAPair*?) The same rule applies to names of datatypes, constructors, and modules, all of which are potentially globally visible.

Tastes differ, of course. If you intensively dislike the name of a library function or type, you are free to introduce synonyms e.g.

type $\mathbb{Z}$ = *Integer*
sort = *insertionSort*

(As an aside, we can also rename module names in qualified imports e.g. **import** *qualified BinarySearchTree as Set*.) However, we cannot introduce synonyms for constructors (unless you use a recent extension of Haskell called pattern synonyms) or class and method names.

5 Higher-order functions

Exercise 5.1 (Warm-up: *foldl* and *foldr*). Redo Exercise 3.2: use the higher-order functions *foldl* and *foldr* to define

1. a function $allTrue :: [Bool] \to Bool$ that determines whether every element of a list of Booleans is true;
2. a function *allFalse* that similarly determines whether every element of a list of Booleans is false;
3. a function $member :: (Eq\ a) \Rightarrow a \to [a] \to Bool$ that determines whether a specified element is contained in a given list;
4. a function $smallest :: [Int] \to Int$ that calculates the smallest value in a list of integers;
5. a function *largest* that similarly calculates the largest value in a list of integers.

If both recursion schemes are applicable, which one is preferable in terms of running time?

Exercise 5.2 (Programming, `Numeral.lhs`). The decimal and the binary number system are both positional number systems. The meaning of the decimal numeral 4711 is $4 * 10^3 + 7 * 10^2 + 1 * 10^1 + 1 * 10^0$ i.e. the weight of a digit depends on its position. For decimal numerals the most significant digit usually comes first. For binary numerals both conventions, MSD and LSD first, are in use: 1011 denotes either $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 11$ or $1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 = 13$.

1. Use *foldl* and *foldr* to define functions

   ```
   type Base  = Integer
   type Digit = Integer
   msdf, lsdf :: Base → [Digit] → Integer
   ```

   which given a base convert a list of digits into a number (MSD and LSD first). *Hint:* recall Exercise 1.4.3.

2. Try to relate *msdf base* and *lsdf base*. (Can you define one in terms of the other?) Are your findings specific to the application at hand? Try to abstract away from the specifics and derive a general law about *foldl* and *foldr*.

Exercise 5.3 (Programming: parser combinators, `Expression.lhs`). The purpose of this exercise is to explore context-free grammars and parser combinators.

1. The grammar below defines the language of (simple) arithmetic expressions.

   $$\begin{aligned} \mathit{expr} &::= \mathit{digit}\ \{\mathit{digit}\} \\ &\ \mid\ \mathit{expr}\ \texttt{'+'}\ \mathit{expr} \\ &\ \mid\ \mathit{expr}\ \texttt{'*'}\ \mathit{expr} \\ &\ \mid\ \texttt{'('}\ \mathit{expr}\ \texttt{')'} \end{aligned}$$

   What's the difference to the grammar shown in the lectures (§5.5, see also below)? Implement the grammar above using parser combinators and test the resulting parser on a few simple examples e.g. `4*71+1` etc. What do you observe? (You also may want to revisit Exercise 1.4.)

2. Reconsider the expression grammar given in the lectures:

   $$\begin{aligned} \mathit{expr} &::= \mathit{term} \\ &\ \mid\ \mathit{term}\ \texttt{'+'}\ \mathit{expr} \\ \mathit{term} &::= \mathit{factor} \\ &\ \mid\ \mathit{factor}\ \texttt{'*'}\ \mathit{term} \\ \mathit{factor} &::= \mathit{digit}\ \{\mathit{digit}\} \\ &\ \mid\ \texttt{'('}\ \mathit{expr}\ \texttt{')'} \end{aligned}$$

   Observe that the alternatives for *expr* and *term* share a common prefix. An important optimization is to *left factor* a grammar to avoid repetitive parses e.g.

   $$\begin{aligned} \mathit{expr} &::= \mathit{term}\ (\epsilon \mid \texttt{'+'}\ \mathit{expr}) \\ \mathit{term} &::= \mathit{factor}\ (\epsilon \mid \texttt{'*'}\ \mathit{term}) \end{aligned}$$

   Apply the optimization to the expression parser given in the lectures. (The hard part is to adapt the semantic actions.) Does the running time improve? (Within GHCi type `:set +s` to ask GHCi to print timing and memory statistics after each evaluation.)

Exercise 5.4 (Programming: parser combinators, `Lambda.lhs`). Write a parser for Haskell expressions built from variables (x, y etc), using λ-abstraction ($\backslash x \rightarrow e$) and application ($e1\ e2$). This tiny language can be seen as the core of Haskell—it is known as the λ-calculus. The *abstract* syntax for λ-expressions is given by the datatype

```
infixl 9 :@
data Lambda var
    = Var var                      — variable
    | Fun var (Lambda var)         — abstraction/λ-expression
    | Lambda var :@ Lambda var     — application
  deriving (Show)
```

which abstracts away from the representation of variables. For example, *Fun* 0 (*Var* 0) :: *Lambda Integer* and *Fun* `"x"` (*Var* `"x"`) :: *Lambda String* represent both the identity but use different types for variables.

1. Define the *concrete* syntax of λ-expressions using a context-free grammar. Haskell's $\backslash x \to x$ is traditionally written $\lambda x.x$. Which syntax do you prefer or, perhaps, you want to support both? How do you want to represent variables: by a single letter or using a full-blown identifier? Take you pick. Capture the syntactic conventions that application associates to the left, i.e. $e_1 e_2 e_3$ is shorthand for $(e_1 e_2) e_3$, and that abstraction extends as far as possible to the right, e.g. $\lambda x.xy$ means $\lambda x.(xy)$ rather than $(\lambda x.x)y$.

2. Implement the grammar for λ-expressions using parser combinators and test the resulting parser on a few examples e.g. is $a\lambda x.x$ legal syntax for *Var* `'a'` :@ *Fun* `'x'` (*Var* `'x'`)?

Exercise 5.5 (Worked example: a tribute a Haskell B. Curry and David Turner, `SKI.lhs`). The purpose of this exercise is to give you an idea how Haskell i.e. the λ-calculus might be implemented. (The technique developed is somewhat outdated, but it was actually used for Haskell's predecessor, David Turner's Miranda.)

The first thing to observe is that a standard stack-based architecture (see §4.5) is not fit for the job. Consider the expression *twice succ*. In a call-by-value regime (eager evaluation) we first push *succ* onto the stack, and then call $twice = \backslash f \to \backslash x \to f\ (f\ x)$. But the function *twice* returns immediately, yielding the λ-expression $\backslash x \to f\ (f\ x)$, which contains the free variable *f*. Upon return the entry *succ* is removed. But *succ* is required, if the λ-expression is later applied to an argument. (As an aside, this is why the language C features only *top-level* functions.)

So λ-expressions are difficult to implement. Perhaps, we can get rid of them? Surprisingly, this is indeed possible. (The approach is based on results from the mathematical field of *combinatory logic*, which Haskell B. Curry founded.) Let us work through two examples: *twice* and *twice twice*. We first compile the λ-expression into “machine code”.

```
⟫⟫  twice = parse expr "λf.λx.f(fx)"
⟫⟫  twice
Fun 'f' (Fun 'x' (Var 'f' :@ (Var 'f' :@ Var 'x')))
⟫⟫  compile twice
S (S (K S) (S (K K) I)) (S (S (K S) (S (K K) I)) (K I))
```

Traditional machine code consists of a *sequence* of instructions. Here we have a *binary tree* of instructions instead. To reduce the tree to a normalform we apply it to two “primitives”.

```
⟫⟫  reduce it [Free 's', Free 'z']
's' ('s' 'z')
```

(You may want to read *s* as *s*uccessor and *z* as *z*ero.) Thus, *twice s z* is *s* (*s z*). The reduction machine has, actually, no notion of primitives: *Free* `'s'` and *Free* `'z'` are really free variables, which are treated purely symbolically. We can also apply *twice* to itself.

```
⟫⟫  compile (twice :@ twice)
S (S (K S) (S (K K) I)) (S (S (K S) (S (K K) I)) (K I)) (S (S (K S)
  (S (K K) I))(S (S (K S) (S (K K) I)) (K I)))
⟫⟫  reduce it [Free 's', Free 'z']
's' ('s' ('s' ('s' 'z')))
```

Thus, *twice twice s z* is *s* (*s* (*s* (*s z*))).

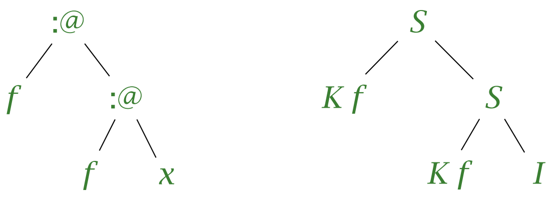
The type of machine instructions *SKI var*, defined below, is a stripped-down version of the type of λ-expressions *Lambda var*, defined in Exercise 5.4. The constructor *Free* is the counterpart of *Var*, and *App* (printed as a space in the examples above) is the counterpart of :@

```
data SKI var
  = Free var                  -- free/unbound variable
  | S
  | K
  | I
  | App (SKI var) (SKI var)   -- application
```

Compared to *Lambda var*, the type misses a case for λ-expressions—of course, we wanted to get rid of those—and features three additional constants instead: S, K, and I. On the face of it, SKI terms can be seen as binary leaf trees with four kinds of leaves. Now, why these constants? They allow us to *simulate* λ-abstractions. To get an idea how this works consider the λ-expression $\lambda x.f(fx)$:

```
⟫⟫  compile (Fun 'x' (Var 'f' :@ (Var 'f' :@ Var 'x')))
S (K 'f') (S (K 'f') I)
```

The body of the λ-expression consists only of applications, which can be seen as a binary tree. The translation is a binary tree of the same shape:



The combinators S, K, and I are machine instructions for performing a substitution: S distributes an incoming argument down the two subtrees, K discards an incoming argument, and I accepts it. Given

```
i :: env → env
i arg = arg
k :: a → (env → a)
k x _arg = x
s :: (env → a → b) → (env → a) → (env → b)
s x y arg = (x arg) (y arg)
```

we can show that $s\ (k\ f)\ (s\ (k\ f)\ i) = \backslash x \to f\ (f\ x)$:

$$
\begin{aligned}
& s\ (k\ f)\ (s\ (k\ f)\ i)\ arg \\
\Longrightarrow\quad & \{\ \text{definition of } s\ \} \\
& (k\ f\ arg)\ (s\ (k\ f)\ i\ arg) \\
\Longrightarrow\quad & \{\ \text{definition of } k \text{ and definition of } s\ \} \\
& f\ ((k\ f\ arg)\ (i\ arg)) \\
\Longrightarrow\quad & \{\ \text{definition of } k \text{ and definition of } i\ \} \\
& f\ (f\ arg)
\end{aligned}
$$

Can you see that S, K, and I propagate the actual parameter arg to the original occurrence(s) of the formal parameter x in the body?

1. Define a function

   $$\mathit{abstr} :: (\mathit{Eq}\ \mathit{var}) \Rightarrow \mathit{var} \to \mathit{SKI}\ \mathit{var} \to \mathit{SKI}\ \mathit{var}$$

   that implements λ-abstraction for SKI terms e.g. *abstr* `'x'` (*Free* `'x'`) = *I*, *abstr* `'x'` (*Free* `'x'` ‘*App*‘ *Free* `'x'`) = *S I I*, etc.

2. Define a compiler from λ-expressions to SKI machine code:

   $$\mathit{compile} :: (\mathit{Eq}\ \mathit{var}) \Rightarrow \mathit{Lambda}\ \mathit{var} \to \mathit{SKI}\ \mathit{var}$$

3. Implement a reduction machine that simplifies SKI terms:

   $$\mathit{reduce} :: \mathit{SKI}\ \mathit{var} \to [\,\mathit{SKI}\ \mathit{var}\,] \to \mathit{SKI}\ \mathit{var}$$

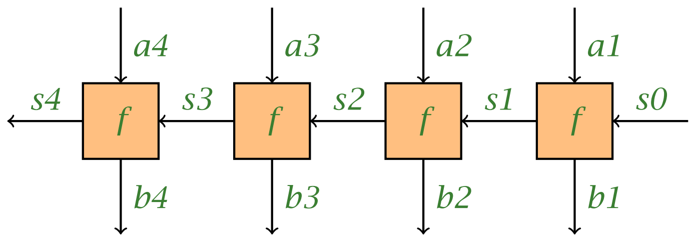
   The second argument of *reduce* serves as a stack. The function traverses the left spine of the binary tree to the leftmost leaf, pushing the visited nodes onto the stack. Then it applies the definitions of *s*, *k*, and *i* as rewrite rules.

4. Test the compiler and the reduction machine on some examples. Do you obtain the same results as in Haskell? In case you are lacking inspiration here are some things to try:

   ```
   ⟫⟫  parse expr "(λx.xx)(λx.xx)"
   ⟫⟫  compile it
   ⟫⟫  reduce it [ ]
   ⟫⟫  parse expr "λf.(λx.f(xx))(λx.f(xx))"
   ⟫⟫  compile it
   ⟫⟫  reduce it [Free 's', Free 'z']
   ```

   You may be surprised to learn that, like the λ-calculus, SKI machine code is Turing-complete.

Exercise 5.6 (Programming and hardware design, `Hardware.lhs`). Complex circuits are often assembled from simpler components using some regular "wiring pattern". A simple example is afforded by the ripple carry adder, which implements the school algorithm for addition in hardware. It consists of a series of full adders:



Each full adder takes two summand bits (top input) and a carry (right input), and produces a sum bit (bottom output) and a carry (left output).

1. Capture the wiring scheme as a higher-order function.

   $$\mathit{mapr} :: ((a, \mathit{state}) \to (b, \mathit{state})) \to (([a], \mathit{state}) \to ([b], \mathit{state}))$$

   The second component of each pair can be seen as a state. In a sense, *mapr* is a stateful version of *map*. Contrary to imperative programming, the state is explicit, not implicit. The diagram above shows that the state is threaded through the gates from right to left (hence the *r* in *mapr*).

2. Use the higher-order function to implement a ripple carry adder. I propose that you define a tailor-made datatype for binary digits.

   ```
   data Bit = O | I
     deriving (Eq, Ord, Show)
   ```

   Recall that a full adder is defined in terms of two half adders plus some additional circuitry.

Hints to practitioners 5. GHC offers a plethora of extension to the standard Haskell language. We have already encountered

```
{-# LANGUAGE UnicodeSyntax #-}
```

which enables the use of Unicode characters e.g. ∷ for :: etc. Several of the language extensions can be characterized as *syntactic sugar*: not in any way essential, but nice to have. One of the convenience features is

```
{-# LANGUAGE LambdaCase #-}
```

Here is a use case:

In §2 we have emphasized a type-driven approach to programming. The idea is that the type of a function suggests the code we might want to write. To illustrate the type-driven approach, consider calling the function $\mathit{fold} :: (\mathit{List}\ \mathit{Int}\ \mathit{Bool} \to \mathit{Bool}) \to ([\mathit{Int}] \to \mathit{Bool})$ where the non-recursive datatype *List* is defined:

```
data List a b = Nil | Cons a b
```

The function *fold* is higher-order: it expects a function as an argument. Functions are created using λ-expressions, so we can start coding:

```
fold (\x → □ )
```

How to fill the hole? Well, the argument function consumes an element of a datatype, which suggests using a case analysis. The cases are, of course, dictated by the datatype:

$$fold\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\ \{Nil \to\ \Box\ ; Cons\ age\ ok \to\ \Box\ \})$$

How to fill the holes? Well, the argument function has to produce a Boolean, an element of a simple enumeration type, which suggests using constructors. So we may replace the first hole by *True* (or by *False*, but these are really the only choices).

$$fold\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\ \{Nil \to True; Cons\ age\ ok \to\ \Box\ \})$$

The second hole is more interesting because some data is available to play with. The pattern match introduces $age :: Int$ and $ok :: Bool$, which we can suitably combine e.g.

$$fold\ (\backslash x \to \mathbf{case}\ x\ \mathbf{of}\ \{Nil \to True; Cons\ age\ ok \to age \geqslant 18\ \&\&\ ok\})$$

Of course, the specifics depend on the task at hand, but I hope you get the general idea.

Now, functions that consume data are quite frequent. The syntactic nicety mentioned in the beginning allows us to write the combination of $\backslash$ and **case** more succinctly,

$$fold\ (\backslash\mathbf{case}\ \{Nil \to True; Cons\ age\ ok \to age \geqslant 18\ \&\&\ ok\})$$

sparing us the invention of a variable—inventing names is hard.

(Just in case you are curious, here is the definition of *fold*.

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

It combines the first two arguments of *foldr*, i.e. $\triangleright$ and e, into a single argument, i.e. *alg*: we have $x \triangleright y = alg\ (Cons\ x\ y)$ and $e = alg\ Nil$. The argument $alg :: List\ a\ ans \to ans$ is known as an *algebra*, hence the name. Recall the *slogan*: fold replaces constructors by functions. The cases $\{Nil \to ..; Cons\ a\ b \to ..\}$ define the replacements for $[\,]$ and :.)

And if you are curious about the many other language extensions see §9 of the GHC Users Guide (https://wiki.haskell.org/GHC).

6 Type classes

Exercise 6.1 (Warm-up: type classes, `BinaryTree.lhs`). Recall the definition of binary trees.

```
data Tree elem = Empty | Node (Tree elem) elem (Tree elem)
  deriving (Eq, Ord)
```

The **deriving** clause automagically generates instance declarations for the type classes *Eq* and *Ord*. To appreciate this convenience, remove the **deriving** clause and program the instances by hand.

```
instance (Eq elem) ⇒ Eq (Tree elem) where ...
instance (Ord elem) ⇒ Ord (Tree elem) where ...
```

Are these instance declarations actually useful? Can you think of a use-case where you would need to compare two binary trees? *Warning:* note that the method name for ordering is <= (two ASCII symbols) and not $\leqslant$ (one Unicode character).

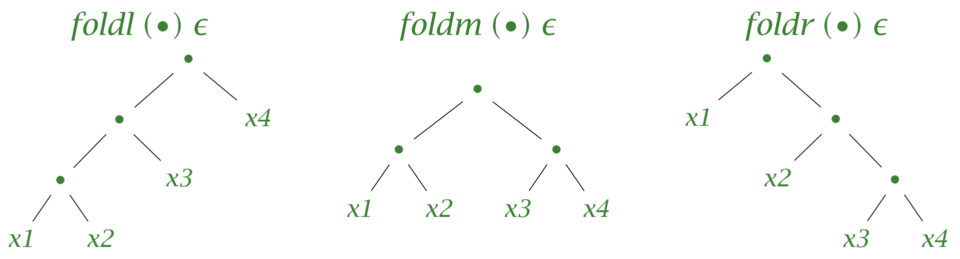
Exercise 6.2 (Warm-up: map-reduce, `MapReduce.lhs`).

1. How many ways are there to turn the type *Bool* into a monoid? (There are sixteen *candidates* as there are sixteen functions of type *Bool* → *Bool* → *Bool*.) Define all of them using **newtype** definitions—think hard about telling names.

2. For each of the Boolean monoids, what is the meaning of *reduce*? Are any of these functions predefined (under a different name)?

Exercise 6.3 (Programming: map-reduce, `MapReduce.lhs`). The type of lists forms a monoid: ϵ is the empty list [ ], and • is list concatenation ++. The type of *ordered* lists also forms a monoid: what are ϵ and •? Provide a suitable instance declaration. Can you use the monoid to implement a sorting function? Exercise 3.3 is also potentially useful.

Exercise 6.4 (Programming: map-reduce, `MapReduce.lhs`). In the lectures we have implemented *reduce* in terms of a higher-order function: *reduce* = *foldr* (•) ϵ. Of course, this is a rather arbitrary choice: *reduce* = *foldl* (•) ϵ works equally well. Since the operation is associative, it does not matter how nested applications of '•' are parenthesized. The overall result is bound to be the same. However, there is possibly a big difference in running time. For many applications, a balanced "expression tree" is actually preferable, see for example Exercise 6.3. In particular, as a balanced tree can in principle be evaluated in parallel!



Define a higher-order function

$$foldm :: (a \to a \to a) \to a \to ([a] \to a)$$

that constructs and evaluates a balanced expression tree, see also Exercise 4.3.

1. The types of *foldl*, *foldm*, and *foldr* are quite different. How come?

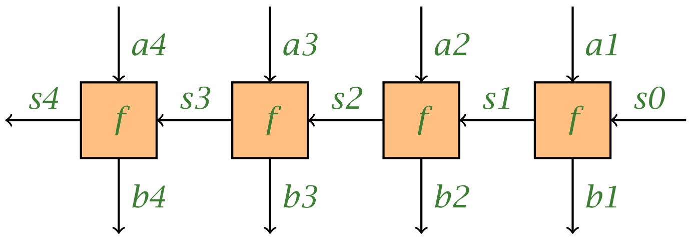
2. Implement *foldm* using a *top-down* approach: Split the input list into two halves, evaluate each half separately, and finally combine the results using the monoid operation (*divide and conquer*).

3. Implement *foldm* using a *bottom-up* approach: Traverse the input list combining two adjacent elements e.g. $[x1, x2, x3, x4]$ becomes $[x1 \bullet x2, x3 \bullet x4]$. Repeat the transformation until the list is a singleton list.

(In general, the task of finding a good expression tree is an optimization problem, which requires some knowledge of the run-time behaviour of the monoid operation e.g. matrix chain multiplication.)

Exercise 6.5 (Programming and hardware design, `MapReduce.lhs`). The art of map-reduce is to find a suitable monoid for the problem at hand. Framing a task as a composition of *map* and *reduce* then opens the door to a massively parallel implementation. The exploitation of parallelism is useful on a very large scale (the internet), but also on a very small scale (electronic circuits). This exercise looks into the latter continuing Exercise 5.6.

Reconsider the ripple carry adder:



At first sight, there seems to be little opportunity for parallelization. The computation of the carry is inherently sequential: it ripples through the gates from right to left (hence the name of the circuit). The leftmost full adder can only operate once all the others have finished their work. Perhaps surprisingly, the computation of the carry can be improved. (For simplicity, we only consider computing the final carry. To speed up hardware addition, all the intermediate carries are needed too. But this is mostly a matter of using a scan instead of a fold—a scan in hardware is called a *parallel prefix circuit*. The resulting adder is called *carry look-ahead adder*.)

Now, the first step is to concentrate on the carries, ignoring the sum bit (you will see in a moment why the function is called *kpg*):

```
kpg :: (Bit, Bit) → (Carry → Carry)
kpg s = \c → snd (fullAdder (s, c))
```

For each pair of summand bits (*s1*, *s2*), the partial application *kpg* (*s1*, *s2*) is a function that takes an input carry to an output carry. However, function composition is associative: invoking map-reduce

```
reduce ∘ map kpg :: [ (Bit, Bit) ] → (Carry → Carry)
```

we can map the given list of summand bits to a function that in turn maps the initial to the final carry. Problem solved!

Well, not quite: how can we possibly implement higher-order functions such as composition in hardware? (Even in Haskell, we haven't gained anything: the nested compositions can be evaluated in parallel, but each composition will create a new function at run-time, a so-called *closure*. When the resulting function is then applied to an initial carry, the evaluation will proceed purely sequential.)

How to make progress? An important observation is that there are not that many total functions of type *Carry* → *Carry* = *Bit* → *Bit*. Four, to be precise. If we partially evaluate *kpg* (*s1*, *s2*), we see that actually only three out of four occur:

```
kpg :: (Bit, Bit) → (Carry → Carry)
kpg (O, O) = \c → O   -- kill
kpg (O, I ) = \c → c   -- propagate
kpg (I,  O) = \c → c   -- propagate
kpg (I,  I ) = \c → I   -- generate
```

If both summand bits are O, the output carry is always O—the input carry is killed. Dually, if both summand bits are I, the output carry is always I—the output carry is generated. In the other two cases, the carry is propagated. The idea is to work with *representation* of functions instead of actual functions.

1. The three functions *const O*, *id*, and *const I* can be represented using a simple enumeration type (we keep the traditional names):

   ```
   data KPG = K | P | G
   ```

   Turn *KPG* into a monoid.

2. Test your implementation exhaustively. To this end define a function *apply* :: *KPG* → (*Carry* → *Carry*) that maps syntax to semantics. We expect the following properties to hold:

   $$\begin{aligned} apply\ \epsilon &= id \\ apply\ (a \bullet b) &= apply\ a \circ apply\ b \end{aligned}$$

   That is, ϵ represents the identity and $\bullet$ corresponds to function composition. (Mathematically speaking, *apply* is a *monoid homomorphism* from the *KPG* monoid to a transformation monoid.)

3. *Optional:* you may argue that using an enumeration type is still cheating. After all, we want to implement addition in silicon. Agreed. Represent *KPG* using two bits

   ```
   newtype KPG = KPG (Bit, Bit)
   ```

   and adapt the monoid instance to the new representation. Again, test the implementation exhaustively.

Exercise 6.6 (Worked example: digital sorting, `DigitalSorting.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 strings. For a more precise analysis, let us assume that comparing two elements, say, a_1 and a_2 takes time $\Theta(N_1 + N_2)$, where N_i is the size of a_i. In this case merge sort takes time $\Theta(N^2)$, where N is the size of the *entire input*. The purpose of this exercise is to show that we can do a lot better: we can sort in *linear* time if we employ the structure of search keys!

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

```
sortBy :: (a → a → Ordering) → [a] → [a]
```

The higher-order function *sortBy* can only call its first argument, but not inspect it: the ordering is treated as an oracle. As a benefit of the abstraction, there is one *generic* sorting function for all types.

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

```
class Rank key where
  sort :: [(key, val)] → [val]
  rank :: [(key, val)] → [[val]]
  sort  = concat ∘ rank
```

Actually, two methods are provided. Both take as input a so-called association list, a list of key-value pairs. The values are treated as *satellite data*: the methods are *parametric* in the value type *val*. The sorting function, *sort*, outputs the value components according to the given order on *key* without, however, returning the key components. The ranking function, *rank*, additionally groups the value components into non-empty runs of values associated with equal keys. The following examples illustrate their use, assuming suitable instances of *Rank* for characters and lists.

```
⟫⟫  sort [("ab",1),("ba",2),("aba",3),("ba",4)]
[1,3,2,4]
⟫⟫  rank [("ab",1),("ba",2),("aba",3),("ba",4)]
[[1],[3],[2,4]]
```

In the last example, we have three runs as there are three different strings in the input. The last run, `[2,4]`, contains the values associated with the key that occurs twice i.e. `"ba"`.

It may seem rather strange that the keys are dropped in the process. This has to do with a promise that we wish to make about the efficiency of *sort* and *rank*. Their running time should be linear in the *total* size of the keys. Even more: each component of each key must not be inspected more than once. The types have been chosen to make it hard (but not impossible) to violate these promises. Observe that there is actually no loss of generality: if we wish to retain the keys, we simply add them to the value component.

```
⟩⟩⟩⟩  kvs = [("ab",1),("ba",2),("aba",3),("ba",4)]
⟩⟩⟩⟩  rank [(k,(k,v)) | (k,v) ← kvs]
[[("ab",1)],[("aba",3)],[("ba",2),("ba",4)]]
```

We can use the generic comparison-based sorter to give an executable, but inefficient specification of the key-based sorter:

```
sort :: (Ord key) ⇒ [(key, val)] → [val]
sort kvs = map snd (sortBy (\kv1 kv2 → compare (fst kv1) (fst kv2)) kvs)
```

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 generic sorter does not and cannot make use of it. To see how we can take advantage of a white-box approach, 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! Sorting is just a matter of extracting the value components.

```
instance Rank () where
  sort kvs  = map snd kvs
  rank kvs = [map snd kvs | not (null kvs)]
```

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

1. Also specify the ranking function in terms of *compare*, using *sortBy* and *groupBy*. The executable specifications of *sort* and *rank* are actually quite useful: they can be used as (default) implementations if we need to quickly provide an instance of *Rank*.

2. Conversely, can you define *compare* in terms of *sort* or *rank*?

3. 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
   ```

   Define a corresponding instance of *Rank* i.e.

   ```
   instance (Rank key1, Rank key2) ⇒ Rank (key1, key2) where ..
   ```

4. 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. Again, define a corresponding instance of *Rank* i.e.

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

5. 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 *Rank*:

```
instance (Rank key) ⇒ Rank [key] where . .
```

What about other datatypes such as binary trees?

6. Re-implement (a variant of) the function *repeatedSegments* from the lectures, see "Case study: DNA analysis" §3.3.

```
repeatedSegments :: (Rank key) ⇒ Int → [key] → [[Integer]]
```

The call *repeatedSegments m* takes a list of keys as an input and returns the *positions* of repeated segments of length *m* e.g.

```
⟫⟫ dna
ATGTAAAGGGTCCAATGA
⟫⟫ repeatedSegments 3 dna
[[0, 14]]
```

The segment *ATG* of length 3 occurs at positions 0 and 14. To be able to test your implementation also provide:

```
instance Rank Base where . .
```

Hint: instances of *Rank* for small enumeration types typically use a technique called *sorting by distribution* also known as bucket sort.

Hints to practitioners 6. In Hint 5 we have touched upon GHC's language extensions. GHC is especially well-known for its many type- and class-system extensions. A useful one is

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

Recall that you can only provide one class instance per type. However, sometimes there is more than one candidate. For example, there are at least four possible instances for *Monoid Int*. This is a typical use-case for **newtype**: we introduce a new type for each instance i.e.

```
newtype Additive = Sum {fromSum :: Int}
  deriving (Eq, Ord, Show)
instance Monoid Additive where ...
```

As the keyword suggests, a **newtype** declaration introduces a new type, which is incompatible with all other existing types. *Slogan:* all data- and newtypes are born unequal. In particular, we cannot say 4711 + *Sum* 0815 as *Int* and *Sum* are different types. This is usually a welcome feature as the main purpose of a newtype is to erect an abstraction barrier—the other use-case for **newtype** is to define abstract datatypes. However, we also cannot say *Sum* 4711 + *Sum* 0815 as *Sum* has not inherited any instances from *Int*. This is where the language pragma *GeneralizedNewtypeDeriving* comes into play. It allows us to write

```
newtype Additive = Sum {fromSum :: Int}
  deriving (Eq, Ord, Show, Num)
```

freeing us from the arduous task of providing an instance of *Num* by hand. Now, *Sum* 4711 + *Sum* 0815 evaluates to *Sum* 5526. (The expression 4711 + *Sum* 0815 actually also type-checks as integer literals are overloaded i.e. 4711 is shorthand for *Sum* 4711.)

7 Reasoning and calculating

Exercise 7.1 (Warm-up: induction). Recall the implementation of insertion sort from the very first lecture (§0.4).

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

Show the partial correctness of *insert* and *insertionSort*:

$$\mathit{ordered}\ \mathit{xs} \implies \mathit{ordered}\ (\mathit{insert}\ x\ \mathit{xs})$$
$$\mathit{ordered}\ (\mathit{insertionSort}\ \mathit{xs})$$

Why *partial* correctness? Well, we do not capture that the output list is a permutation of the input list. (The definitions *insert x xs* = [ ] and *insertionSort xs* = [ ] trivially satisfy the specification above.) In case you feel adventurous: how would you capture the latter property?

Exercise 7.2 (Pencil and paper: induction).

1. We have emphasized in the lectures that every datatype comes with a pattern of induction. What is the induction scheme for *Tree elem* i.e. for binary trees?

2. Define functions that count the number of inner nodes (*Node*) and the number of outer nodes (*Empty*). Prove by induction that the latter number is one more than the former. (Put differently, show that a binary tree of size n has $n + 1$ leaves.)

3. Show by induction

$$2^{\mathit{minHeight}\ t} - 1 \leqslant \mathit{size}\ t \leqslant 2^{\mathit{maxHeight}\ t} - 1$$

Exercise 7.3 (Pencil and paper: program derivation). The implementation of *inorder*

```
inorder :: Tree elem → [elem]
inorder Empty        = [ ]
inorder (Node l a r) = inorder l ++ [a] ++ inorder r
```

has a quadratic running time because of the repeated invocations of list concatenation—recall that the running time of $+\!\!+$ is linear in the length of its first argument. To improve the running time we solve a more complicated task (see also *reverseCat* and Exercise 7.5 below):

$$inorderCat\ t\ xs = inorder\ t +\!\!+ xs \tag{1}$$

At first sight, it may seem slightly paradoxical that a more difficult problem is actually easier i.e. more efficient to solve. This is why this technique is known as *inventor's paradox* or, more prosaically, as *strengthening the induction assumption*. The important observation is that while the problem is more difficult, the recursive call (or the induction assumption) is also more powerful—remember, you only have to program a step (or prove an implication)!

1. Derive an implementation of *inorderCat* from the specification above (1) and then define *inorder* in terms of *inorderCat*.

2. Test the resulting program on a few test cases. (You may want to define a function $skewed :: Integer \to Tree\ ()$ that generates a left-skewed tree of the given height.) Is the new implementation of *inorder* actually more efficient?

3. Repeat the exercise for *preorder* and *postorder*.

4. What is the relation of the derivation to an inductive proof? Put differently, can you rearrange the derivation into an inductive proof?

5. Finally, what's the relation to Hughes' lists, see §6.1 and §6.2.

Exercise 7.4 (Pencil and paper: program derivation). Like *foldr* captures a common recursion scheme (canned recursion), *foldr* fusion captures a common induction scheme (canned induction). The purpose of this exercise is to train the use of canned induction.

1. Prove the *foldr*-*map* fusion law:

$$foldr\ (\triangleright)\ e \circ map\ f = foldr\ (\backslash a\ b \to f\ a \triangleright b)\ e$$

   Of course, you should *not* use induction. Instead, apply *foldr* fusion making use of the fact that *map* can be defined in terms of *foldr*.

2. Use *foldr*-*map* fusion to prove the second functor law:

   $$map\ (f \circ g) = map\ f \circ map\ g$$

   (There is actually a second option: alternatively, you can establish the functor law using *foldr* fusion.)

3. Use *foldr*-*map* fusion to prove the 'bookkeeping law':

   $$reduce \circ concat = reduce \circ map\ reduce$$

   Here we assume that $reduce = foldr\ (\bullet)\ \epsilon$. In what sense does the bookkeeping law generalize the second homomorphism condition? *Hint:* specialize the law to 2-element lists.

Exercise 7.5 (Worked example: program derivation, `Text.lhs`). Haskell's *show* method converts a value into its string representation. The approach is, however, a tad simplistic and the resulting string is usually not fit for human consumption. To illustrate, consider testing the function *balanced* of Exercise 4.3. Showing *balanced* $[1\,..\,6]$ produces:

```
Node (Node Empty 1 (Node Empty 2 Empty)) 3 (Node (Node
Empty 4 Empty) 5 (Node Empty 6 Empty))
```

The purpose of this exercise is to implement a small library for *pretty printing* that allows us to emphasize the *structure* of data. By contrast, pretty printing *balanced* $[1\,..\,6]$ produces:

```
Node
    Node
        Empty
        1
        Node
            Empty
            2
            Empty
    3
    Node
        Node
            Empty
            4
            Empty
        5
```

```
Node
    Empty
    6
    Empty
```

The nesting level is nicely indicated through indentation.

The pretty printing interface is minimalist by design: it introduces a datatype of "text with indentation" and a handful of operations.

```
data Text
infixr 5 ◇
text   :: String → Text   — without '\n'
nl     :: Text
indent :: Int → Text → Text
(◇)    :: Text → Text → Text
render :: Text → String
```

The function *text* converts a string into a text where the string must not contain any newline characters. Line breaks are introduced explicitly using *nl*. The function call *indent i t* inserts *i* spaces *after* each line break in *t*. The operator ◇ is the counterpart of ++: it concatenates two pieces of text. Finally, *render* transforms a text back to a string. The pretty printer for binary trees illustrates the use of the combinators.

```
prettyTree :: (Show elem) ⇒ Tree elem → Text
prettyTree Empty
  = text "Empty"
prettyTree (Node l a r)
  = indent 4 (text "Node"   ◇ nl ◇
              prettyTree l  ◇ nl ◇
              text (show a) ◇ nl ◇
              prettyTree r)
```

The string "Node", the left sub-tree, the root element, and the right sub-tree are shown on separate lines using an indentation of 4 spaces.

Let us nail down the semantics of the combinators by providing a reference implementation. The implementation is actually interesting in its own right as it uses the *interpreter design pattern*. Each operation that *produces* text is implemented by a constructor i.e. the operation does (almost) nothing.

```
data Text = Text String    -- without '\n'
          | Nl
          | Indent Int Text
          | Text :<> Text
          deriving (Show)

text   = Text
nl     = Nl
indent = Indent
(<>)   = (:<>)
```

There is only one active operation: *render*, which *consumes* a text. It can be seen as an interpreter, which interprets the other “commands”.

```
render (Text s)     = s
render (Nl)         = "\n"
render (Indent i d) = tab i (render d)
render (d1 :<> d2)  = render d1 ++ render d2
```

The helper function *tab i s* inserts *i* spaces *after* each newline in *s*.

```
tab :: Int -> String -> String
tab _i ""      = ""
tab i (c : s)
  | c == '\n'  = c : replicate i ' ' ++ tab i s
  | otherwise  = c : tab i s
```

An alternative or, perhaps, complimentary approach to providing a reference implementation is to list algebraic properties of the combinators. For example, we expect that *text* "" and ⋄ form a monoid and that *render* is a monoid homomorphism. Table 2 provides a comprehensive list of properties. Is the list exhaustive? These properties are, in particular, useful for optimizing pretty printers e.g. Equation (8) applied from left to right replaces two indentations by a single one. (As an aside, the reference implementation does *not* satisfy all of these laws. Many of them are only valid *under observation*: instead of $d1 = d2$ we only have $render\ d1 = render\ d2$.)

1. The reference implementation is quite slow. (Do you see why?) Like in Exercise 7.3, we can improve the performance by solving a more complicated task (applying the *inventor's paradox*):

$$renderWith\ doc\ i\ x = render\ (indent\ i\ doc) +\!\!+\ x \qquad (13)$$

$$\mathit{text}\ \texttt{""} \diamond d = d \tag{2}$$
$$d \diamond \mathit{text}\ \texttt{""} = d \tag{3}$$
$$d1 \diamond (d2 \diamond d3) = (d1 \diamond d2) \diamond d3 \tag{4}$$
$$\mathit{indent}\ 0\ d = d \tag{5}$$
$$\mathit{indent}\ i\ (\mathit{text}\ s) = \mathit{text}\ s \tag{6}$$
$$\mathit{indent}\ i\ \mathit{nl} = \mathit{nl} \diamond \mathit{text}\ (\mathit{replicate}\ i\ \texttt{' '}) \tag{7}$$
$$\mathit{indent}\ i\ (\mathit{indent}\ j\ d) = \mathit{indent}\ (i + j)\ d \tag{8}$$
$$\mathit{indent}\ i\ (d1 \diamond d2) = \mathit{indent}\ i\ d1 \diamond \mathit{indent}\ i\ d2 \tag{9}$$
$$\mathit{render}\ (\mathit{text}\ s) = s \tag{10}$$
$$\mathit{render}\ \mathit{nl} = \texttt{"\textbackslash n"} \tag{11}$$
$$\mathit{render}\ (d1 \diamond d2) = \mathit{render}\ d1 +\!\!+ \mathit{render}\ d2 \tag{12}$$

Table 2: Properties of the pretty printing combinators.

The specified function *renderWith* does several things at a time: it indents a text, transforms the indented text into a string, and then appends another string to the result.

Derive an implementation of *renderWith* from the specification above (13) and then define *render* in terms of *renderWith*. Make use of the properties listed in Table 2. (Do you need all of them?)

2. An advantage of the using the *interpreter design pattern* is that we can easily debug pretty printers: showing *pretty* (*balanced* [1 . . 6]) produces:

```
Indent 4 (Text "Node" :◇ (Nl :◇ (Indent 4 (Text "Node" :◇ (Nl :◇
(Text "Empty" :◇ (Nl :◇ (Text "1" :◇ (Nl :◇ Indent 4 (Text "Node"
 :◇ (Nl :◇ (Text "Empty" :◇ (Nl :◇ (Text "2" :◇ (Nl :◇ Text
"Empty")))))))))))) :◇ (Nl :◇ (Text "3" :◇ (Nl :◇ Indent 4 (Text
"Node" :◇ (Nl :◇ (Indent 4 (Text "Node" :◇ (Nl :◇ (Text "Empty"
 :◇ (Nl :◇ (Text "4" :◇ (Nl :◇ Text "Empty")))))) :◇ (Nl :◇ (Text
"5" :◇ (Nl :◇ Indent 4 (Text "Node" :◇ (Nl :◇ (Text "Empty" :◇
(Nl :◇ (Text "6" :◇ (Nl :◇ Text "Empty")))))))))))))))))
```

OK, perhaps we should pretty print the pretty printing combinators. Implement a pretty printer for the type *Text*. (*For geeks:* to debug the pretty printer for *Text* we can pretty print its output. But the result is again an element of *Text*, so we can apply the pretty printer a second time. It's fun to iterate this process a few more times . . .)

3. A slight disadvantage of the using the *interpreter design pattern* is the interpretive overhead. Remove the original definitions of *text*, *nl*, *indent*, and $\diamond$, which define the operations as constructors. Instead, implement the operations by actual functions. Derive the implementations from the following specifications:

$$\begin{aligned} \mathit{text}\ s &= \mathit{renderWith}\ (\mathit{Text}\ s) && (14)\\ \mathit{nl} &= \mathit{renderWith}\ (\mathit{Nl}) && (15)\\ \mathit{indent}\ i\ (\mathit{renderWith}\ d) &= \mathit{renderWith}\ (\mathit{Indent}\ i\ d) && (16)\\ \mathit{renderWith}\ d1 \diamond \mathit{renderWith}\ d2 &= \mathit{renderWith}\ (d1 :\!\diamond\ d2) && (17) \end{aligned}$$

4. *Optional:* design a type class *Pretty*, a replacement for *Show*, that builds on the pretty printing library.

Exercise 7.6 (Programming and program derivation). Here are some problems to work on, just in case you need some inspiration for the seasonal break.

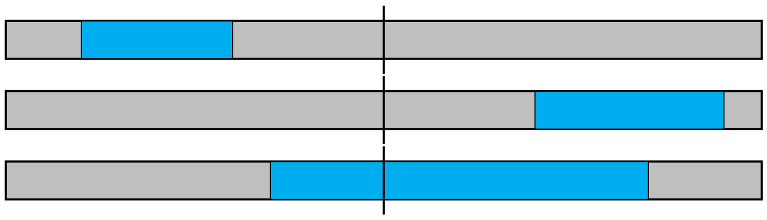
1. In the lectures we have tackled the *maximum segment sum* problem. We have derived a program from an executable, but inefficient specification. Can you use a similar approach to solve the maximum segment *product* problem?

   ```
   maximumSegmentProduct :: [Integer] → Integer
   maximumSegmentProduct = maximum ∘ map product ∘ segments
   ```

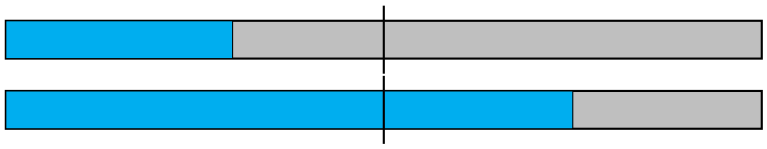
   What's the problem? Carefully study the derivation given in the lectures: which properties of maximum ($\uparrow$) and sum ($+$) are actually needed? Can you abstract away from these two operators?

2. The solution to the maximum segment sum problem that we derived in the lectures runs in linear time. However, it seems to be inherently sequential as the argument to *scanr* is not associative. Can you frame the problem as an instance of map-reduce so that it is easily parallelizable? Have a go! *Remember:* the art of map-reduce is to find a suitable monoid.

Hint: apply the *inventor's paradox* solving a more complicated problem. To see what's needed, consider a division step (map-reduce is an instance of divide-and-conquer). There are three possible cases: the maximum segment sum lies entirely in the left half, entirely in the right half, or it crosses the dividing line.



Thus, to calculate the maximum *segment* sum, we also require the maximum *prefix* sum and the maximum *suffix* sum. Now, consider computing the maximum prefix sum. There are only two cases: the maximum prefix lies entirely in the left half, or it crosses the dividing line.



Thus, to compute the maximum prefix sum, we also require the overall sum, the total. Overall, we need to compute four values simultaneously:

(a) maximum segment sum;

(b) maximum prefix sum;

(c) maximum suffix sum;

(d) overall sum.

3. In the lectures we initially considered the *maximum profit problem*. The maximum segment sum problem was only the result of transforming the original problem. But is the transformation really necessary? How would you specify the original problem? Can you derive a program from the specification?

4. The original problem only allows for a *single* transaction: you buy one unit of stock and then sell it at a later point in time. Consider allowing multiple, *non-overlapping* transactions.

Hints to practitioners 7. Equational proofs and derivations shown in textbooks are often the result of a lot of polishing—like the proofs in the lectures. It has been said that proving is not a spectator sport. So perhaps you appreciate some advice on conducting proofs.

To show the equation $f = g$ one typically starts at both ends, and tries to meet in the middle. First apply obvious rewrites such as plugging in definitions. Perhaps you can identify an intermediate goal such as applying the induction assumption or e.g. Horner's rule. Try to systematically work towards this intermediate goal. When the proof is finalized, double check whether you have used all of the assumptions. If this is not the case, there is some chance that either the proof is wrong or the theorem is actually more general.

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:

```
maybe     :: b → (a → b) → Maybe a → b
catMaybes :: [Maybe a] → [a]
```

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:

```
empty  :: Map k v
insert :: (Ord k) ⇒ k → v → Map k v → Map k v
update :: (Ord k) ⇒ k → Map k v → Maybe v
```

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.