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.

```haskell
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* $\Longrightarrow$ 33. (In case you wonder why some variables have a leading underscore see Hint 1.)

   ```haskell
   age :: Person -> Age
   age (_n, a, _c) = a
   ```

   Define functions

   ```haskell
   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"` $\Longrightarrow$ `"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 $increaseAge :: Person \to Person$ 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]`
   ⟫⟫ $map\ (\backslash p \to (age\ p, name\ p))\ students$
   `[(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.

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

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

Recall that Haskell has a very simple computational model: an expression is evaluated by repeatedly replacing equals by equals. Evaluate the expression $\mathit{insertionSort}\ (7 : (9 : (2 : [\,])))$—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.

$$\mathit{twice}\ f\ x = f\ (f\ x)$$

(Like *map* and *filter*, it is an example of a higher-order function as it takes a function as an argument.) Evaluate $\mathit{twice}\ (+1)\ 0$ and $\mathit{twice}\ \mathit{twice}\ ({*}2)\ 1$ 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.

$$\mathit{twice} = \backslash f \to \backslash x \to f\ (f\ x)$$

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

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

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 $twice\ twice\ (*2)\ 1 = ((twice\ twice)\ (*2))\ 1$). 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. $sin\ pi$, 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: $Integer \to Integer \to Integer$ means $Integer \to (Integer \to Integer)$.) 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. $1 \otimes 0 \otimes 1 \otimes 1 \implies 11$ and $(1 \otimes 1 \otimes 0) + 4711 \implies 4717$. 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 $Bool$.

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

   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?

   $$
   \begin{array}{l}
   (abs\circ) \circ (\circ negate) \\
   (div\circ) \circ (\circ mod)
   \end{array}
   $$

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

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

   $$
   \begin{array}{l}
   i\ x = x \\
   k\ (x, y) = x \\
   b\ (x, y, z) = (x\ z)\ y \\
   c\ (x, y, z) = x\ (y\ z) \\
   s\ (x, y, z) = (x\ z)\ (y\ z)
   \end{array}
   $$

   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.

$\rangle\rangle\rangle\rangle$: *info map*
$map :: (a \to b) \to [a] \to [b]$ — Defined in 'GHC.Base'
$\rangle\rangle\rangle\rangle$: *type* $map \circ map$
$map \circ map :: (a \to b) \to [[a]] \to [[b]]$

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 $allTrue :: [Bool] \to Bool$ 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.

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

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

that returns a list of runs such that $concat \circ runs = id$, 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 $contains :: Segment \to DNA \to Bool$ 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 $longestOnlyAs :: DNA \to Integer$ that computes (the length of) the longest segment that contains only the base A.

3. Define a function $longestAtMostTenAs :: DNA \to Integer$ 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, $A \to B \to C$ 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.

$$\begin{aligned}
&\mathbf{type}\ Probes\ a &&= [\,a\,] \\
&\mathbf{type}\ Property\ a &&= a \to Bool
\end{aligned}$$

Given $pa :: Probes\ A$, $pb :: Probes\ B$, and $pc :: Property\ C$, the expression $pa \dashrightarrow pb \dashrightarrow pc$ is then a test procedure for functions of type $A \to B \to C$. Applied to f i.e. $(pa \dashrightarrow pb \dashrightarrow pc)\ f$ 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

$$(\mathit{permutations}\ [0\,..\,9] \dashrightarrow \mathit{ordered})\ \mathit{niftySort}$$

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 $\mathit{niftySort}\ \mathit{xs} = [\,]$. A better approach is to check his implementation against a *trusted* sorting algorithm. This is accomplished using a variant of $\dashrightarrow$:

$$(\mathit{permutations}\ [0\,..\,9] \Longrightarrow \backslash \mathit{inp}\ \mathit{res} \to \mathit{trustedSort}\ \mathit{inp} == \mathit{res})\ \mathit{niftySort}$$

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

Turning to the implementation, $\dashrightarrow$ and $\Longrightarrow$ 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 $\mathit{runs} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [[a]]$ 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 $\mathit{isIntegerSqrt} :: \mathit{Property}\ (\mathit{Integer} \to \mathit{Integer})$ to exercise the program. Can you actually figure out how it works?

5. Define a combinator

$$\mathbf{infixr}\ 4 \otimes$$
$$(\otimes) :: \mathit{Probes}\ a \to \mathit{Probes}\ b \to \mathit{Probes}\ (a, b)$$

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.

$\rangle\rangle\rangle\rangle$ $\mathit{bools} = [\mathit{False}, \mathit{True}]$
$\rangle\rangle\rangle\rangle$ $\mathit{bools} \otimes \mathit{bools}$
$[(\mathit{False}, \mathit{False}), (\mathit{False}, \mathit{True}), (\mathit{True}, \mathit{False}), (\mathit{True}, \mathit{True})]$
$\rangle\rangle\rangle\rangle$ $\mathit{bools} \otimes \mathit{bools} \otimes \mathit{bools}$
$[(\mathit{False}, (\mathit{False}, \mathit{False})), (\mathit{False}, (\mathit{False}, \mathit{True})), (\mathit{False}, (\mathit{True}, \mathit{False})), (\mathit{False}, (\mathit{True}, \mathit{True})),$
$(\mathit{True}, (\mathit{False}, \mathit{False})), (\mathit{True}, (\mathit{False}, \mathit{True})), (\mathit{True}, (\mathit{True}, \mathit{False})), (\mathit{True}, (\mathit{True}, \mathit{True}))]$
$\rangle\rangle\rangle\rangle$ $\mathit{chars} = \texttt{"Ralf"}$
$\rangle\rangle\rangle\rangle$ $\mathit{bools} \otimes \mathit{chars}$
$[(\mathit{False}, \texttt{'R'}), (\mathit{False}, \texttt{'a'}), (\mathit{False}, \texttt{'l'}), (\mathit{False}, \texttt{'f'}), (\mathit{True}, \texttt{'R'}),$
$(\mathit{True}, \texttt{'a'}), (\mathit{True}, \texttt{'l'}), (\mathit{True}, \texttt{'f'})]$

If as contains m elements, and bs contains n elements, then $\mathit{as} \otimes \mathit{bs}$ 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 $\mathit{length}\ (\mathit{unwords}\ \mathit{ws}) \leqslant n$. Define a function

$$\mathit{format} :: \mathit{Int} \to [\mathit{Word}] \to [[\mathit{Word}]]$$

that given a maximal line width and a list of words returns a list of fitting lines so that $\mathit{concat} \circ \mathit{format}\ n = \mathit{id}$. Is this always possible? (As an aside, a function f with the property $\mathit{concat} \circ f = \mathit{id}$ 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:

$\rangle\rangle\rangle\rangle$ $\mathit{putStr}\ \$\ \mathit{unlines}\ \$\ \mathit{map}\ \mathit{unwords}\ \$\ \mathit{format}\ 40\ \$\ \mathit{words}\ \mathit{lorem}$

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 $[\mathit{Base}]$ with the singleton list $[\mathit{base}]$. The first is a type; the second is a value, a shorthand for $\mathit{base} : [\,]$. 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.

$$
\begin{array}{l}
\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \to [a] \to [a] \\
\mathit{insert}\ a\ [\,] \qquad = [a] \\
\mathit{insert}\ a\ (b : \mathit{xs}) \\
\quad \mid a \leqslant b \qquad = a : b : \mathit{xs} \\
\quad \mid \mathit{otherwise} = b : \mathit{insert}\ a\ \mathit{xs}
\end{array}
$$

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.

$$\mathit{insert} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow \mathit{elem} \to [\mathit{elem}] \to [\mathit{elem}]$$

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.

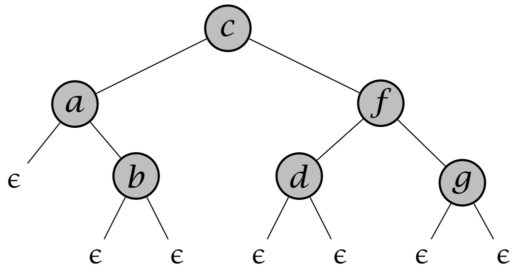


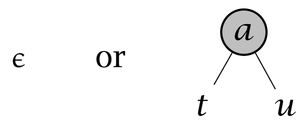
Figure 1: A binary tree of size 6.

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 $preorder, inorder, postorder :: Tree\ elem \to [elem]$ 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 $layout :: (Show\ elem) \Rightarrow Tree\ elem \to String$ 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.)

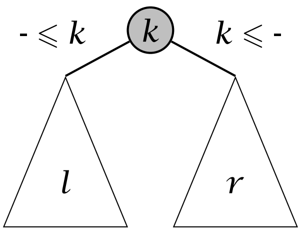
Exercise 4.3 (Programming, `BinaryTree.lhs`). 1. Define a function $build :: [elem] \to Tree\ elem$ that constructs a binary tree from a given list of elements such that $inorder \circ build = id$. The shape of the tree does not matter. (Apply the list design pattern.)

2. Define a function $balanced :: [elem] \to Tree\ elem$ 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 $\mathit{create} :: \mathit{Int} \to \mathit{Tree}\ ()$ can construct a tree of size n in logarithmic time i.e. $\mathit{create}\ n$ 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.

$\mathit{registry} :: \mathit{Tree}\ \mathit{String}$
$\mathit{registry} = \mathit{Node}\ (\mathit{Node}\ (\mathit{Node}\ \mathit{Empty}\ \texttt{"Frits"}\ \mathit{Empty})\ \texttt{"Peter"}\ \mathit{Empty})\ \texttt{"Ralf"}\ \mathit{Empty}$

1. Define a function $\mathit{member} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow \mathit{elem} \to \mathit{Tree}\ \mathit{elem} \to \mathit{Bool}$ 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 $\mathit{insert} :: (\mathit{Ord}\ \mathit{elem}) \Rightarrow \mathit{elem} \to \mathit{Tree}\ \mathit{elem} \to \mathit{Tree}\ \mathit{elem}$ that inserts an element into a search tree e.g.

   $\rangle\rangle\rangle\ \ \mathit{insert}\ \texttt{"Nienke"}\ \mathit{registry}$
   $\mathit{Node}\ (\mathit{Node}\ (\mathit{Node}\ \mathit{Empty}\ \texttt{"Frits"}\ (\mathit{Node}\ \mathit{Empty}\ \texttt{"Nienke"}\ \mathit{Empty}))$
   $\quad \texttt{"Peter"}\ \mathit{Empty})\ \texttt{"Ralf"}\ \mathit{Empty}$

   (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 $delete :: (Ord\ elem) \Rightarrow elem \to Tree\ elem \to Tree\ elem$ 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 $isSearchTree :: (Ord\ elem) \Rightarrow Tree\ elem \to Bool$ 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 $trees :: [elem] \to Probes\ (Tree\ elem)$ that generates *all* trees whose inorder traversal yields the given list of elements: $and\ [inorder\ t == xs \mid t \leftarrow trees\ xs]$. Applied to an ordered list, $tree$ will then generate search trees. (For the mathematically inclined: which sequence does $[length\ (trees\ [1\,..\,i]) \mid i \leftarrow [0\,..]]$ 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.

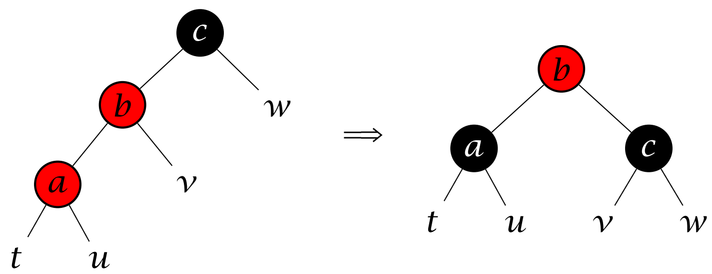
1. Adapt the membership test to red-black trees.

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

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 $\lambda\ x \rightarrow x$, *Const r* represents $\lambda\ x \rightarrow r$, *Prim Sin* :+: *Const* 2 :∗: *Id* represents $\lambda\ x \rightarrow \mathit{sin}\ x + 2 * x$. In general, if *e1* represents *f1* and *e2* represents *f2*, then *e1*:+:*e2* represents the function $\lambda\ x \rightarrow f1\ x + f2\ x$.

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:

$\rangle\rangle\rangle\rangle$ $\mathit{derive}\ (\mathit{Const}\ 1 :+: \mathit{Const}\ 2 :*: \mathit{Id})$
$\mathit{Const}\ (0 \% 1) :+: (\mathit{Const}\ (0 \% 1) :*: \mathit{Id} :+: \mathit{Const}\ (2 \% 1) :*: \mathit{Const}\ (1 \% 1))$
$\rangle\rangle\rangle\rangle$ $\mathit{derive}\ (\mathit{Id} :\circ: \mathit{Id} :\circ: \mathit{Id})$
$\mathit{Const}\ (1 \% 1) :\circ: (\mathit{Id} :\circ: \mathit{Id}) :*: (\mathit{Const}\ (1 \% 1) :\circ: \mathit{Id} :*: \mathit{Const}\ (1 \% 1))$

Implement a function $\mathit{simplify} :: \mathit{Function} \to \mathit{Function}$ 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.

$$\begin{aligned}&\mathit{swap} :: (a, b) \to (b, a)\\&\mathit{swap}\ (x, y) = (y, x)\end{aligned}$$

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?

$$\begin{aligned}&\mathit{swap}\ (\mathit{firstComponent}, \mathit{secondComponent})\\&\quad = (\mathit{secondComponent}, \mathit{firstComponent})\end{aligned}$$

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?

$$\begin{aligned}&\mathit{swap} :: (\mathit{typeOfFirstComponent}, \mathit{typeOfSecondComponent})\\&\qquad\qquad \to (\mathit{typeOfSecondComponent}, \mathit{typeOfFirstComponent})\end{aligned}$$

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.

$$\textbf{type}\ \mathbb{Z} = \mathit{Integer}$$
$$\mathit{sort} = \mathit{insertionSort}$$

(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.

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:

```haskell
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.

```haskell
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 $send :: POP3 \to IO\ ()$ and $receive :: IO\ POP3$ within your module, then these could be exported explicitly by listing them in the module declaration:

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

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

A.2 Importing modules

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

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

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

import *Data.Maybe*

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

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

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

A.3 Qualifying and hiding imports

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

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

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

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

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

import *qualified Data.Map*

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

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

import *qualified Data.Map as M*

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

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

import *Prelude hiding* (*lookup*)

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