

1 Programming with expressions and values

Exercise 1.1. Describe what each of the following Haskell functions mean:

```
f1      :: Int
f1      = 1 + 5
```

```
f2      :: Int → Int → Int
f2 m n
| m < n  = m
| otherwise = n
```

```
f3      :: String → Int → String
f3 s n
| n ≤ 0   = ""
| otherwise = s ++ f3 s (n-1)
```

```
f4      :: Int → Int → Int
f4 x 0   = x
f4 x y   = f4 y (x mod y)
```

```
f5      :: (Int,Int) → Int
f5 x     = fst x + snd x
```

```
f6      :: (a,b) → (b,a)
f6 (a,b) = (b,a)
```

```
f7      :: (a,a) → (a,a)
f7 x     = f6 (f6 x)
```

Exercise 1.2. Determine the result of the expressions below by rewriting. First add parentheses `()` in the expressions illustrating the priorities of the operations correctly. Then apply rewrites as shown in the lecture: place every rewrite step on a new line, and underline the part of the expression that you rewrite (the *redex*, *reducible expression*).

```
e1 = 42
e2 = 1 + 125 * 8 / 10 - 59
e3 = not True || True && False
e4 = 1 + 2 == 6 - 3
e5 = "1 + 2" == "6 - 3"
e6 = "1111 + 2222" == "1111" ++ " + " ++ "2222"
```

Exercise 1.3 (Warm-up: definitions, `Database.hs`). Consider the following definitions, which introduce a type of persons and some sample data.

```
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 the hint)

```
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" = "hello, world"`. *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:

- increment the age of all students by two;
- promote all of the students (attach "dr " to their name);
- find all students named Frits;
- find all students whose favourite course is Functional Programming;
- find all students who are in their twenties;
- find all students whose favourite course is Functional Programming and who are in their twenties;
- find all students whose favourite course is Imperative Programming or who are in their twenties.

Exercise 1.4 (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.

```
insert 7 (2 : (9 : []))
⇒ {definition of insert and 7 > 2}
2 : (insert 7 (9 : []))
⇒ {definition of insert and 7 ≤ 9}
2 : (7 : (9 : []))
```

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 1.5 (Pencil and paper: λ -expressions). 1. An alternative definition of `twice` builds on λ -expressions.

```
twice' = \ f → \ x → 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).

```
(\ x → body) arg ⇒ 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.

```
(\ x → x + x) 47 ⇒ x + x { x := 47} ⇒ 47 + 47 ⇒ 94.
```

- 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 1.6 (Worked example: prefix and infix notation).

- 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!
- Infix notation is inherently ambiguous: `x ⊗ y ⊗ z`. What does this mean: `(x ⊗ y) ⊗ z` or `x ⊗ (y ⊗ 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: `"F" ++ "P" ++ "1" = "F" ++ ("P" ++ "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: $\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$ means $\text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})$.) 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

```
infixl **
a ** b = 2 * a + b
```

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

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

```
infixl 7 *
infixl 6 +
```

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

```
f x ≥ 0 && a || g x y * 7 + 10 == b - 5
```

Exercise 1.7 (Programming). Define the string

```
thisOldMan :: String
```

that produces the following poem (if you type `putStr 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 1.8 (Programming, `Shapes.hs`). 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 equations, 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.)

Exercise 1.9. The *greatest common divisor* function is defined in roughly the following way:

```

gcd :: Int → Int → Int
gcd x y = gcd' (abs x) (abs y)
    where gcd' a 0 = a
          gcd' a b = gcd' b (a `rem` b)

```

Rewrite the following expressions using this definition:

```

gcd -42 0
gcd 0 -42
gcd 18 42
gcd 123456789 987654321

```

Exercise 1.10 (`Strings`, `Strings.hs`). The standard environment of Haskell provides various functions for working with texts of type `String`. In this exercise you are only allowed to use the following functions as `String` operations:

- `length`: returns the number of characters of the given `String`.
Example: `length ""` returns 0.
Example: `length "0123456789"` returns 10.
- `!!`: returns the `Char` value on the given `Int` index in a string. The indices of a non-empty string `s` go from 0 up to and including `(size s)-1`. The empty string `""` does not have any valid indices.
Example: `"0123456789" !! 4` returns `'4'`.
Example: `"0123456789" !! -1` returns `*** Exception: Prelude.!!: negative index`.
Example: `"0123456789" !! 10` returns `*** Exception: Prelude.!!: index too large`.

Using only the above functions, write the following ones:

1. Write your own implementation of `head :: String → Char` and `tail :: String → String` that given a non-empty `String` return the first and the remaining elements of the `String` respectively. If the argument is the empty `String`, it should show an error.
Example: `head "Madam, I'm Adam" = 'M'`.
Example: `tail "Madam, I'm Adam" = "adam, I'm Adam"`.
2. Write a function `is_equal` that determines the equality of two `String` arguments. This means that for every pair s_1, s_2 of type `String` the following should hold: `is_equal s1 s2 = s1 == s2`.
3. Write a function `is_substring` that returns a `Bool` when given two `String` arguments. The result should be `True` if and only if the first argument is a *substring* of the second argument.
Example: `is_substring "there" "Is there anybody out there?"`¹ returns `True`, after all: "Is there anybody out there".
Example: `is_substring "there" "Just for the record"`² returns `False` because there is a space between `the` and `re`.
4. Write a function `is_sub` that returns a `Bool` when given two `String` arguments. The results should be `True` if and only if the characters of the first argument appear in the same order in the second argument. Characters of the second argument may be skipped.
Example: `is_sub "there" "Just for the record"` returns `True` because the space will be skipped.
Example: `is_sub "she and her" "Is there anybody in there?"` returns `True`, after all: "Is there anybody in there".
Example: `is_sub "There there" "Is there anybody in there?"`³ returns `False` because `T` is no part of the second `String`.
5. Write a recursive function `is_match` that returns a `Bool` when given two `String` arguments (*pattern* and *source*). The function determines whether the *pattern* can be applied on the source (the pattern *matches* the source). The *pattern* may contain *wildcard* characters:
 - `.`: this matches the next, arbitrary single character in the *source*.
 - `*`: this matches zero or more consecutive characters in the *source*.

All other characters in the *pattern* have to match exactly with the corresponding characters in the *source*.

Example: `is_match "here" "here"` returns `True`.

Example: `is_match "here" " here"` returns `False` because the first characters of the two strings do not match.

Example: `is_match "here" "here "` returns `False` because the second text has one extra character.

Example: `is_match ".*" "AB"` returns `True` because the second text consists of exactly

¹Pink Floyd – The Wall (1979)

²Marillion – Clutching at straws (1987)

³Radiohead – Hail to the thief (2003)

two characters.

Example: `is_match "?"` "Is there anybody in there?" returns **True** because the second text ends with a `'?'` character.

Example: `is_match "*?!*?!"` "Answers? Questions! Questions? Answers!" ⁴ returns **True**.

Example: `is_match ".*here*.here*."` "Is there anybody in there?" returns **True**.

Example: `is_match ".here.here."` "Is there anybody in there?" returns **False**.

Exercise 1.11 (Determining prime numbers). Write the *recursive* function

`isPrime :: Int → Bool`

that determines whether the argument is a prime number. A prime number is a positive integer divisible by exactly two *different* positive integers; namely itself and 1 (hence the reason why 1 is not a prime itself). Test the function with `[x | x ← [1 .. 1000], isPrime x]`. This should return all primes in the range of one up to and including a thousand. The result should be:

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]

Note: Use the function `rem` for integer division with remainder. The function `mod` is practically an alias of this function.

Exercise 1.12 (Prime factors). Every integer $x > 1$ can be decomposed in prime factors in a canonical way. This means that you can find a range of prime numbers $p_0 \dots p_k$ where $k \geq 0$ for which $p_i < p_j$ for all $i < j$ and as many positive numbers $n_0 \dots n_k$ in such a way that:

$$x = p_0^{n_0} \cdot \dots \cdot p_k^{n_k} = \prod_{i=0}^k p_i^{n_i}$$

Examples:

36	=	$2^2 \cdot 3^2$
52	=	$2^2 \cdot 13$
133	=	$7 \cdot 19$
123456789	=	$3^2 \cdot 3607 \cdot 3803$

Write the function `primeFactors :: Int → String` that decomposes an integer into its prime factors. The prime factors found should be glued together in the resulting **String**.

⁴Focus – Focus III (1973)

Examples: `primeFactors 52` = `"2*2*13"`
`primeFactors 36` = `"2*2*3*3"`
`primeFactors 133` = `"7*19"`
`primeFactors 123456789` = `"3*3*3607*3803"`

Use the function `isPrime` from one of the previous exercises.

The following operations on **Strings** and **Ints** are useful: `s1 ++ s2` glues **String** `s2` to **String** `s1`; `toString n` creates a **String** value from an **Int**.

Exercise 1.13 (Adding numbers). Write the function `numberSum :: Int → Int` that sums the digits of a positive integer.

Example: `numberSum 9876543` = 9+8+7+6+5+4+3 = 42.

Example: `numberSum 1000` = 1+0+0+0 = 1.

Exercise 1.14 (O Tannenbaum). Write the function `triangle` that receives an **Int** argument `n`. This function will *draw* a triangle as shown on the right with `n = 5`. The *drawing* is actually returning a **String** of which every *line* ends with a *newline*.

Hence, the string corresponding with the image to the right is:

```
"      *\n     ***\n    *****\n   *********\n  *********"

```

The output will be:

```

      *
     ***
    *****
   *********
  *********

```

Write the function `christmasTree` that receives an **Int** argument `n`. This function *draws* a Christmas tree as a **String** (as shown on the right with `n = 4`). For every sub triangle the function should use a generalized version of the above defined `triangle` function.

```

      *
     *
    ***
   *
  ***
 *****
  *
 ***
 *****
 *****

```

The following operations on **Strings** and **Chars** are useful: `s1 ++ s2` glues **String** `s2` to **String** `s1`; `toString c` creates a **String** value from a **Char** `c`. `'\n'` is the *newline* character.

Exercise 1.15 (Trajectory). An object, such as a ball, which is shot away from a flat surface with an angle of $\theta_0 (0 < \theta_0 \leq \frac{\pi}{2})$ ⁵ and initial velocity $v_0 (0 < v_0)$, follows an arclike trajectory influenced by gravitational acceleration⁶. If we ignore effects such as air resistance and wind, the trajectory of the ball is a curve defined in a vertical flat surface in which the x -axis shows the distance to the starting position on the ground and the y -axis shows the height of the ball.

Because we ignore air resistance, the velocity of the ball in the x -direction ($v_x(t)$) is constant. The velocity of the ball in the y -direction ($v_y(t)$) depends on the time t and the gravitational acceleration $g = 9.81 \frac{m}{s^2}$:

$$\begin{aligned} v_x(t) &= v_0 \cdot \cos(\theta_0) \\ v_y(t) &= v_0 \cdot \sin(\theta_0) - g \cdot t \end{aligned}$$

You can calculate the distance $x(t)$ and the height $y(t)$ of the ball on time t as follows:

$$\begin{aligned} x(t) &= v_0 \cdot \cos(\theta_0) \cdot t \\ y(t) &= v_0 \cdot \sin(\theta_0) \cdot t - \frac{1}{2} \cdot g \cdot t^2 \end{aligned}$$

Height h , expressed in distance x is:

$$h(x) = \tan(\theta_0) \cdot x - \frac{g}{2 \cdot (v_0 \cdot \cos(\theta_0))} \cdot x^2$$

The ball reaches the highest point at time $t_{\max y}$:

$$t_{\max y} = \frac{v_0 \cdot \sin(\theta_0)}{g}$$

Thus you can find the maximal height reached at that time by plugging $t_{\max y}$ into $y(t)$. The ball takes just as much time to reach the highest point as it takes to get back on the ground. Therefore the ball is airborne for $t_2 = 2 \cdot t_{\max y}$ seconds. You can find the distance the ball travels by plugging t_2 into $x(t)$. The velocity of the ball when touching the ground can be found by plugging t_2 into $v_x(t)$ and $v_y(t)$. Then, the combined value is:

$$v_1 = \sqrt{v_x(t_2)^2 + v_y(t_2)^2}$$

Functions Implement the aforementioned functions v_x , v_y , x , y and h in Haskell with the respective names `v_x`, `v_y`, `x_at`, `y_at` and `h`. The equations given above are parameterized with the initial velocity v_0 and angle θ_0 and need to be passed as parameters to the Haskell functions.

Best angle Write a *recursive* function `best_angle` that, given an initial velocity v_0 , calculates the angle $\theta \in \{\frac{1}{100}\pi, \frac{2}{100}\pi \dots \frac{50}{100}\pi\}$ in such a way that when the ball is shot away with velocity v_0 and angle θ it travels the *greatest* distance.

Experiment with different values for v_0 . Does this result in the same return value?

⁵Angle in radians.

⁶Source: Halliday, Resnick. *Physics, Parts I and II, combined edition*, Wiley International Edition, 1966, ISBN 0 471 34524 5

Exercise 1.16 (`Stringnum`, `Stringnum.hs`). In the module `Stringnum` you develop functions to work with `Strings` representing arbitrarily large non-negative integers. The functions are:

- `isAStringNum` tests whether the given `String` represents a correct positive integer (including zero).
- The functions `smaller`, `bigger` and `equal` implement the operations `<`, `>` and `==`.
- The functions `plus`, `decrement`, `times` and `divide` implement the operations `+`, `-`, `*` and `/`, where `decrement a b = "0"` if `bigger b a`.

The following functions on `Strings` and `Chars` are useful (they are in module `Data.Char`): `isDigit c` tests whether a `Char c` is a digit; `digitToInt c` returns the ASCII-code of `Char c`; `size s` returns the length of the `String s`; `++` glues two `Strings` together; `s !! i` returns the character from `s` at index `i`;

Exercise 1.17 (`Stringnum2`, `Stringnum.hs`). If you can calculate with arbitrarily large non-negative integers, then, using these operations, you can also construct the implementations for all arbitrarily large integers (negative, positive, and 0). In this implementation, use the exported functions from `Stringnum` to create a new module `Stringnum2` implementing the same operations for all integers. Add the following operations besides the existing ones:

- `isNegative` tests whether the given number is negative (`< 0`).
- `absolute` calculates the absolute value of the argument.
- `changeSign` flips the sign of the number.

Exercise 1.18 (`Time`, `Time.hs`). The length of movies and music is often displayed in the `m+ : ss` format where `m+` is the number of minutes and `ss` the number of seconds. For example, the length of the movie “*The Lord of the Rings: The Fellowship of the Ring*” is 178:00, i.e. two hours and 58 minutes. The length of the song “*Tea*” of Sam Brown on the album “*Stop!*” is 0:41, i.e. 41 seconds. Moreover, time in between songs (silence) is often indicated via negative time. For instance, three seconds of silence is -0:03. Implement the following two functions:

- `showTime n`: with `n` the number of seconds (positive or negative), this function shows the time as explained in the above format;
- `parseTime x`: with `x` a string, this function attempts to extract the format explained above and return the corresponding (positive or negative) number of seconds. If `x` is ill formatted, then the result should be zero.

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 $k \times y = x$ will provoke the warning “Defined but not used: y ”. Variables with a leading underscore are not reported, so changing the definition to $k \times _y = x$ suppresses the warning.