1 Programming with expressions and values

Exercise 1.1 (*Warm-up:* getting GHC up and running, `Hello.hs`). Install the Glasgow Haskell Compiler on your system, and load `Hello.hs` in GHCi. This file contains the following code.

```
lyrics 0 = "no more seats in the lecture hall!\n"
lyrics n = show n ++ " " ++ seats ++ " in the lecture hall! " ++
           "Only " ++ show n ++ " " ++ seats ++ " left!\n" ++
           "A students walks in, and sits down, now there are\n" ++ lyrics (n-1)
  where seats = if n /= 1 then "seats" else "seat"

song = lyrics 75
```

You can probably guess its meaning. Evaluate it by typing `putStr song`. You can also simply type `song`, but you will notice that this outputs an actual string *denotation* instead of just its contents.

- Discuss how the function `lyrics` computes its result.
- In Haskell it is considered proper etiquette to add *type declarations* (of the form `f :: A → B`) to all top-level definitions. Add a type declaration for `song` and `lyrics` and reload the file (`:reload` or `:r` in GHCi) to see if your declarations are valid.
- What do you think happens if you would type `lyrics (-5)` in GHCi? Try it and see if you were right. (Note: Ctrl+C can be used to interrupt GHCi)
- If you type `lyrics -5` instead, you get a type error message for the '`-`' operator. The first of many! What is causing it in this case?

Exercise 1.2 (*Warm-up*: reading Haskell, naming). Coming up with good function names is very important. Haskell programmers like brevity, but in the code below things got out of hand! Come up with succinct names for these definitions, that actually tell the reader what they compute.

```
f1 :: Integer → Integer → Integer
f1 m n
  | m < n     = m
  | otherwise = n

f2 :: String → Integer → String
f2 s n
  | n ≤ 0     = ""
  | otherwise = s ++ f2 s (n-1)

f3 :: Integer → Integer → Integer
f3 x 0 = x
f3 x y = f3 y (x `mod` y)
```

Exercise 1.3 (*Warm-up*: term rewriting). Determine the result of the expressions below by rewriting. First add parentheses `()` in the expressions illustrating the priorities of the operations. Then apply rewrites as shown in the lecture: place every rewrite step on a new line and explicitly state the rewrite rule used.

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

Check your answers using GHCi!

Exercise 1.4 (Pen-and-paper; term rewriting). Given this definition:

```
double x = incr (incr 0)
  where incr y = x + y
```

- Compute the result of `double 5` by rewriting as shown in the lecture: place every rewrite step on a new line and explicitly state the rewrite rule used.
- State the evaluation order you used (e.g., *applicative order*, *normal order*, ...)

Exercise 1.5 (Definitions, local functions, `Database.hs`). 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

elena, peter, pol :: Person
elena  =  ("Elena",  33,  "Functional Programming")
peter  =  ("Peter",  57,  "Imperative Programming")
pol    =  ("Pol",    36,  "Object Oriented Programming")

students :: [Person]
students = [elena, peter, pol]
```

Note: `students :: [Person]` means that `students` is a *list* of `Person`'s; we will explore lists in more detail in coming weeks.

1. Add your own data (e.g. yourself) and/or invent some additional entries.
2. The function `age` defined below extracts the age from a person, e.g. `age elena = 33`. In case you wonder what the underscores are for, see Hint 1.

   ```
   age :: Person → Age
   age (_, n, _) =  n
   ```

Define the 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. As in Exercise 1.1, the operator `++` (concatenation) and function `show` (converting expressions into strings) will be useful.

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 elena
("Elena",34,"Functional Programming")
```

6. We introduce some simple functions on lists:

   - The operator `++` concatenates lists, the function `length` provides the length of a list, and the function `sum` computes the sum of a list of integers (i.e. `[Integer]`);
   - The function `map` takes a function and a list and applies the function to each element of the list:

```
⋙ map age students
[33,57,33]
⋙ let ageName p = (age p, name p) in map ageName students
[(33,"Elena"),(57,"Peter"),(36,"Pol")]
```

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

```
⋙ let elderly p = age p > 50 in filter elderly students
[("Peter",57,"Imperative Programming")]
```

Create expressions to solve the following tasks:

- increment the age of all students by two;
- promote all of the students (prefix `"dr. "` to their name);
- find all students named Frits;
- find all students who are in their twenties;
- compute the average age of all students (see Hint 2)
- promote the students whose favourite course is Functional Programming

Exercise 1.6 (Recursion, patterns and guards, `Say.hs`, `SayTest.lhs`). Below is an start of a function `say :: Integer → String` which should transform an integer into its natural language representation, e.g. `say 23 = "twenty three"`.

```
module Say where

say :: Integer → String
say  0 = "zero"
say  1 = "one"
say  2 = "two"
say  3 = "three"
say  4 = "four"
say  5 = "five"
say  6 = "six"
say  7 = "seven"
say  8 = "eight"
say  9 = "nine"
say 10 = "ten"
say 11 = "eleven"
say 12 = "twelve"
say 20 = "twenty"
```

This function is not yet defined for all inputs, for instance you will get an error message if you ask for `say 23`. Complete it so that it produces understandable English renderings for all non-negative integers less than one million.

- Obviously, just adding all missing cases is not the right approach.
- You may assume that the inputs are in the desired range, i.e. between 0 and 999999.
- The file `SayTest.lhs` can be used to perform a non-exhaustive test of your solution. You can load it in GHCi (e.g. by typing `:load SayTest`), and then run the function `testme` to check your function. To resume manually testing the `Say` module, type `:module +Say`. If you do not pass all tests, that doesn't automatically mean you have failed the assignment!

  (Note: `SayTest.lhs` is a literate Haskell script; you don't have to understand it fully, but it doesn't contain many Haskell concepts not covered yet.)
- *Optional/extra*: Can you change `Hello.hs` from Exercise 1.1 so that the song text it produces prints the numbers in natural language, without cut-and-pasting your `say` function there?

Exercise 1.7 (*Extra*: Recursion).

1. Write the function `triangle` that receives an `Integer` 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 below is:

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

   You can print this to the screen with `putStr (triangle 5)`, for which the output will be:

   ```
       *
      ***
     *****
    *******
   *********
   ```

2. Write the function `christmasTree` that receives an `Integer` argument n. This function *draws* a Christmas tree as a `String` (as shown below with $n = 4$), which consists of ever growing triangles. You will need to change the function `triangle` created above slightly so that it can shift the triangle it creates a certain number of spaces to the right.

   ```
      *
      *
     ***
      *
     ***
    *****
      *
     ***
    *****
   *******
   ```

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`". Haskell programmers will usually replace such arguments with underscores, as in `k x _ = x`. Variables with a leading underscore are not reported, so changing the definition to `k x _y = x` also suppresses the warning, but can result in more readable code.

Hints to practitioners 2. Like every programming language, Haskell has some technical details that may trip you up from time to time. For example, Haskell actually has *two* integer types! `Int` and `Integer`. The difference is this: `Int` is like a C/C++ long int: a signed integer of N bits; `Integer` is an arbitrary-precision integer (like int in Python3). You can convert an `Int` to an `Integer` using the function `toInteger`.

Also, when it comes to division, Haskell has a `/` operator, functions `div` and `mod`, as well as `quot` and `rem`. The `/` operator only divides two real-valued numbers. To convert an integer to a real-valued number you can use the `fromIntegral` function.

The function `div` can be used to divide integers: it rounds the result towards $-\infty$, like the // operator in Python: `7 ‘div‘ 2 = 3` and `(-7) ‘div‘ 2 = -4`. The function `quot` truncates the result (i.e. rounds the result towards zero), like the / operator in Java: `7 ‘quot‘ 2 = 3` and `(-7) ‘quot‘ 2 = -3`

The `mod` and `rem` functions compute remainders so that `x = (x ‘div‘ y)*y + (x ‘mod‘ y)` and `x = (x ‘quot‘ y)*y + (x ‘rem‘ y)`. So, if you used `div` to divide, you should use `mod` to compute the remainder.

The functions `div` and `mod` are usually preferred because they have nicer mathematical properties; for instance `x ‘mod‘ 3` will always be either 0, 1 or 2, but `x ‘rem‘ 3` can be negative. For non-negative integers both `div` and `quot` behave exactly the same.