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 → 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) → 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 → Bool → Bool`?

Exercise 2.2 (`Types.hs`). Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions. You can assume the following types:

```
length  :: [a] → Int
isUpper :: Char → Bool
isLower :: Char → Bool
(!!)    :: [a] → Int → a
(<:=)   :: [a] → (Int,a) → [a]
```

Finally, in an expression `x <:= (y,z)`, `x` has type `[a]`, `y` has type `Int`, `z` has type `a`, and the result has type `[a]` (the same list as `x`, except that the element at index position `y` has been replaced by `z`). Please note that you can use ghci to check your answer.

```
f1 x y z
   | z < 0 || z ≥ length x = -1
   | x !! z == y           = z
   | otherwise             = f1 x y (z+1)

f2 x y = f1 x y 0

f3 x y
   | y < 0 || y ≥ length x = x
   | isUpper x !! y        = f3 (x <:= (y,toLower (x !! y))) (y+1)
   | isLower x !! y        = f3 (x <:= (y,toUpper (x !! y))) (y+1)
   | otherwise             = f3 x (y+1)

f4 x = f3 x 0

f5 x y z
   | y > z               = x
   | y < 0               = x
   | z ≥ length x        = x
   | otherwise           = f5 (x <:= (z,x !! y) <:= (y,x !! z)) (y+1) (z-1)

f6 x  = f5 x 0 (length x - 1)
```

Exercise 2.3 (Programming, `Char.hs`).
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

   ```
   isNumeral :: String → Bool
   isBlank   :: String → Bool
   ```

   that test whether a string consists solely of digits or white space. You may find the predefined function `and :: [Bool] → 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

3. Define functions

   ```
   fromDigit :: Char → Int
   toDigit   :: Int → Char
   ```

   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 → Char → 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 2.4 (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 2.5 (Programming, `Swap.hs`).

1. Define a function

   ```
   swap :: (Int, Int) → (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) → (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 2.6 (Warm-up: static typing).

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

   ```
   (+ 4)
   div
   div 7
   (div 7) 4
   div (7 4)
   7 ‘div‘ 4
   + 3 7
   (+) 3 7
   (b, ’b’, "b")
   (abs, ’abs’, "abs")
   abs ∘ negate
   (* 3) ∘ (+ 3)
   ```

2. What about these?

   ```
   (abs ∘ ) ∘ ( ∘ negate)
   (div ∘ ) ∘ ( ∘ mod)
   ```

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

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

```
i x           = x
k (x, _y)     = x
b (x, y, z)   = (x z) y
c (x, y, z)   = x (y z)
s (x, y, z)   = (x z) (y z)
```

If you get stuck see Hint 1. Are any of these functions predefined (perhaps under a different name)? Again, see Hint 1.

Exercise 2.7 (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 → Int`

   (b) `a → a`

   (c) `(Int, Int) → Int`

   (d) `(a, a) → a`

   (e) `(a, b) → a`

   How many total functions are there of type `Int → Int`? By contrast, how many total functions are there of type `a → a`?

2. Define total functions of the following types:

   (a) `(a, a) → (a, a)`

   (b) `(a, b) → (b, a)`

   (c) `(a → b) → a → b`

   (d) `(a, x) → a`

   (e) `(x → a → b, a, x) → b`

   (f) `(a → b, x → a, x) → b`

   (g) `(x → a → b, x → a, x) → b`

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

3. Define total functions of the following types:

   (a) `Int → (Int → Int)`

   (b) `(Int → Int) → Int`

   (c) `a → (a → a)`

(d) `(a → a) → a`

How many total functions are there of type `(Int → Int) → Int`? By contrast, how many total functions are there of type `(a → a) → a`?

Exercise 2.8. Below a number of functions are given without their type signatures. Derive the *most general type* for each of these functions.
You can assume the following types: `fst :: (a,b) → a` and `snd :: (a,b) → b`. Please note that you can use GHCi to check your answers.

```
f1 x          = x

f2 x
  | x < 0       = -1
  | x > 0       =  1
  | otherwise   =  0

f3 x      = (snd x, fst x)

f4 (x,y) = (y,x)

f5 x y    = (x,y)

f6 x y    = (x,(y,y),x)

f7 (x,(y,z)) = ((x,y),z)
```

Hints to practitioners 1. 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.

```
  ⋙ :info map
map :: (a → b) → [a] → [b]    -- Defined in 'GHC.Base'
  ⋙ :type map ∘ map
map ∘ map :: (a → b) → [[a]] → [[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.