2 Types and polymorphism

Exercise 2.1 (*Warm-up*: Polymorphism, tuples).

1. Create a Haskell file with the function definition:

   ```
   swap :: (Int,Int) → (Int,Int)
   swap (x,y) = (y,x)
   ```

   that swaps the two components of a tuple. Define at least two other functions of this type (use your creativity).

2. What happens if we change the type to

   ```
   swap :: (a,b) → (b,a)
   ```

   Is the original definition of `swap` still valid? What about changing the type of the other two functions that you have invented to `(a,b)→(b,a)`?

3. Remove all top-level type declarations, and inspect the types GHCi infers for all the functions by using `:type swap`, and similar for your own functions. What types does GHCi infer?

4. 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 2.2 (*Warm-up*: type inference, `WhatType.hs`).
What are the inferred types of the function definitions `f0` to `f7`?

```
f0 (x,y)    = x == 'F' && y == 'P'

f1 s        = s ++ ", cruel world!"

f2 x (y,z) = (z,x,y)

f3 ' '      = '_'
f3 c        = c

f4 x y
  | x == ""    = y
  | otherwise = x

f5 b x y    = if b then (x,y) else (y,x)

f6 x        = \y → x

f7          = ("Haskell" ++)
```

Apply the type inference method used in the lecture! If you are stumped by `f6` and `f7`, see Hint 1.

After you think you have the answers (or decide to have thought hard enough), the answers can be revealed to you by using `:type` in GHCi, for more info on how to use that see Hint 2. Try to run the functions on arguments of the proper type. Is the output what you expected?

Exercise 2.3 (*Mandatory*: Programming with ad-hoc polymorphism, `Pow2.hs`).
We will explore writing a function that works on various numerical types.

1. Write a recursive definition of the function `pow2 :: Integer → Integer` that computes 2^n, e.g., `pow2 0 = 1`, `pow2 1 = 2`, `pow2 16 = 65536`, etc.

2. Change the type declaration into the overloaded type `pow2 :: (Ord n, Num n, Num a) ⇒ n → a`, and reload. Does your program still type check? If not, repair your function so that it does.

   You should be able to reproduce this in GHCi:

   ```
   ⋙ pow2 16 :: Integer
   65536
   ⋙ pow2 16 :: Int
   65536
   ⋙ pow2 16 :: Float
   65536.0
   ⋙ pow2 16 :: Double
   65536.0
   ```

   The type annotation "`:: Int`" instructs the compiler to evaluate the expression pow2 16 with the expression type forced to `Int`; likewise for the other annotations. In the case of `Int` it will make `pow2` compute its result using machine integers. Also see Hint 2.

3. What is the maximum n for which 2^n can still be represented by the types `Integer`, `Int`, `Float`, and `Double`?

Exercise 2.4 (*Mandatory:* Standard type classes, types of polymorphism, `PolyType.hs`).
The functions `f8` to `f11` all take two arguments of the same type, and compute a result of that same type. So, their type is of the form ❶→❶→❶, where we need to fill in something for ❶.

```
f8 x y  = if x ≤ y then x else y

f9 x y  = not x || y

f10 x y
  | x == 0    = y
  | otherwise = x + y

f11 x y = get 0
  where get n = if n == 0 then x else y
```

1. Which of these functions can be used on arguments of type `String`?

2. For each function, determine if it is *parametric polymorphic*, *ad-hoc polymorphic* (also known as *overloaded*), or not polymorphic.

   If a function is not polymorphic, state what ❶ is. Otherwise, determine the type classes (if any) that the type used as ❶ must be an instance of.

You can use the list of type classes shown on the lecture slides. Compare your findings with the output of `:type` in GHCi. Also, again: see Hint 2.

Exercise 2.5 (*Mandatory:* Data.Char, map, `Char.hs`, `CharTest.hs`).
Haskell's `String` is really a list 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 "Marc" ⟹ "marc"
map (\c→ord c - ord 'A') "TWAN" ⟹ [19,22,0,13]
```

(Recall that `map` takes a function and a list and applies the function to each element of the list.)

Most operations on `Char` are not part of Haskell's default environment (called the Prelude). These are in the module Data.Char; to access it add `import Data.Char` at the top of your own module. The documentation for it can be found at: `https://hackage.haskell.org/package/haskell2010/docs/Data-Char.html`.

1. Define an 'equality' operator `(~~)` for strings that, unlike `(==)`, disregards case, e.g.

   `"Sjaak" == "sjaaK" = False` but `"Sjaak" ~~ "sjaaK" = True`.

2. Define a function `reverseCase` that reverses the case of a string, e.g.

   `reverseCase "Sjaak" == "sJAAK"`

3. Create a function `shift :: Int → Char → Char`, which shifts all uppercase letters in the ASCII table by a given number of positions. E.g., `shift 3 'A'` becomes `'D'`, `'B'` becomes `'E'`, …, `'Y'` becomes `'B'`, and `'Z'` becomes `'C'`. Any other character should be left untouched.

4. Implement the Caesar cipher `caesar :: Int → String → String`, which shifts all ASCII letters by a certain amount. Make sure `caesar` also works on lowercase letters by converting them to uppercase before shifting.

   If you like, the file `CharTest.lhs` can be used to perform a non-exhaustive test of your solution. You can load it in GHCi (e.g. by typing `:load CharTest`), and then run the function `testme` to check your cipher.

5. *(Extra, optional)* Try to decode the following message.

   ```
   msg :: String
   msg = "ADMNO D HPNO NKMDIFGZ TJP RDOC AVDMT YPNO"
   ```

Exercise 2.6 (*Extra*: Records, Type-driven refactoring, `Database.hs` from last week).
Last week we implemented some queries on a database of students. If at some point in the future this code needs to handle more information about persons, then tuples are of course not the elegant solution. Time to refactor!

1. Rewrite the file so that it uses records; at the start it will now read:

   ```
   data Person = Person { name::String, age::Integer, favouriteCourse::String }

   elena, peter, pol :: Person
   elena  =  Person {name="Elena", age=33, favouriteCourse="Functional Programming"}
   peter  =  Person {name="Peter", age=57, favouriteCourse="Imperative Programming"}
   pol    =  Person {name="Pol",   age=36, favouriteCourse="Object Oriented Programming"}
   ```

   Observe that in the old code, `Person` was a type synonym for an existing data type (a tuple) introduced with the `type` keyword. Here, it is a new data type and so has to be introduced with the `data` keyword.

   You will no longer need the explicit accessor functions `age`, `name`, etc; as these are generated by the record type. Rewrite the other functions as needed. You can try to use Haskell's type system to point out all places where you forgot to adapt the old code to the new data structure.

2. Last week we generated queries for the following expressions:

   - 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;
   - promote the students whose favourite course is Functional Programming

   Update these to work with the new data type.

   Wherever you used helper functions, try to rewrite them using lambda expressions (i.e., use anonymous functions instead of named ones).

Hints to practitioners 1. Try to rewrite `f6` and `f7` in *declaration style* instead of using an anonymous function, and see if that allows you to figure out the type.

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.

```
⋙ :type length "Hello"
length "Hello" :: int
⋙ :type (+)
(+) :: Num a ⇒ a → a → a
⋙ :info (+)
class Num a where
  (+) :: a → a → a
  ...
        -- Defined in 'GHC.Num'
infixl 6 +
```

This is particularly useful if your program does not type check (or, if you are lazy.) You can also try to coerce expressions to a certain type by using '`::`'. Haskell will complain if the expression is not of the desired type.

```
⋙ 1 + 2 :: Integer
3
⋙ 1 + 2 :: Double
3.0
⋙ 1 + 2 :: String
<interactive>:3:1: error:
    * No instance for (Num String) arising from a use of '+'
    * In the expression: 1 + 2 :: String
      In an equation for 'it': it = 1 + 2 :: String
```

To get a recap on the kinds of polymorphism, `https://wiki.haskell.org/Polymorphism` is a good overview.

Hints to practitioners 3. 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 `Char → Bool` into the search field, Hoogle will display all predicates on characters.