Solution sheet 1

Introduction

Please note that there can be other solutions than those listed in this document.

This is a literate Haskell file which is available as PDF, as well as literate Haskell source code (.lhs extension). If you are looking at the PDF version, you should download the .lhs file, which can be loaded into ghci or compiled with ghc. This document uses ***Markdown*** syntax, compiled to PDF using *Pandoc*.

Everything that is not enclosed in a code block like below is treated as text.

```haskell
-- This is code
main :: IO ()
main = undefined
```

Exercise 1 - Types of values

`['a', 'b', 'c']` Is a list of characters `[Char]`

`('a', 'b', 'c')` Is a 3-Tuple of characters `(Char, Char, Char)`

`[(False, '0'), (True, '1')]` Is a list of `(Bool, Char)` tuples. `[(Bool, Char)]`

`([False, '0'], [True, '1'])` Is not a valid type, as lists cannot contain elements of different types.

`[tail, init, reverse]` Is a list of functions which take a list of some type `a`, and returns another list of same type. `[[a] -> [a]]`

Note: Function types are defined by their parameter type(s) and return value type, enclosed by parenthesis. As such, a function with N parameters would have the type: `(arg1type -> arg2type -> ... -> argNtype -> resType)`

Exercise 2 - Constructors

To create the list `['a', 'b', 'c']` explicitly with constructors, we would write `'a':'b':'c':[]`. Haskell does allow and understand `['a', 'b', 'c']`, but that is just syntactic sugar, and behind the scenes the explicit version is used.

`(,,) 'a' 'b' 'c'` Will construct the 3-Tuple `('a', 'b', 'c')`

Note: A 2-Tuple would use single comma `(,) a b`, 4-Tuple would use 3 commas `(,,,) a b c d` and so on.

The two can be combined: `(,) False '0' : (,) True '1' : []`, and we get the list `[(False, '0'), (True, '1')]`

Finally, we can also construct a list of functions this way:

```
-- Loadable in ghci
functionList = tail : init : reverse : []
```

Notice how calling `functionList` in *ghci* gives an error, this is because it has not been told how to *Show* these functions, only their results. You can however still use the functions for their intended use, from the list.

Exercise 3 - Function types

All functions will be loadable into ghci with the .lhs document.

The function `second` returns the second element of a list by getting the head (first element) of the tail (all but the first element). This will work of any list type, as such we define the type as `[a] -> a`. In general, if a type can be anything, the letters *a, b, c...* are used in order, as needed.

```
second :: [a] -> a -- The type declaration
second xs = head (tail xs)
```

The function `swap` swaps the order of elements in a 2-Tuple, as such we can define the type as `(a, b) -> (b, a)` Note the usage of `a` **and** `b`. We use two different letters as we cannot guarantee that the first and second element of the tuple is of same type.

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

The function `pair` takes two values and returns a tuple of the two values, somewhat similarly to previous, we can define the type as `a -> b -> (a, b)`. Again, we do not know that `x` and `y` will be of same type, Additionally, notice the arrow between `a` and `b`, this is because we no longer get a tuple, but two arguments instead.

```
pair :: a -> b -> (a, b)
pair x y = (x, y)
```

The function `double` simply multiplies a number by 2. That does however mean that we need to limit the type to a number. A naive option would be allowing only integer values: `Int -> Int`. However, a much better option is available: Type constraints. These constraints will allow any type belonging to a given typeclass (e.g. Num for number). This allows us to define the function type `Num a => a -> a`.

```
double :: Num a => a -> a
double x = x*2
```

The function `palindrome` takes a list and performs an equivalence test on its reverse. As such, we can define the type as `Eq a => [a] -> Bool`. The Eq typeclass constraint is added as we need to make sure it is possible to do equality comparison on the elements of the given list.

```haskell
palindrome :: Eq a => [a] -> Bool
palindrome xs = reverse xs == xs
```

The function `twice` applies a function *f* twice. One should note that the return type of f has to be the same as the argument type. We write the function type as `(t -> t) -> t -> t`.

```haskell
twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

Exercise 4 - Types and functionality

`(:) 'a'` is a function `[Char] -> [Char]` which appends an a to a list.

`(,) 5` is a function `b -> (Int, b)` which constructs a 2-Tuple with first element being the number 5, and second element the argument given.

`(+) 2` is a function `Num a => a -> a` which adds 2 to the number provided as argument.

Exercise 5 - orElse implementation

Two examples using pattern matching only.

```haskell
orElse :: Maybe t -> t -> t
orElse (Just e) _ = e
orElse _ f = f
```

Another option:

```haskell
orElse_ :: Maybe t -> t -> t
orElse_ Nothing f = f
orElse_ (Just e) _ = e
```

Exercise 6 - Function examples

This exercise is one that you really should play around with yourself.

```haskell
f :: Num a => (a->a) -> a
f func = func (func 1) -- same as twice except always called with 1.

g :: Num a => a -> (a->a)
g x y = x*x + y*y -- adds squares of x and y
```

```haskell
h :: Num a => (a->a) -> (a->a)
h func x = x + func 42 -- adds x to func called with 42
```

Exercise 7 - “sign” implementation

While possible (As seen below), Creating such a function with only the typeclass `Num`, is difficult to a point where neither I or Søren have been able to do it nicely.

As sign is already implemented as signum for all instances of Num, we can simply:

```haskell
sign :: Num a => a -> a
sign = signum -- Not very interesting option
```

An implementation using strictly the Num typeclass seems like a lot of work, instead we include the Ord typeclass so we can use `<`, `=` and `>`.

```haskell
signOrd :: (Num a, Ord a) => a -> a -- If-then-else
signOrd 0 = 0
signOrd x = if x < 0
  then -1
  else 1
```

Another one

```haskell
signOrd' :: (Num a, Ord a) => a -> a -- Guards, will be covered later
signOrd' x
  | x == 0 = 0
  | x < 0 = -1
  | x > 0 = 1
```

If you are interested in the use of guards, I suggest reading the chapter on Syntax in functions of *Learn you a Haskell for great good*!

Some versions using other typeclasses

```haskell
signIntegral :: Integral a => a -> a
signIntegral 0 = 0
signIntegral x = x `div` abs x
```

```haskell
signFrac :: (Fractional a, Eq a) => a -> a
signFrac 0 = 0
signFrac x = x / abs x
```

Exercise 8 - More types

`one x = 1` could have the type `t -> Int`, ghc generalises to `one :: Num t => t1 -> t`

`apply f x = f x` could have the type `(a -> b) -> a -> b` ghc: `apply :: (t1 -> t) -> t1 -> t`

`compose f g x = f (g x)` could have the type `(b -> c) -> (a -> b) -> a -> c`, as g is applied before f. ghc writes this slightly differently: `compose :: (t2 -> t1) -> (t -> t2) -> t -> t1`