Type Inference Exercise 1

Infer the types of the following Haskell functions:

foo (*x*,*y*) = (*y*,*x*)

bar (*x*,*y*) = *x* ++ *y*

msg **True** = "okay"
msg **False** = "error"

baz [ ] = [ ]
baz (*x* : *xs*) = *xs* : *baz xs*

strange [ ] = "empty"
strange (*f* : *fs*) = *f* 42

Let's Play the Evaluation Game

```
h1 :: String -> (Int, String)
h1 str = (length str, ' ' : str)

g h = fst (h "") : [limit]
```

Then:

```
g h1
= fst (h1 "") : [limit]
= fst (length "", ' ' : "") : [limit]
= length "" : [limit]
= 0 : [limit]
= [0, 100]
```


Let's Play the Evaluation Game Again

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (`notElem` str) ['A' .. 'Z']

g h = fst (h "") : [limit]
```

Then:

```
g h2
= fst (h2 "") : [limit]
= fst (sum (map ord (notOccCaps "")), head "") : [limit]
= sum (map ord (notOccCaps "")) : [limit]
= …
= 2015 : [limit]
= [2015, 100]
```


Higher-Order Functions

g h = *fst* (*h* "") : [*limit*]

Functional Programming: **Functions are first-class citizens**

- Functions can be **arguments of other functions**: *g h2*
- Functions can be **components of data structures**: (7,*h1*), [*h1*, *h1*]
- Functions can be **results of function application**: *succ* ∘ *succ*

A **first-order function** accepts only non-functional values as arguments.

A **higher-order function** expects functions as arguments.

g is a second-order function: it expects first-order functions like *h1*, *h2* as arguments.

Type Inference Examples

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

fst ('c', False)                :: Char

["hello", fst (x, 17)]      ⇒    x :: String

f p = limit + fst p         ⇒    p :: (Int,a)
                                 f :: (Int,a) -> Int

g h = fst (h "") : [limit]
    ⇒     h ::  String -> (Int,a)
          g :: (String -> (Int,a)) -> [Int]
```


Curried Functions

- **Function application associates to the left**, i.e.,

  ```
  f x y  =  (f x) y
  ```

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., "they accept their arguments one at a time":

  ```
  cylVol r h = (pi :: Double) * r * r * h
  ```

  Since the right-hand side, `r`, and `h` obviously all have type `Double`, we have;

  ```
  (cylVol r) :: Double -> Double
  cylVol     :: Double -> (Double -> Double)
  ```

- **Function type construction associates to the right**, i.e.,

  ```
  a -> b -> c  =  a -> (b -> c)
  ```


Application of Curried Functions

Let values with the following types be given:

$f :: a \to b \to c$
$x :: a$
$y :: b$

The type of f is the function type $a \to (b \to c)$, with

- argument type a,
- result type $b \to c$.

Therefore, we can apply f to x and obtain:

$(f\ x) :: b \to c$

The application of a "two-argument function" to a single argument is a "one-argument function", which can then be applied to a second argument:

$(f\ x)\ y :: c \quad = \quad f\ x\ y$

Operations on Functions

```
id :: a -> a                                  -- identity function
id x = x

(.) :: (b -> c) -> (a -> b) -> (a -> c)       -- function composition
(f . g) x  =  f (g x)

flip :: (a -> b -> c) -> (b -> a -> c)        -- argument swapping
flip f x y  =  f y x

curry :: ((a,b) -> c) -> (a -> b -> c)        -- currying
curry g x y = g (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y
```

Exercise (*necessary!*): Copy only the definitions to a sheet of paper, and then infer the types yourself!

Operator Sections

- Infix operators are turned into functions by surrounding them with parentheses:

```
(+) 2 3  =  2 + 3
```

- This is necessary in type declarations:

```
(+)  :: Int -> Int -> Int      -- not the "natural" type of (+)
(:)  ::  a  -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
```

- It is also possible to supply only one argument (which has to be an atomic expression):

```
    (2   +) 3     =    2   + 3       =  (+ 3  ) 2
    (8.3 /) 2.5   =    8.3 / 2.5     =  (/ 2.5) 8.3
    (7   :) []    =    7   : []      =  (: [] ) 7
((2^17) :) (16:[]) = (2^17) : 16 : [] = (: (16:[])) (2^17)
```


Turning Functions into Infix Operators

Surrounding a function name by **backquotes** turns it into an infix operator.

Frequently used examples (not the "natural" types throughout):

```
div, mod, max, min :: Int -> Int -> Int
elem :: Int -> [Int] -> Bool

12 `div`  7          =    1
12 `mod`  7          =    5
12 `max`  7          =   12
12 `min`  7          =    7
12 `elem` [1 .. 10]  =  False
```


Type Inference Exercise 2

Infer the types of the following Haskell functions:

$pupd1\ f\ (x, y) = (f\ x, y)$

$keep1\ f\ p = (fst\ p, f\ p)$

$filter\ p\ xs = [x \mid x \leftarrow xs, p\ x]$

$(\ldots) = (.) \circ (.)$

$(f\ x) :: b$ iff for exactly one type a we have $f :: a \to b$ and $x :: a$.