# Exercise 11.8

Define a function total

```
total :: (Integer → Integer) → (Integer → Integer)
```

so that `total f` is the function which at value `n` gives the total:

```
f 0 + f 1 + … + f n
-------------------------------------------
```

# Exercise 11.9+10

Given a function `f` of the type `a → b → c`, write down a lambda abstraction that describes the function of type `b → a → c` which behaves like `f` but which takes it arguments in the other order

Using this expression, give a definition of the function:

```
flip :: (a → b → c) → (b → a → c)
```

which reverses the order in which its functions arguments takes its arguments

---

# Exercise 11.14

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)

($) :: (a -> b) -> a -> b

(:) ::  a -> [a] -> [a]

(.)     :: (b -> c) -> (a -> b) -> a -> c
```

# Exercise 11.14

- What is the effect of `uncurry ($)` ?
- What is its type?
- Answer a similar question for `uncurry (:)` and `uncurry (.)`

---

# Exercise 11.15

What are the effects and types of

1. `uncurry uncurry`
2. `curry uncurry`

---

# Exercise 11.17

Can you define functions:

```
curry3 :: ((a, b, c) → d) → (a → b → c → d))
```

```
uncurry3 :: (a → b → c → d)→ ((a, b, c) → d)
```

which perform the analogue of `curry` and `uncurry` but for three arguments rather than two? Can you use `curry` and `uncurry` in these definitions?

---

# Exercise 11.19

Give calculations of

```
iter 3 double 1
(comp2 succ (*)) 3 4
comp2 sq add 3 4
```

```
iter :: Integer -> (a -> a) -> (a -> a)
iter n f
  | n > 0     = f . iter (n-1) f
  | otherwise = id
```

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

```
add :: Num a => a -> a -> a
add x y = x + y
```

```
sq :: Num a => a -> a
sq x = x*x
```

```
succ :: Integer → Integer
succ n = n+1
```

```
comp2 :: (a -> b) -> (b -> b -> c) -> (a -> a-> c)
comp2 f g = (\x y -> g (f x) (f y))
```

# Exercise 11.20

What is the type and effect of the function:

```
\n → iter n succ
```

---

# Exercise 11.21

Give an alternative ‘constructive’ definition of `iter` which creates the list of `n` copies of `f`

```
[f,f…,f]
```

and then composes these functions by folding the operator ‘.’ to give

```
f . f . … . f
```

---

# Exercise 12.13

Define the function

```
splits :: [a] → ([a], [a])
```

which defines the list of all the ways that a list can be split in two

e.g.

```
splits “Spy” = [(“”,”Spy”),(“S”,”py”),(“Sp”,”y”),(“Spy”,””) ]
```

---

# Exercise 17.2

Using the list comprehension notation, define the functions:

```
sublists, subsequences :: [a] → [[a]]
```

which return all the sublists and subsequences of a list. A sublist is obtained omitting some of the elements of a list; a subsequence is a continous block from a list. E.g. `[2,4]` and `[3,4]` are sublists of `[2,3,4]` but only `[3,4]` is a subsequence.

---

# Exercise 17.23

Define the infinite lists of factorial and Fibonacci numbers,

```
factorial = [1,1,2,6,24,120,720,…]
fibonacci = [0,1,1,2,3,5,8,13,21]
-------------------------------------------------
```

# Exercise 17.24

Give a definition of the function

```
factors :: Integer -> [Integer]
```

which returns a list containing the factors of a positive integer.

```
factors 12 = [1,2,3,4,6,12]
```

Using this function or otherwise, define the list of numbers whose only prime factors are 2, 3 and 5. So called Hamming Numbers

```
hamming = [2,3,4,5,6,8,9,10,12,15]
```

---

# Exercise 17.25

Define the function

```
runningSums :: [Int] → [Int]
```

which calculates the running sums

```
[0,a0, a0+a1,a1+a2+a3,...]
```

of a list

```
[a0, a1, a2,...]
```

---

# Exercise 17.29

- How would you merge two infinite lists, assuming that they are sorted?
- How would you remove duplicates from the list which results?
- As an example, how would you merge the lists of powers of 2 and 3?

---