Seq:

- seq is used for killing lazy evaluation where you deem it unsuitable.
- It is a built in function.
- seq is a special function that is used to force expressions to be evaluated. seq evaluates the first parameter and passes it to the second parameter.
- To evaluate "seq x y": evaluate x to "**weak head normal form**", then continue with y.
- Weak head normal form (WHNF) means:
  - for built-in number types: until you have the number
  - for algebraic data types: until you have a data constructor
  - for functions: until you have a lambda
- Naturally, "seq x y" is most meaningful when x is something that y will need.
- E.g. mySumV2, mySumV3 and mySumV4 are used to sum a list.

```
mySumV2 xs = g 0 xs
  where
    g accum [] = accum
    g accum (x:xs) = g (accum + x) xs
```

Evaluation of mySumV2 [1,2,3]:
mySumV2 (1 : 2 : 3 : [])
→ g 0 (1 : 2 : 3 : [])
→ g (0 + 1) (2 : 3 : [])
→ g ((0 + 1) + 2) (3 : [])
→ g (((0 + 1) + 2) + 3) []
→ ((0 + 1) + 2) + 3
→ (1 + 2) + 3
→ 3 + 3
→ 6

Now, we will use seq.

```
mySumV3 xs = g 0 xs
  where
    g accum [] = accum
    g accum (x:xs) = seq accum (g (accum + x) xs)
    -- Note: accum is something that "g (accum + x) xs" will need.
```

Evaluation of mySumV3 [1,2,3]:
g 0 (1 : 2 : 3 : [])
→ seq 0 (g (0 + 1) (2 : 3 : []))
→ g (0 + 1) (2 : 3 : [])
→ seq a (g (a + 2) (3 : [])) with a = 0 + 1
→ seq a (g (a + 2) (3 : [])) with a = 1
→ g (1 + 2) (3 : [])
→ seq b (g (b + 3) []) with b = 1 + 2
→ seq b (g (b + 3) []) with b = 3

→ g (3 + 3) []
→ 3 + 3
→ 6

We can still decrease number of the iterations:

```
mySumV4 xs = g 0 xs
  where
    g accum (x:xs) = let a1 = accum + x
                     in seq a1 (g a1 xs)
    -- Note: a1=accum+x is something that "g a1 xs" will need.
```

g 0 (1 : 2 : 3 : [])
→ seq b (g b (2 : 3 : [])) with b = 0 + 1
→ seq b (g b (2 : 3 : [])) with b = 1
→ g 1 (2 : 3 : [])
→ seq c (g c (3 : [])) with c = 1 + 2
→ seq c (g c (3 : [])) with c = 3
→ g 3 (3 : [])
→ seq d (g d []) with d = 3 + 3
→ seq d (g d []) with d = 6
→ g 6 []
→ 6

Fold Functions:

- Back when we were dealing with recursion, we noticed a theme throughout many of the recursive functions that operated on lists. Usually, we'd have an edge case for the empty list. We'd introduce the x:xs pattern and then we'd do some action that involves a single element and the rest of the list. It turns out this is a very common pattern, so a couple of very useful functions were introduced to encapsulate it. These functions are called **folds**. They're sort of like the map function, only they reduce the list to some single value.
- A fold takes a binary function, a starting value, called an **accumulator**, and a list to fold up. The binary function itself takes two parameters. The binary function is called with the accumulator and the first or last element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first or last element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.
- Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that. Whenever you want to traverse a list to return something, chances are you want a fold. That's why folds are, along with maps and filters, one of the most useful types of functions in functional programming.
- Foldl and foldr are two fold functions.

Foldl:

- Also called the left fold.
- It folds the list up from the left side. The binary function is applied between the starting value and the head of the list. That produces a new accumulator value and the binary function is called with that value and the next element, etc.

- E.g. Consider **foldl (\acc x -> acc + x) 0 xs**, 0 is the accumulator or starting value, xs is the list, acc is the accumulator value and x is the first element of the list.
- E.g. The below functions all take in a list of numbers and sum up the numbers.

```
mySum :: (Num a)=> [a] -> a
mySum xs = foldl (\acc x -> acc + x) 0 xs

mySum2 :: (Num a)=> [a] -> a
mySum2 = foldl (+) 0
```

```
*Main> mySum [3,5,3,1]
12
*Main> mySum2 [3,5,3,1]
12
```

Here's the evaluation of mySum [3,5,3,1]:
→ **0** + **3** [3, 5, 3, 1]
→ **3** + **5** [5, 3, 1]
→ **8** + **3** [3, 1]
→ **11** + **1** [1]
→ **12**
The bolded dark green numbers represent the accumulator value.

- **Note:** Instead of **(\acc x -> acc + x)**, we can use **(+)** instead.
- **Note:** In the second example, we can omit the xs as the parameter because calling foldl (+) 0 will return a function that takes a list. Generally, if you have a function like foo a = bar b a, you can rewrite it as foo = bar b, because of currying.
- E.g.

```
reverseList :: [a] -> [a]
reverseList xs = foldl (\a x -> x:a) [] xs
```

```
*Main> reverseList [1,2,3]
[3,2,1]
*Main> reverseList [5, 4, 2, 1]
[1,2,4,5]
```

Foldr:

- The right fold, foldr, works in a similar way to the left fold, only the accumulator eats up the values from the right. Also, the left fold's binary function has the accumulator as the first parameter and the current value as the second one (so \acc x -> ...), the right fold's binary function has the current value as the first parameter and the accumulator as the second one (so \x acc -> ...). It kind of makes sense that the right fold has the accumulator on the right, because it folds from the right side.

- E.g.

```
mySum3 :: (Num a)=> [a] -> a
mySum3 xs = foldr (\x acc -> acc + x) 0 xs

mySum4 :: (Num a)=> [a] -> a
mySum4 = foldr (+) 0
```

```
*Main> mySum3 [3,5,3,1]
12
*Main> mySum4 [3,5,3,1]
12
```

Here's the evaluation of mySum3 [3,5,3,1]:
→ **0** + **1** [3, 5, 3, 1]
→ **1** + **3** [3, 5, 3]
→ **4** + **5** [3, 5]
→ **9** + **3** [3]
→ **12**
The bolded dark green numbers represent the accumulator value.

- One big difference is that right folds work on infinite lists, whereas left ones don't. To put it plainly, if you take an infinite list at some point and you fold it up from the right, you'll eventually reach the beginning of the list. However, if you take an infinite list at a point and you try to fold it up from the left, you'll never reach an end.
- **Note:** If you have **foldr op z (xs:xt)**
  where
  z = …
  op x r = …
  z is the starting value, x is the first element/value of the list and r = **foldr op z xt**.

Foldl1 and foldr1:

- The foldl1 and foldr1 functions work much like foldl and foldr, only you don't need to provide them with an explicit starting value. They assume the first or last element of the list to be the starting value and then start the fold with the element next to it. With that in mind, the sum function can be implemented like so: sum = foldl1 (+). Because they depend on the lists they fold up having at least one element, they cause runtime errors if called with empty lists. foldl and foldr, on the other hand, work fine with empty lists. When making a fold, think about how it acts on an empty list. If the function doesn't make sense when given an empty list, you can probably use a foldl1 or foldr1 to implement it.

Type Classes:

- In Haskell, every statement is considered as a mathematical expression and the category of this expression is called as a Type. You can say that "Type" is the data type of the expression used at compile time.
- In a generic way, Type can be considered as a value, whereas Type Class can be considered as a set of similar kinds of Types.
- A "type class" declares a group of overloaded operations ("methods").
- Syntax:
  class ClassName typeVar where

```
methodName :: type sig containing typeVar
-- Optional: default implementations
```

- Example: Methods == and /= are grouped under the Eq class. Its declaration in the standard library goes like this:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default implementation for (==)
  x == y = not (x /= y)
  -- default implementation for (/=)
  x /= y = not (x == y)
  -- default implementations are deliberately circular so you just have to
  -- implement one of them to break the cycle
```

The role of type variable "a" is a placeholder for Integer, Char, etc.

To implement these methods for a type, e.g., the standard library has this for Bool:

```
instance Eq Bool where    -- (so a=Bool here)
  False == False    = True
  True == True      = True
  _ == _            = False

  -- default implementation for (/=) takes hold
```

We say "Bool is an instance of Eq".

- **Note:** When you do "instance Num a where …", you are making "a" a Num type.
- WARNING:
  1. A class is not a type. Eq is not a type. These are illegal:
     foo :: Eq -> Eq -> Bool
     bar :: Eq a -> Eq a -> Bool
  2. A type is not a "subclass". Bool is not a "subclass" of Eq.
- Method types outside classes look like this:
  (==) :: Eq a => a -> a -> Bool
  The additional "Eq a =>" is marked for polymorphic but the user's choice of a must be an instance of Eq. This marker also appears when you write polymorphic functions using the methods.
- Type classes use => while types use ->.
  I.e. **function :: (Type Class) => (type 1) -> … -> (type n)**
  E.g.
  sum :: (Num a) => a -> a -> a
- Built in Types and Type Classes:
  - **Int:** Int is a type representing the Integer types data. Every whole number within the range of 2147483647 to -2147483647 comes under the Int type class.
  - **Integer:** Integer can be considered as a superset of Int. This value is not bounded by any number, hence an Integer can be of any length without any limitation.
  - **Float:** Float is a floating point number with single precision at the end.
  - **Double:** Double is a floating point number with double precision at the end.
  - **Bool:** Bool is a Boolean Type. It can be either True or False.

- **Char:** Char represents Characters. Anything within a single quote is considered as a Character.
- **EQ:** EQ type class is an interface which provides the functionality to test the equality of an expression. Any Type class that wants to check the equality of an expression should be a part of this EQ Type Class. Whenever we are checking any equality using any of the types mentioned above, we are actually making a call to the EQ type class. EQ is used for == or !=.
- **Ord:** Ord is another type class which gives us the functionality of ordering. Like EQ interface, Ord interface can be called using ">", "<", "<=", ">=", "compare". An instance of Ord is also an instance of Eq. We say “Ord is a subclass of Eq”, but beware that this is unrelated to OOP subclasses.
- **Show:** Show is a type class that has a functionality to print its argument as a String. Whatever may be its argument, it always prints the result as a String.
- **Read:** Read is a type class that does the same thing as Show, but it won’t print the result in String format.
- **Enum:** Enum is another Type class which enables the sequential or ordered functionality in Haskell.
- **Bounded:** All the types having upper and lower bounds come under this Type Class.
  E.g.

```
main = do
   print (maxBound :: Int)
   print (minBound :: Int)
```

```
sh-4.3$ main
9223372036854775807
-9223372036854775808
```

- Number operations are grouped into several type classes:
  - **Num:**
    - some methods: +, -, *, abs
    - instances: all number types
  - **Integral:**
    - some methods: div, mod
    - instances: Int, Integer
  - **Fractional:**
    - some methods: /, recip
    - instances: Rational, Float, Double, Complex a
- E.g. Why is the following a type error?

```
let xs :: [Double]
   xs = [1, 2, 3]
in sum xs / length xs
```

Answer:
sum xs :: Double, but length xs :: Int
(/) wants the two operands to be of the same type.
How to fix: sum xs / fromIntegral (length xs) or use realToFrac

```
fromIntegral :: (Integral a, Num b) => a -> b
realToFrac :: (Real a, Fractional b) => a -> b
```

- Often it is straightforward but boring to write instances for these classes, so the computer offers to auto-gen for you. However, restrictions apply. You can request it at the definition of your algebraic data type like this:
  data MyType = ... deriving (Eq, Ord, Bounded, Enum, Show, Read)

Foldable:

- The Foldable type class provides a generalisation of list folding (foldr and friends) and operations derived from it to arbitrary data structures. Besides being extremely useful, Foldable is a great example of how monoids can help formulating good abstractions.
- The purpose of this section is twofold:
  1. This explains why some of the library functions for lists have types like
     length :: Foldable t => t a -> Int
     instead of the simpler
     length :: [a] -> Int
  2. This familiarizes you with things like “Foldable t” and how it is not “Foldable (t a)”.
- E.g.

```
Prelude> :type length
length :: Foldable t => t a -> Int
Prelude> :type minimum
minimum :: (Foldable t, Ord a) => t a -> a
Prelude> :type sum
sum :: (Foldable t, Num a) => t a -> a
Prelude> :type foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
Prelude> :type foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

- A few library functions that consume a list and compute a “summary” are:
  - **length :: [a] -> Int**
  - **sum :: Num a => [a] -> a**
  - **minimum :: Ord a => [a] -> a**
    -- assumes non-empty list
  - **foldr :: (a -> b -> b) -> b -> [a] -> b**

  These make sense for other data structures representing sequences too, not just linked lists.
  E.g. sum should be used for vector and seq, too.

```
sum :: Num a => Vector a -> a
-- Vector is an array, 0-based Int index. Third-party but popular library.

sum :: Num a => Seq a -> a
-- Seq is a middle ground between array and linked list,
-- O(1) prepend and append, log time random access.
```

  Haskell supports this generalization with a type class:

```
class Foldable t where
  length :: t a -> Int    -- implicitly ∀a, similarly below
  sum :: Num a => t a -> a
  minimum :: Ord a => t a -> a
  foldr :: (a -> b -> b) -> b -> t a -> b    -- implicitly ∀a,b
```

```
-- and others
```

Note: It is not “class Foldable (t a)”. Likewise, instances go like “instance Foldable []”, not “instance Foldable ([] a)”.

Good type classes and bad type classes:

- A good type class has these traits:
  - You have multiple instances.
  - Methods satisfy useful laws or expectations, therefore can be used to build useful general algorithms.
  - Example: Ord: <= is reflexive, transitive, anti-symmetric, total. These laws are the basis of sorting algorithms and binary search tree algorithms.
- Bad type class: Created for no further benefit than:
  - Common method name.
  - Procrastinating writing actual code, procrastinating making up your mind what to do.