Laws!

George Wilson

Data61/CSIRO

george.wilson@data61.csiro.au

28th August 2018

```haskell
class Monoid m where
  mempty :: m
  (<>)   :: m -> m -> m
```

```haskell
class Monoid m where
  mempty :: m
  (<>)   :: m -> m -> m
```

Left identity:	`mempty <> y = y`
Right identity:	`x <> mempty = x`
Associativity:	`(x <> y) <> z = x <> (y <> z)`

```haskell
data Sum = Sum Int

instance Monoid Sum where
  mempty            = Sum 0
  Sum x <> Sum y = Sum (x + y)
```

Left identity:

`0 + y = y`

Right identity:

`x + 0 = x`

Left identity:

$0 + 5 = 5$

Right identity:

$x + 0 = x$

Left identity:

$0 + 5 = 5$

Right identity:

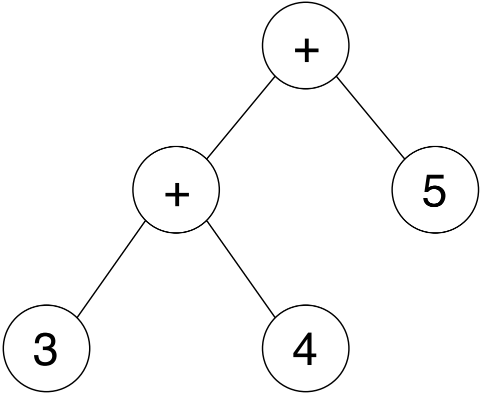
$7 + 0 = 7$

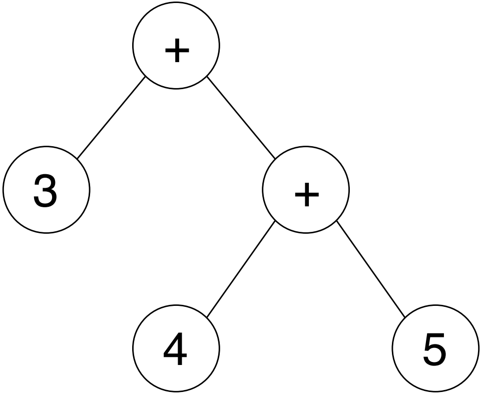
Associativity:

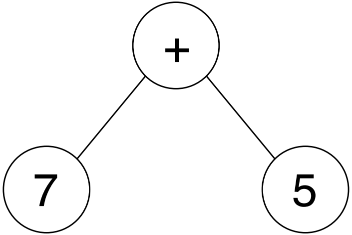
`3 + 4 + 5`

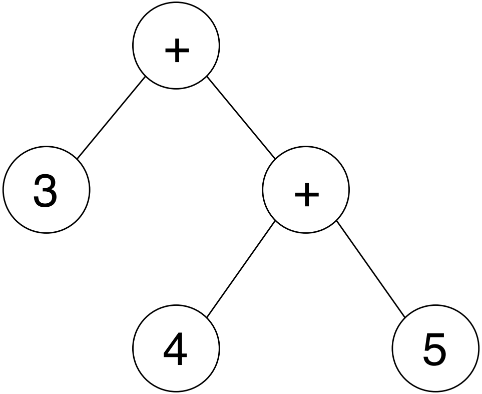
`(3 + 4) + 5`

`3 + (4 + 5)`

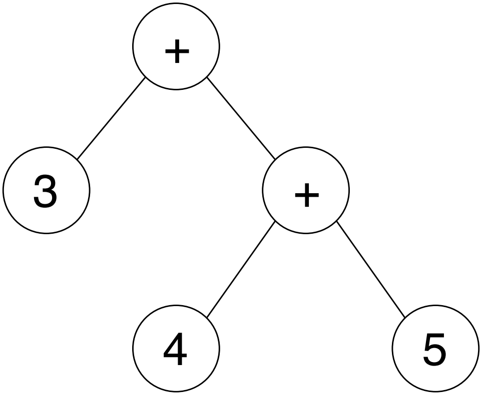

+
+
5
3
4

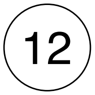

+
3
+
4
5

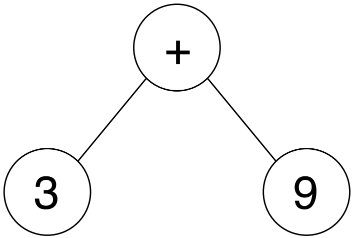

+
7
5


+
3
+
4
5


12


+
3
+
4
5

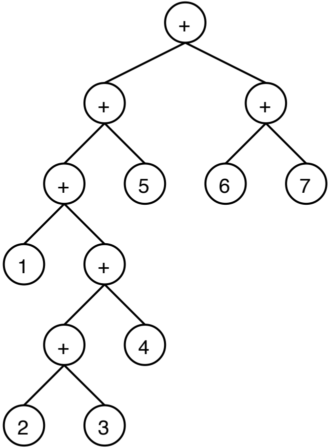

12

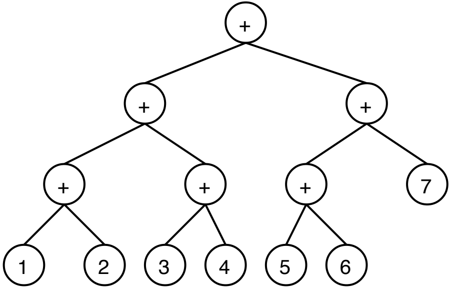

+
3
9

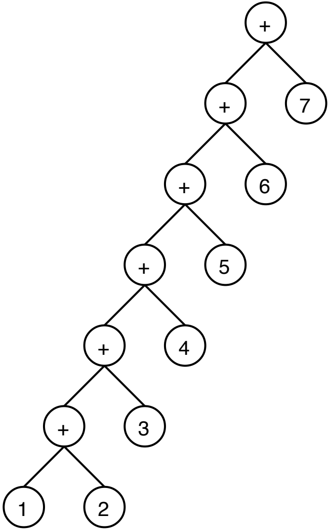
12

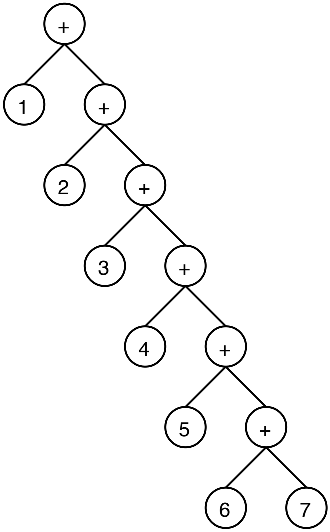
12

So?


+
+
+
+
5
6
7
1
+
+
4
2
3


+
+
+
+
+
+
+
1
2
3
4
5
6
7


+
+
7
+
6
+
5
+
4
+
3
1
2


+
1
+
2
+
3
+
4
+
5
+
6
7

```haskell
mconcat :: Monoid m => [m] -> m
```

```haskell
mconcat :: Monoid m => [m] -> m

mconcat list =
  case list of
    [] -> mempty
    (h:t) -> h <> mconcat t
```

```
mconcat [Sum 1, Sum 2, Sum 3, Sum 4]
```

```haskell
mconcat [Sum 1, Sum 2, Sum 3, Sum 4]
```

```haskell
Sum 1 <> (Sum 2 <> (Sum 3 <> (Sum 4 <> mempty)))
```

```
mconcat [Sum 1, Sum 2, Sum 3, Sum 4]
```

```
Sum 1 <> (Sum 2 <> (Sum 3 <> (Sum 4 <> mempty)))

==>  Sum 10
```

```haskell
mconcatR :: NotMonoid m => [m] -> m

mconcatR list =
  case list of
    []    -> mempty
    (h:t) -> h <> mconcatR t
```

```haskell
mconcatR :: NotMonoid m => [m] -> m
mconcatR list =
  case list of
    []     -> mempty
    (h:t) -> h <> mconcatR t


mconcatL :: NotMonoid m => [m] -> m
mconcatL list =
  helper mempty list
    where
      helper acc xs =
        case xs of
          []     -> acc
          (h:t) -> helper (acc <> h) t
```

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b

foldl :: (b -> a -> b) -> b -> [a] -> b
```

Laws give us **freedom** when working **in terms of** our abstractions

```haskell
instance Monoid [a] where
  mempty    = []
  left <> right =
    case left of
      []    -> right
      (h:t) -> h : (t <> right)
```

```haskell
instance Monoid [a] where
  mempty    = []
  left <> right =
    case left of
      []     -> right
      (h:t) -> h : (t <> right)
```

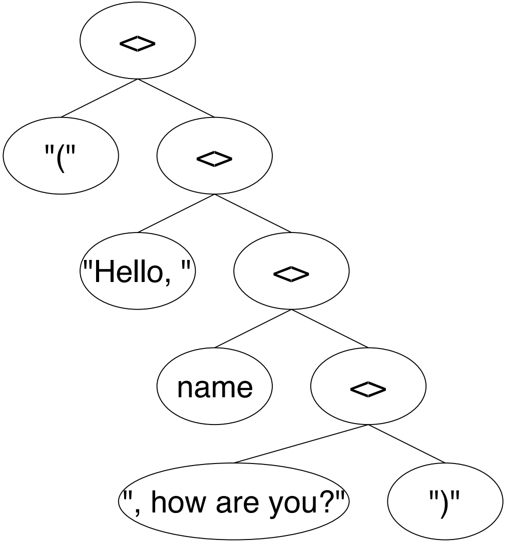
Left identity:	`[] ++ y    = y`
Right identity:	`x ++ []    = x`
Associativity:	`(x ++ y) ++ z = x ++ (y ++ z)`

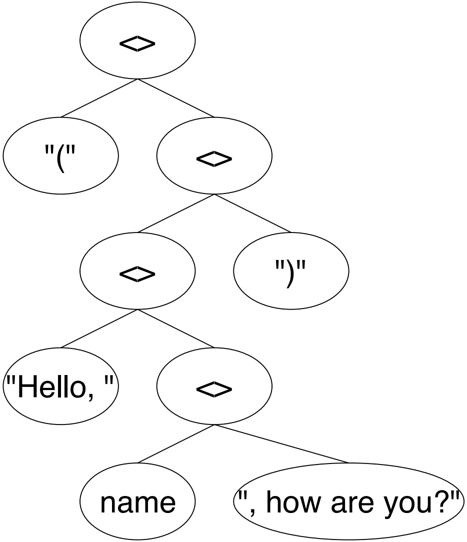
```haskell
greeting :: [Char] -> [Char]
greeting name =
  "(" <> "Hello, " <> name <> ", how are you?" <> ")"
```

```haskell
greeting :: [Char] -> [Char]
greeting name =
  "(" <> "Hello, " <> name <> ", how are you?" <> ")"



between op cl x =
  op <> x <> cl
```

```haskell
greeting :: [Char] -> [Char]
greeting name =
  between "(" ")" $
          "Hello, " <> name <> ", how are you?"


between op cl x =
  op <> x <> cl
```

```haskell
greeting :: [Char] -> [Char]
greeting name =
  between "(" ")" $
          between "Hello, " ", how are you?"
            name

between op cl x =
  op <> x <> cl
```


<>
"("
<>
"Hello, "
<>
name
<>
", how are you?"
")"


<>
"("
<>
<>
")"
"Hello, "
<>
name
", how are you?"

Laws let us **refactor** and **reuse** more

```
([1,2,3] <> [4,5,6]) <> [7,8,9]
```

```
([1,2,3] <> [4,5,6]) <> [7,8,9]
```

:(

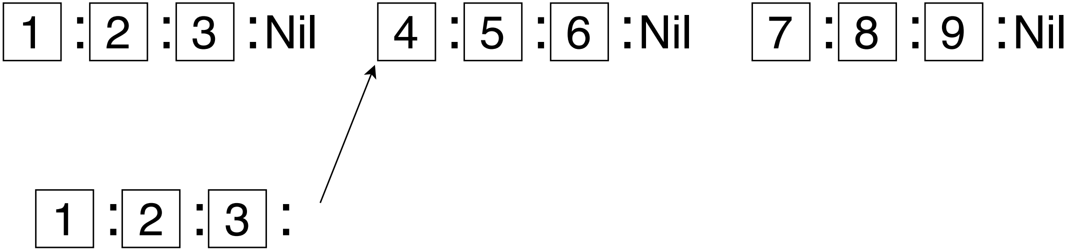
$\boxed{1} : \boxed{2} : \boxed{3} : \mathrm{Nil}$ $\quad$ $\boxed{4} : \boxed{5} : \boxed{6} : \mathrm{Nil}$ $\quad$ $\boxed{7} : \boxed{8} : \boxed{9} : \mathrm{Nil}$

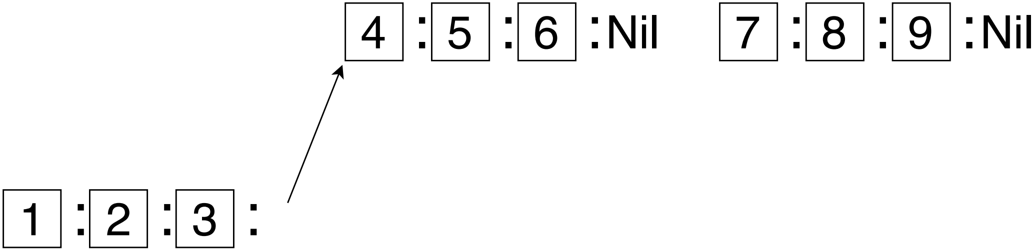
$\boxed{1} : \boxed{2} : \boxed{3} : \mathrm{Nil} \quad \boxed{4} : \boxed{5} : \boxed{6} : \mathrm{Nil} \quad \boxed{7} : \boxed{8} : \boxed{9} : \mathrm{Nil}$

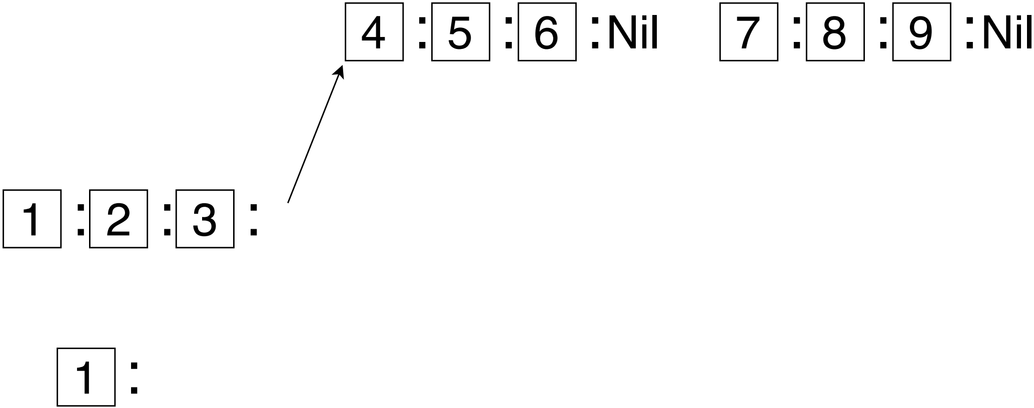
$\boxed{1} :$

$\boxed{1} : \boxed{2} : \boxed{3} : \text{Nil} \quad \boxed{4} : \boxed{5} : \boxed{6} : \text{Nil} \quad \boxed{7} : \boxed{8} : \boxed{9} : \text{Nil}$

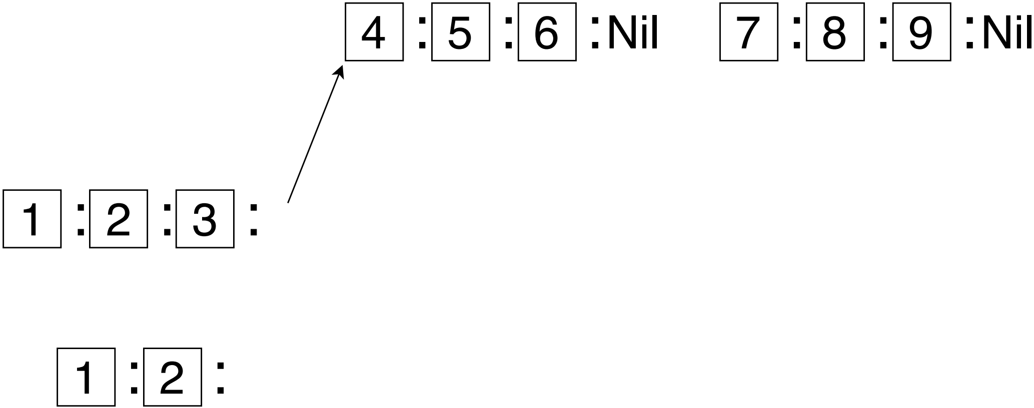
$\boxed{1} : \boxed{2} :$

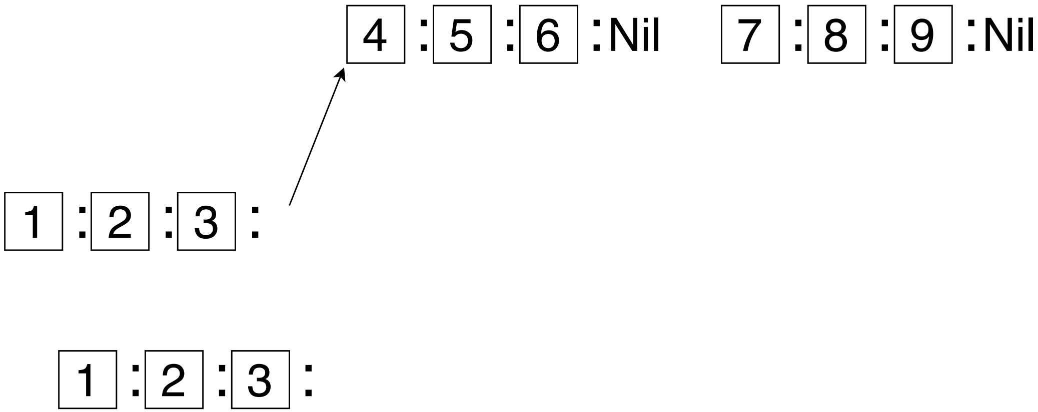

1 : 2 : 3 : Nil
4 : 5 : 6 : Nil
7 : 8 : 9 : Nil
1 : 2 : 3 :

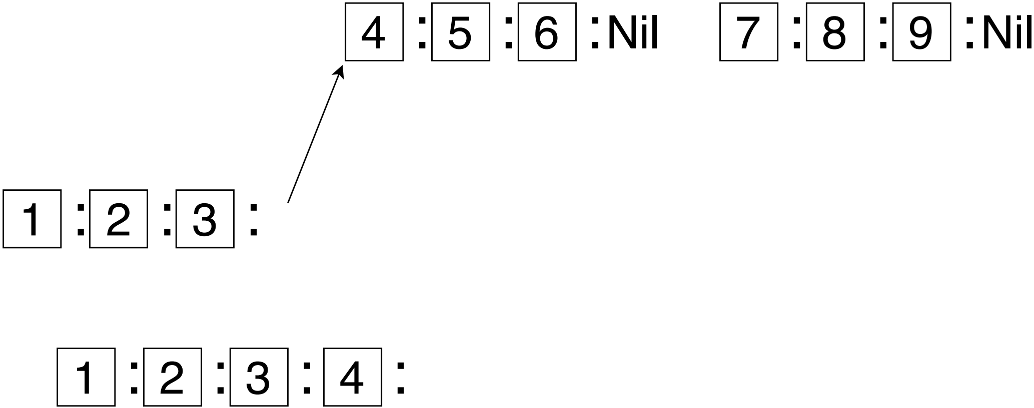

4 : 5 : 6 : Nil
7 : 8 : 9 : Nil
1 : 2 : 3 :

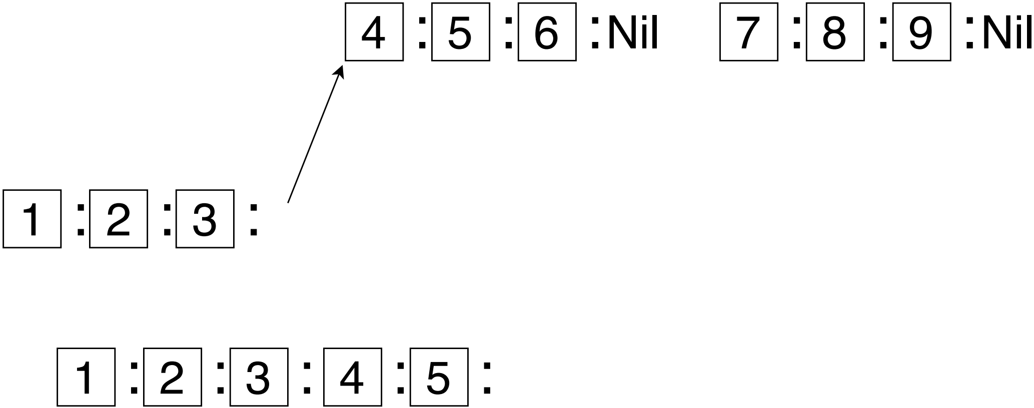


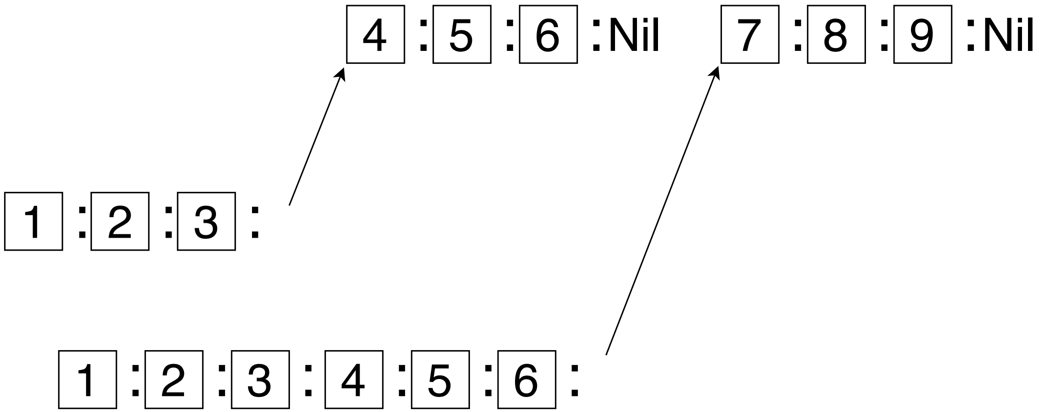
4 : 5 : 6 : Nil
7 : 8 : 9 : Nil
1 : 2 : 3 :
1 :


4 : 5 : 6 : Nil
7 : 8 : 9 : Nil
1 : 2 : 3 :
1 : 2 :


4 : 5 : 6 : Nil
7 : 8 : 9 : Nil
1 : 2 : 3 :
1 : 2 : 3 :


4 : 5 : 6 :Nil
7 : 8 : 9 :Nil
1 : 2 : 3 :
1 : 2 : 3 : 4 :


4 : 5 : 6 : Nil
7 : 8 : 9 : Nil
1 : 2 : 3 :
1 : 2 : 3 : 4 : 5 :


4 : 5 : 6 : Nil
7 : 8 : 9 : Nil
1 : 2 : 3 :
1 : 2 : 3 : 4 : 5 : 6 :

```
data DList a
```

```haskell
data DList a

instance Monoid (DList a)                  -- O(1) append
```

```haskell
data DList a

instance Monoid (DList a)                    -- O(1) append

fromList :: [a]     -> DList a               -- O(1)
toList   :: DList a -> [a]                   -- O(n)
```

```
result :: [a]
result = (((((((x <> y) <> z) <> ...
```

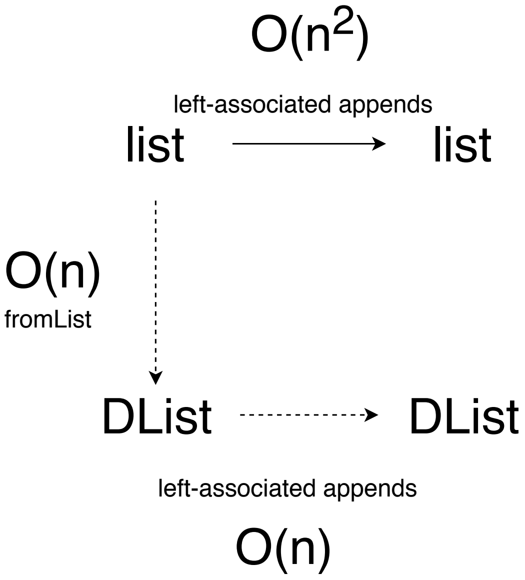
```haskell
result :: [a]
result = (((((((x <> y) <> z) <> ...
```

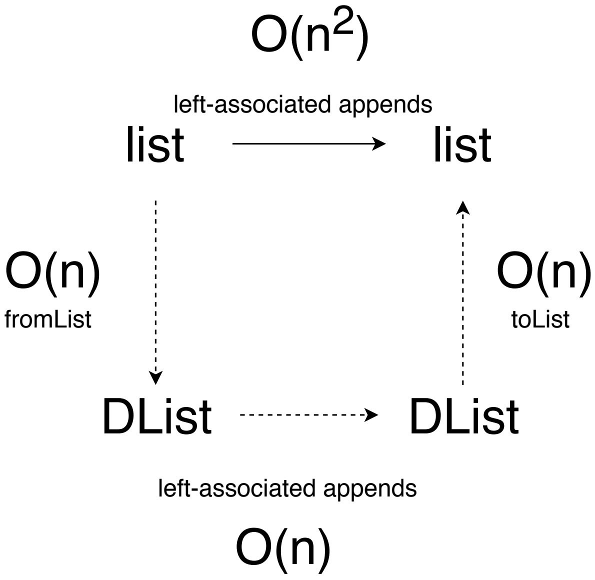
```haskell
appended :: DList a
appended = (((((((fromList x <> fromList y) <> fromList z) <> ...

result' :: [a]
result' = toList appended
```

$O(n^2)$

$$\text{list} \xrightarrow{\text{left-associated appends}} \text{list}$$

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\text{list} \arrow[r, "\substack{O(n^2) \\ \text{left-associated appends}}"] \arrow[d, dashed, "\substack{O(n) \\ \text{fromList}}"'] & \text{list} \\
\text{DList} &
\end{tikzcd}
$$


O(n2)
left-associated appends
list
list
O(n)
fromList
DList
DList
left-associated appends
O(n)



O(n2)
left-associated appends
list
list
O(n)
fromList
O(n)
toList
DList
DList
left-associated appends
O(n)

Optimisation is altering the program
to get **the same answer**
more efficiently

`toList` is the left inverse of `fromList`

```
toList (fromList x) = x
```

`fromList` is a monoid homomorphism

```
fromList :: [a] -> DList a
```

`fromList` is a monoid homomorphism

```
fromList :: [a] -> DList a
```

```
fromList mempty = mempty
```

```
fromList(x <> y) = fromList x <> fromList y
```

```
x <> y <> z
```

```
x <> y <> z
```

Left inverse: `toList (fromList (x)) = x`

```
toList (fromList (x <> y <> z))
```

Left inverse: `toList (fromList (x)) = x`

```
toList (fromList (x <> y <> z))
```

Monoid homomorphism:

```
fromList (x <> y <> z)
  = fromList x <> fromList y <> fromList z
```

```
toList (fromList x <> fromList y <> fromList z)
```

Monoid homomorphism:

```
fromList (x <> y <> z)
  = fromList x <> fromList y <> fromList z
```

What about a world without laws?

```haskell
class Default a where
  def :: a
```

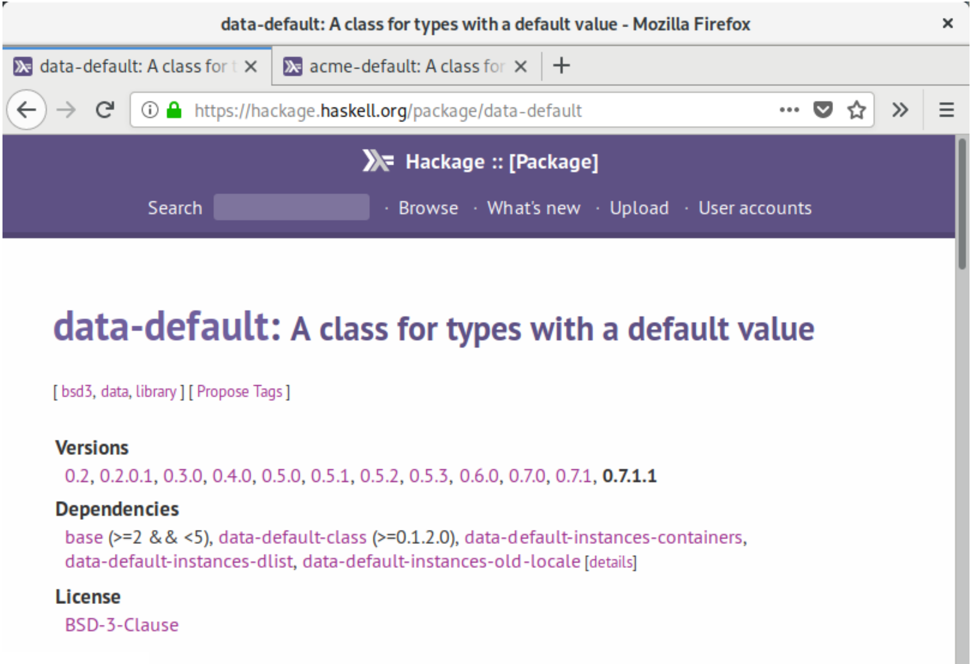
```haskell
class Default a where
  def :: a

instance Default [a] where
  def = []
```

```haskell
class Default a where
  def :: a

instance Default [a] where
  def = []

instance Default Int where
  def = 0
```

```haskell
orDefault :: Default a => Maybe a -> a
orDefault ma =
  case ma of
    Just a  -> a
    Nothing -> def
```

```haskell
orDefault :: Default a => Maybe a -> a
orDefault ma =
  case ma of
    Just a  -> a
    Nothing -> def
```

```haskell
orElse :: a -> Maybe a -> a
orElse d ma =
  case ma of
    Just a  -> a
    Nothing -> d
```


data-default: A class for types with a default value - Mozilla Firefox

data-default: A class for t | acme-default: A class for

https://hackage.haskell.org/package/data-default

Hackage :: [Package]

Search · Browse · What's new · Upload · User accounts

data-default: A class for types with a default value

[bsd3, data, library] [Propose Tags]

Versions

0.2, 0.2.0.1, 0.3.0, 0.4.0, 0.5.0, 0.5.1, 0.5.2, 0.5.3, 0.6.0, 0.7.0, 0.7.1, **0.7.1.1**

Dependencies

base (>=2 && <5), data-default-class (>=0.1.2.0), data-default-instances-containers, data-default-instances-dlist, data-default-instances-old-locale [details]

License

BSD-3-Clause


acme-default: A class for types with a distinguished aesthetically pleasing value - Mozilla Firefox
data-default: A class for t
acme-default: A class for
https://hackage.haskell.org/package/acme-default
Hackage :: [Package]
Search
Browse
What's new
Upload
User accounts
acme-default: A class for types with a distinguished aesthetically pleasing value
[acme, library] [Propose Tags]
This package defines a type class for types with certain distinguished values that someone considers to be aesthetically pleasing. Such a value is commonly referred to as a default value.
This package exists to introduce artistic variety regarding the aesthetics of Haskell's base types, but is otherwise identical in purpose to data-default.
[Skip to Readme]

```haskell
-- | Current default -1 chosen by ertes,
--   the largest negative number.
instance Default Int64 where
  def = -1
```

```haskell
-- | Current default -1 chosen by ertes,
--   the largest negative number.
instance Default Int64 where
  def = -1


-- | Current default 'False' chosen by ertes,
--   the answer to the question
--   whether mniip has a favourite 'Bool'.
instance Default Bool where
  def = False
```

```haskell
-- | Current default -1 chosen by ertes,
--   the largest negative number.
instance Default Int64 where
  def = -1


-- | Current default 'False' chosen by ertes,
--   the answer to the question
--   whether mniip has a favourite 'Bool'.
instance Default Bool where
  def = False


instance Default String where
  def = "Call me Ishmael. Some years ago - never mind how long preci
```

How do I know whether I obey the laws?

`QuickCheck + checkers`

Property-based testing for laws!

```haskell
monoid :: (Monoid a, Show a, Arbitrary a, EqProp a)
       => a -> TestBatch
```

```haskell
monoid :: (Monoid a, Show a, Arbitrary a, EqProp a)
       => a -> TestBatch

functor :: (Functor t,
            Arbitrary a, Arbitrary b, Arbitrary c,
            CoArbitrary a, CoArbitrary b,
            Show (t a),
            Arbitrary (t a), EqProp (t a), EqProp (t c))
        =>  t (a, b, c) -> TestBatch
```

```haskell
data Subtraction = Subt Int

-- totally dodgy
instance Monoid Subtraction where
  mempty              = Subt 0
  Subt x <> Subt y = Subt (x - y)
```

```haskell
data Subtraction = Subt Int

-- totally dodgy
instance Monoid Subtraction where
  mempty            = Subt 0
  Subt x <> Subt y = Subt (x - y)


main :: IO ()
main = do
  quickBatch (monoid (Sum 0))
  quickBatch (monoid (Subt 0))
```

```
Sum monoid:
  left  identity: +++ OK, passed 500 tests.
  right identity: +++ OK, passed 500 tests.
  associativity:  +++ OK, passed 500 tests.

Subtraction "monoid":
  left  identity: *** Failed! Falsifiable (after 2 tests)
  right identity: +++ OK, passed 500 tests.
  associativity:  *** Failed! Falsifiable (after 2 tests)
```

Laws give rise to useful functions

Laws allow us to refactor more

Laws help us to optimise

Laws are the difference between
an **overloaded name**
and an **abstraction**

Thanks for listening!

References

- Daniel J. Velleman “How To Prove It”
- Edward Kmett “Introduction to Monoids” http://comonad.com/reader/wp-content/uploads/2009/08/IntroductionToMonoids.pdf
- Tom Ellis “Demystifying DList” http://h2.jaguarpaw.co.uk/posts/demystifying-dlist/
- Edward Kmett “Why not Pointed?” https://wiki.haskell.org/Why_not_Pointed%3F
- Tim Humphries “Continuations All The Way Down” http://lambdajam.yowconference.com.au/slides/yowlambdajam2017/Humphries-ContinuationsAllTheWayDown.pdf
- Edward Kmett “The Free Theorem for fmap” https://www.schoolofhaskell.com/user/edwardk/snippets/fmap

What's up with Foldable?

It sort of has laws.

- Gershom Bazerman wrote a paper:
  `http://gbaz.github.io/slides/buildable2014.pdf`
- Then started a mailing list discussion:
  `https://mail.haskell.org/pipermail/libraries/2015-February/024943.html`
- ...and then another one:
  `https://mail.haskell.org/pipermail/libraries/2018-May/028761.html`

Are there reasonable cases of law breakage?

Are there reasonable cases of law breakage?

Yes! Both `QuickCheck` and `hedgehog` break the `Applicative` and `Monad` laws