PROGRAMMING IN HASKELL

Chapter 11.1 – Semigroups and Monoids

Composability

e.g. to use ++ for composing a string:

```
"this" ++ " " ++ "is" ++ " " ++
"a" ++ " " ++ "bit" ++ " " ++ "much"
```

Yes, it is a bit too much! We need a better way!

Composability

Composability means that you create something new by combining two like things, e.g.

```
myLast :: [a] -> a
myLast = head . reverse

myMin :: Ord a => [a] -> a
myMin = head . sort

myMax :: Ord a => [a] -> a
myMax = myLast . sort
```

Composing types – Semigroups and Monoids

A *semigroup* is a type class that has one class has only one important method you need, the <> operator.

You can think of <> as an operator for combining instances of the same type. You can trivially implement Semigroup for Integer by defining <> as +.

```
instance Semigroup Integer where
    (<>) x y = x + y
```

Semigroups

Note the type of <>

```
(<>) :: Semigroup a => a -> a -> a
```

This simple signature is the heart of the idea of composability

You can take two like things and combine them to get a new thing of the same type.

Semigroups - example

Looking at combining colours:

- Adding Blue and yellow make green.
- Red and yellow make orange.
- Blue and red make purple.

```
data Color = Red |
   Yellow |
   Blue |
   Green |
   Purple |
   Orange |
   Brown deriving (Show,Eq)
```

Semigroups - example


```
instance Semigroup Color where
    (<>) Red Blue = Purple
    (<>) Blue Red = Purple
    (<>) Yellow Blue = Green
    (<>) Blue Yellow = Green
    (<>) Yellow Red = Orange
    (<>) Red Yellow = Orange
    (<>) a b = if a == b
               then a
               else Brown
```

How we combine the colours

Semigroups - associativity

Associativity : e.g.
(1 + (2 + 3)) ==
(1 + 2) + 3

```
instance Semigroup Color where
    (<>) Red Blue = Purple
    (<>) Blue Red = Purple
    (<>) Yellow Blue = Green
    (<>) Blue Yellow = Green
    (<>) Yellow Red = Orange
    (<>) Red Yellow = Orange
    (<>) a b | a == b = a
             | all (`elem` [Red,Blue,Purple]) [a,b] = Purple
             | all (`elem` [Blue,Yellow,Green]) [a,b] = Green
             | all (`elem` [Red,Yellow,Orange]) [a,b] = Orange
             | otherwise = Brown
```

Theoretically, all instances of semigroups should be associative but these laws are not enforceable by the Haskell compiler. Associativity should be implemented.

Semigroups -> Monoids

A monoid is a semigroup

Monoids are similar to semigroups

we call <> mappend

we have mempty (identity)

we have mconcat

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

Monoids


```
GHCi> [1,2,3] ++ []
[1,2,3]

GHCi> [1,2,3] <> []
[1,2,3]

GHCi> [1,2,3] `mappend` mempty
[1,2,3]
```

All three are the same

Monoids

Type definition of mconcat:

```
mconcat :: Monoid a => [a] -> a
```

Takes a list and

```
GHCi> mconcat ["does"," this"," make"," sense?"]
"does this make sense?"
```

returns the flattened list

Monoids

Definition of mconcat:

```
mconcat = foldr mappend mempty
```

++

[]

```
GHCi> mconcat ["does"," this"," make"," sense?"]
"does this make sense?"
```

```
“does” : ( “this” : (" make”: (" sense?”: mempty ) ) )
```

++

[]


ANY
QUESTIONS?