Category Theory for Programmers Cheat Sheet

compiled by Fabricio Olivetti de França

How to use this guide

- Match a type signature with one of those in the following boxes.
- Check if the properties hold.
- Create an instance of the matched class.

Monoid

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

Properties:

```
x <> mempty = mempty <> x = x
(a <> b) <> c = a <> (b <> c)
a <> b = b <> a
```

(the last property only for commutative Monoids)

Semiring

```
class Semiring a where
    plus  :: a -> a -> a
    times :: a -> a -> a
    zero  :: a
    one   :: a
```

Properties:

```
zero + x = x + zero = x
x + (y + z) = (x + y) + z
x + y = y + x
one * x = x * one = x
x * (y * z) = (x * y) * z
x * (y + z) = (x * y) + (x * z)
(x + y) * z = (x * z) + (y * z)
zero * x = x * zero = zero
```

Ring

```
class Semiring a => Ring a where
    negate  :: a -> a
```

Properties:

```
negate x + x = zero
```

Functor

```
class Functor F where
    fmap :: (a -> b) -> F a -> F b
    (<$>) = fmap
```

Properties:

```
fmap id = id
fmap (g . f) = fmap g . fmap f
```

Contravariant

```
class Contravariant f where
    contramap :: (a -> b) -> f b -> f a
    (>$) :: b -> f b -> f a
```

Properties:

```
contramap id = id
contramap f . contramap g = contramap (g . f)
```

Representable Functor

```
class Contravariant f => Representable f where
    type Rep f :: *
    tabulate :: (a -> Rep f) -> f a
    index :: f a -> a -> Rep f
```

Properties:

```
tabulate . index    = id
index    . tabulate = id
```

Adjunction

```
class (Functor f, Representable y) =>
  Adjunction f u | f -> u, u -> f where
    unit   :: a -> u (f a)
    counit :: f (u a) -> a
    leftAdjunct  :: (f a -> b) -> a -> u b
    rightAdjunct :: (a -> u b) -> f a -> b
```

Properties:

```
unit         = leftAdjunct id
counit       = rightAdjunct id
leftAdjunct  = fmap g . unit
rightAdjunct = counit . fmap g
```

Applicative

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>)  :: f a -> f b -> f b
  (<*)  :: f a -> f b -> f a
```

Properties:

```
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

Monad

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  join   :: m (m a) -> m a
```

Properties:

```
return a >>= k  =  k a
m >>= return  =  m
pure f <*> pure x = pure (f x)
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Comonad

```
class Functor w => Comonad w where
  extract   :: w a -> a
  duplicate :: w a -> w (w a)
  extend    :: (w a -> b) -> w a -> w b
  (=>>)     :: w a -> (w a -> b) -> w b
```

Properties:

```
extend extract      = id
extract . extend f  = f
extend f . extend g = extend (f . extend g)
```