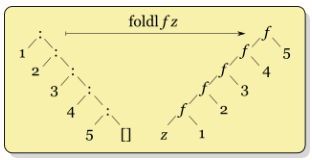
General libraries (base ≥ 4.8)

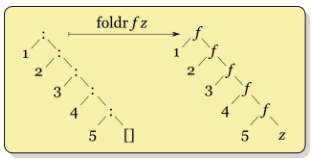
Right associative (foldr)

If you operating over potentially infinite structures and/or building another structure, then you probably want **foldr**.



Left associative (foldl)

If you are reducing to a single value, then you will probably get more performance from a strict left fold (**foldl'**).




```
foldl :: Foldable t => (a -> b -> a) -> a -> t b -> a
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

toList :: Foldable t => t a -> [a]

and, or               :: Foldable t => t Bool -> Bool
any, all              :: Foldable t => (a -> Bool) -> t a -> Bool
sum, product          :: (Foldable t, Num a) => t a -> a
minimum, maximum      :: (Foldable t, Ord a) => t a -> a
minimumBy, maximumBy :: Foldable t => (a -> a -> Ordering) -> t a -> a

elem :: (Foldable t, Eq a) => a -> t a -> Bool
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

```
> foldl' (flip (:)) [0] [1,2,3]
[3,2,1,0]

> foldr (:) [5] [1,2,3,4]
[1,2,3,4,5]

> take 5 $ foldr (:) [] [1..]
[1,2,3,4,5]
```

```
> all even [1,2,3]
False

> any even [1,2,3,undefined]
True

> find (> 42) [1..]
Just 43
```

Applicative Traversals/Folds

```
traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)
for      :: (Traversable t, Applicative f) => t a -> (a -> f b) -> f (t b)
sequenceA :: Applicative f => t (f a) -> f (t a)

> for [1000000, 2000000] $ \t -> threadDelay t >> getCurrentTime
[2015-04-29 05:24:30.040399 UTC,2015-04-29 05:24:32.042665 UTC]
```

Functor

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
```

Control.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
```

Control.Monad

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

(>=>) :: Monad m
      => (a -> m b) -> (b -> m c) -> a -> m c

join  :: Monad m => m (m a) -> m a
ap    :: Monad m => m (a -> b) -> m a -> m b
```