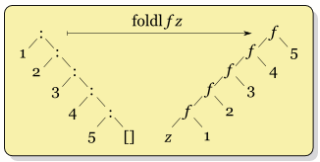
Base – Folds and Typeclasses

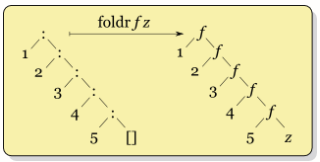
Left associative (foldl)

If you are **reducing to a single value,** then you may get more performance from a strict left **foldl'**.



Right associative (foldr)

If what you are reducing is **potentially infinite**, or you are **building a structure**, use **foldr**.




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

Base – Lists and misc

Data.List

```
intersperse :: a -> [a] -> [a]
> intersperse ',' "abcde" == "a,b,c,d,e"

intercalate :: a -> [a] -> [a]
> intercalate " love " ["ponies", "ducks"]
"ponies love ducks"

subsequences, permutations :: [a] -> [[a]]
> subsequences "abc"
["","a","b","ab","c","ac","bc","abc"]
> permutations "abc"
["abc","bac","cba","bca","cab","acb"]

scanl :: (b -> a -> b) -> b -> [a] -> [b]
> scanl f z [x1, x2, ...]
[z, z `f` x1, (z `f` x1) `f` x2, ...]
> take 8 $ fix (\fib -> scanl (+) 1 (0:fib))
[1,1,2,3,5,8,13,21]

iterate :: (a -> a) -> a -> [a]
> iterate f x
[x, f x, f (f x), ...]
> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]

replicate :: Int -> a -> [a]
repeat :: a -> [a]
cycle :: [a] -> [a]

 :: Int -> [a] -> ([a], [a])
> splitAt n xs
(take n xs, drop n xs)

takeWhile, dropWhile
  :: (a -> Bool) -> [a] -> [a]
> takeWhile (< 3) [1,2,3,4,1,2,3,4]
[1,2]

isPrefixof, isSuffixOf, isInfixOf
  :: Eq a => [a] -> [a] -> Bool

lines, words :: String -> [String]
words, lines :: [String] -> String

nub :: Eq a => [a] -> [a]
> nub [1,2,2,3,2]
[1,2,3]

delete :: Eq a => a -> [a] -> [a]
> delete 'a' "banana"
"bnana"

(\\), union, intersect
  :: Eq a => [a] -> [a] -> [a]
> (xs ++ ys) \\ xs -- difference
ys
```

Commonly re-defined

```
strip :: String -> String
strip =
  join fmap (reverse . dropWhile isSpace)

> strip "  a "
"a"

pairs :: [a] -> [(a, a)]
pairs = zip <*> tail

> pairs [1]
[]
> pairs [1,2,3]
[(1,2),(2,3)]
```

Data.Function

```
fix :: (a -> a) -> a
> fix $ \f n ->
>   if n < 0 then [] else n : f (n - 1)) 5
[5,4,3,2,1,0]

on :: (b -> b -> c)
   -> (a -> b) -> a -> a -> c
> (*) `on` f
\x y -> f x * f y.
> sortBy (compare `on` length) ["bb", "a"]
["a","bb"]
```

Debug.Trace

The usual output stream is **stderr**.

```
trace       :: String -> a -> a
traceShow   :: Show a => a -> b -> b
traceShowId :: Show a => a -> a
traceStack  :: String -> a -> a
traceIO     :: String -> IO ()
traceM      :: Monad m => String -> m ()
traceShowM  :: (Show a, Monad m) => a -> m ()

> trace ("call f with x = " ++ show x) (f x)
> g x y = traceShow (x, y) (x + y)
```

Control.Arrow

```
class Category a => Arrow a where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b,d) (c,d)
  second :: a b c -> a (d,b) (d,c)
  (***) :: a b c -> a b' c' -> a (b,b') (c,c')
  (&&&) :: a b c -> a b c' -> a b (c,c')
```