**Catamorphism**

*"Cata"* means "down" or "against", as in "catacombs". Catamorphisms are means of deconstructing data. If the spine of a list is the structure of a list, then a fold is what can reduce that structure.

Where a fold allows to break down a list into an arbitrary datatype, a catamorphism is a means of breaking down the structure of any datatype (bool func).

# 1 Fold right

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z xs =
    case xs of
    []      -> z
    (x:xs)  -> f x (foldr f z xs) --<---- this one
                -- |rest of the fold|

```

If *f* doesn't evaluate its 2nd argument (rest of the fold), no more spine will be forced. For this reason, foldr can be used with infinite lists.

In other words,

```
foldr (+) 0 [2, 3] =
case [2, 3] of
    [] -> 0 -- this didn't match again
    (2 : [3]) -> (+) 2 (foldr (+) 0 [3]) -- (1+(2+(3+0)))
```

Since `(+)` is **strict in both arguments**, and also *unconditionally* so, so it jump to the **next recursion**. Bouncin' between *f* - `foldr`, give controls to the folding functions.

The difference between `foldl` and `foldr` is just how it associates, or - direction of folding.

```
-- not strict in both args
Prelude> myAny even [1..]
True
-- but not this
Prelude> myAny even (repeat 1) -- bottom

```

The first piece of the spine, the first `:` (cons) can't be `undefined` for folding. Since `f x` forces the `(x:xs)`.

```
foldr f z (x:xs) = f x (foldr f z xs)
      -- see there
Prelude> foldr (\_ _ -> 9001) 0 [undefined, undefined] --9001
Prelude> foldr (\_ _ -> 9001) 0 ([1, 2, 3] ++ undefined) -- 9001
Prelude> foldr (\_ _ -> 9001) 0 undefined -- or undefined ++ [1,2]
*** Exception: Prelude.undefined
```

# 2 Fold left

Because `foldl` evaluate its whole spine before it starts evaluating in each cell, it accumulates a pile of unevaluated values as it traverses the spine.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []      = acc
foldl f acc (x:xs)  = foldl f (f acc x) xs
-- ((0+1)+2)+3)
```

`foldl'` (foldl prime) works the same except it is strict, has **less negative effect** on performance over long lists. Only beginning to produce values **after reaching the end of the list**. Nearly useless, gotta use foldl'.

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q ls =
q : (case ls of                -- Wow, this is possible!
        [] -> []
        x:xs -> scanl f (f q x) xs)

-- Fibonacci numbers
fibs = 1 : scanl (+) 1 fibs
fibsN x = fibs !! x
```