Cheatsheet

Recursive

project

```haskell
-- project swaps the first layer of your type with the recursive functor
-- version leaving the next layer untouched
-- >>> project [1, 2, 3]
-- Cons 1 [2,3]
```

cata

```haskell
-- >>> sumList [1, 2, 3]
-- 6

sumList :: [Int] -> Int
sumList = cata alg
 where
  alg :: ListF Int Int -> Int
  alg Nil               = 0
  alg (Cons next total) = next + total
```

para

```haskell
-- >>> paraTails [1, 2, 3, 4]
-- [[1,2,3,4],[2,3,4],[3,4],[4]]

paraTails :: [a] -> [[a]]
paraTails = para alg
 where
  alg :: ListF a ([a], [[a]]) -> [[a]]
  alg Nil                   = []
  alg (Cons x (xs, tails')) = (x : xs) : tails'
```

zygo

```haskell
-- >>> zygoDedupeList [1, 1, 3, 2, 3]
-- [1,2,3]

zygoDedupeList :: [Int] -> [Int]
```

```haskell
zygoDedupeList = zygo setBuilderAlg listBuilderAlg
 where
  setBuilderAlg :: ListF Int (S.Set Int) -> S.Set Int
  setBuilderAlg Nil              = S.empty
  setBuilderAlg (Cons n allNums) = S.insert n allNums
  listBuilderAlg :: ListF Int (S.Set Int, [Int]) -> [Int]
  listBuilderAlg Nil = []
  listBuilderAlg (Cons n (allNums, ns)) =
    if S.member n allNums then ns else n : ns
```

Co-Recursive

ana

```haskell
-- >>> countDown 5
-- [5,4,3,2,1]

countDown :: Int -> [Int]
countDown = ana coalg
 where
  coalg :: Int -> ListF Int Int
  coalg 0 = Nil
  coalg n = Cons n (n - 1)
```