New York Yankees
$114,457,768
VS
$39,722,689
Oakland Athletics

RUNS SCORED² / RUNS SCORED² + RUNS ALLOWED² = WIN %
WIN %
RECORD
EXPECTED
ACTUAL
EXPECTED
ACTUAL
OAK 2001
884² / 884² + 645² = 781456 / 1197481 =
.6525
.6296
106 - 56
102 - 60
OAK 2002 PROJECTION
814² / 814² + 645² = 662596 / 1078621 =
* LOSE GIAMBI, DAMON
.6143
99 - 63
0-0
SEA 2001
927² / 927² + 627² = 859329 / 1252458 =
.6861
.7160
111 - 51
850² = 722500 / 1184900 =
.6097
EDGAR AGES
99 - 63
804² / 804² + 713² = 646416 / 1154785 =
.5597
.5864
91 - 71
890² / 890² + 700² = 792100 / 1282100 =
PROJECTION
* LOSE JUSTICE, GAIN →
.6178
GIAMBI, VENTURA
100 - 62

001

$$\frac{884^2}{884^2+645^2} = \frac{781456}{1197481} =$$

AK
002
OJECTION

$$\frac{814^2}{814^2+645^2} = \frac{662596}{}$$

* LOSE GIAM.

EA

$$\frac{927^2}{927^2+627^2} = \frac{859329}{1252458} =$$

An Introduction to Category Theory

Kyle Oba
@mudphone
pasdechocolat.com

Swift

An Introduction to Functional Programming with Haskell

Kyle Oba
@mudphone
pasdechocolat.com

github.com/PasDeChocolat/LearningSwift

Map, Filter, Reduce

Map, Filter, Reduce

```
/// Haskell's fmap for Optionals.
func map<T, U>(x: T?, f: (T) -> U) -> U?


/// Return an `Array` containing the results of mapping `transform`
/// over `source`.
func map<C : CollectionType, T>(source: C, transform: (C.Generator.Element) -> T) -> [T]


/// Return an `Array` containing the results of mapping `transform`
/// over `source`.
func map<S : SequenceType, T>(source: S, transform: (S.Generator.Element) -> T) -> [T]
```

Extension on Array…

```
/// Return an `Array` containing the results of calling
/// `transform(x)` on each element `x` of `self`
func map<U>(transform: (T) -> U) -> [U]
```

See also ContiguousArray, ImplicitlyUnwrappedOptional, LazyBidirectionalCollection, LazyForwardCollection, LazyRandomAccessCollection, LazySequence, Optional, Range, Slice

Map, Filter, Reduce

```
/// Return an `Array` containing the elements of `source`,
/// in order, that satisfy the predicate `includeElement`.
func filter<S : SequenceType>(source: S, includeElement: (S.Generator.Element) -> Bool) -> [S.Generator.Element]
```

Extension on Array…

```
/// Return an `Array` containing the elements `x` of `self` for which
/// `includeElement(x)` is `true`
func filter(includeElement: (T) -> Bool) -> [T]
```

See also ContiguousArray, LazyBidirectionalCollection, LazyForwardCollection, LazyRandomAccessCollection, LazySequence & Slice

Map, Filter, Reduce

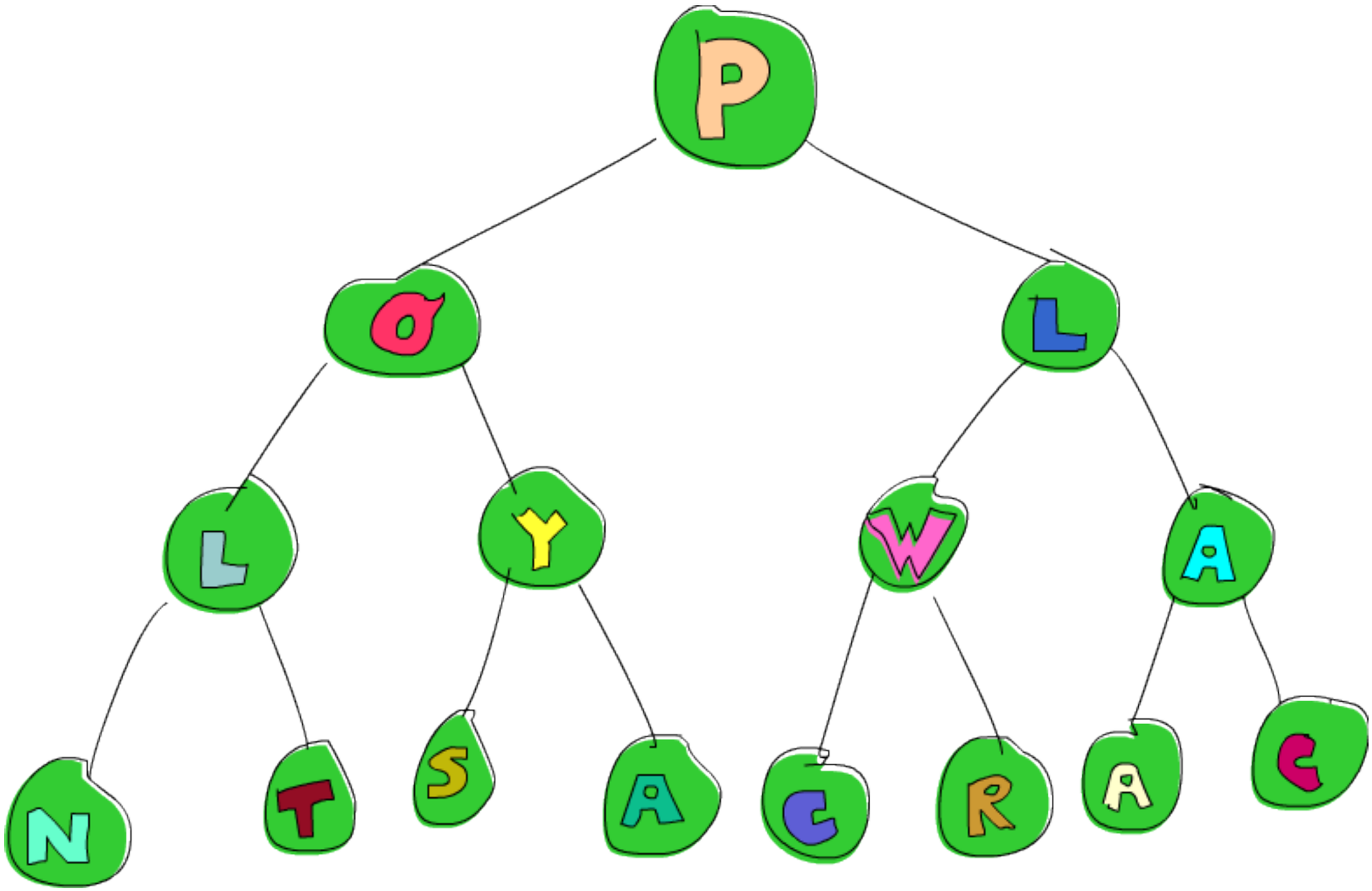
```
/// Return the result of repeatedly calling `combine` with an
/// accumulated value initialized to `initial` and each element of
/// `sequence`, in turn.
func reduce<S : SequenceType, U>(sequence: S, initial: U, combine: (U, S.Generator.Element) -> U) -> U
```

Extension on Array…

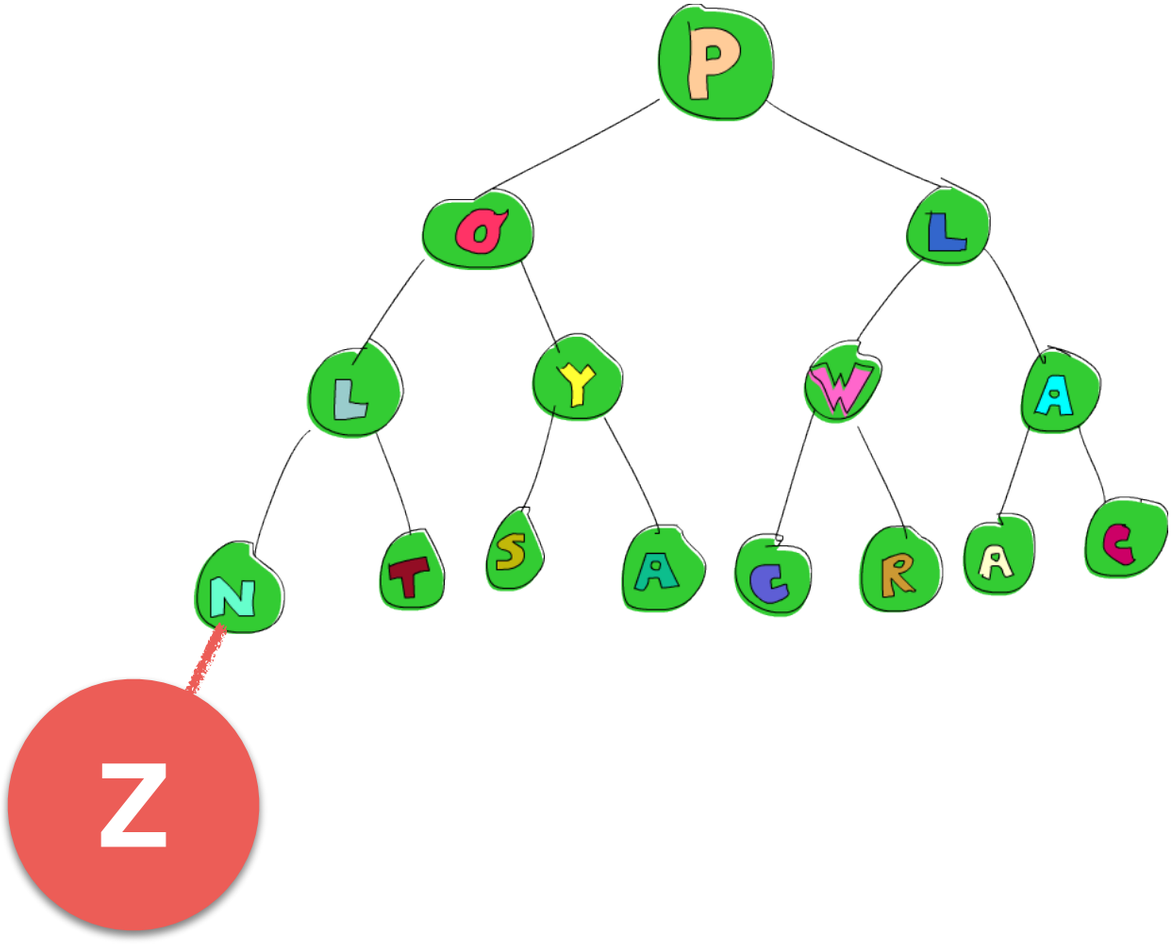
```
/// Return the result of repeatedly calling `combine` with an
/// accumulated value initialized to `initial` and each element of
/// `self`, in turn, i.e. return
/// `combine(combine(...combine(combine(initial, self[0]),
/// self[1]),...self[count-2]), self[count-1])`.
func reduce<U>(initial: U, combine: (U, T) -> U) -> U
```

See also ContiguousArray, & Slice

Function Zippers

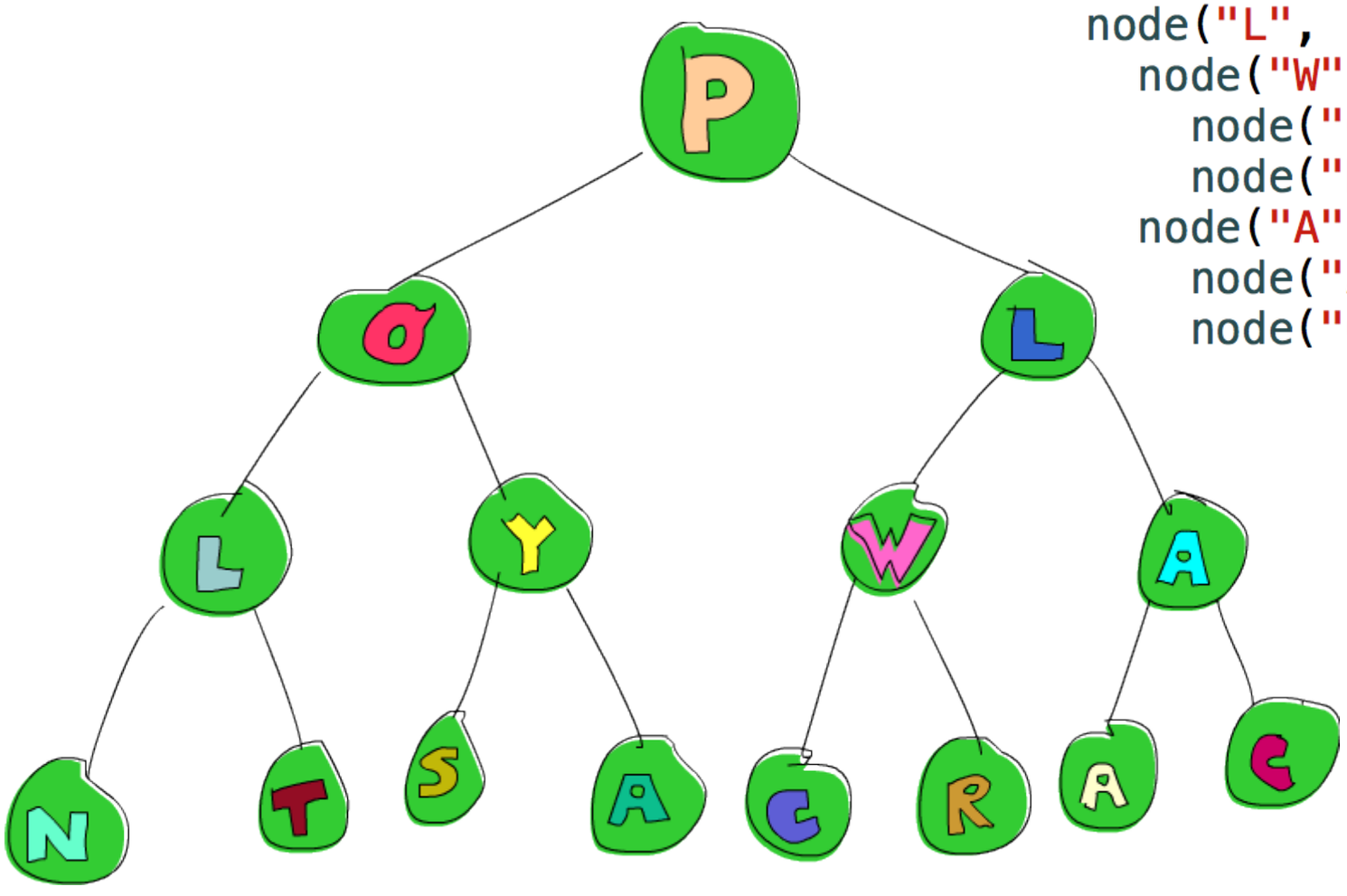

P
σ
L
L
Y
W
A
N
T
S
A
C
R
A
C

```
let freeZipper = (freeTree, [Crumb<String>]())

let farLeft = goLeft( goLeft( goLeft( goLeft( freeZipper )!)!)!)!
let newFocus3 = attach(node("Z", empty, empty), farLeft)
elements(newFocus3.0)
                          => ["Z"]

func goLeft<T>(zipper: (Tree<T>, [Crumb<T>])) -> (Tree<T>, [Crumb<T>])?
```



```
class Box<T> {
  let unbox: T
  init(_ value: T) { self.unbox = value }
}


/*----------------------------------------------------------------------------/
//  Tree
/----------------------------------------------------------------------------*/
enum Tree<T> {
  case Empty
  case Node(Box<T>, Box<Tree<T>>, Box<Tree<T>>)
}

let empty: Tree<String> = Tree.Empty
let five: Tree<String> = Tree.Node(Box("five"), Box(empty), Box(empty))
```

```swift
func node<T>(x: T, l: Tree<T>, r: Tree<T>) -> Tree<T> {
  return Tree.Node(Box(x), Box(l), Box(r))
}

let freeTree = node("P",
                    node("O",
                      node("L",
                        node("N", empty, empty),
                        node("T", empty, empty)),
                      node("Y",
                        node("S", empty, empty),
                        node("A", empty, empty))),
                    node("L",
                      node("W",
                        node("C", empty, empty),
                        node("R", empty, empty)),
                      node("A",
                        node("A", empty, empty),
                        node("C", empty, empty))))
```



```swift
/*----------------------------------------------------------------------*/
//  elements
/*----------------------------------------------------------------------*/
// list elements
func elements<T>(tree: Tree<T>) -> [T] {
  switch tree {
  case .Empty:
    return []
  case let .Node(x, left, right):
    return elements(left.unbox) + [x.unbox] + elements(right.unbox)
  }
}
```

```
/*-----------------------------------------------------------------------*/
//  Breadcrumbs
/*-----------------------------------------------------------------------*/
enum Crumb<T> {
  case Left(Box<T>, Tree<T>)
  case Right(Box<T>, Tree<T>)
}

func goLeft<T>(zipper: (Tree<T>, [Crumb<T>])) -> (Tree<T>, [Crumb<T>])? {
  let (t, crumbs) = zipper
  switch t {
  case .Empty:
    return nil
  case let .Node(x, l, r):
    return (l.unbox, [Crumb.Left(x, r.unbox)] + crumbs)
  }
}

func goRight<T>(zipper: (Tree<T>, [Crumb<T>])) -> (Tree<T>, [Crumb<T>])? {
  let (t, crumbs) = zipper
  switch t {
  case .Empty:
    return nil
  case let .Node(x, l, r):
    return (r.unbox, [Crumb.Right(x, l.unbox)] + crumbs)
  }
}
```

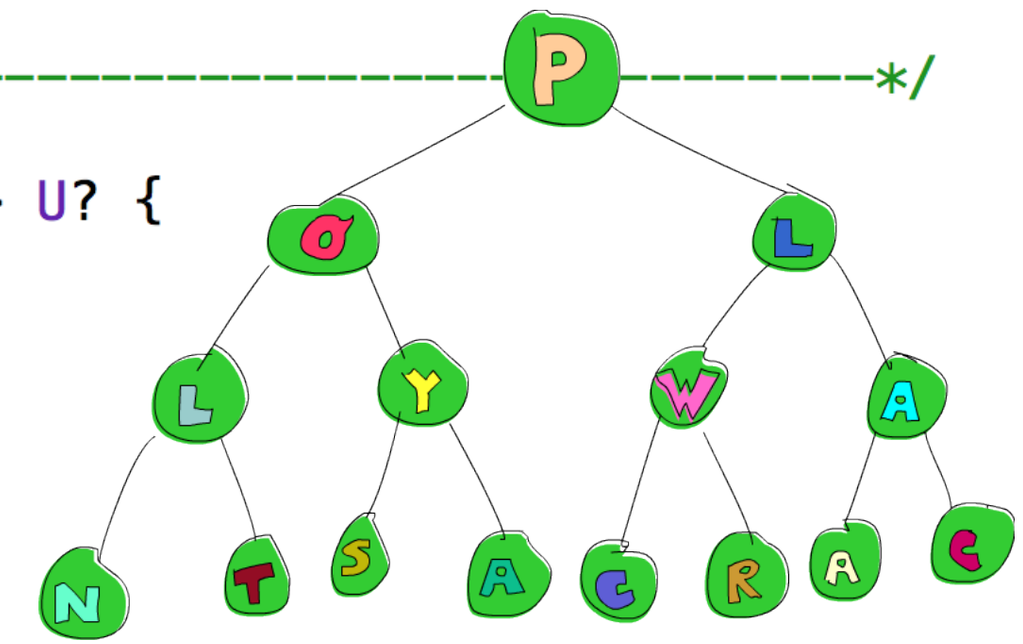
```
/*-----------------------------------------------------------------------------/
//  `decompose` is a helper for `goUp`
/-----------------------------------------------------------------------------*/
extension Array {
  var decompose: (head: T, tail: [T])? {
    return (count > 0) ? (self[0], Array(self[1..<count])) : nil
  }
}

func goUp<T>(zipper: (Tree<T>, [Crumb<T>])) -> (Tree<T>, [Crumb<T>])? {
  let (t, crumbs) = zipper
  if let (c, bs) = crumbs.decompose {
    switch c {
    case let .Left(x, r):
      return (node(x.unbox, t, r), bs)
    case let .Right(x, l):
      return (node(x.unbox, l, t), bs)
    }
  } else {
    return nil
  }
}
```

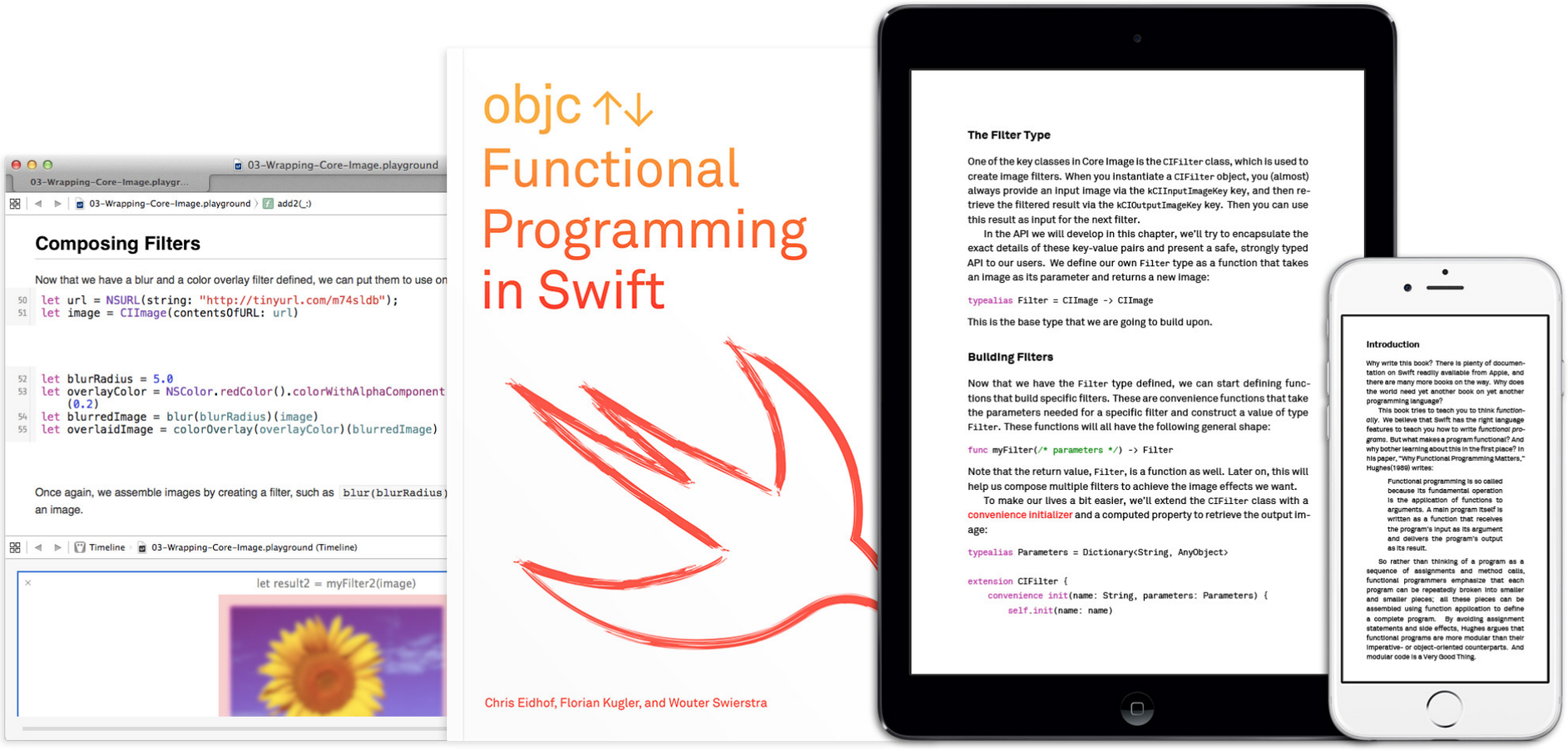
```swift
/*-----------------------------------------------------------------------------/
//  modify
/-----------------------------------------------------------------------------*/
func modify<T>(f: T->T, zipper: (Tree<T>, [Crumb<T>])) -> (Tree<T>, [Crumb<T>]) {
  let (t, crumbs) = zipper
  switch t {
  case .Empty:
    return (Tree.Empty, crumbs)
  case let .Node(x, l, r):
    return (node(f(x.unbox), l.unbox, r.unbox), crumbs)
  }
}


/*-----------------------------------------------------------------------------/
//  attach
/-----------------------------------------------------------------------------*/
func attach<T>(t: Tree<T>, zipper: (Tree<T>, [Crumb<T>])) -> (Tree<T>, [Crumb<T>]) {
  let (_, crumbs) = zipper
  return (t, crumbs)
}


/*-----------------------------------------------------------------------------/
//  topMost
/-----------------------------------------------------------------------------*/
func topMost<T>(zipper: (Tree<T>, [Crumb<T>])) -> (Tree<T>, [Crumb<T>]) {
  let (t, crumbs) = zipper
  if crumbs.isEmpty { return zipper }
  return topMost(goUp( zipper )!)
}
```

```
let farLeft = goLeft( goLeft( goLeft( goLeft( freeZipper )!)!)!)!
let newFocus3 = attach(node("Z", empty, empty), farLeft)
elements(newFocus3.0)
```

=> [“Z”]

```
/*-----------------------------------------------------------------------------/
//  Optionals are monads!... use >>=
/-----------------------------------------------------------------------------*/
infix operator >>= { associativity left}
func >>=<U, T>(optional: T?, f: T -> U?) -> U? {
  return optional.map { f($0)! }
}

func opt<T>(x: T) -> T? {
  return x
}


let bindRight = opt(freeZipper) >>= goRight
elements(bindRight!.0)
                        ["C", "W", "R", "L", "A", "A", "C"]


let bindLeft = opt(freeZipper) >>= goLeft
elements(bindLeft!.0)
                        ["N", "L", "T", "O", "S", "Y", "A"]


let bindLLLeft = opt(freeZipper) >>= goLeft >>= goLeft >>= goLeft
elements(bindLLLeft!.0)
                        ["N"]
```





Functional Programming in Swift
by Chris Eidhof, Florian Kugler, and Wouter Swierstra

Learn You a Haskell for Great Good!
by Miran Lipovača

Thank you.