

New York Yankees

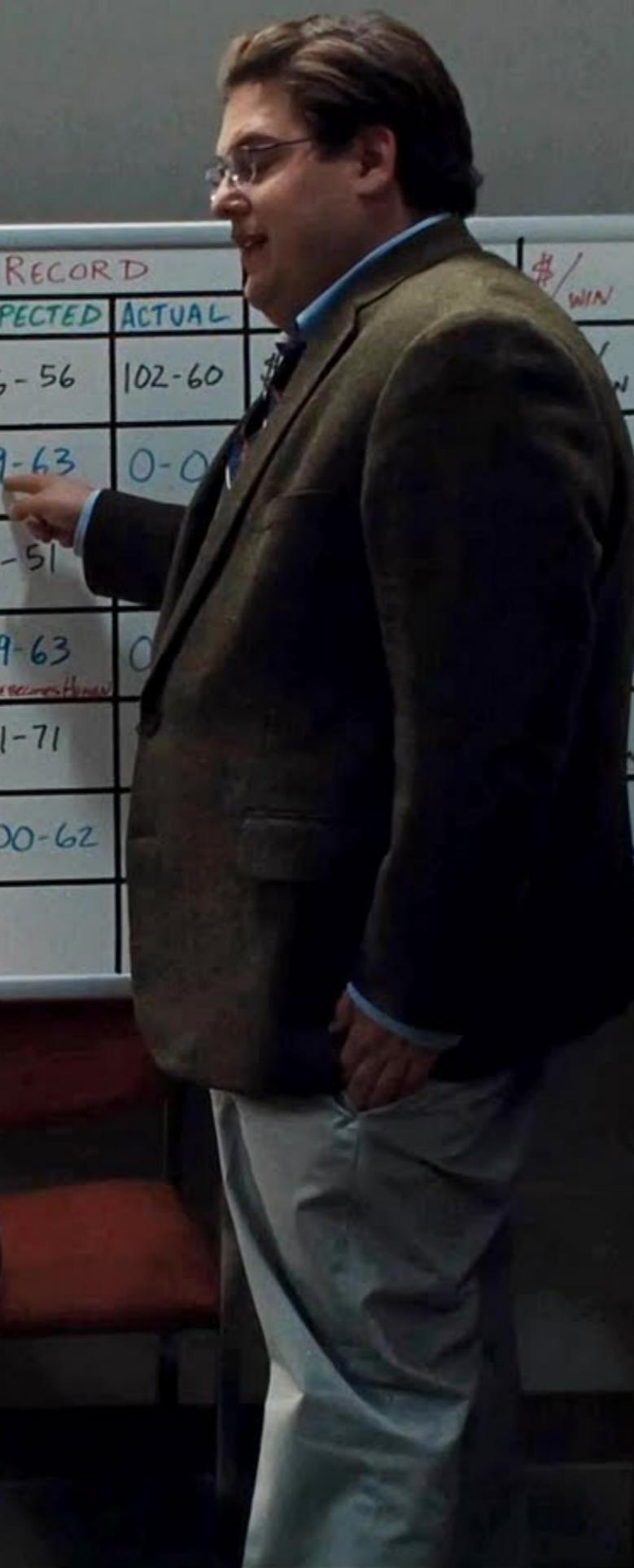
\$114,457,768

vs

\$39,722,689

Oakland Athletics

	Runs Scored ² Runs Scored ² + Runs Allowed ²	WIN %		RECORD	
		EXPECTED	ACTUAL	EXPECTED	ACTUAL
OAK 2001	$\frac{884^2}{884^2 + 645^2} = \frac{781456}{1197481} = .6525$.6296	.6296	106 - 56	102 - 60
OAK 2002 PROJECTION	$\frac{814^2}{814^2 + 645^2} = \frac{662596}{1078621} = .6143$ + Lose Giambi, Damon, Trumbo	—	—	99 - 63	0 - 0
SEA 2001	$\frac{927^2}{927^2 + 627^2} = \frac{859329}{1252458} = .6861$.7160	.7160	111 - 51	0 - 0
	$\frac{850^2}{850^2 + 680^2} = \frac{722500}{1184900} = .6097$ + Lose Self, Bell	—	—	99 - 63	0 - 0
	$\frac{804^2}{804^2 + 713^2} = \frac{646416}{1154785} = .5597$.5864	.5864	91 - 71	0 - 0
	$\frac{890^2}{890^2 + 700^2} = \frac{792100}{1282100} = .6178$ Projection + Lose Justice, Gain → Giambi, Venturo	—	—	100 - 62	0 - 0



001 $\frac{884^2 + 645^2}{1197481} = \frac{781456}{\dots}$

AK
002 $\frac{814^2}{814^2 + 645^2} = \frac{662596}{\dots}$

OBJECTION * LOSE GIAM.

E4 $\frac{927^2}{927^2 + 627^2} = \frac{859329}{1252458} = \dots$

I



MATH

An Introduction to Category Theory

Kyle Oba
@mudphone
pasdechocolat.com



I
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?)  
  
/// 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

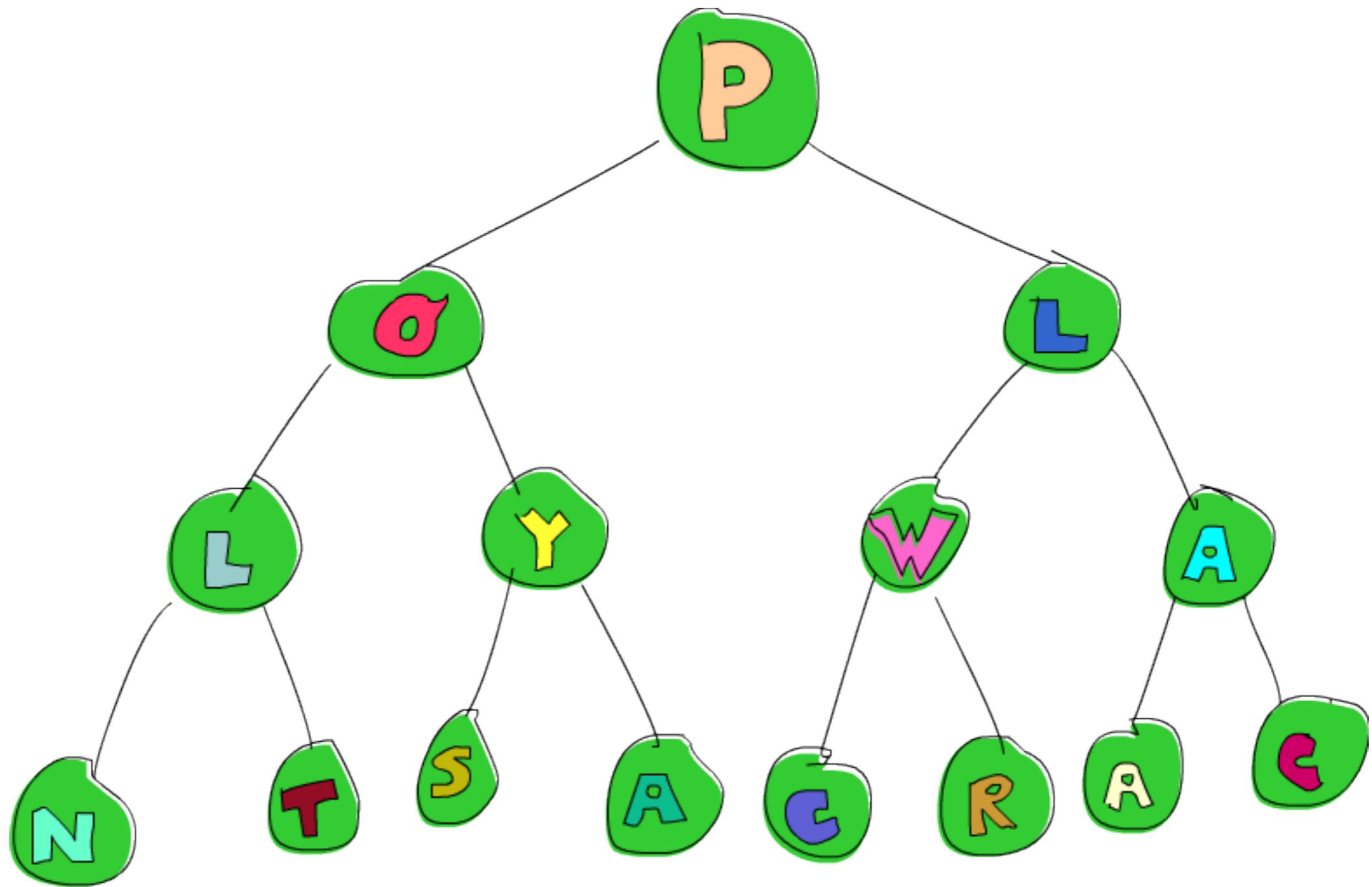
```
/// Return the result of repeatedly calling `combine` with an
/// accumulated value initialized to `initial` and each element of
/// `sequence`, in turn.
func reduce<S : SequenceType, U>(sequence: S, initial: U, combine: (U, S.Generator.Element) -> U) -> U
```

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

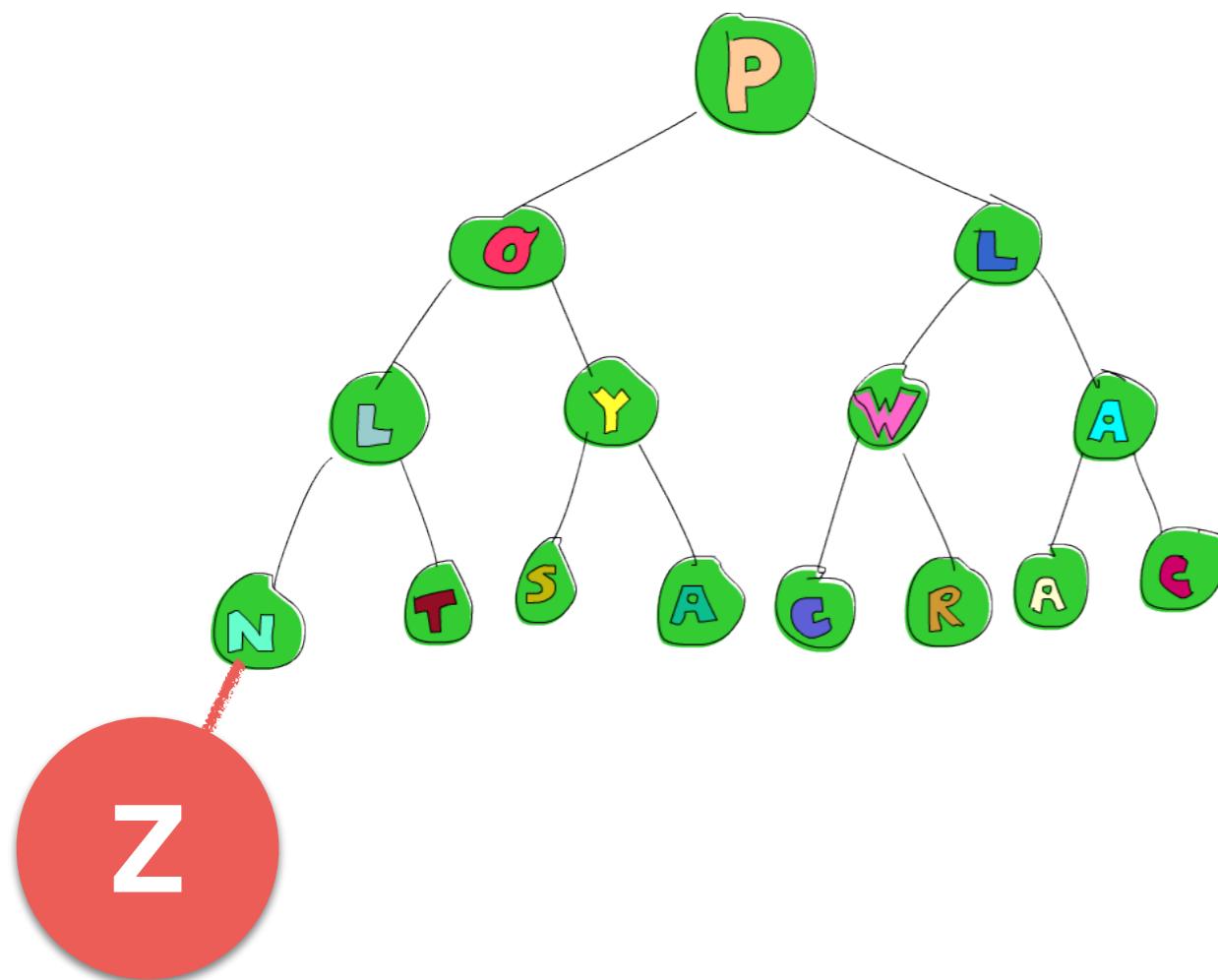


```

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>])?

```



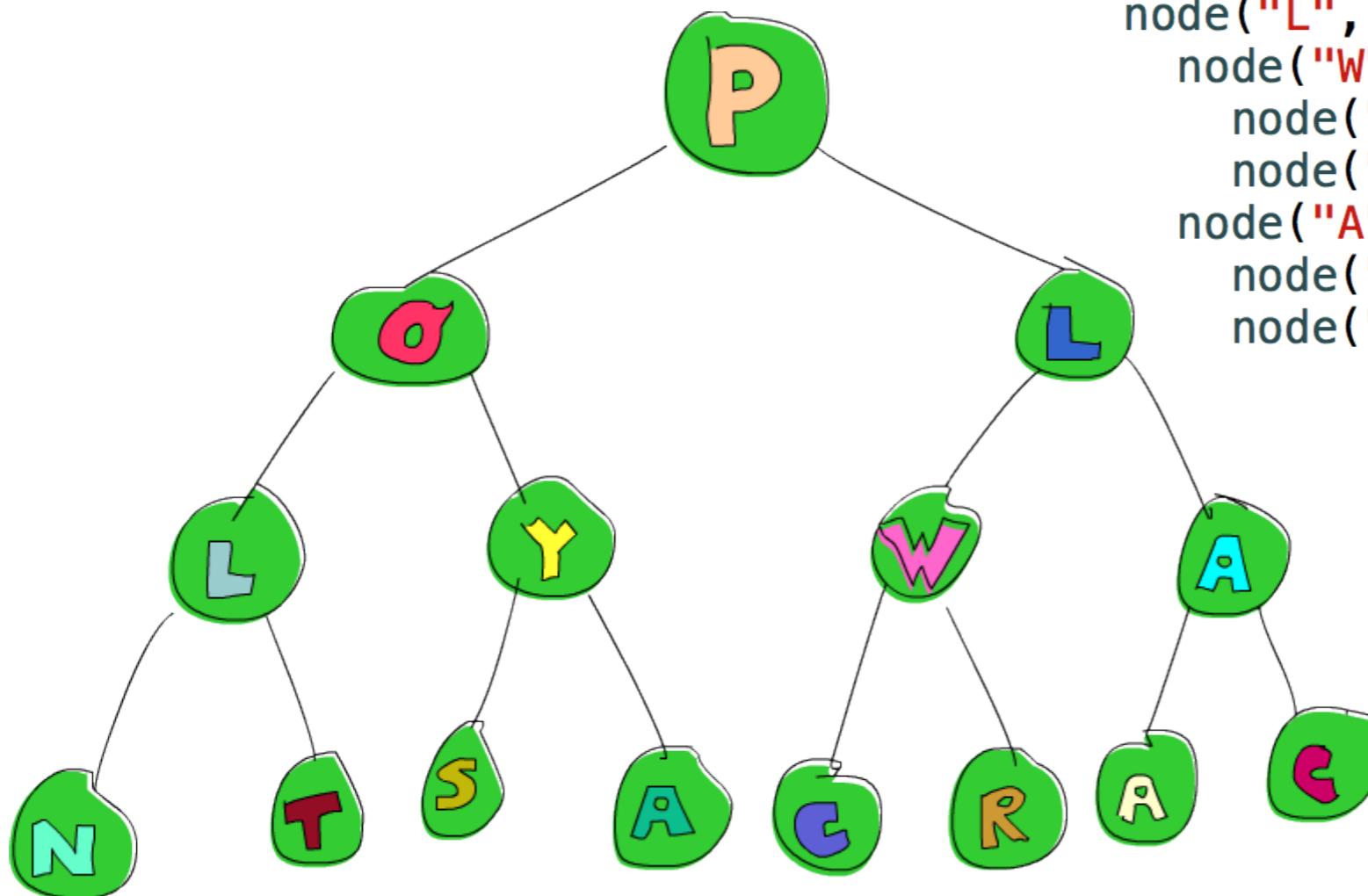
```
class Box<T> {
    let unbox: T
    init(_ value: T) { self.unbox = value }
}
```

```
/*-----/
// 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))
```

```
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))))
```



```
/*-----/
// 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..    }  
}  
  
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  
    }  
}
```

```
/*-----/
//  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)! }
}

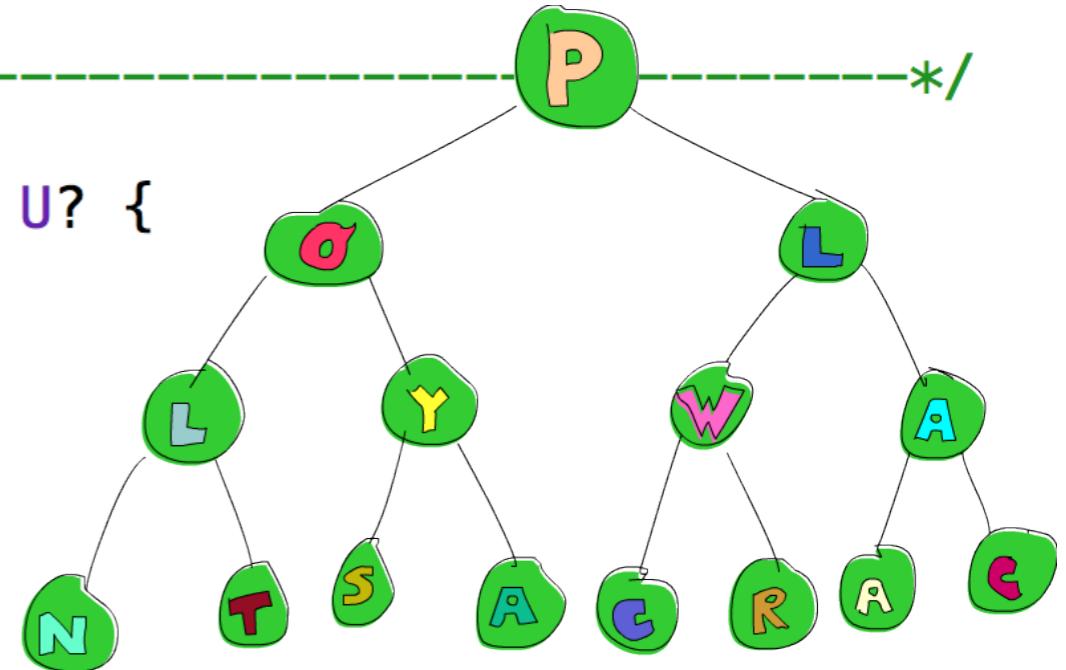
func opt<T>(x: T) -> T? {
    return x
}

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

objc ↑↓

Functional Programming in Swift

Composing Filters

Now that we have a blur and a color overlay filter defined, we can put them to use on

```
50 let url = NSURL(string: "http://tinyurl.com/m74sldb");
51 let image = CIImage(contentsOfURL: url)

52 let blurRadius = 5.0
53 let overlayColor = NSColor.redColor().colorWithAlphaComponent(0.2)
54 let blurredImage = blur(blurRadius)(image)
55 let overlaidImage = colorOverlay(overlayColor)(blurredImage)

Once again, we assemble images by creating a filter, such as blur(blurRadius) on an image.
```

Building Filters

Now that we have the Filter type defined, we can start defining functions that build specific filters. These are convenience functions that take the parameters needed for a specific filter and construct a value of type Filter. These functions will all have the following general shape:

```
func myFilter(/* parameters */) -> Filter
```

Note that the return value, Filter, is a function as well. Later on, this will help us compose multiple filters to achieve the image effects we want.

To make our lives a bit easier, we'll extend the CIFilter class with a `convenience initializer` and a computed property to retrieve the output image:

```
 typealias Parameters = Dictionary<String, AnyObject>

extension CIFilter {
    convenience init(name: String, parameters: Parameters) {
        self.init(name: name)
    }
}
```

The Filter Type

One of the key classes in Core Image is the CIFilter class, which is used to create image filters. When you instantiate a CIFilter object, you (almost) always provide an input image via the `kCIInputImageKey` key, and then retrieve the filtered result via the `kCIOutputImageKey` key. Then you can use this result as input for the next filter.

In the API we will develop in this chapter, we'll try to encapsulate the exact details of these key-value pairs and present a safe, strongly typed API to our users. We define our own Filter type as a function that takes an image as its parameter and returns a new image:

```
typealias Filter = CIImage -> CIImage
```

This is the base type that we are going to build upon.

Introduction

Why write this book? There is plenty of documentation on Swift readily available from Apple, and there are many more books on the way. Why does the world need yet another book on yet another programming language?

This book tries to teach you to think functionally. We believe that Swift has the right language features to teach you how to write functional programs. But what makes a program functional? And why bother learning about this in the first place? In his paper, "Why Functional Programming Matters," Hughes (1989) writes:

Functional programming is so called because it is based on the notion that computation is the application of functions to arguments. A main program itself is written as a function that receives the program's input as its argument and delivers the program's output as its result.

So rather than thinking of a program as a sequence of assignments and method calls, functional programmers emphasize that each program can be represented by a single, larger and smaller pieces; all these pieces can be assembled using function application to define a complete program. By avoiding assignment statements and side effects, Hughes argues that functional programs are more modular than their imperative- or object-oriented counterparts. And modular code is a Very Good Thing.

Chris Eidhof, Florian Kugler, and Wouter Swierstra



Learn You a Haskell for Great Good!

by Miran Lipovača



Thank you.