

# A Brief Intro to Functional Programming

## And Its Influence on React

Chase Zhang

Splunk Inc.

May 17, 2017

# Disclaimer

- ▶ The internal content of Functional Programming is too tremendous to be put into a single speech.
- ▶ This speech is only intended for a shallow impression.
- ▶ Don't hesitate to tell me if I was wrong!

# Prerequisite

- ▶ Basic knowledge about ES6
- ▶ Basic knowledge about React/Flux

```
1 // ES6 Arrow Functions
2 (x, y) => x + y
3
4 (x, y) => {
5   // something
6   return x + y;
7 }
8
9 let func = (x) => x * 2
```

# Warming Up

What's your impression of FP?

- ▶ Properties?
- ▶ Paradigms?
- ▶ Slogans?

# Table of Contents

## Paradigms of FP

- Closures

- Lazy Evaluation

- Y-Combinator

## The Core of FP

- A Spoon of History

- Problem: Make a Chocolate Cake

  - The Imperative Way

  - The FP Way

- Conclusion

## FP & React

## References

# Paradigms of FP: Closures

Most of us know closures.

But do you have a comprehensive understanding of its power?

# Paradigms of FP: Closures

## Question

*How do you build a linked list without objects?*

# Paradigms of FP: Closures

```
1 let getNode = (value, next) => {  
2   return (x) => x ? value : next;  
3 }  
4  
5 let value = (node) => node(true);  
6  
7 let next = (node) => node(false);
```



# Paradigms of FP: Closures

```
1 let a = getNode(1, getNode(2, getNode(3, null)));  
2 /* 1 -> 2 -> 3 -> null */  
3  
4 value(a); // 1  
5  
6 let b = next(a); value(b); // 2  
7  
8 let c = next(b); value(c); // 3  
9 next(c) // null
```

# Paradigms of FP: Closures

What happened?

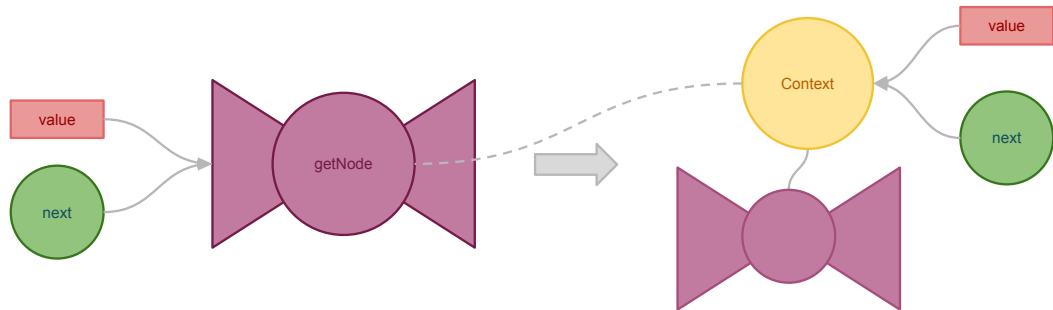


Figure: Closures as Nodes

# Paradigms of FP: Closures

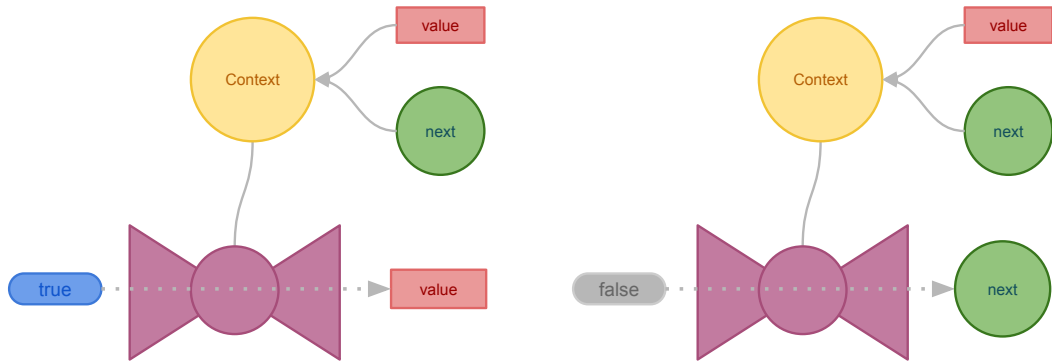


Figure: Get value and next

# Paradigms of FP: Closures

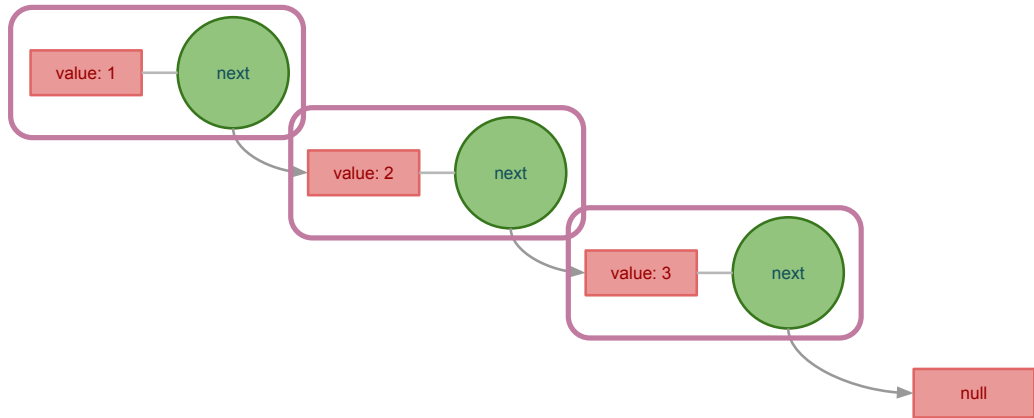


Figure: Closures as Linked List

# Paradigms of FP - Closures

A more complex example: reverse a linked list

```
1 let append = (n, v) => {  
2   if (n == null) return getNode(v, null);  
3   else return getNode(value(n), append(next(n), v));  
4 }  
5  
6 let reverse = (list) => {  
7   if (list == null) return null;  
8   else return append(reverse(next(list)), value(list));  
9 }  
10  
11 let a = getNode(1, getNode(2, getNode(3, null)));  
12  
13 reverse(a); // 3 -> 2 -> 1 -> null;
```

# Paradigms of FP: Closures

Closures can do a lot more than just linked list:

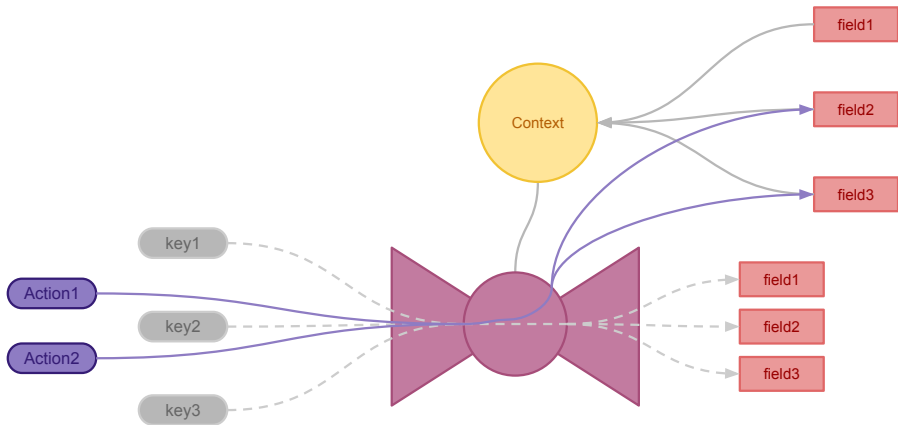


Figure: Closures as Objects

# Paradigms of FP: Closures

## Conclusion

- ▶ *Closures* are a poor man's *objects*<sup>1</sup>
- ▶ *Recursion*!
- ▶ *Abstraction*!<sup>2</sup>

---

<sup>1</sup>The other half of this slogan is: Objects are a poor man's Closures[1]

<sup>2</sup>`getNode`, `value` and `next` are usually named as `cons`, `car` and `cdr` in `lisp`

# Paradigms of FP: Lazy Evaluation

We can do more with the abstraction of **getNode**:

```
1 let naturalNumber = (n) => {
2   return (x) => x ? n : naturalNumber(n + 1);
3 }
4
5 let take = (n, count) => {
6   if (n == null || count <= 0) return null;
7   else return (x) => x ? value(n) : take(next(n), count - 1);
8 }
9
10 take(naturalNumber(1), 10) // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
11 // *naturalNumber* generates an infinite list of integers!
12 // But don't worry, we won't run out of memory because it is lazy
   evaluated
```



# Paradigms of FP: Lazy Evaluation

What happened?

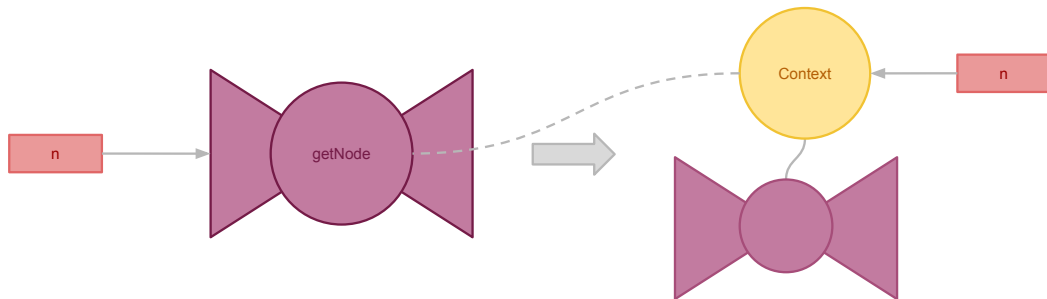


Figure: Lazy Evaluate Node

# Paradigms of FP: Lazy Evaluation

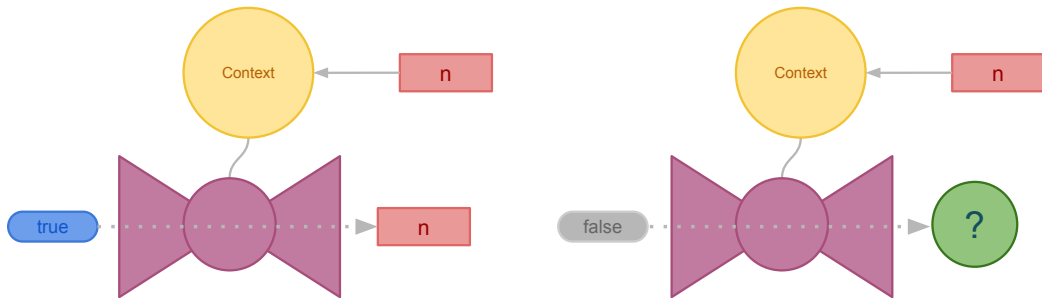


Figure: Get value and ?

# Paradigms of FP: Lazy Evaluation

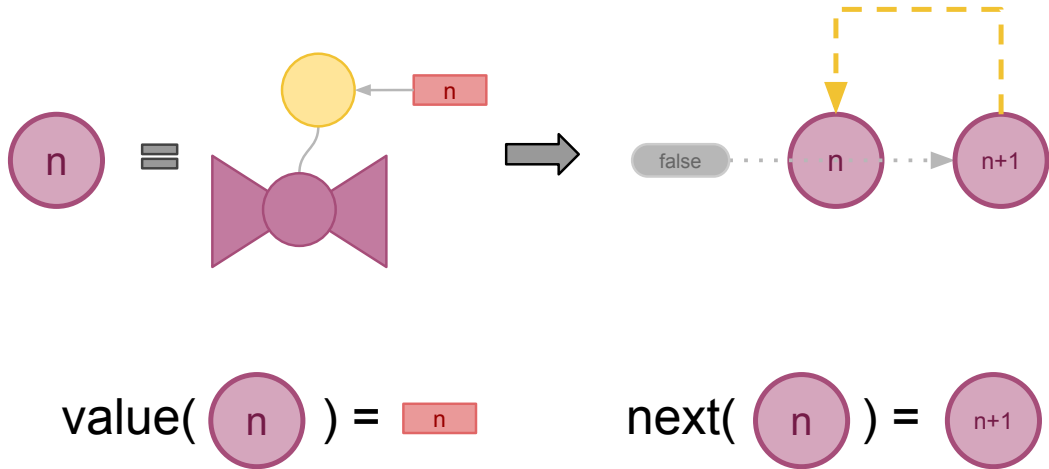


Figure: Recursive Defined Node

# Paradigms of FP: Lazy Evaluation

With infinite sequence, we gain great power of expression.

For example, how to produce a sequence of prime numbers with sieve method?

**Algorithm.** Sieve of Eratosthenes:

1. At first we have an infinite sequence of natural numbers starting from 2, ( $\mathcal{S} = \{k | k \in \mathbb{N}, k \geq 2\}$ )
2. Pick a number  $p$  from the sequence  $\mathcal{S}$ , this number  $p$  must be a prime
3. We sieve out all multiples of  $p$  from the sequences. That is, remove  $2p, 3p, \dots, np, \dots (n \in \mathbb{N})$  from  $\mathcal{S}$
4. Repeat (2) and (3), we'll get a list of primes

# Paradigms of FP: Lazy Evaluation

```
1  let nextMatch = (seq, fn) => {  
2      return fn(value(seq)) ? seq : nextMatch(next(seq), fn);  
3  }  
4  
5  let filter = (seq, fn) => {  
6      let mseq = nextMatch(seq, fn);  
7      return (x) => x ? value(mseq) : filter(next(mseq), fn);  
8  }  
9  
10 let sieve = (seq) => {  
11     let aPrime = value(seq);  
12     let nextSeq = filter(next(seq), (v) => v % aPrime != 0);  
13  
14     return (x) => x ? aPrime : sieve(nextSeq);  
15 }
```

# Paradigms of FP - Lazy Evaluation

Magic!

```
1 let primes = sieve(naturalNumber(2));  
2 // primes represents for the infinite sequence of prime numbers!  
3  
4 take(primes, 15);  
5 // 2, 3, 5, 7, 13, 17, 19, 23, 29, 31, 37, 41, 43
```

## Paradigms of FP: Lazy Evaluation

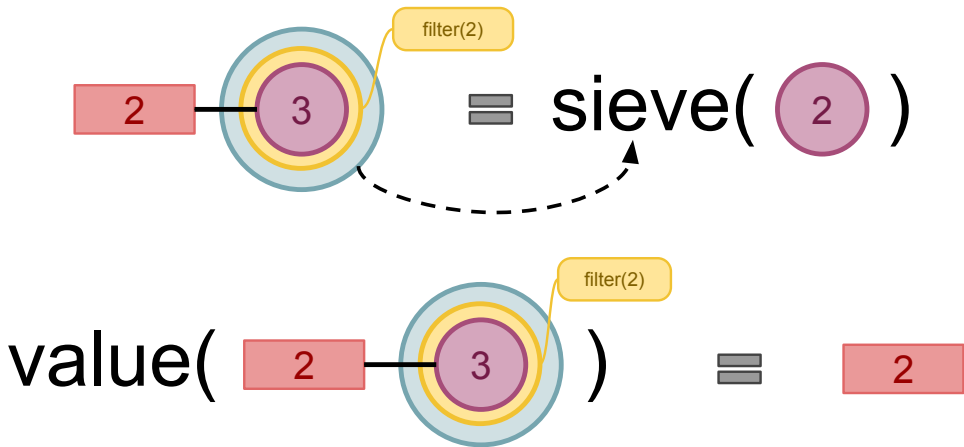


Figure: Sieving Prime Numbers

## Paradigms of FP: Lazy Evaluation

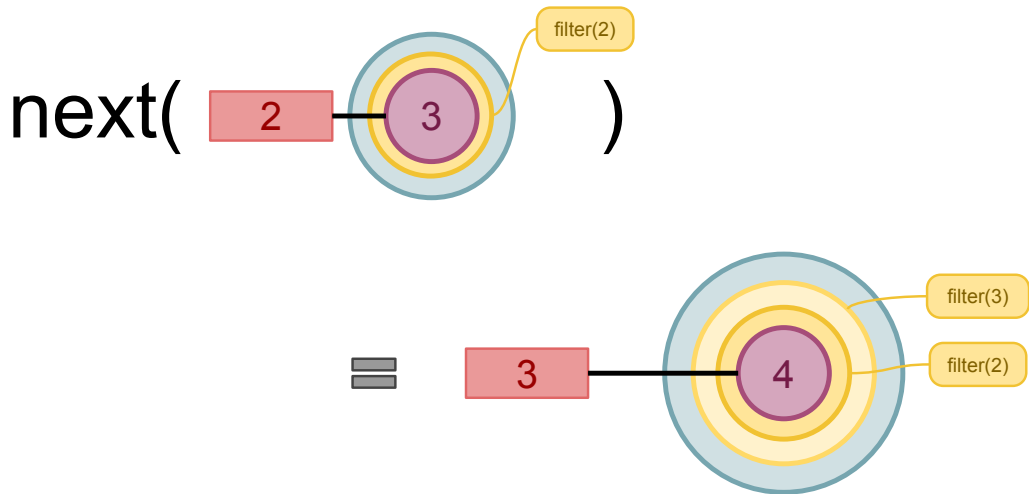


Figure: Sieving Prime Numbers: 2 to 3



## Paradigms of FP: Lazy Evaluation

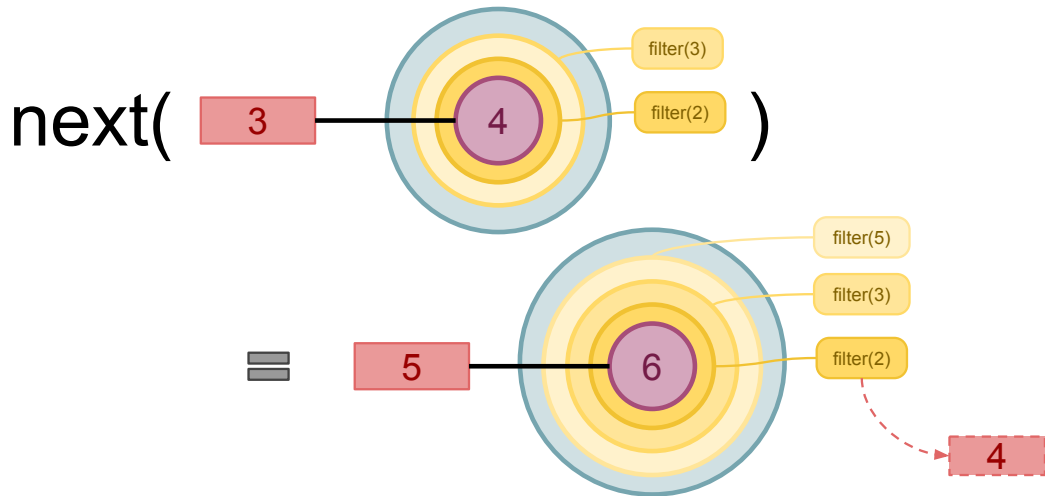


Figure: Sieving Prime Numbers: 2 to 3

# Paradigms of FP: Lazy Evaluation

Why lazy evaluation is so important?

1. It enables us to express infinite sequences and apply operations on it naturally
2. It is the underlying model for stream processing, which is becoming more and more important nowadays
3. It may save memory space and speed up calculation if previous items can be thrown away<sup>3</sup>
4. It might be an important part of the core of FP (will explain later)

---

<sup>3</sup>Python's **generator** might be a good example for this

# Paradigms of FP: Y-Combinator



Figure: Y-Combinator, an American seed accelerator

# Paradigms of FP: Y-Combinator

As to know the origin of YC's name, we'll start with...

# Paradigms of FP: Y-Combinator

## Question

*How do you implement recursion, without a named function?*

# Paradigms of FP: Y-Combinator

```
1 // What if we can't give *fact* a name?  
2 let fact = (n) => {  
3   if (n == 1) return 1;  
4   else return n * fact(n - 1);  
5 }
```

# Paradigms of FP: Y-Combinator

Although we can't give function a name, we can give parameters of a function names. What if:

1. We define a hyper function which accepts a function as parameter
2. Then the function passed in has a name, can we make use this name to implement recursion?

```
1 // *F* is a hyper function accepts a function as parameter ,
2 // inside its context. We'll then get a named function *f*
3 //
4 // The name of *F* will be erased later
5 let F = (f) => {
6   return (n) => {
7     if (n == 1) return 1;
8     else return n * f(n - 1);
9   }
10 }
```

# Paradigms of FP: Y-Combinator

Do we have what we want? Let's check:

1. Firstly we have a hyper function  $\mathcal{F} : f \rightarrow f'$ , this function maps a function  $f$  to  $f'$
2. As in  $f'$ , we recursively called  $f$ ,  $f'$  will be the recursive function we want if and only if  $f \equiv f'$
3. Can we do something to find such an  $f$ , s.t.  $f \equiv f'$ ?



# Paradigms of FP: Y-Combinator

For any  $\mathcal{F} : f \rightarrow f'$ , the input  $f$  which lets  $\mathcal{F}(f) \equiv f$ , ( $f \equiv f'$ ) is called the fixed point of  $\mathcal{F}$ .

And yes, we DO HAVE something that can produce the fixed point of any such  $\mathcal{F}$ , which is known as **Y-Combinator**.

```
1 let Y = (F) => {  
2   return (x => F(v => x(x)(v)))(x => F(v => x(x)(v)));  
3 }
```

# Paradigms of FP: Y-Combinator

As Y-Combinator  $\mathcal{Y}$  can return  $\mathcal{F}$ 's fixed point, we can get our recursive function simply with  $f = \mathcal{Y}(\mathcal{F})$ . Then just call  $f$  to get the result of the recursive function

```
1 Y(F)(10); // 10! = 3628800
```

At last we can erase all the names. It is crazy, isn't it? We now have a recursive function with unnamed functions only!

```
1 (F => (x => F(v => x(x)(v)))(x => F(v => x(x)(v))))(f => (n => n
    == 1 ? 1 : n * f(n - 1)))(10)
```

# Paradigms of FP: Y-Combinator

How does it work? Well, it's a long story. For now, we only need to know:

1. In the 1930s, **Alonzo Church** invented  $\lambda - calculus$ , which might be the very beginning of FP. That is also why we now have tons of programming languages have something named **lambda expression**
2. Under the framework of  $\lambda - calculus$ , **Haskell Brooks Curry** found Y-Combinator and proved it can find a fixed point of such hyper function  $\mathcal{F}$ . A little more about him:
  - 2.1 There are three programming languages are named after him: Haskell[2], Brooks[3] and Curry[4]
  - 2.2 Curry seems familiar? You must know **Currying**[5]

Enough history for now, let's first have a conclusion of Y-Combinator.

# Paradigms of FP: Y-Combinator

## Conclusion

- ▶ Y-Combinator shows us, in FP, we not only regard function as the first class member, but we also perform calculus upon it
- ▶ FP is closely related to mathematic and the early development of computer science, it is never something new!

# The Core of FP

## Question

*What is Functional Programming all about?*

# The Core of FP: A Spoon of History

In 1936, **Alonzo Church** created  $\lambda$  – *calculus*[6], which is a method for defining function. It looks like:

1.  $\lambda x. x$
2.  $\lambda x. \lambda y. x + y$  (Currying)
3.  $\lambda g. (\lambda x. g (x x)) (\lambda x. g (x x))$  (Y-Combinator)

He also invented **Church numerals** to represent natural numbers with functions. Then he defined  $\lambda$  – *computable* which means a function on natural numbers can be represented by a term of  $\lambda$  – *calculus*.

# The Core of FP: A Spoon of History

In 1936, Alan Turing invented Turing Machine[7]. He also defined a similar Turing computable as some Turing machine can compute the function on encoded natural numbers. With Turing Machine, Alan Turing solved the famous halting problem. Turing machine also became a base of modern computers.

It has been proved,  $\lambda$  – *calculus* and Turing machine are actually equivalent.

# The Core of FP: A Spoon of History

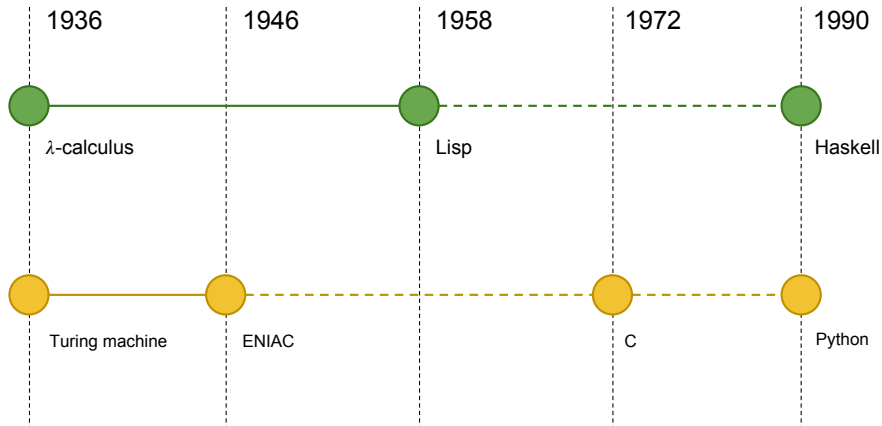


Figure: A Brief Timeline



# The Core of FP: A Spoon of History

- ▶ **Lisp** is the second-oldest high-level programming language (The first is Fortran)
- ▶ Lisp pioneered many ideas including:
  - ▶ Higher-order functions
  - ▶ Recursions
  - ▶ REPL
  - ▶ Dynamic typing
- ▶ Lisp is said to be most suitable for AI, but it's not true. Such saying is because Lisp was the only language providing some advanced feature like recursion for a long time
- ▶ Lisp is now famous for its macro system

# The Core of FP: A Spoon of History

- ▶ **Haskell** is a general purpose **purely functional** programming language
- ▶ Haskell is famous for its typing system
  - ▶ Type inference
  - ▶ Type classes (dependent types)
- ▶ All collections are lazy evaluated in Haskell
- ▶ You might want to learn Category Theory (and get two Ph.Ds) before learning Haskell



# The Core of FP: A Spoon of History

- ▶ A lot of multiple-paradigm programming languages are influenced by FP
  - ▶ Java 8 added lambda expression
  - ▶ Most languages supports recursion now
  - ▶ Function as first member and higher order function are supported by some languages (ex. Python, Golang)
- ▶ Many libraries are influenced by FP (ex. ReactiveX[8])

# The Core of FP

## Summary

Functional Programming is not decided by if it is static typing or dynamic typing, if it has macros and if we're writing things in Haskell or other purely FP language.

# The Core of FP: Make a Chocolate Cake



Figure: Chocolate Cake

# The Core of FP: The Imperative Way

Recipe (FAKE) of making a cream cake:

1. Break 3 eggs and whisk them into egg sauce
2. Mix 500ml of milk with 200ml of water and egg sauce into mixed sauce
3. Mix mixed sauce from step (2) with 200g of plain flour into plaste
4. Bake plaste from (4) for 20 minutes to get base cake
5. Mix 100g chocolate flour with 300 milk cream into chocolate cream
6. Apply chocolate cream onto base cake
7. Finished

# The Core of FP: The Imperative Way

Questions:

1. Which steps can be done in parallel?
2. What tools are shared and should be used exclusively?
3. What if we have multiple people working on this? How can they cooperate?
4. What if we'd like to start a cake factory? How could we turn that recipe into something can be execute with large scale?



# The Core of FP: The FP Way

Let's have a review on the recipe, this time with some keywords highlighted:

1. Break 3 eggs and whisk them into egg sauce
2. Mix 500ml of milk and egg sauce into mixed sauce
3. Mix mixed sauce from step (2) with 200g of plain flour into plaste
4. Bake plaste from (4) for 20 minutes to get base cake
5. Mix 100g chocolate flour with 300ml of milk cream into chocolate cream
6. Apply chocolate cream onto base cake
7. Finished

# The Core of FP: The FP Way

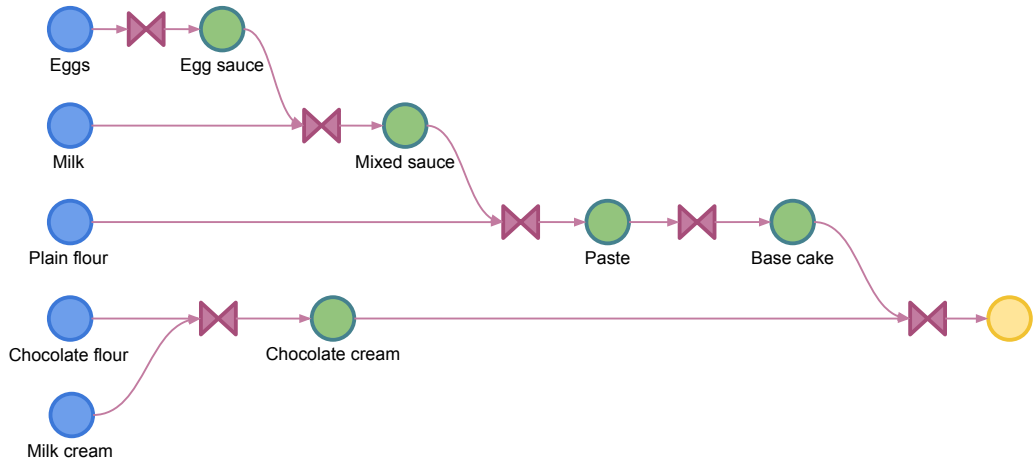


Figure: The Flow of Bake a Cake

# The Core of FP: The FP Way

## Summary

1. Parallel steps are obvious
2. Each transform has now side effect thus on exclusive resource locking
3. Let each person work on different branch of work flow, they can cooperate well
4. Let resource material a lazy evaluated list (or streams), we have pipelines to build factory

# The Core of FP: Conclusion

## Conclusion

1. The difference between imperative programming and functional programming exists in the perspective of handling program
2. The core of Functional Programming is thinking about **data-flow** rather than **control-flow**[9]

# FP & React

## Question

*What's the relationship between FP and React?*

# FP & React

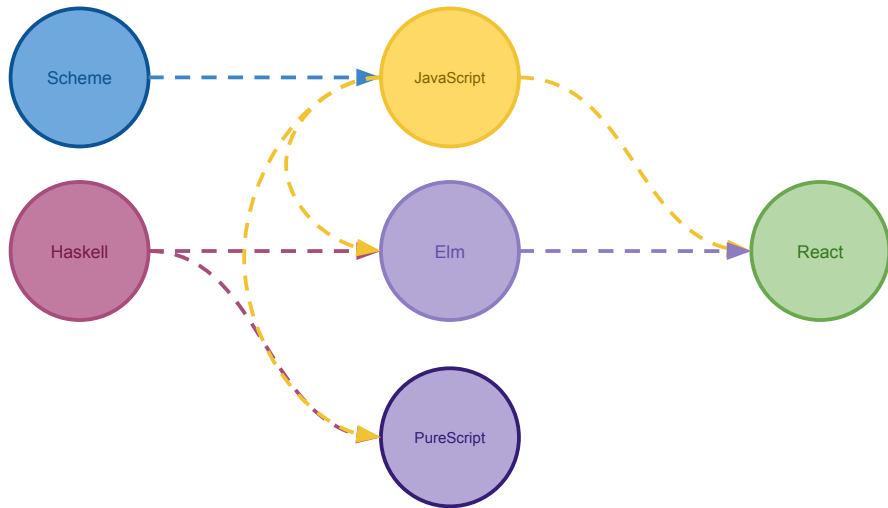


Figure: Influence Flow

# FP & React: Component

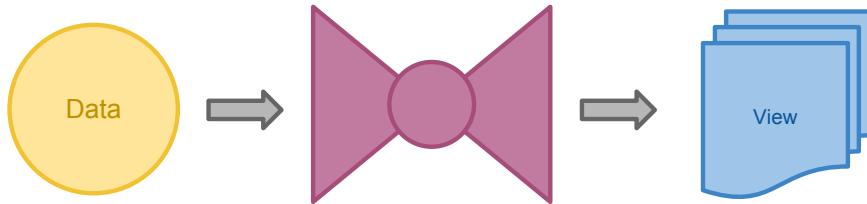


Figure: React Component

# FP & React: PureComponent

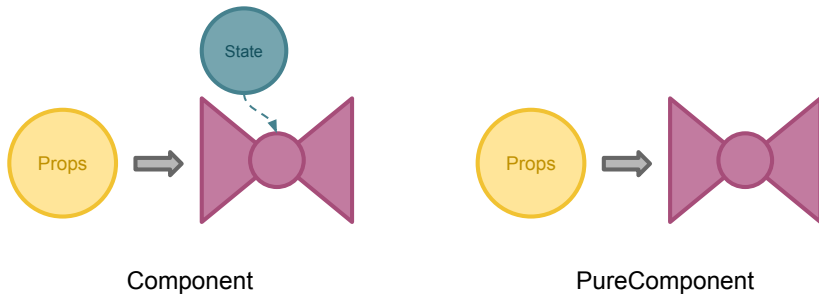


Figure: Pure Component



# FP & React: Pure App

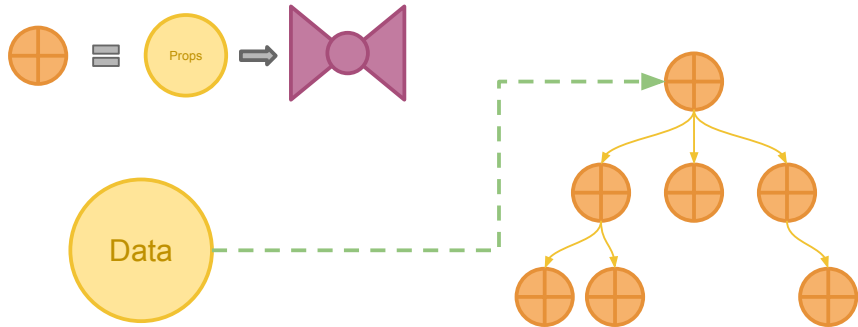
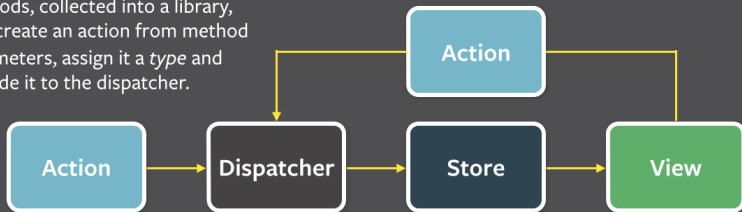


Figure: Pure App

# FP & React: Flux

*Action creators* are helper methods, collected into a library, that create an action from method parameters, assign it a *type* and provide it to the dispatcher.



Every action is sent to all stores via the *callbacks* the stores register with the dispatcher.

After stores update themselves in response to an action, they emit a *change* event.

Special views called *controller-views*, listen for *change* events, retrieve the new data from the stores and provide the new data to the entire tree of their child views.

Figure: Flux

# FP & React: Flux

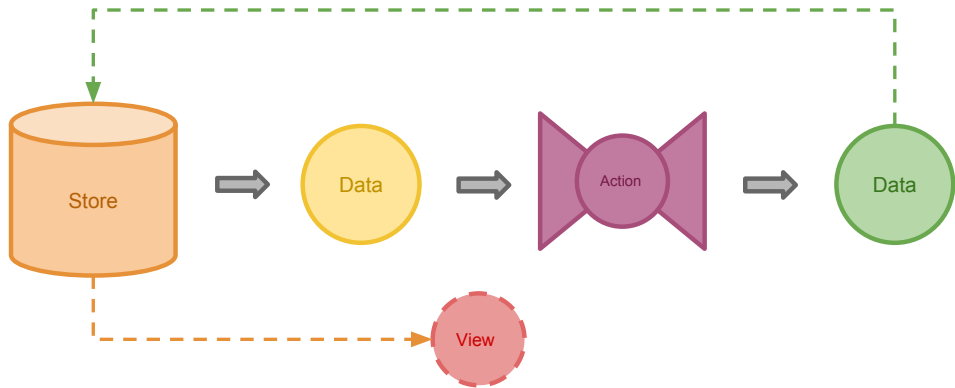


Figure: Flux in a perspective of data flow

## FP & React: Action

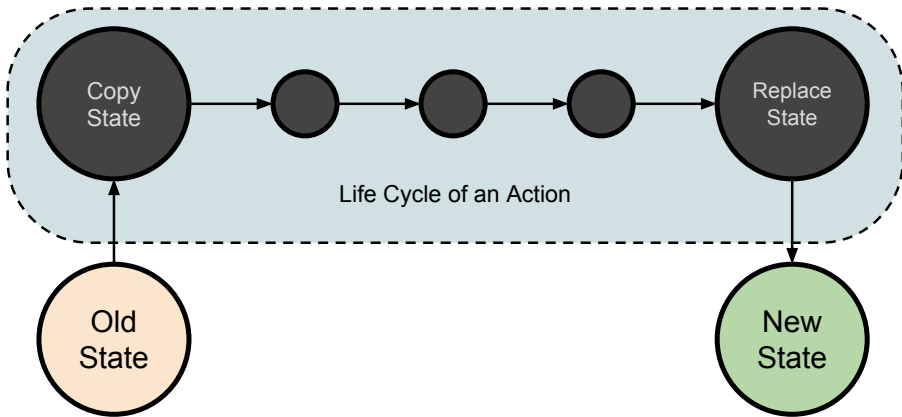


Figure: Action executed success

## FP & React: Action

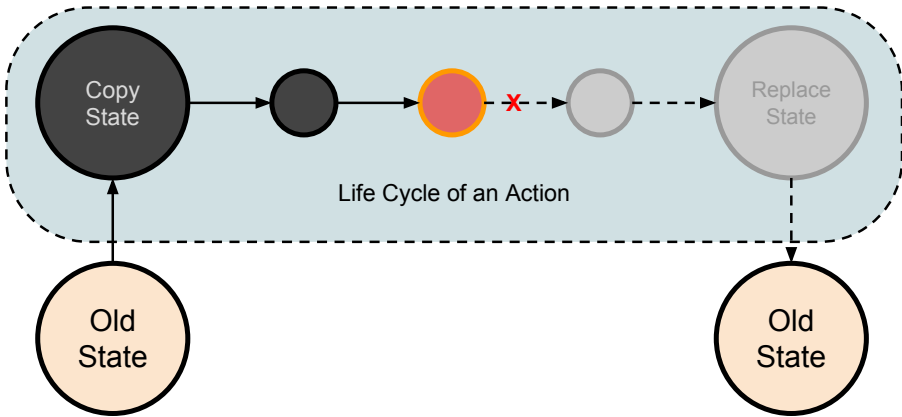


Figure: Action executed fail

# FP & React: ImmutableJS

ImmutableJS implements a collection of immutable data structures.





1. It is inspired by similar implementation from Scala and Clojure
2. It can speed up props comparison
3. It avoid copying the whole object when performing actions: modified versions shared structures internally with the old
4. The underlying data structures are **Vector Trie** and **HAMT**<sup>4</sup>

---

<sup>4</sup>Interested in details? I've a blog series about it.[10]

Thank you!

# References I

-  Anton van Straaten.  
What's so cool about scheme.  
<http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html>.
-  Haskell (programming language).  
<https://en.wikipedia.org/wiki/Haskell>.
-  Brooks (programming language).  
[https://en.wikipedia.org/wiki/Brooks\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Brooks_(programming_language)).
-  Curry (programming language).  
<https://en.wikipedia.org/wiki/Curry>.
-  Currying.  
<https://en.wikipedia.org/wiki/Currying>.



## References II



[Lambda calculus.](#)

[https://en.wikipedia.org/wiki/Lambda\\_calculus.](https://en.wikipedia.org/wiki/Lambda_calculus)



[Turing machine.](#)

[https://en.wikipedia.org/wiki/Turing\\_machine.](https://en.wikipedia.org/wiki/Turing_machine)



[Reactivex.](#)

[http://reactivex.io/.](http://reactivex.io/)



[Haoyi Li.](#)

What's functional programming all about.

[http://www.lihaoyi.com/post/  
WhatsFunctionalProgrammingAllAbout.html.](http://www.lihaoyi.com/post/WhatsFunctionalProgrammingAllAbout.html)



[Functional go.](#)

[https://io-meter.com/categories/Functional-Go/.](https://io-meter.com/categories/Functional-Go/)