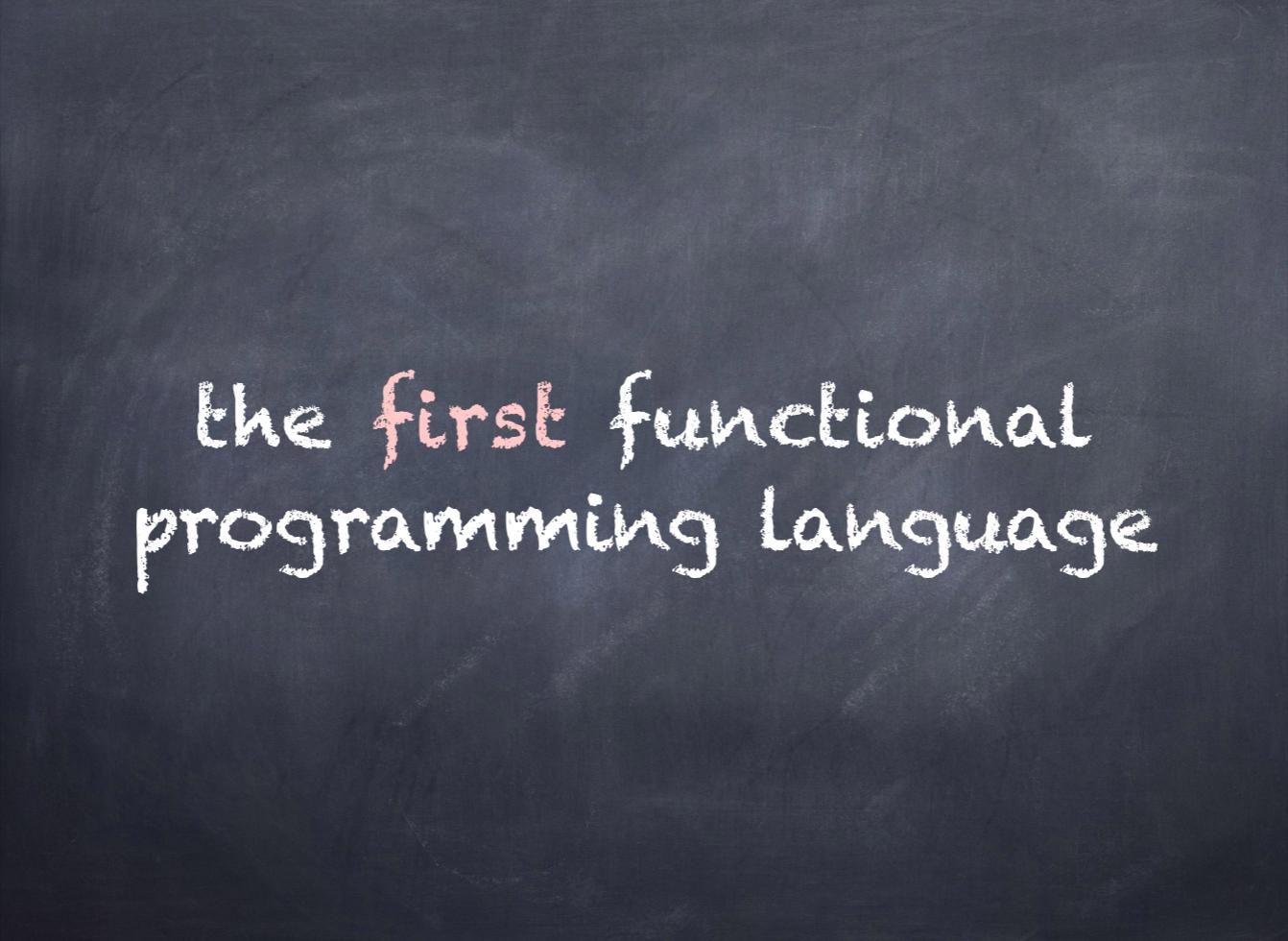


awesome Lisp

What is a  
Lisp?



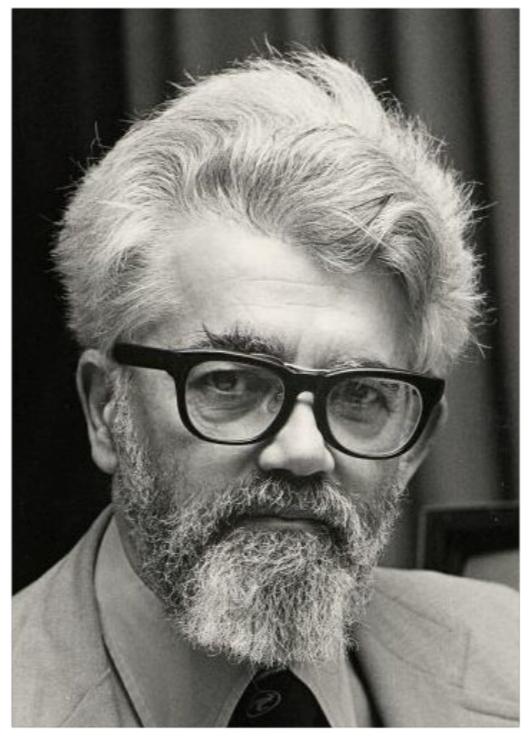
[www.playingwithwords365.com](http://www.playingwithwords365.com)



the first functional  
programming language

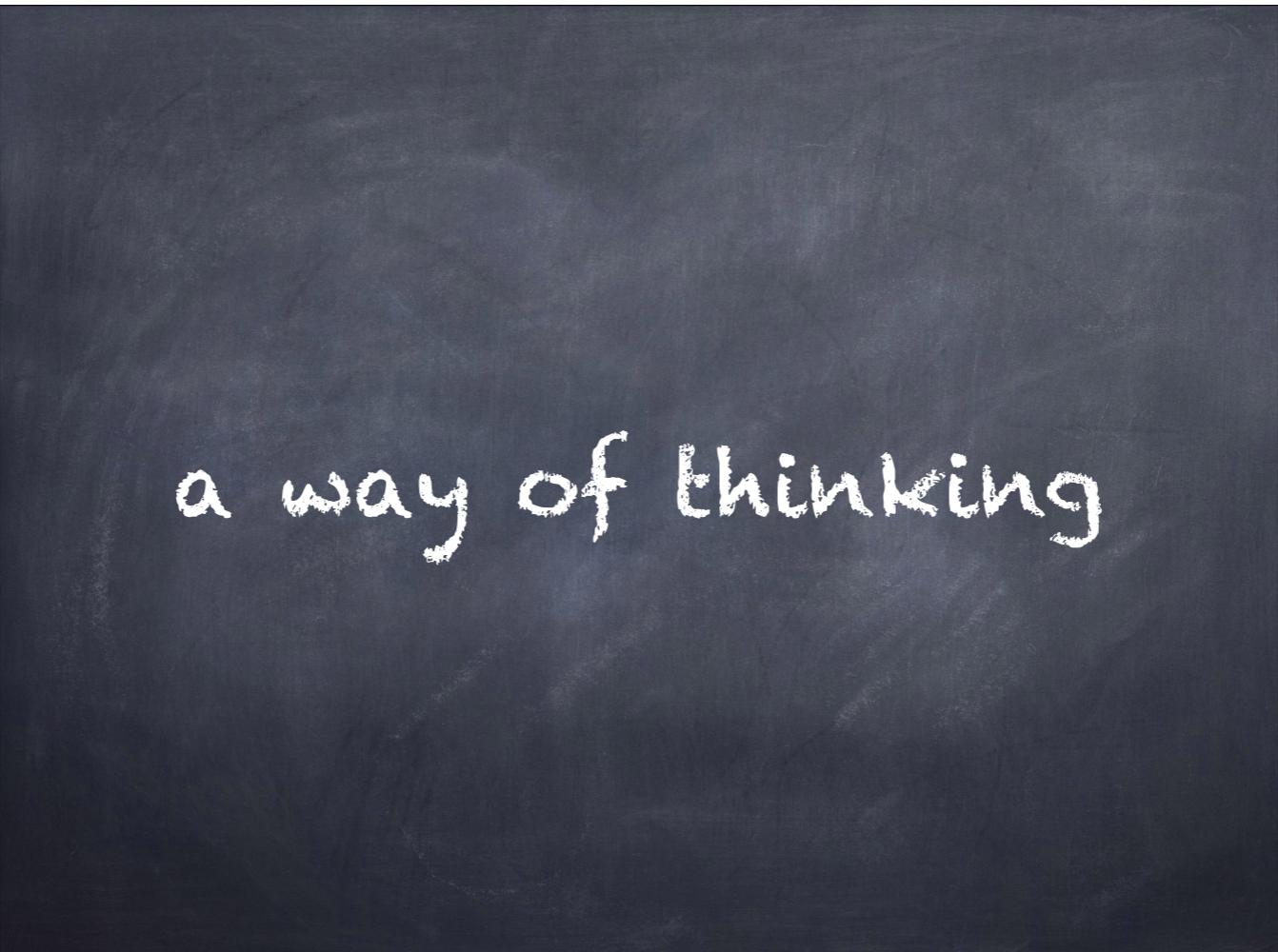
lisp是地球上第一门函数式编程语言。

# John McCarthy



约翰·麦卡锡在1958年基于λ演算创造了Lisp。演化至今，是历史第二悠久的高级语言，仅次于Fortran。

————— from wikipedia



a way of thinking

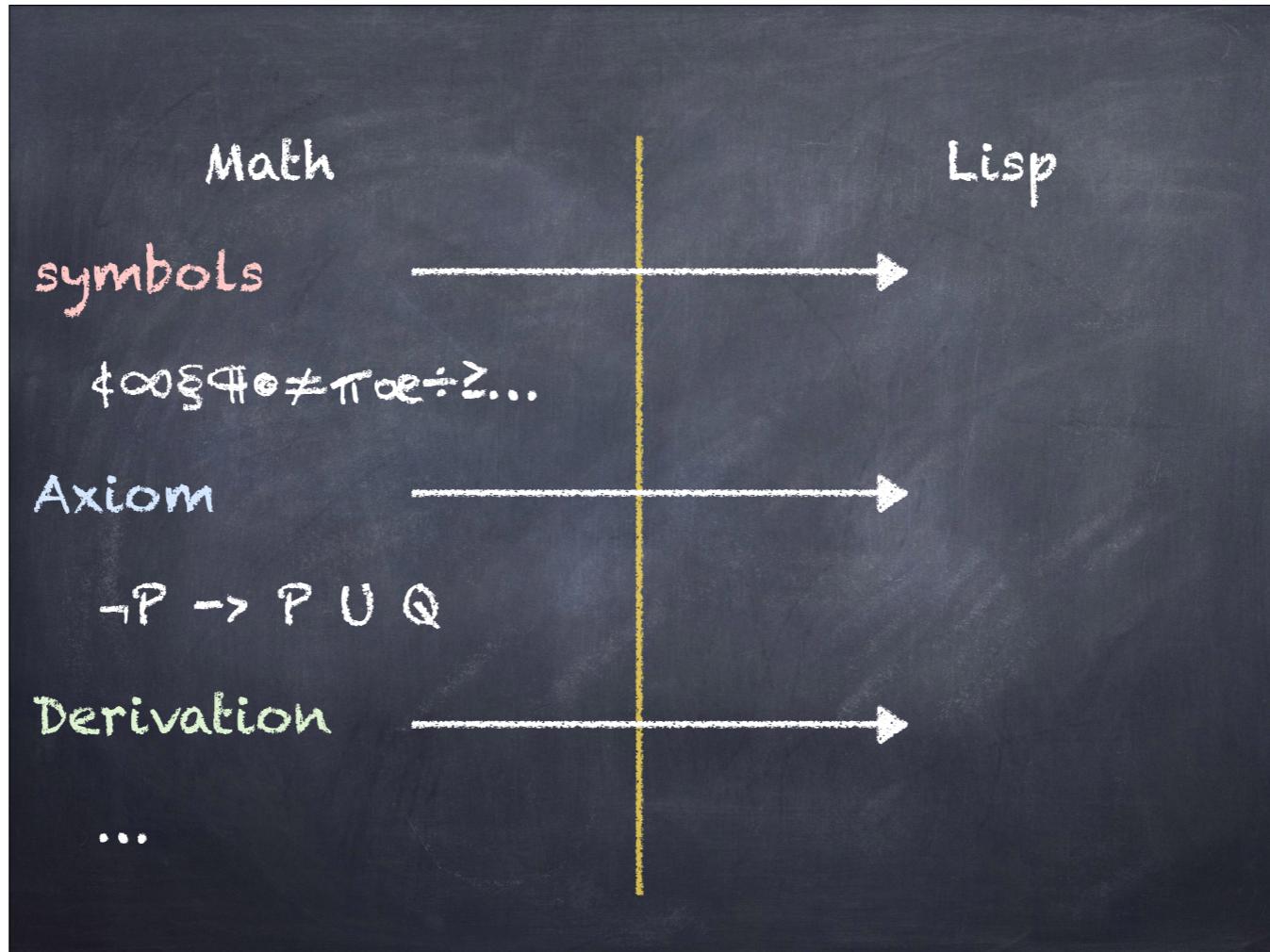
lisp也是一种思考方式。

针对一门语言，往往需要契合其特点，用特定的思维去写程序。

Based on  
Mathematics



以数学理论，形式化定义为基础，加以推导，演算。



Lisp的核心很精简，这里可以拿数学和Lisp做个类比。



仔细学习Lisp你会惊讶发现，编程领域很多当下流行的东西早在Lisp中就已经提出甚至实现了。

比如广为流行的设计模式：面向对象。

自动内存分配与垃圾回收，

类型推导系统

闭包

.....

S - expression

- (cons 'a '())
- (if (zerop lst)  
"is zero"  
(car lst))

S-表达式是Lisp的语法构成

# define a function

```
(define fib (n)
  (if (zerop n)
      0
      (+ n (fib (- n 1))))))
```

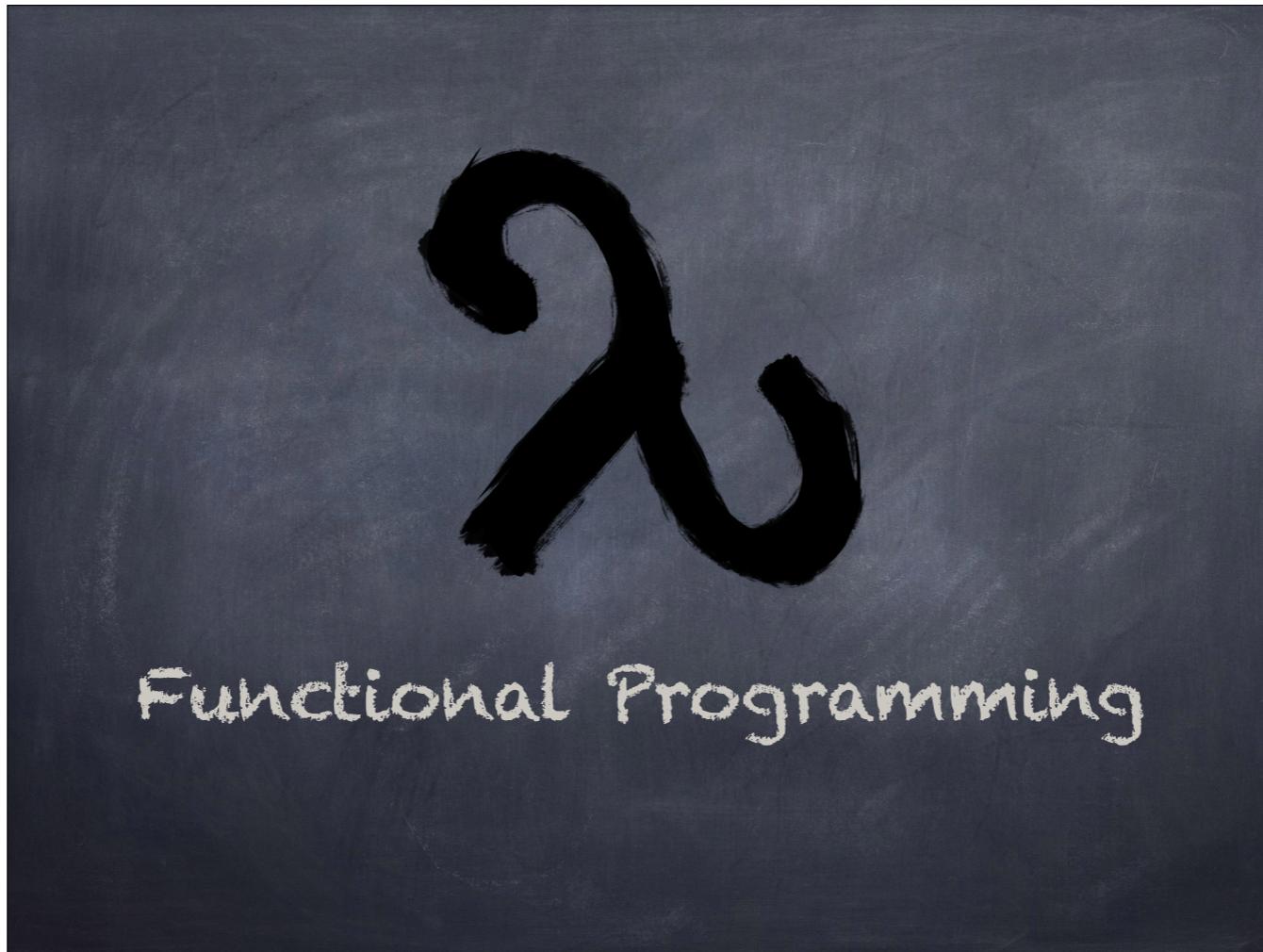
Excerpt from a typical LISP program:

```
) ) ) ) ) ) ) ) ) ) ) ) )
```

Excerpt from a typical Node.js program:

# What's the hell !?

— dont worry about "))))",  
editor will deal it instead of u

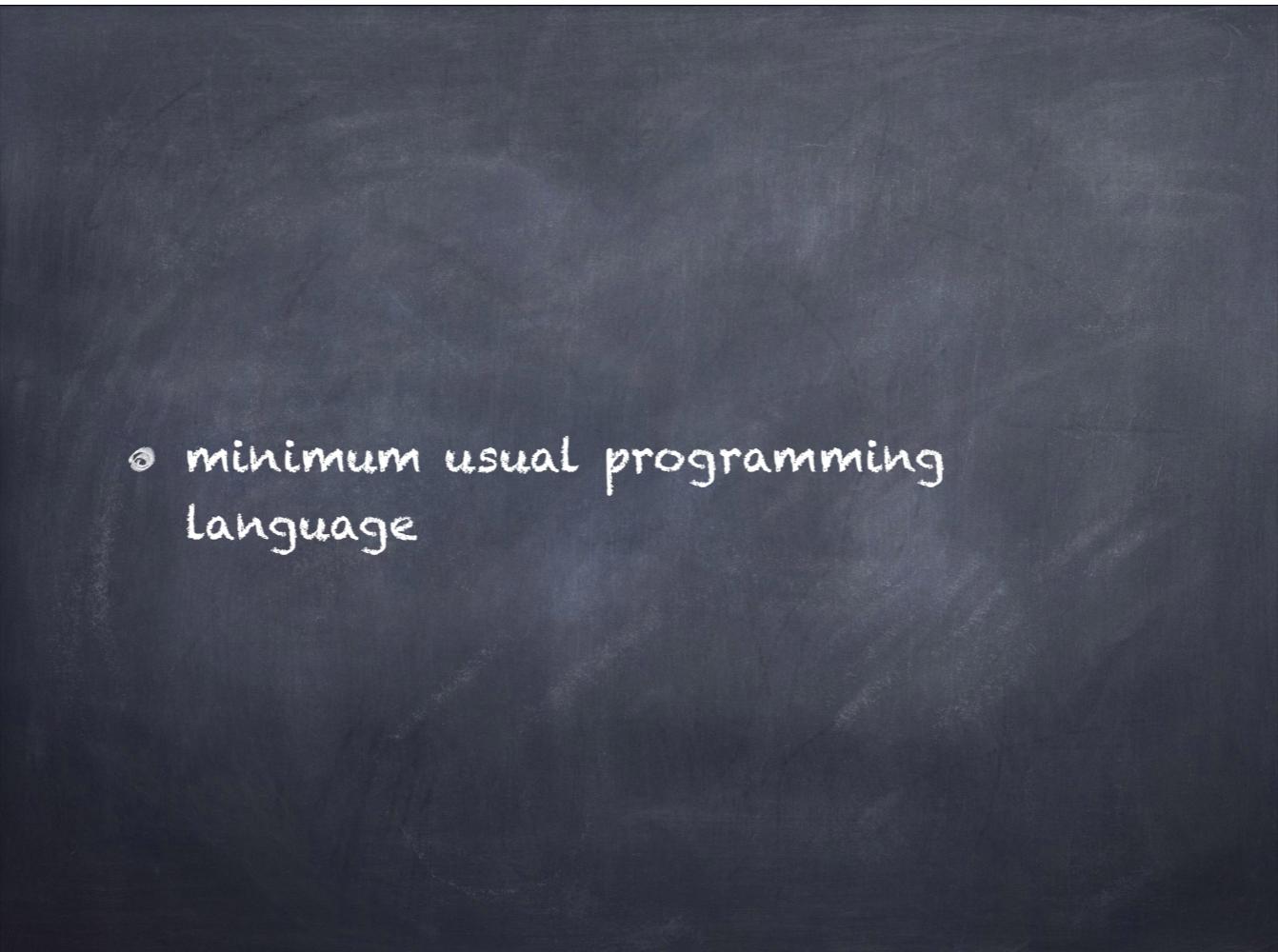


函数式特性：

- + lambda演算为基础
- + 不改变程序状态（不变量）
- + 函数作为一种数据类型存在
- + 将复杂问题拆分成若干单元（通用型的函数）
- + 不断渐进计算结果
- + 逐层推导复杂计算

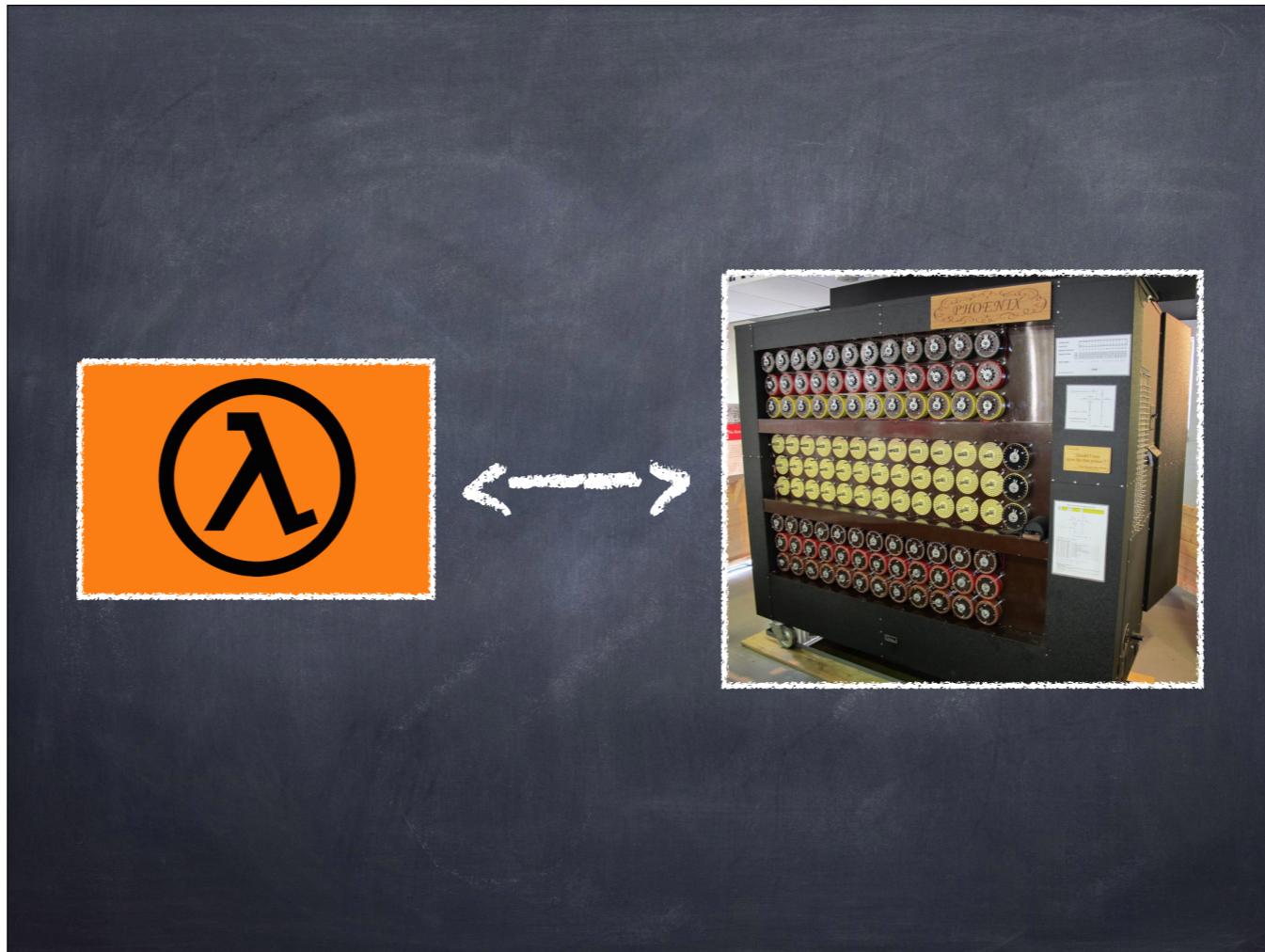


Lambda ?



• minimum usual programming  
language

lambda演算可以看做最精简的通用程序设计语言。  
任何一个“可计算函数”可以通过lambda演算表达、求值。



图灵等价

除此之外还有：

- + 马可夫链模型
- + 细胞自动机
- + .....

- definition:  $y = \lambda x. x + 1$
- apply:  $y 8$
- type: single argument function

包括两部分：

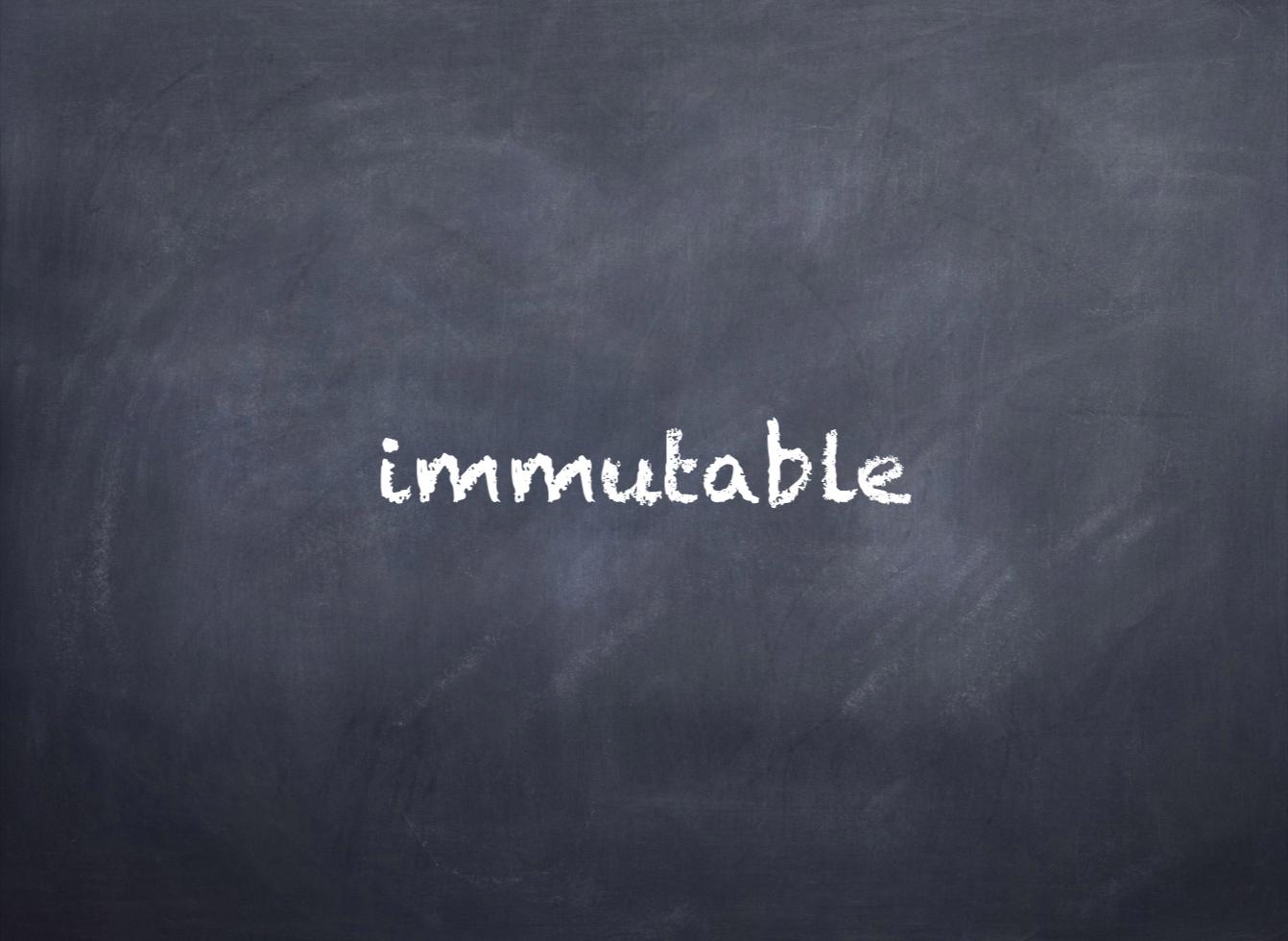
- + 表达式定义
- + 函数应用 (apply)

除此之外，也暗示了“函数也作为一种类型”

# the first class object

第一类对象：可以被作为参数传递、返回值，并且可以被赋值给变量的对象。  
Lisp函数和JavaScript函数就属于第一类对象。

treat function as data  
— in lambda calculating



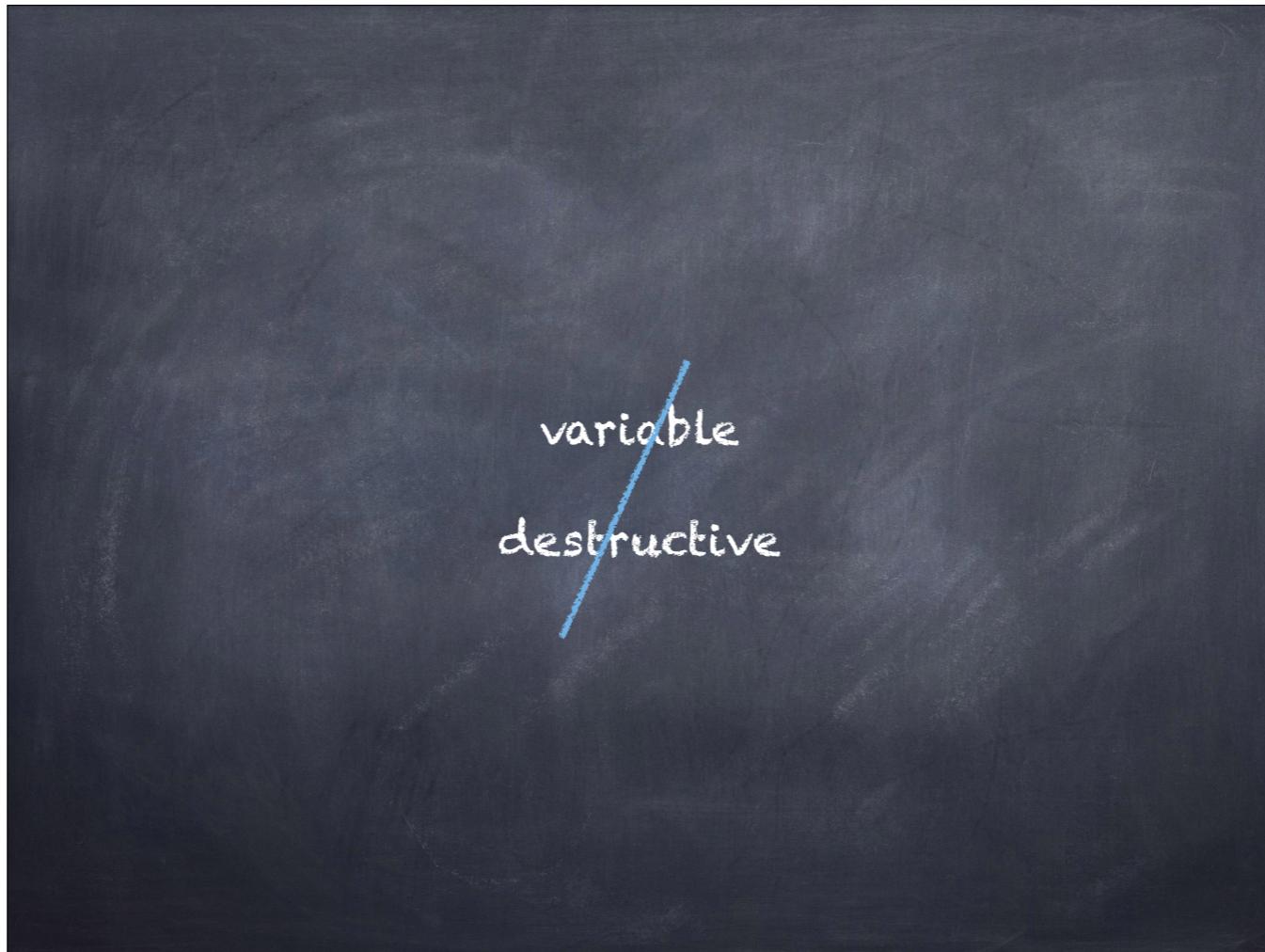
immutable

## 不变量

函数式编程中一条约定：编写利用返回值工作的程序，而不是修改什么东西，亦即不要破坏性、副作用



它是纯函数式编程语言

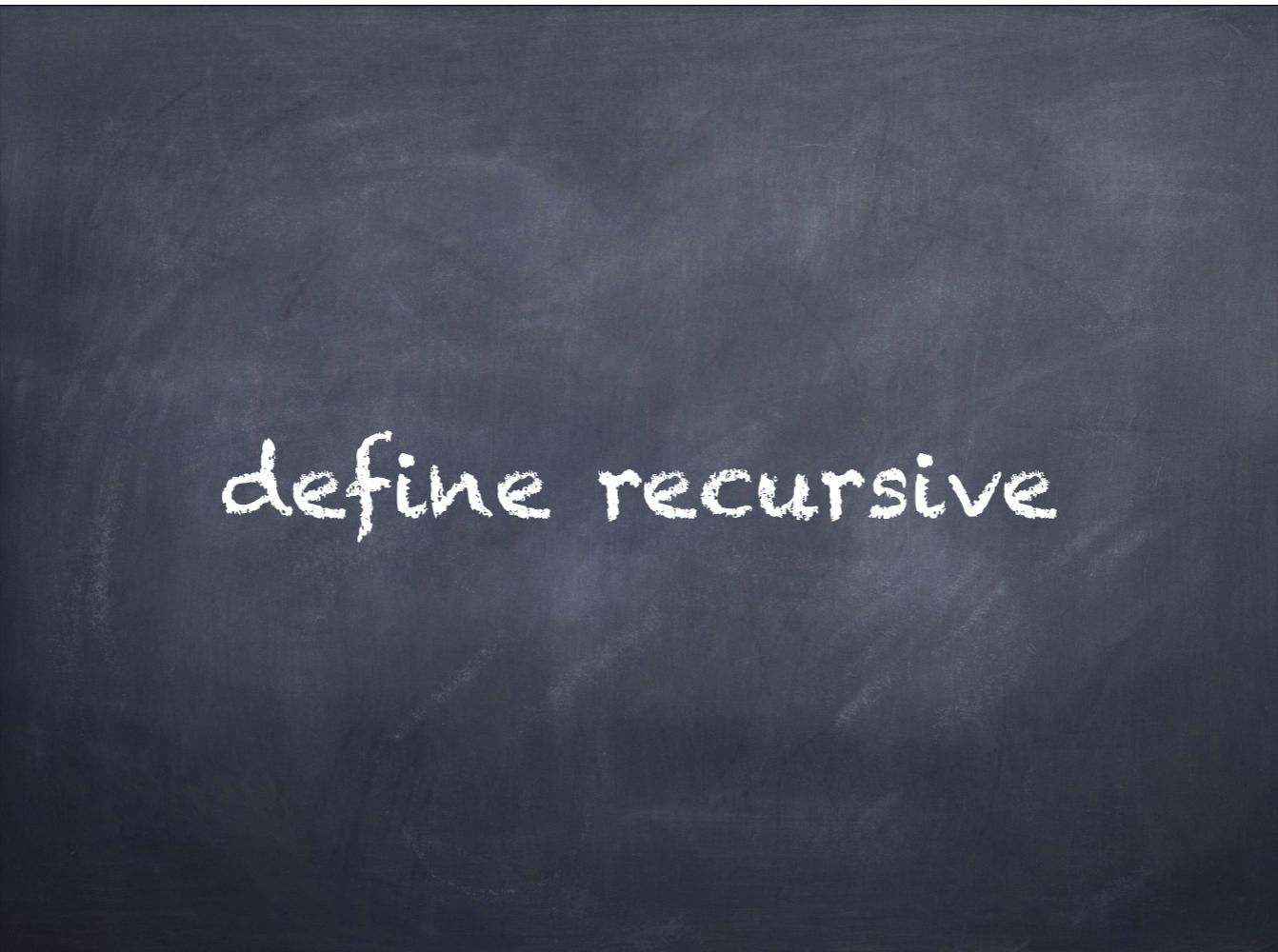


纯粹的函数式编程中没有变量和破坏性操作（副作用）。

纯粹的函数式编程与数学几乎没有区别：

赋值只允许一次，如： $a = 5$ ，就是说 $a$ 只能是5，而不能是别的值了，

数学中的函数从来不会改变这些“变”量，只能创建新变量并赋值： $y = f(x) = ax^2 + f(x - 1)$



define recursive

“递归”对于已经熟悉了xx的人来说比较费解：一个机器在运行时如何在次执行他自己？  
但这个想法是错误的，递归的概念其实是“一种策略的反复调用，而不是一个实体的自身调用”。

fact =

Lambda n . (if (or (zerop n)

(equal n 1))

1

(n o (fact (- n 1))))

fact doesn't exist now :(

- bind x to lambda expression:

  - Lambda x . x + 1

- y isn't binded:

  - Lambda x . x \* y

Fixed Point

$$\circ f(x) = x$$

$$\circ f(f(x)) = x$$

$$\circ f(f(f(x))) = x$$

$$\circ f(f(f(\dots))) = x$$

$\circ x$  is a fixed point of  $f(x)$

# $\lambda$ - Combinator

- $f = g f$
- $f$  is a fixed point of  $g(x)$
- $\lambda g = g(\lambda g)$
- $f = \lambda g$

via Y

fact =

(lambda f . n . (if (or (zerop n)

(equal n 1))

1

(n o (f (- n 1)))) fact

fact =  $\lambda\ g$   
 $= g(\lambda\ g)$   
 $= g(g(\lambda\ g))$   
 $= g(g(g(\dots(\lambda\ g)\dots))))$

So, what's the expression of Y?

same

$$Y = \text{Lambda } f . (\text{Lambda } x . f x x) (\text{Lambda } x . f x x)$$



$$(Y g) = (g (Y g))$$

(only? (in functional lisp))  
=> NIL

# Meta-programming

元编程，在代码执行的过程中诞生另一块具有其他功能代码，生成的称之为元代码。

元编程旨在用更少量的人工代码去完成更多的任务。这种代码的生成可以是另外一种语言，也可以是自身代码，最典型的当属编译器，就是一种元编程的典型。



eval

最初的Lisp解释器就是eval函数。

eval函数最先靠汇编代码编写，随后的Lisp语言皆由eval完成解释。

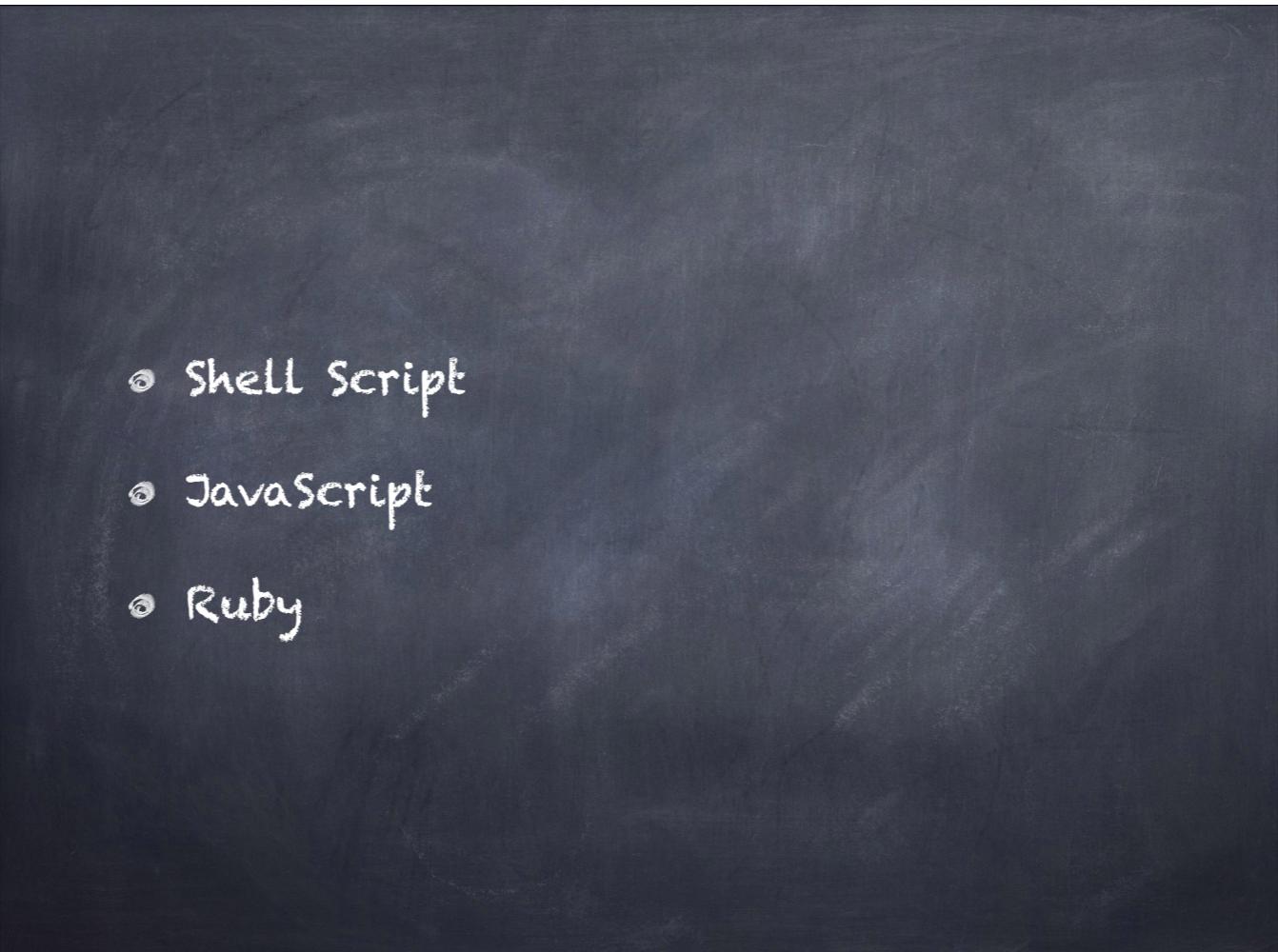
# Reflection

反射是元编程中的特例：意思是代码在运行时对自己的结构动态修改执行。

元循环解释器（eval）就是最好的示例。

JavaScript之父在github上开源了一个js元循环解释器“Narcissus”

compiler / interpreter



- Shell Script

- JavaScript

- Ruby

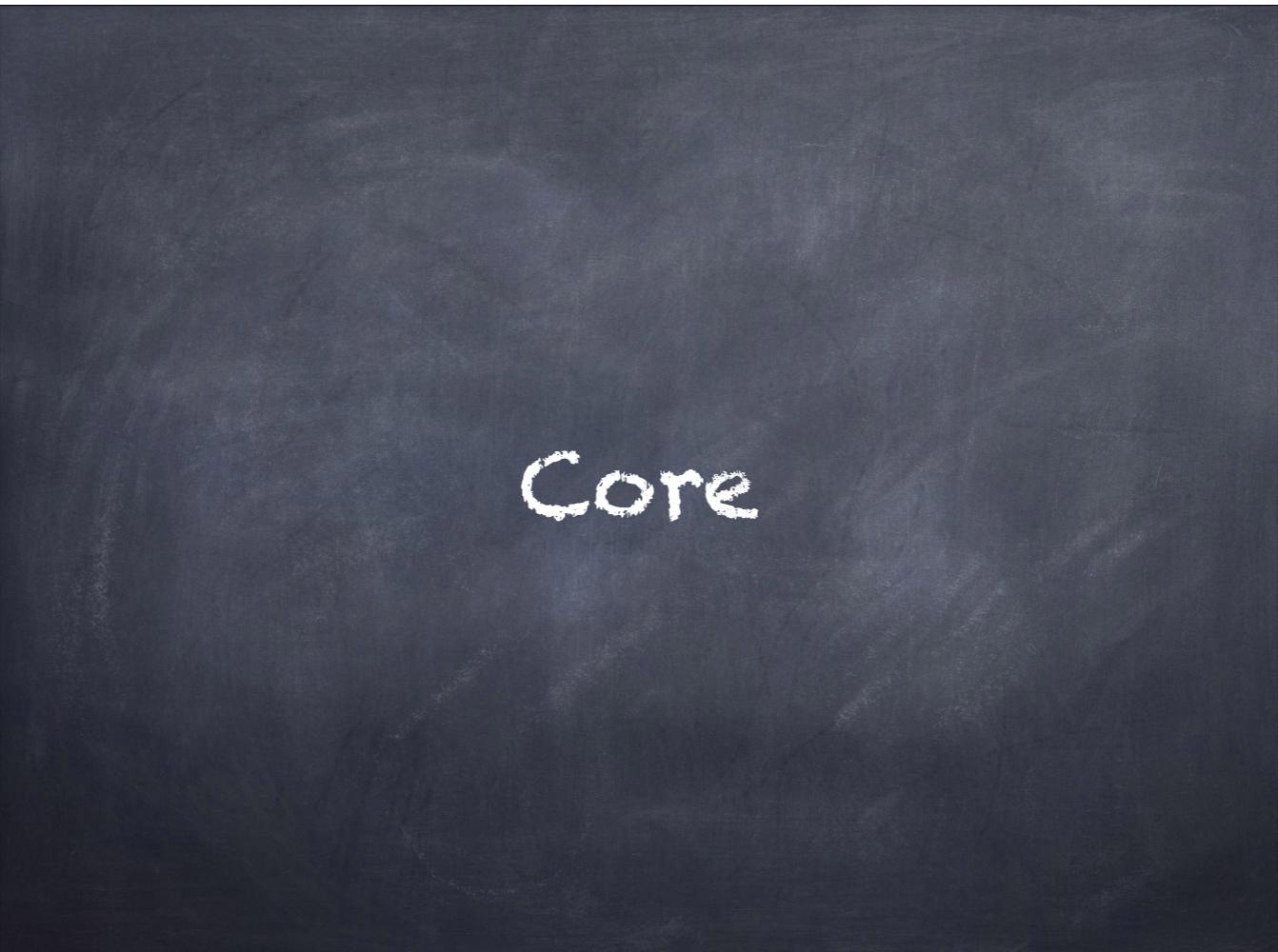
他们都是拥有反射能力的语言



这也是lisp与众不同之处，其本身的设计允许自身不断进化。

Lisp可以通过一些标准特性来扩展，例如Lisp宏（程序自我进行的编译时代码重排）和阅读器宏（赋予用户自定义的保留字以特殊意义的符号扩展）

———— from wikipedia



Core

C系与Lisp系从两个不同的设计角度出发，走的完全不同的路线。  
其核心分别是图灵机与lambda演算

# diff -Lisp -c

- Lisp family: we tell machine **what** to do (in Math)
- C family: we teach machine **how** to do (in Machine)

Why Lisp ?

# Lisp philosophy

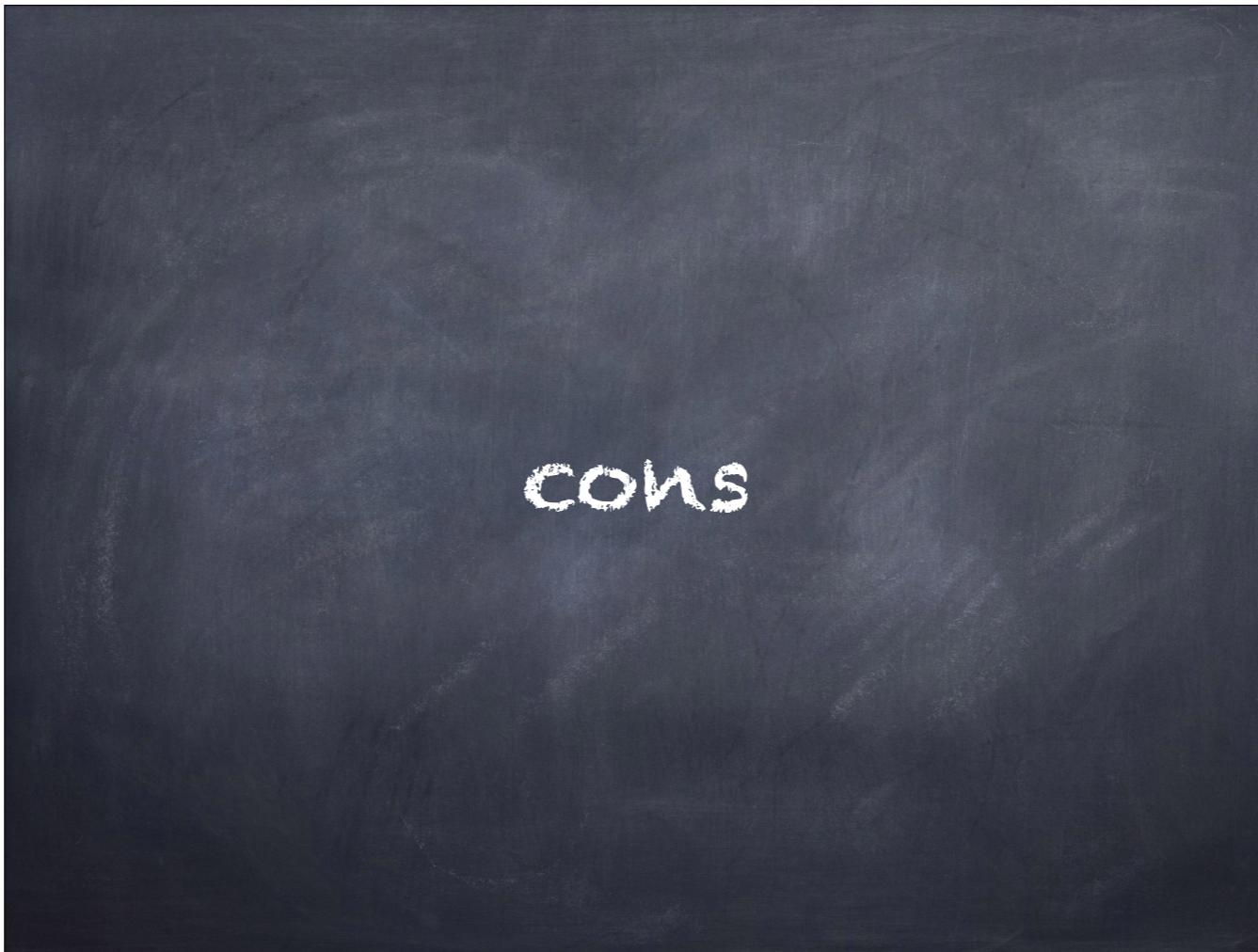
- + 不需要改变想法来迎合语言，因为可以改变语言来配合想法
- + 强调结果而非过程
- + 自底向上设计程序，每一层都是上一层的语言
- + 将写程序视为另一门语言的设计过程 (DSL)

Type & Value

list

(Lis)t (p)rocess → Lisp

一种表处理语言

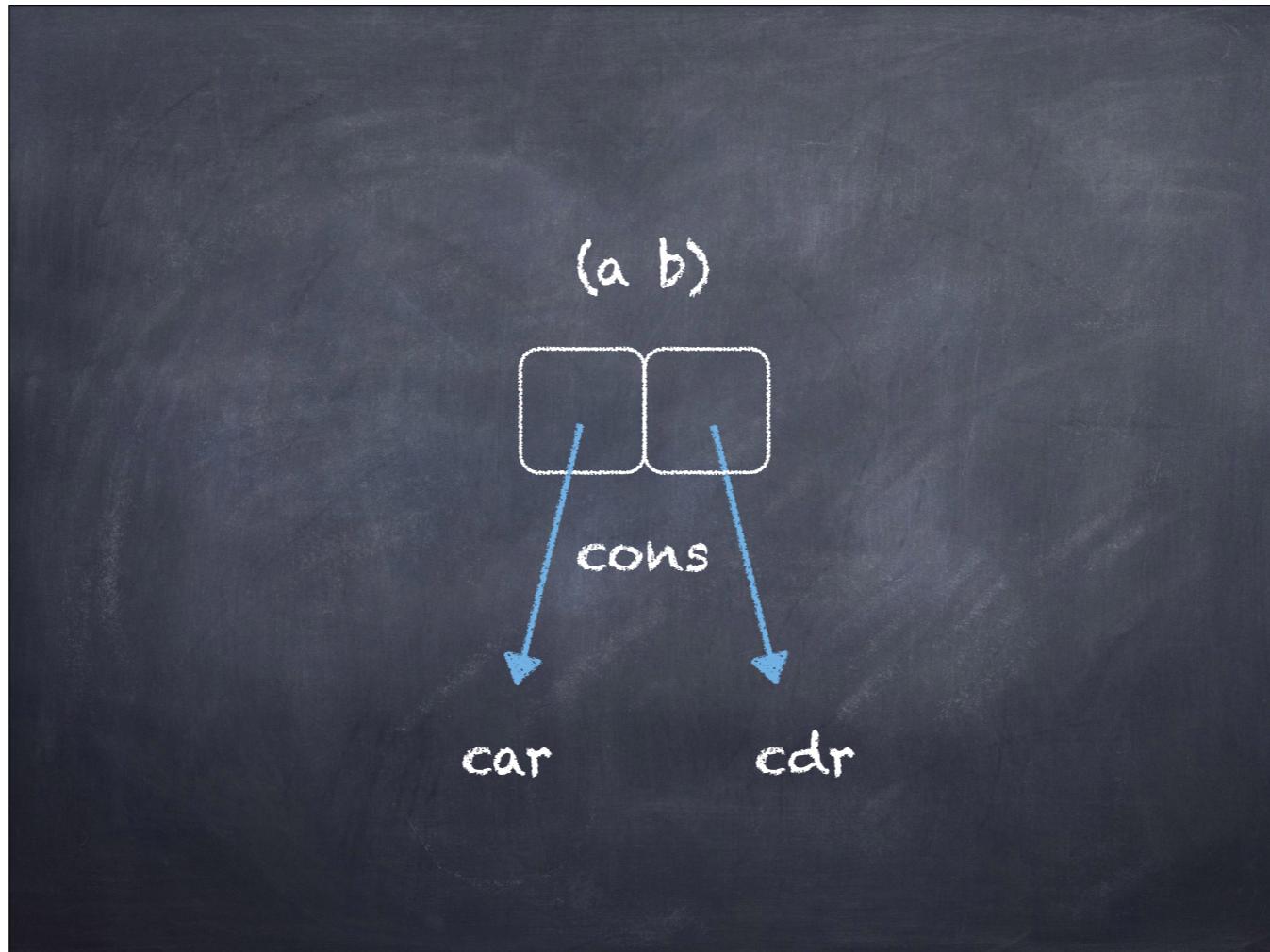


cons对象Lisp中一个二元序对，也是几乎全部代码的结构组成单元。

cons包括car和cdr，分别表示cons对的第一个元素和剩余元素

常见list (lisp代码) 是由多个cons组成，也称cons串。结构：

列表本身的结构就可以表达成一颗二叉树，每个cons为一个节点，包含car和cdr。结构：

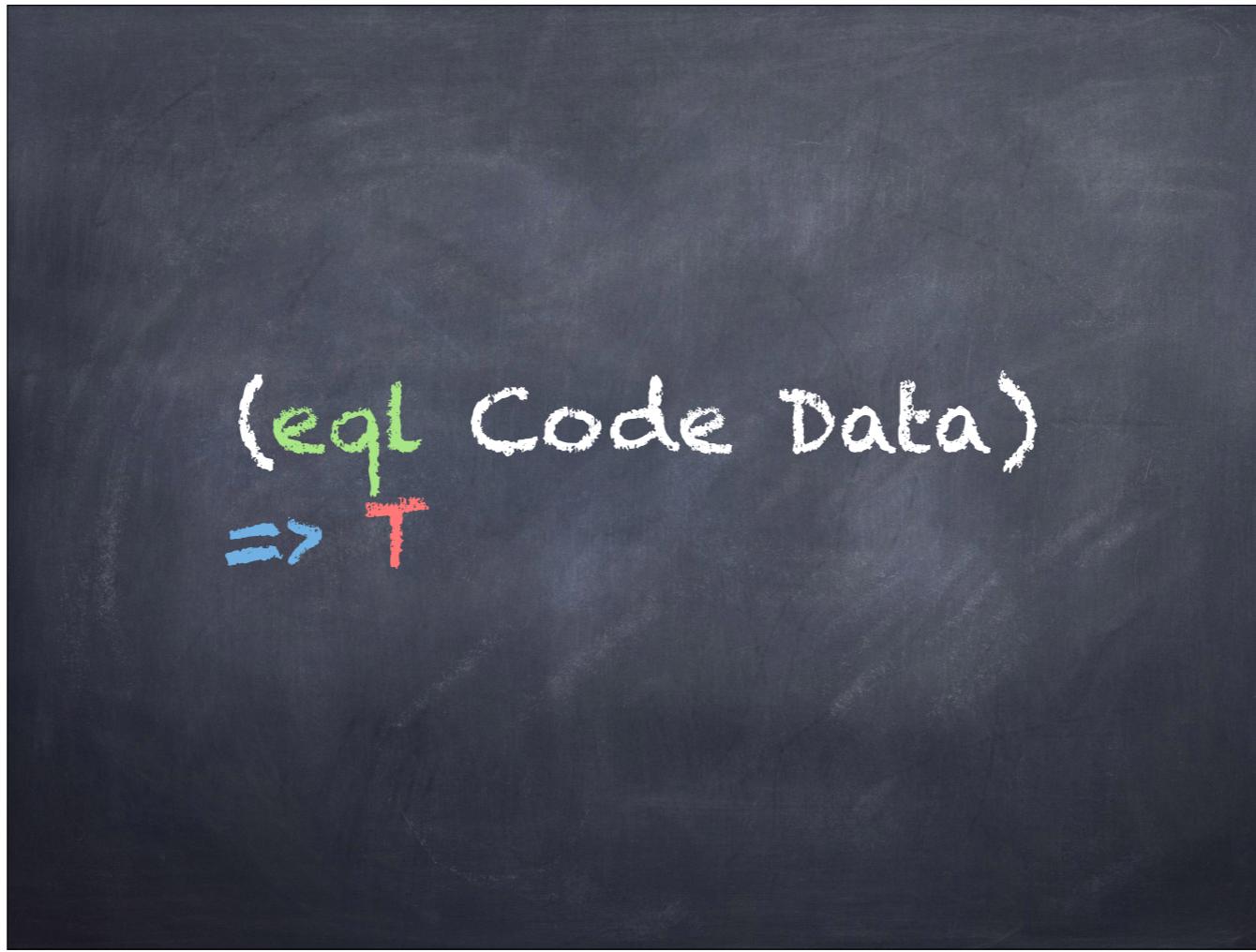


cdr不是b，而是(b)，正规列表中cdr必须是list



同向性：程序结构（src源码）与其语法结构（AST抽象语法树）相似。（因此易于通过阅读程序来推测其内在涵义）

代码与数据同构。



+ 代码即数据

source code  
is similar to  
AST



代码可以作为数据来修改、交换

- as code:
  - (List "a" 3 'b nil)
- as data:
  - '(a b c "str" & nil (List List))

一线之隔， quota

# Closure

当一个函数引用到外部定义的变量时，称这些变量为“自由变量”，而这个函数称之为“闭包”

`lambda x.y. (lambda z. (+ x y z))`

x, y为自由变量



Common Lisp中的宏是独一无二的，和C语言中的宏的机制相同，但是在宏扩展的过程中由于可以使用所有现有的Common Lisp功能，因此宏的功能就不再仅限于C语言中简单的文本替换，而是更高级的代码生成功能。宏的使用形式和函数一致，但是宏的参数在传递时不进行求值，而是以字面形式传递给宏的参数。宏的参数一旦传递完毕，就进行展开。展开宏的过程将一直进行到这段代码中的所有宏都展开完毕为止。宏完全展开完毕后，就和当初直接手写在此处的代码没有区别，也就是嵌入了这段代码上下文中，然后Lisp系统就对完整的代码上下文进行求值。

----- from wikipedia



curry

- + 惰性求值
- + 部分求值
- + 高阶函数

(Lambda x. y. y + x) 4  
=>  
Lambda y. y + 4

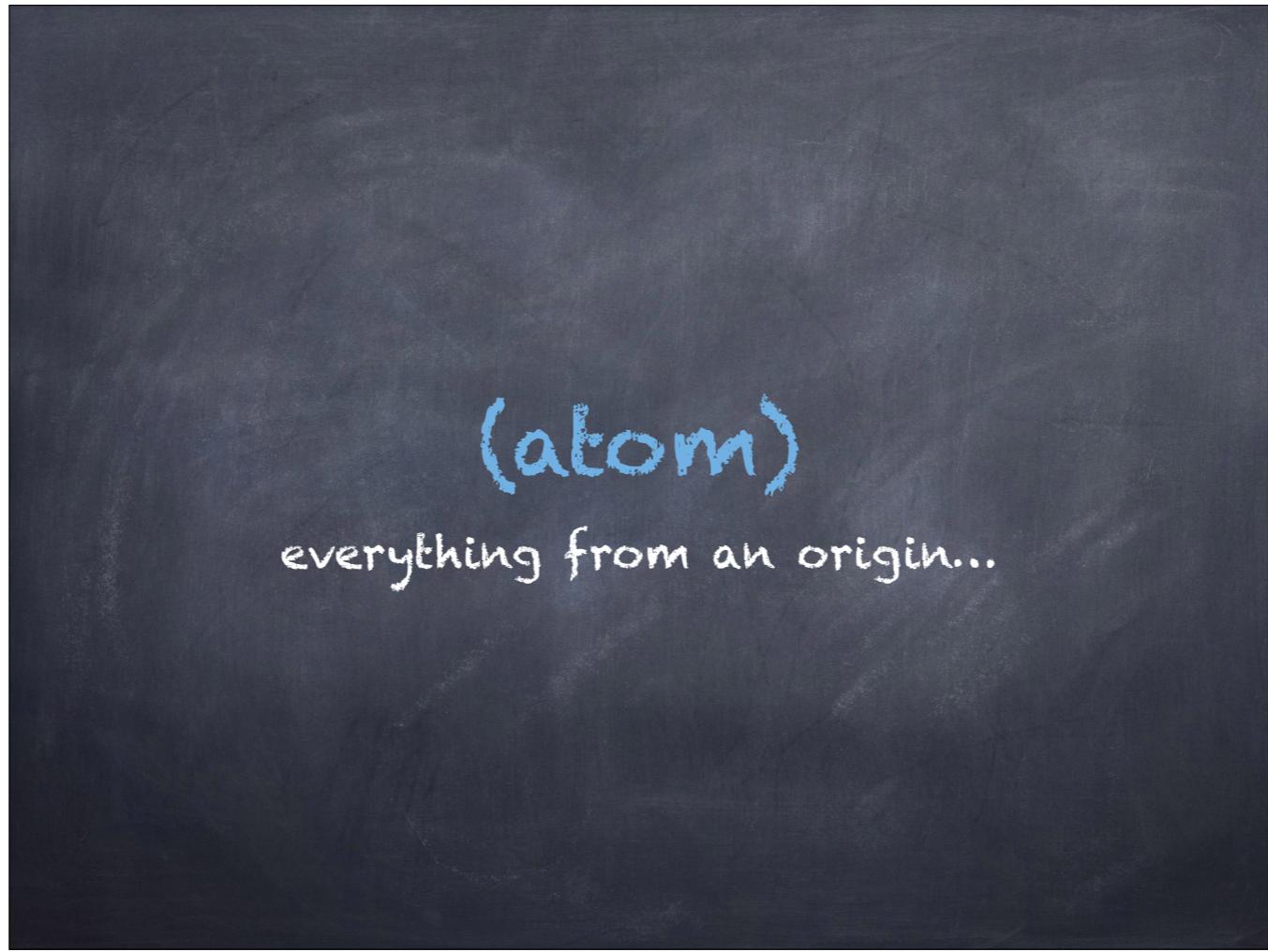
JavaScript中不允许通过部分传参来直接进行函数的curry化，但可以用函数原型的“.bind”方法实现：

```
isFriendOfThem = ((n1, n2) -> @isFriend n1, n2 ).bind person, "gloff"  
isFriendOfThem "carl"
```

other goodies from Lisp

- destructing-bind
- rest arguments
- default argument
- multiple value
- auto-mm (GC)
- type deduction system

195x年的设计如此惊艳，这个先进的思想比某某大肆宣扬其自动垃圾回收功能的语言早了几十年...



原子是一种抽象表示，它没有硬编码的解释，只要在其上的操作满足逻辑定义即可。

True

• t = lambda x . y . x

False

•  $f = \lambda x . y . y$

If

• if = Lambda p . x . y . p x y

$(\text{if } t \ 1 \ 2) \rightarrow (\text{if } (\text{Lambda } x.y. \ x) \ 1 \ 2)$   
 $\quad \rightarrow (\text{Lambda } x. \ y. \ x) \ 1 \ 2$   
 $\quad \quad \quad \rightarrow 1$

$(\text{if } f \ 1 \ 2) \rightarrow (\text{if } (\text{Lambda } x.y. \ y) \ 1 \ 2)$   
 $\quad \rightarrow (\text{Lambda } x. \ y. \ y) \ 1 \ 2$   
 $\quad \quad \quad \rightarrow 2$

zero

- $0 = \lambda f. x. x$
- definition same 2 f

# numbers

- $1 = \lambda f. x. f x$
- $2 = \lambda f. x. f(f x)$
- $3 = \lambda f. x. f(f(f x))$
- ...
- $n = \lambda f. x. f(f(\dots(f x)\dots))$

# SUCC

• SUCC = Lambda n. f. x. f (n (f) x)

SUCC 3

→ Lambda f. x. f(f(f(f x)))

→ 4

+

$\circ + = \text{Lambda } m. n. m \text{ SUCC } n$

(+ 2 3)

$\rightarrow 2 \text{ SUCC } 3$

$\rightarrow \text{SUCC } (\text{SUCC } 3)$

$\rightarrow \text{SUCC } 4$

$\rightarrow 5$

X

•  $x = \lambda m. n. n (+ m) o$

$x 2 3$

$\rightarrow 2 (+ 3) o$

$\rightarrow (+ 3) ((+ 3) o)$

$\rightarrow (+ 3) (+ 3 o)$

$\rightarrow (+ 3) 3$

$\rightarrow (+ 3) 3$

$\rightarrow 6$

# Lisp Localism

- Scheme
- Clojure
- ANSI Common Lisp
- Emacs Lisp

到现在为止，Lisp家族已经发展出了N多分支，并影响了世界各地的N多种语言。其坚如磐石而又贯通古今的理论基础与可自身进化的特点也让它在时代的浪潮中经久不衰。

(exit 0)