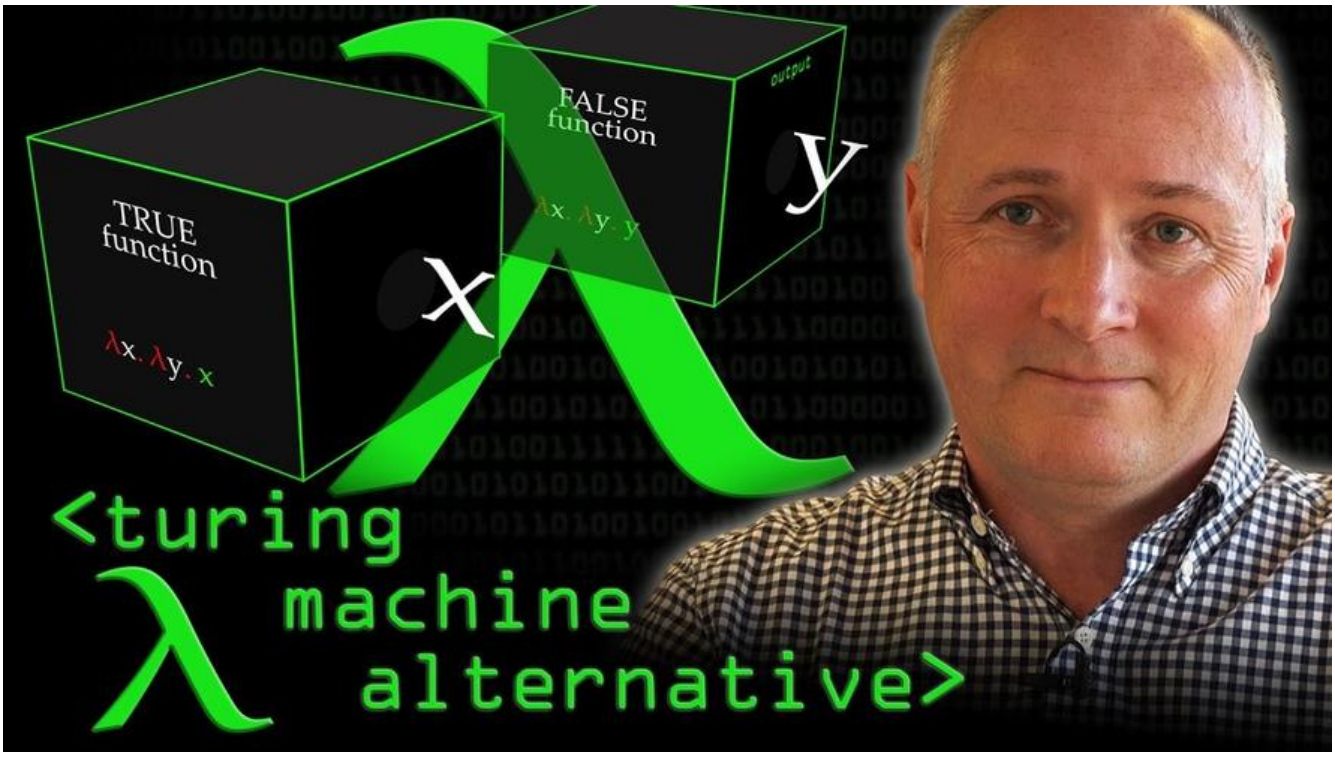
链滴

lambda 演算

作者：zhaozhizheng
原文链接：https://ld246.com/article/1619087739116
来源网站：链滴



一篇给小白们看的lambda演算教学文章，和图灵机一样，lambda演算也是计算机理论基础的重要组部分。也是理解函数式编程的一扇窗户。这篇文章的作者是一位来自MIT media lab的工作人员，从绍上看他是一位认知科学家。原文的标题叫做： The Lambda Calculus for Absolute Dummies。

原文链接：http://palmstroem.blogspot.hk/2012/05/lambda-calculus-for-absolute-dummies.html

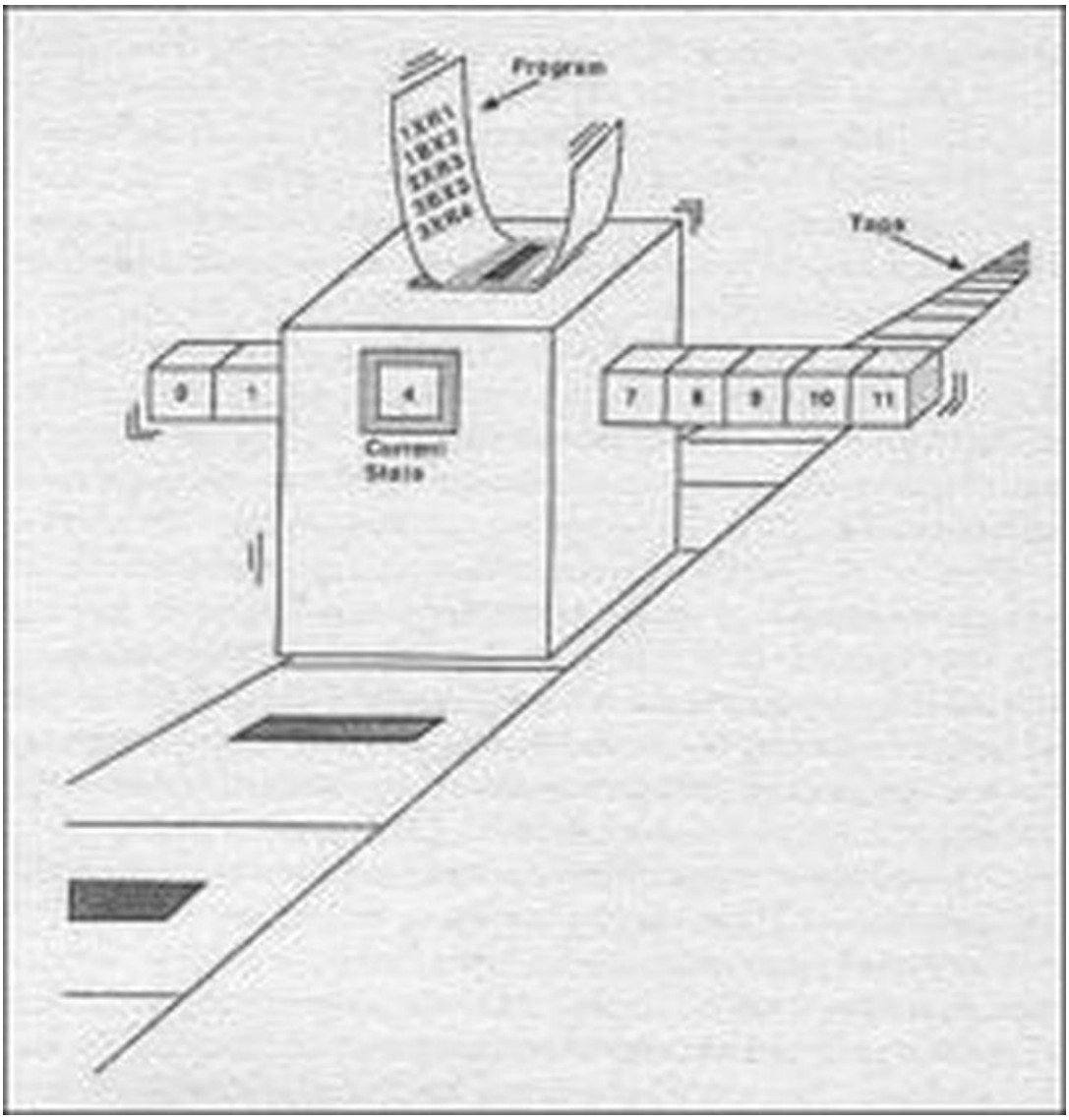
作者：Joscha Bach, Cognitive Scientist

发明Lambda演算的作者叫做阿隆佐邱奇（Alonzo Church）， 发明图灵机的作者叫做阿兰图灵（Ala Turing）。他们几乎活跃在同一个时代，他们那个时代的数学界有个领袖，叫希尔伯特（David Hilbet， 德国数学家），当然比图灵和邱大不少。简单说，他鼓舞大家去将证明过程纯机械化，这样机器可以通过形式语言推理出大量定理（是不是有点像人工智能，机器自己把定理枚举了）。当然那个时没有今天计算能力如此强大的机器，但当时的科学家们已经在思考今天的事情了。图灵和邱奇都受到股思潮的影响，几乎从不同的角度解决了同一个问题。无论是图灵机还是λ演算，都可以模拟出我们天的所有程序。

λ演算是一种形式系统(formal system)，什么是形式系统呢？大家知道，数学语言是是可以脱离现实存在的——大家把数学想成了一种符号游戏，脱离生活常识，从公理开始，进行大量的推导和证明—最终产生了一个系统，里面有公里、定理、推论、猜想...上述这种自成体系，有公理又承认推理证明法的体系，称为形式系统。那么什么是形式语言呢？形式系统需要语言去描绘，这种语言就是形式语言formal language)。

如果有什么理论在哲学界被严重低估，那就是**计算**(computation)。为什么计算这么重要？ 因为**计算义**是新的机械论（一种认为自然界整体就是一个复杂的机器或工艺品，其不同组成部分间并没有内在系的哲学）。近一千年，哲学家们一直努力用机械方法描述整个宇宙，但没有成功，因为什么是机械或什么不是）太难解释。而**计算**刚好解决了这个问题——计算定义了机械可以做什么，不可以做什么如果宇宙/思想/大脑/小兔子/上帝是可以用机械的方法解释的，那么他们就是电脑，反之亦然。

不幸的是，大多数不是计算机或程序领域的人不明白什么是计算。大多数人听说过图灵机，但这些往对理解利大于弊，因为他们最终对纸带、轮子留下了很有感觉的印象，而不是理解了图灵机到底在做么——计算的本质。



什么是图灵机？ 图灵机是阿兰图灵（Alan Turing）发明的机器，最早是想用来解决一个叫做Entscheidung Problem（德文：判定问题）的问题。 判定问题其实是数学世界的核心，就是用机械化的方法证明、去推断，解决了判定问题，数学领域很多公式定理就可以让机器自己跑啦（1930年左右，人类已经在构思今天的AI了）。就是上图那个有着纸带、读写头、寄存器、规则表的奇怪机器，可以用来行计算，可以实现我们今天说的所有的算法，计算机就是根据图灵机制造出来的。

λ演算和图灵机做一类事情，但是没有轮子混乱你的视线。你也许会被它吓到，毕竟有个大家不认识希腊字母λ嘛，所以非学术领域几乎不怎么去学习他——但是它真的很简单，非常容易理解！如果你解了它，你会对**计算**有更好的直觉。（lisp就是受它启发的语言哦， 所以说开阔下视野很有意义，特是函数式编程又重新回归的今天）

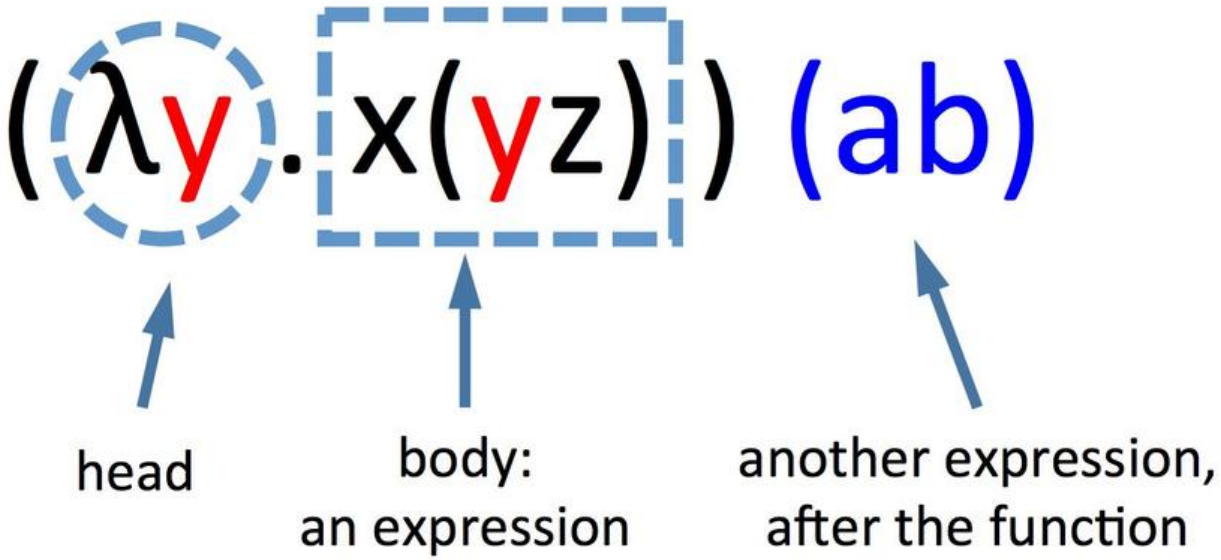
λ演算是由阿隆佐邱奇发明，和图灵机几乎同时代被发明。不要被演算(Calculus)这个词吓到。这里没任何的公式或者操作。λ演算其实就是将一行字母进行查找替换。你会看到它凭借简单的剪切和粘贴几乎可以计算所有东西。

在λ演算中，一行符号被叫做表达式。例如表达式长成这样子：(λx.xy) (ab)。表达式只包含以下符号：

- 单个字母（abcd...），被称作变量。一个表达式可以是单个字母，或多个字母。一般的，我们可以把个表达式写在一起组成一个新的表达式。
- 括号()。括号表明表达式被括起来的部分是一个整体（就像句子中的括号表示这部分是一个整体）当我们没有括号时，我们从左到右分析表达式。

原文链接：lambda 演算

● 希腊字母λ(发音：Lambda)，和点(.)。 λ和点，我们可以描述函数。函数由λ和变量开头，跟上一个，然后是表达式。λ没有任何特别的含义，它只是说函数由此开始。在λ后面，在点之前的字母，我们作的变量，点之前的部分，被称作头部(head)，点后面的表达式，被称作体(body)部。



提问：变量有什么含义?

回答：没有任何含义。它不代表任何。他们只是空的名字。甚至名字也不重要。唯一重要的是，如果个变量有相同的名字，他们是代表相同的东西。你可以任意修改变量的名字，而不会影响表达式。

提问：函数在计算什么?

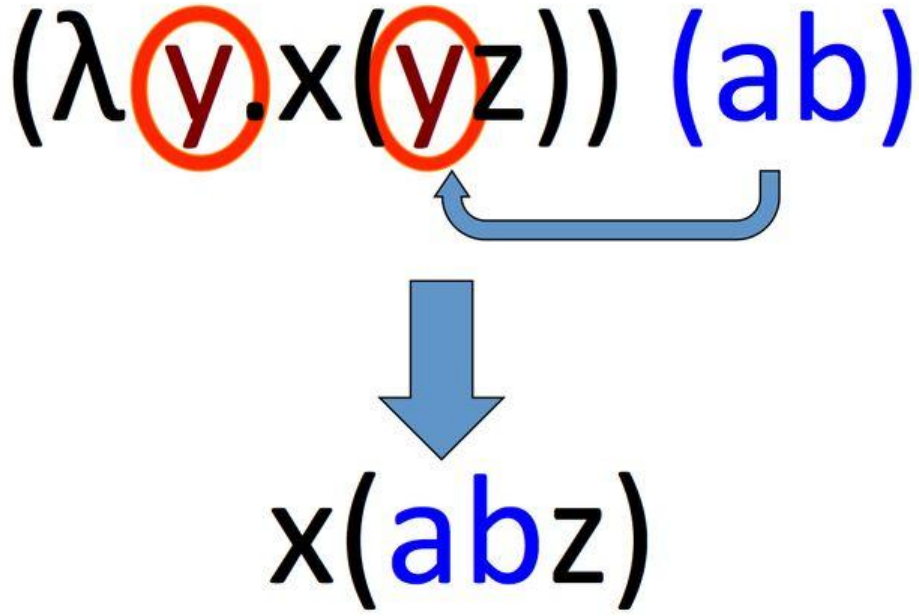
回答：什么也不计算。这就是一个表达式，有头和体。它就在那，我们唯一可做的事情就是解析(resoling)它。

提问：为什么是λ?

回答：偶然因素。也许一开始邱奇画了一个顶部符号上去，像这样：(ŷ xy) ab。在手稿中，他写成了样(y.xy) ab。最后排字工人，把它变成了这样(λy.xy) ab。

我们可以更加正式的说：a. 所有变量都是λ形式（所有变量都是合法的λ表达式）；b. 如果x和y是λ形，那么(xy)是λ形式; c. 那么(λx.y)是λ形式。通过这三条规则，我们可以写出所有的λ表达式。如果我从左向右读λ表达式，我们可以少写一些括号：(λy.xy) ab是 (((λy.(x y)) a) b)的化简版。

剪切和粘贴



在函数后边又跟了一个表达式时，它可以被解析。解析过程就是将头部一个变量去掉，然后将它所有体部的出现的这个变量替换成写在函数后边跟着的表达式。 也就是说，我们剪切函数后面的表达式，后粘贴进体，替换和头部同名的那个变量；做完这一步，我们把头部的那个变量删了，因为它已经完了它的使命，告诉我们替换哪个变量。

解析函数是我们在λ演算过程中唯一可以做的事情。当我们已经没有任何东西可以解析时，也就是说们不可以再替换任何东西。我们就可以回家了（完成）。

提问：函数可以包含函数吗?

回答：当然。因为函数是表达式，表达式可以包含其他表达式，所以函数可以是其他表达式体（body的一部分。事实上我们有这样的表达式λx.λy.xzy，但通常我们这样缩写成λxy.xzy。这表示我们会先尝在体（xzy)中替换头部的x，然后是y，依此类图（从左到右）

在头部中提到的变量被称作**约束变量**，没有提到的称作**自由变量**。因为函数可以是其他函数的一部分所以一个变量可以同时是约束变量，又是自由变量。

提问：我觉得有点懵

回答：这样想。 假设你在编辑一个极简主义的八卦报纸。所有的内容都是人名（报纸上只有人名，没文章，动词）。人们不愿意在报纸上被认出，所以你只好用假名去替代他们的真名。（比如马云替换杰伦，然后猫咪替代孙燕姿）。所以名字没有了任何含义，但是如果两个名字一样，那么就代表同一人。（读者看到所有的马云， 虽然不知道其实是周杰伦，但他们知道这是一个人）。

所有报纸中的文本被组织成了段落，一个文本段落中只有很多人名。文本段落也可能有标题，标题也名字。标题用大字黑体，而且只有一个名字。所有出现在标题的名字是出名的（比如马云），它代表新闻人物。所有不在标题中的人是普通人物。文本段落，也可能再包含文本段落（嵌套关系，子文本落也可以有标题）。所以一个名字可能在一个段落中是普通的，在一个段落中是出名的。

这和λ表达式非常像，名字就是变量，文本段落就是表达式，头条就是函数头，区别是函数头用λ和点着，而头条用大写黑体。

解析操作就是简单的查找替换，我们找到所有标题名字在内容中的出现，然后进行替换。（为什么是

换标题人物，因为标题人物出名，不容易重复）

在这里我们也许会遇到问题，比如当替换文本中也有人名已经被原文提到了。所有的人名都是假名，我们的假名已经被用作不用的人名了。当我们合并这段文本的时候，我们必须确保不同的被不同的假引用了，所以我们有时候需要改名。（比如之前周杰伦用马云替代了，但替换文本中出现的马云其实是马云，所以需要重新命名了）

另一种方法是，我们坚持在两段文本中不用相同的名字。（换句话说：要么在没有关联的表达式中使不同的名字，要么你不要忘记在替换操作中检查命名冲突并改名。）

提问：如果有一个变量在头部绑定了，但在体部中没有出现，这样的情况会怎样？

回答：这个变量是有约束的。但在替换过程中，替换的表达式将会消失，因为没有地方需要插入它。实没有关系，它简化了我们的结果，所以为什么要纠结呢？

数字

我们已经讲完了所有的技术细节（是不是很简单？），让我们开始学习一些λ演算的技巧。你也许会问说计算应该可以对数字做一些事情，所以让我们来做一些。 数学家们总是喜欢从自然数开始，然后这里开始，定义各种各样的操作，给我们各种各样的数字类型。

定义所有自然数最简单的方法就是从第一个开始（0），然后定义后继操作（successor operation）通过在自然数上定义后继操作，我们的到一个比它更大的自然数，然后一个一个定义所有的自然数。

让我们这样定义0：

0 :□ λ sz.z

（记住：这是 **λs.λz.z**的缩写，它和**λab.b**, 或 **λqx.x**是一个意思。）

这个表达式有一个有意思的特性:当它被解析，它会把第一个表达式丢掉，然后第二个原封不动。它的束变量**s**会被空字符串替换（因为它不在体中出现），所以最后留下一个**z**。

类似的，

1 = λ sz.s(z)

2 = λ sz.s(s(z))

3 = λ sz.s(s(s(z)))

4 = λ sz.s(s(s(z))))

...

换句话说，我们的计数法其实就是在z之上嵌套表达式s(...)，数字多大嵌套多少次。（也就是说：如果们解析数字n，上述过程被复制n次）。我们可以这样说：我们对z应用了n次s。

一个好的后继函数是：

S :□ λ abc.b(abc)

让我们用这个后继函数计算0：

S0 = (λ **a**bc.b(**a**bc)) **(λ sz.z)**

= λ bc.b((λ **s**z.z) **b**c)

= λ bc.b((λ **z**.**z**) **c**)

*= λ bc.b(c)

*

最后一个表达式不可以再简化了（没有函数了），然后——

λ bc.b(c) = λ sz.s(z) = 1

换句话说，这个后继函数应用在0上产生了1，让我们再重复一次：

S1 = (λ **a**bc.b(**a**bc)) **(λ sz.s(z))**

= λ bc.b((λ **s**z.**s**(z)) **b**c)

= λ bc.b((λ **z**.b(**z**)) **c**)

= λ bc.b(b(c))

瞧！看哪~

λ bc.b(b(c)) = λ sz.s(s(z)) = 2

就像我们看到的，我们的后继函数完成了我们期待的工作：从0开始，它产生自然数。它用s(...)将传的自然数括起来，从而得到下一个数。AH，有魔力的复制和粘贴。（疑惑外国人的感叹能力，鄙视3）

提问：这样写数字很奇怪-喂-

回答：事实上，从数学家的角度，这并不比1，2，3...，罗马数字(I, II, III, IV, V...)，或者中国数字 (一, 二 三, 四, 五, ...)，或者二进制数字（1, 10, 11, 100, 101...）更奇怪。没有真正对的写数字的方式，只有习。自然数并不在意我们怎样称呼他们。

加法

把数字加起来其实就是自动化后继函数。如果我们把5和3加起来，可以理解成在3上调用5次后继函数。

幸运的是，我们写数字的方式已经将这中操作自动完成了。就像上文提到的，对n求值就是我们重复达式n遍。如果数字后面的表达式是后继函数，它会被阐述n次，当我们解析它，后继函数会被应用到后面数字n次。

3+5 = 3S5 = (λ sz.s(s(s(z)))) (λ abc.b(abc)) (λ xy.x(x(x(x(x(y))))))

如果你自己尝试解析，你会发现最后结果是**8:λ xy.x(x(x(x(x(x(x(x(y))))))))**

（大家有木有发现++这个操作和x+y这个操作在lambda表达式上是统一的，后继函数都是**λabc.b(ab)**）

乘法

一个类似的后继函数实现了乘法的功能：

MULTIPLY:□ λ abc.a(bc)

这个函数有两个参数，举例子：

2 x 3 = MULTIPLY2 3 :□ (λ abc.a(bc)) (λ sz.s(s(z))) (λ xy.x(x(x(y))))

= λ c.(λ sz.s(s(z)))((λ xy.x(x(x(y))))c)

= λ cz.((λ xy.x(x(x(y))))c)(((λ xy.x(x(x(y))))c)(z))

= λ cz.(λ y.c(c(c(y)))) (c(c(c(z))))

= λ cz.c(c(c(c(c(c(z)))))) = 6

这是如何工作的？ 如果我们仔细看，我们的乘法函数同时传入了两个参数（2,3）：

MULTIPLY 2 3 = (λ abc.a(bc)) 2 3 = λ c.2(3c)

（这是上述公式演算的一个化简版）

解析*λ c.2(3c)*得到λ cz.(3c)(3c(z))， (*忘记了吧， 2 = λ sz.s(s(z))) ，*这等价于对z应用3次第2个：c(c(c(z))),然后再对结果应用3次第1个c，得到：c(c(c(c(c(c(z))))))。和函数头λcz一起，最后得到6。

避免还留有疑惑的最好方式就是你也找一张纸自己算一遍，你会发现你很快就上手了。

（注意加法是λabc.b(abc)，乘法是λ abc.a(bc) ，而且乘法函数一次传入两个参数。）

倒数

到这里，我们讲述了如何从小的数字推导大的数字。对于减法，我们也许想要拥有一个前趋操作（pre ecessor function)。我们如构造一个减法操作呢?

根据我们之前的定义，一个数字是另一个数字的后继，除了0（记作：λsz.z)。按照定义我们可以定义前趋函数，通过对一个数字应用前趋函数，我们得到它的原数字。

通常的，数学上，我们说：**y = P(x) □ x = S(y) **——y是x的前置前趋(predecessor)，x是y的后继(uccessor)。不幸的是，这只是一个阐述，而不是计算。一个阐述告诉我们前趋函数必须符合哪些条，但不告诉我们前趋函数是怎样工作的。λ演算是去计算，必须严谨地给出具体的，如何从x得到y。

一种可能的方式是从0开始，然后调用后继函数x次：

x S 0 = x (λ abc.b(abc)) (λ sz.z)

这个结果等价于数字x。如果我们可以找到一种方式记住在第x-1次后继计算的值（倒数第一次），我就可以找到x的前趋值。

让我们用一对数来完成这件事情。我们用(y,y-1)代替x。这样我们定义一个后继函数将(y,y-1)变成(y+1, y)。我们从y=0开始，然后调用这个后继函数x次，这样这个对的值是（x,x-1)。最后，取出这个对中第二个值，我们就完成了。 (x-1居然是从0数数到x-1，哈哈哈)

我们这样定义一个对(a,b)， **λp.pab**。 最小的对是**λp.p00**，写出来是**λp.p (λ sz.z) (λuv.v)。**我们以拿出这个对的第一个成员，a，然后构造（a+1,b)。

第一个成员可以将对**λp.pab**，应用于**λxy.x**：

(λp.p a b) (λxy.x)

= (λxy.x) a b

= (λy.a) b

= a

（如上，**λxy.x**的作用是，保留第一个跟着它的表达式，然后删除第二个。类似的，对的第二项可以过这个函数取得:**λxy.y**）

新对(a+1,a)可以使用用后继函数S = λ abc.b(abc) 应用到a得到， 然后将Sa和a填充进新的对。

NEXT-PAIR pair :≡ (λ pair z.z S (pair λxy.x) pair λxy.x) = (a+1, a)

(是不是很像lisp?函数的嵌套)

（不要疑惑我用λpair，这只是表示(a,a-1)将被插入到表达式体中），然后让我们对(0,0)应用这个表式n次，然后取出对的第二项，就是最终的结果。你会发现我这里耍了一个小聪明:(0,0）和(0,-1)是非不同的，所以对（a,a-1)不是一个很好的例子。尽管如此，我们还没有使用过负数，而且我们的NEXT-AIR函数会忽略对的第二项，所以这不影响结果。重复对(0,0)调用NEXT-PAIR会产生(1,0), (2, 1), (3,2), 4,3)...

P n :≡ (λn.n NEXT-PAIR(0, 0)) λxy.y

使用这个前继函数P，我们可以倒着数自然数了。 值得注意的是，当我们数到0，我们会停到0，这也是件好事。——因为我会把负数、除法、幂运算、超数留给读者当做一个练习（眼前一黑！）。（好，严格的说，我们可以在数字运算上再娱乐很多页纸，但不会对我们的基础理解有多少帮助了）

提问：我发现我们做减法，需要应用好多次前继函数。 而且每次，都要从0开始产生所有的中间数字有没有办法只使用一次前置函数？这样不是很影响性能么？

回答：谁在意性能？！λ演算只强调有效地计算所有可以被计算的，但它不承诺性能。而且，它是一数学概念，它可以在数学世界以极端的时间执行。）

（性能优化是那些计算机科学家和程序员的事情，认知科学家不必在意这些细节。）

逻辑

λ演算不仅仅局限于计算数字，它进行布尔运算也同样有效。还记得之前从对中取出第一项和第二项函数吗？ 这也是**真**和**假**的定义：

**TRUE:≡ λ xy.x

FALSE :≡ λ xy.y

布尔运算还有一系列的操作：与(AND) 或(OR) 非(NOT)，用来求取逻辑表达上的值。我们可以把这概念放进λ表达式。例如：我们可以这样定义求反函数（逻辑非）：

NOT :≡ λ a.a (λ bc.c) (λ de.d)

这是怎么工作的？如果我们写NOT TRUE（写出来是λ a.a (λ bc.c) (λ de.d)) λ xy.x）,第一个λ会把TRU放到表达式 (λ bc.c) (λ de.d) 前面，也就是FALSE TRUE。之前提到TRUE其实和从一对中取第一个成的函数是一样的，所以真假真，其实就是从（FALSE，TRUE）中取出第一个成员。

这里还有AND和OR，感兴趣你可以自己找出为什么他们是工作的：

AND :≡ λ ab.ab (λ xy.y)

OR :≡ λ ab.a (λ xy.x) b

条件

不可能仅仅从一个逻辑值计算另一个逻辑值。下面的函数告诉我们**如果...**。如果是0，返回真，如果是0, 返回假。这样的测试在写程序的过程中非常有用。

IS-ZERO **:≡ λ a.a FALSE NOT FALSE**

如果将IS-ZERO应用到n，我们得到

IS-ZERO n = n FALSE NOT FALSE

（注意λ表达式的自然数有个特性，就是将它自己应用n次）

这会对NOT应用n次FALSE，然后结果再应用于FALSE。每一次， 第一个FALSE(= λ xy.y) 直接删除后面的表达式。最后，n次后，得到NOT FALSE，擦掉NOT得到 FALSE。这样IS-ZERO的结果总是FALSE。（你如果觉得迷惑，自己动手写一下）。只有在n=0的情况下，我们对FALSE NOT FALSE应用0他会擦掉第一个表达式，于是NOT FALSE = TRUE,计算过程：

IS-ZERO 0

= (λ sz.z)* FALSE NOT FALSE*

= NOT FALSE

= TRUE

演算下1: IS-ZERO 1 = λ sz.s(z) FALSE NOT FALSE

= λz.FALSE(z) NOT FALSE

= FALSE(NOT FALSE) = FALSE

已知一个数字是0，我们可以找出另一个数字是不是大于等于0，我们可以这样描述：

GREATER-OR-EQUAL n **m :≡ λ n m. IS-ZERO(n P m)

换句话说，我们应用n次前继函数给m。如果n和m相等，结果是0。如果n比m大，结果是0。只有在比m小的情况下，结果大于0。

使用≥，我们可以确定相等性：n=m 如果m≥n且n≥m。我们可以这样描述：

EQUAL n m :≡ λ n m. AND (GREATER-OR-EQUAL(n m) GREATER-OR-EQUAL(m n))

= λ n m. AND (IS-ZERO(n P m) IS-ZERO(m P n))

更多...

正如你看到的，λ表达式是一种极简主义的编程语言。必然的，所有可能的编程语言都可以最终映射到表达式子上。事实上，优雅的lisp正是建立在λ演算的思想上，只是对语法稍微进行了修改，同时增加一些宏和数据类型。

我们这篇介绍是基于Raúl Rojas卓越的Tutorial Introduction to the Lambda Calculus，里面也讲了递归，而且更加学术，因为它是针对计算机科学的学生。另外，使用你在这篇文章中新发现的理解你可以开始进军更加学术的介绍了，好比维基百科上那篇。（维基百科上面的名词解释，对于小白们说太过于晦涩了。）

都是可以计算的

λ演算不可以完成全部的数学运算，因为有一部分数学问题是没有答案的（比如说两条线和一条线相，内角都是90度，则这两条线永不相交。之前大家都想要努力证明说，这个是成立的，但没有人可以明；更好玩的是，假设他们相交，所有的定理都还成立。），而且许多数学公式不可以被计算。那么图灵机和λ演算谁更强大？结果是，你总是可以将图灵机的脚本转换成为λ表达式（包括状态还有读写头），而且可以实现一个λ函数，就如同改变图灵机的状态一样。

反之，也可以将所有的λ表达式转换成为图灵机的纸带，然后构造一个图灵机去实现所有的剪切粘贴这样，可以证明λ演算和图灵机有着等同的功效。同样的，所有的电脑也具有这样的功效。包括个人脑、超级电脑、量子计算机甚至iphone（当然也包括小米）。唯一的区别是从实现上将，内存的大小和获得结果需要的步骤。所有计算机都拥有同样的基础能力被称作图灵测试。

在一定的精度内，通过描述个体神经元之间的联系的强度和，神经元的激励值，还有在极段时间内刺的传播，λ演算同样也可以被用于神经网络。理论上，任何管理信息的可行系统都是可以计算的，任可以用λ演算描述的可计算系统（或者可以用其他方式描述λ演算）都是电脑。

在学习函数式编程中有一个非常大的疑惑，就是函数式编程体系是不是可以完美表达面向对象的体系不产生副作用。学习完λ表达式，我看到纯函数可以模拟图灵机，也就是说纯函数可以模拟所有的面对象程序。

@梨梨喵

Lambda calculus引论