λ演算

λ演算是Church提出的一套形式系统，用以刻画可计算性。相比图灵机，它离物理层面更远，但离数学直觉更近。可以说，λ演算完全就是数学中函数(映射)的符号化，借助有限多步的「形式演算」来实现「计算」。正是由于此，λ演算活脱脱就是一门极简的编程语言——除了函数，别无其它。从λ演算开始揭示编程语言的奥秘，是个好的入口。

概要地说，λ演算的哲学观念如下：

(1) 所有对象皆函数，甚或不存在「数」的概念——因为所有「数」都被定义成特别的函数，比如用 $(x,y)\mapsto y$ 来代指自然数0。函数的参量当然也是函数，计算结果也是函数。在λ演算中「编程」，所做的无非两件事：定义一堆函数，而又将一部分函数「作用」在另一部分函数上。

(2) 函数皆匿名。所谓函数，追根结底就是自变量和因变量之间的绑定关系，或说映射关系。数学上 $f(x)=x^3$ 的写法常使人误认为「f」是函数，其实恰切地说应为「f所指的映射关系是函数」；亦即，$x\mapsto x^3$ 这样的绑定才是真正的函数，而「f」只不过是个名称罢了。λ演算

将函数匿名化，从而更加突出 $x \mapsto x^3$ 这样的映射关系。

以下我们遵循语法⟹语义的次序介绍λ演算。和学习逻辑时一样，在叙述语义前应视语法元素为无意义，但与此同时，我们心里早已设想好了应给它们赋予何种意义，只是由于得不到语义保障而徒为空想。唯有当语义定义清楚、研究完备了以后，那种一开始便期待的意义方才实现。

在叙述过程中，我们当然会把空想和期待陈述出来，以提供直观动机，你应当能够剥除识之而不致迷失。

def λ演算的字符集：

$$\Sigma := \{\lambda, (,), .\} \uplus V$$

其中 V 是一个可数的符号集，作用与一阶逻辑中的变元符号集类似。这里，为了讨论方便，我们取 $V = \{a, b, c, \cdots\}$ 就够用了。实际操作中，或许须将 V 取为无穷集。

remark. 因为λ演算的字符集中本身就带有「变元符号」，易与判断中的不定元弄混，因此，我们用蓝色字来标识元语言中的不定元。

def λ演算的语言.

判断形式 $term(t)$.

R 中包含如下规则：

(1) $$\frac{}{term(x)} \quad (\forall x \in V)$$ ← [这实际上是26条规则]

(2) $$\frac{term(t)}{term((\lambda x.t))} \quad (\forall x \in V)$$

(3) $$\frac{term(t) \quad term(t')}{term((t t'))}$$

它们归纳定义了λ演算的语言 Λ.

写成BN记号或许更简明：

$$
\begin{aligned}
t ::= \;& x && (\forall x \in V) \\
\mid\;& (\lambda x. t) && (\forall x \in V) \\
\mid\;& (t t)
\end{aligned}
$$

e.g. 以下字符串均 $\in \Lambda$.

① a

② $(\lambda x. x)$

③ $((\lambda x. (\lambda y. (xy))) z)$

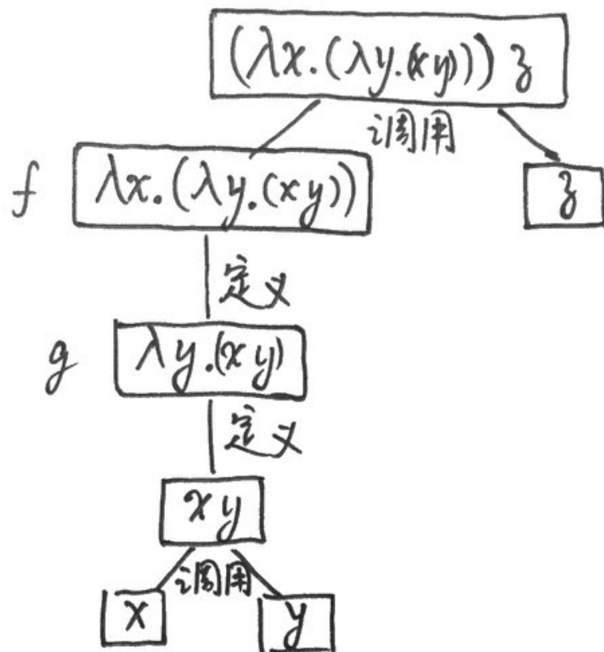
让我们重点关注后两条规则。

$\lambda x.t$ 希望表达的含义是：定义函数 $x \mapsto t$。
「λ」仅仅起到标记(提示)作用，而「.」分隔变元与函数体。

$t t'$ 希望表达的含义是：把 t' 作为参数传入 t 之中，亦即「调用」函数。

比如例②中，定义了函数 $x \mapsto x$。这是一个恒等函数。

例③则复杂许多。为了看清楚它在说什么，不妨按其构造方式画出一棵树：



从最上面往下说。③表达的是：把 z 作为参数传入函数 f 中，其中 $f: x \mapsto g$，而 $g: y \mapsto (xy)$。(xy) 的含义是把 y 作为参数传入 x 之中。这一例彰显了「一切皆函数」的内涵。

到这里，λ演算的语法就定义完了。这语法是没有歧义的，即，给定 Λ 中的字符串，其构造过程唯一。

Lemma 1

$\forall t \in \Lambda$，t的构造（生成）方式唯一。

proof. 关于 term(t) 的结构做归纳。

CASE 1°. $\overline{term(x)}$

因x只能取变元符号，而规则(2)(3)会引入变元符号以外的东西，故x只能由(1)生成。

CASE 2° $\dfrac{term(t)}{term((\lambda x.t))}$

因$(\lambda x.t)$以「(」及「λ」起头，故上一步不可能由规则(1)推得。若它由规则(3)推得，那么应呈现$((\cdots)\cdots)$或者$(x\cdots)$的样式，同样不匹配。是故，它只能由规则(2)推得。根据归纳假设，t的生成方式唯一，故$(\lambda x.t)$的生成方式唯一。

$\cdots \xrightarrow{\text{唯一}} t \longrightarrow (\lambda x.t)$

CASE 3° $\dfrac{term(t) \quad term(t')}{term((tt'))}$

与2°类似，略。 ∎

Lemma 1说明，每个Λ中的字符串都唯一地对应一棵「生成树」，正与一阶逻辑中每个wff对应一棵生成树相类。我们甚至可以这么说：Λ从一个角度看是字符串之集合，从另一个角度看是生成树之集合，二者等价。Benjamin C. Pierce在他的书中有一语中的的总结：在此处，字符串无非是树的扁平化，而括号的存在正是为了保持树结构。（说开去，wff、正则表达式全都如此。）

从这样的角度来理解，我们就可以按照人们惯常所为那样，去除一些「不必要」的括号以增强可读性——以不破坏树结构为限度。

约定

(1) 最外层括号可省略

(2) λ未加括号之处，作用域一直向右延伸

(3) $t\,t'\,t''$ 左结合，即相当于 $((t\,t')\,t'')$.

e.g.

① $(\lambda x.((\lambda y.y)z))$ 可写作 $\lambda x.\lambda y.yz$

[因为无括号约束，所以 λx 的作用域延伸至末尾。因为有括号约束，所以 λy 的作用域仅限于 z 以前]

② $(((\lambda x.(yx))(z(\lambda x.x)))u)$ 可写作

$(\lambda x.yx)(z\,\lambda x.x)u$

③ $\lambda x.xa$ 常被误以为是 $(\lambda x.x)a$，即把 a 作为参数传入恒等函数 $x\mapsto x$ 中。非也！约定(2)指出 λ 的作用域尽可能往右延伸，故 $\lambda x.xa$ 相当于 $(\lambda x.(xa))$，即 $x\mapsto(xa)$。切不要忘记「万物皆函数」的理念——x 这一参数也只接收函数，因而 (xa) 这样的写法不是无稽的。

在进入语义以前，我们先从程序员的视角来看看，λ演算的「程序」怎么写。

（作为程序员，并不需要弄清语义**如何被保证**，而只需了解语法和语义之对应即可。目前所讲内容对于程序员而言足够了。）

1° 多参数（多元）函数

如何表达诸如 $(x,y)\mapsto t$ 这样的多元函数呢？用「函数套函数」即可：$\lambda x.\lambda y.t$。为何奏效呢？让我们试着给它传参：

$(\lambda x.\lambda y.t)\,a\,b$

首先将 a 传入 $x\mapsto(\lambda y.t)$，得到一个函数 $\lambda y.t'$，其中 t' 是 t 传入 $x:=a$ 后得到的。接下来，再将 b 传入之[illegible]，得到 t''，其中 t'' 即将 t' 传入 $y:=b$。综合来看，a 和 b 分批传入，却达成了同时传入的效果。

2° 真假与分支

怎么能用函数来表达真与假呢？

定义 $True := \lambda f.\lambda g.f$ [接收两个参数，返回第一个]

$False := \lambda f.\lambda g.g$ [接收两个参数，返回第二个]

及 $Jump := \lambda b.\lambda p.\lambda q.bpq$

[接收一个真假值和两个参数，若 $b=True$ 则 p，否则，若 $b=False$ 则 q]

比方说 $Jump\ True\ (\lambda x.x)\ ((\lambda y.y)z)$

即 $True\ (\lambda x.x)((\lambda y.y)z)$

即 $\lambda f.\lambda g.f\ (\lambda x.x)((\lambda y.y)z)$

即 $(\lambda x.x)$

作为思考题，试表达与、或、非三种运算.

3° 自然数

自然数之定义方式有很多种，我们采用Church的定义。

$0 := \lambda s.\lambda z.z$

$1 := \lambda s.\lambda z.sz$

$2 := \lambda s.\lambda z.s(sz)$

......

而加法函数定义为

$Add := \lambda x.\lambda y.\lambda s.\lambda z.xs(ysz)$

大致的思想是：把「数」y 作为「数」x 的零点，相当于作了一次偏移。

乘法函数等也可定义，在此不表。

你也许留意到 0 与 False 实为同一个函数，这种现象在C++之中很常见。它也表明，λ演算中缺乏「类型」的概念，所有函数都无差别对待。

我们当然可以往 Jump 中传入非 True/False 的别的函数，只不过那样的话，结果将不受控制。这正是人们开发「类型系统」的主要动机。

接下来，我们从语言设计者的视角来看看，λ演算的「程序」究竟是怎样一步步「演算」的。我们将用严格的数学定义来明确其行为，就好比我们定义图灵机是如何执行的一样。

这里出现了一对矛盾：一方面，用自然语言来表述数学概念比起用归纳定义而言更符合直观；另一方面，后者又比前者精确，能帮我们规避「想当然」的错误。因此以下讨论一式两份，兼容并蓄。

来看一个引例。我们有一个程序

$$(\lambda x.\lambda y.xx\lambda x.x)\ 3\ 5$$

（x 与 xx 绑定；x 与 x 绑定）　如前所述，它们也是函数。

应如何运行之？首先，应当把「3」代入被调用的函数 $(\lambda x.\lambda y.xx\lambda x.x)$ 中，取代占位符 x，得到

$$\rightarrow (\lambda y.33\lambda x.x)5$$

接下来再将「5」代入被调用的函数 $(\lambda y.33\lambda x.x)$ 中，取代占位符 y.

$$\rightarrow 33\lambda x.x.$$

（还可继续，此处略过）

可以说，程序运行的过程就是不断「调用」或说「以参数取代变元符号」的过程。此间有个问题：所谓「取代」，并非见到相同的符号就一股脑拿掉，而应有所区别。唯有与λ绑定在一起的方可取代，比如第一步不能弄成 $\rightarrow \lambda y.33\lambda x.3$ 甚至 $\lambda y.333$，否则就把内部恒等函数 $(\lambda x.x)$ 的语义都改掉了。为了解决这一问题，我们才要定义「自由变元」的概念。然后，在它基础上，定义正确的「取代」概念。最后，借「取代」的概念定义程序「运行」的概念。

def 自由变元.

判断形式: $v \in FV(t)$

规则:

$$\frac{}{v\in FV(v)}\quad (\forall v\in V)$$

$$\frac{v\in FV(t)\quad term(t)}{v\in FV(\lambda x.t)}\quad (\forall v\neq x\in V)$$

$$\frac{v\in FV(t)\quad term(t)\quad term(t')}{v\in FV(tt')}\quad (\forall v\in V)$$

$$\frac{v\in FV(t')\quad term(t)\quad term(t')}{v\in FV(tt')}\quad (\forall v\in V)$$

用自然语言简述，即

$$\begin{cases} FV(v) := \{v\} \\ FV(\lambda x.t) := FV(t) - \{x\} \\ FV(tt') := FV(t)\cup FV(t') \end{cases}$$

Problem
仿此定义判断形式 $v\notin FV(t)$

当我们运行 $(\lambda x.t)t'$ 时，只希望用 t' 取代掉 t 中自由的 x。非但如此，我们还不希望 t' 中的自由变元被某些 λv 捉住，形成不该有的绑定。若被捉住怎么办呢？可以将 λv 及其相关的 v 先行改名为 $v'\notin FV(t')$。

def 取代.

判断形式: $t[x/t']=t''$

规则:

① $$\frac{}{v[v/t']=t'}\quad (v\in V)$$

② $$\frac{}{v[x/t']=v}\quad (v\neq x\in V)$$

③ $$\frac{}{(\lambda x.t)[x/t']=(\lambda x.t)}\quad (x\in V)$$

④ $$\frac{v'\notin FV(t')\quad t[v/v']=t^*\quad t^*[x/t']=t''}{(\lambda v.t)[x/t']=\lambda v'.t''}\quad (v\neq x\in V)$$

⑤ $$\frac{t_1[x/t']=t_1'\quad t_2[x/t']=t_2'}{(t_1t_2)[x/t']=(t_1't_2')}\quad (x\in V)$$

用自然语言简述:

①在变元符号 v 中，将 v 用 t' 取代，得 t'（直接替换）

②在变元符号 v 中，将 x 用 t' 取代，得 v（无可替换）

③在 $(\lambda x.t)$ 中，希望用 t' 代 x，却由于缺乏自由的 x 而无法成行，结果和原来一样。（无自由变元可换）

④ 在$(\lambda v.t)$中，欲以t'代x。因为要避免FV(t')里的变元被λv逮住，所以我们干脆把v先行改名为$v' \notin FV(t')$，得到$(\lambda v'.t^*)$，再做我们想干的事，深入t^*之中以t'代x。（规避捕捉）

⑤ 在$(t_1 t_2)$中，欲以t'代x，那么只需分开两边各自取代即可。（分别取代）

remark. 规则④将造成同一个式子有多种取代方式，因为v'通常有多种选择。但没关系，无论选哪种，意思都一样。（这唤作「α-重命名」。两个式子的差别如果仅在于约束变元名称，那么称其α等价。你可以尝试用归纳定义来刻画α等价，但它过于繁琐，在此不述。）

现在，我们可以定义「单步运行」了，这与图灵机的格局转换关系类似。

def 单步运行

判断形式 $t_1 \to t_2$ （即t_1可单步转移至t_2）

规则：

① $$\frac{term(t) \quad term(t') \quad t[x/\lambda y.t'] = t''}{(\lambda x.t)(\lambda y.t') \to t''}$$

② $$\frac{term(t) \quad t' \to t''}{(\lambda x.t)t' \to (\lambda x.t)t''}$$

③ $$\frac{t \to t''}{t t' \to t'' t'}$$

用自然语言简述：

① 当程序呈现「左右两半均为$(\lambda \cdots . \cdots)$」的格式时，即可把右半边作为参数传入左半边的函数中去。

② 当程序的左半边已准备好但右半边尚未准备好，则去「化简」右半边。

③ 当程序的左半边还未准备好（即还不成其为$(\lambda \cdots . \cdots)$的格式），则去「化简」左半边。

一言以蔽之，左半边长得不像$x \mapsto t$的模样就先去把它整成这模样，然后再

把右半边整成这模样。两端皆就绪后即可「代入」。

那你要问，若程序打一开始就没分两半（即长成 $(\lambda x. \cdots)$），那怎样执行？答案是：执行完了——毕竟，你无论如何也找不出 t：$(\lambda x. \cdots) \to t$。

（这是理解的关键点：$\to$ 只是一个二元关系，不是映射关系，也不是全关系。）

你也许还要质疑刚才用自然语言给的解释与归纳定义对不上号。下面的引理将使你信服。

Lemma 2 $\to$ 的定义（构造）路径唯一。
即 $\forall$ ~~...~~ $t_1 \to t_2$，由①②③生成它的方式唯一。（类比 Lemma 1）

proof. 对 $\to$ 的结构做归纳。

CASE① $(\lambda x.t)(\lambda y.t') \to t''$ 若能由②生成，则意味着 $(\lambda y.t') \to t^*$ 对某个 t^* 成立，但这是无法做到的。同理，它也无法由③生成。因此，它只能由①生成。

CASE② $(\lambda x.t)t' \to (\lambda x.t)t''$ 若能由① ~~...~~ 推得，则意味着 $t' = (\lambda y.t^*)$。又据 ~~...~~ 假设有 $t' \to t''$，但这是不可能的。类似地，它也不能由③推得。故它只能由②推得。据归纳假设，此前生成路径唯一，故整条生成路径唯一。

CASE③ 类似②. ■

该引理直接提示我们有以下快速分析算法：

Algorithm OneStep(t_1)
  // 输入：λ演算「程序」
  // 输出：某个 t_2 满足 $t_1 \to t_2$。若不存在则输出 ~~ERROR~~ HALT.

1° 若 $t_1 = (\lambda x. \cdots)$ 或 x 即完整的「一块」，则 ~~...~~ return HALT

2° 将 t_1 切分成 l 与 r 两部分.

```
3° if l≠(λx.…) then
     // Case③
     l' := OneStep(l)
     if l'=HALT then return HALT
     else return l'r
   else if r≠(λx.…) then
     // Case②
     r' := OneStep(r)
     if r'=HALT then return HALT
     else return l r'
   else
     return 把r代入l后所得.
```

Theorem 3

若 $t_1 \to t_2$ 且 $t_1 \to t_2'$，则 t_2 与 t_2' α等价。

proof. 习题. ∎

该定理说明：虽然 t 的单步运行「结果」不唯一，但在 α 等价意义下是唯一的。

在单步运行的基础上，我们可以定义多步运行。

def 多步运行.

判断形式 $t_1 \xrightarrow{*} t_2$

规则 $\dfrac{term(t)}{t \xrightarrow{*} t}$ $\dfrac{t_1 \xrightarrow{*} t_2 \quad t_2 \to t_3}{t_1 \xrightarrow{*} t_3}$

换言之，$\xrightarrow{*}$ 是 $\to$ 的自反、传递闭包。

def 终止.

设 t 是一个 λ 演算程序。如果存在 t^* 使得 $t \xrightarrow{*} t^*$ 而且 t^* 无法进一步单步运行，那么称 t 可终止，t^* 为终态。

并非所有λ演算程序皆能终止。例如 $(\lambda x.xx)(\lambda x.xx)$ 将永远陷入死循环。

即便一个程序能终止，其终态也未必是函数。例如 $(\lambda x.xx)(y\ \lambda z.z)$，由于 y 是个变元，故无从使用 → 规则进一步执行，程序在 y 处「卡住了」。

在下一章我们会谈到类型系统，其一大功用即确保程序的终止，以及终止

时不发生「卡住」这类的异常。

本章最后，让我们探索一下λ演算的能量。先前，我们已展示了如何用它完成真值及自然数的编码，也了解了基本的算数。现在，我们来考虑「递归」。

假设让你用编程语言来实现阶乘函数，最简单明了的写法是这样：

$$\text{Fact}(n) := \text{if } n=0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1)$$

最关键的特性在于「自引用」：定义fact时引用了fact自己。λ演算显然没法直接做到这点。但是，我们可以投机取巧，绕道而行。如果把「自己」抽象成一个参数，传递到函数体中去，不就能实现「自引用」了吗？

$$\text{Helper} := \lambda f.\ \lambda n.\ \text{Jump}\ (\text{IsZero}\ n)\ 1\ \ n \times (f\ f\ (n-1))$$

（λf ← 准备接收自己）

$$\text{Fact} := \lambda n.\ \text{Helper Helper}\ n$$

定义Helper时，我们并未用到Helper自己。它就像别的普通函数那样，接收两个参数f与n，然后展开一系列操作。可是，我们背地里盘算着令参数 $f := \text{Helper}$，以便Helper「递归」时永远记着自己是谁。就这样，Helper被骗入了陷阱，间接地实现了自引用。

当然，类似的技巧可用于任何涉及递归调用的场合。你可以利用Python的lambda表达式来实验之。（别用诸如ML这样的静态强类型语言。本课程结束以后你就会知道为什么。）

有递归调用作为工具，λ演算之图灵完备性就易证了。下面给出概要。

def λ演算可计算

设 $f: \mathbb{N}^k \to \mathbb{N}$，若存在一个λ演算程序t，使得 $\forall n \in \mathbb{N}^k$，将n的数码输入t以后，程序总能终止且终态为 $f(n)$ 的数码，

那么称 f 是λ演算可计算的。

Theorem 4

设 $f: \mathbb{N}^k \to \mathbb{N}$。$f$ 是图灵可计算的，当且仅当 f 是λ演算可计算的。

proof.

(⇐) 前面已介绍过单步运行λ演算的算法，因此显然图灵机可以模拟λ演算的行为。

(⇒) 由于 f 是图灵可计算的 ⇔ f 是递归函数（见递归论），所以只需证 f 是递归函数 ⇒ f 是λ演算可计算的。

1° Z, $Succ$, π_n^i 显然都是λ演算可计算的。

2° λ可计算类对复合操作封闭。(显然)

3° λ可计算类对原始递归操作封闭。

[只需用递归技巧去模拟
$h(x_1, \dots, x_k, x_{k+1}+1) := g(x_1, \dots, x_{k+1}, h(x_1, \dots, x_{k+1}))$]

4° λ可计算类对极小正则化操作封闭。

[只需用递归技巧由小往大搜索。]

综合1°-4°点即得证。■