让我们谈谈 λ 演算

王盛颐
txyyss@gmail.com

缘起

“写篇介绍 λ 演算的文章” 这一条目在我的待办事项列表里已经躺了很久了。我本科看 “计算机程序的构造和解释”(SICP)一书 [1] 时,接触到了 λ 表达式这个概念。当时我觉得它就是一种表示匿名函数的记法。有了这个记法,把函数作为值来传递、返回以及组合都方便了很多,别的也没多想。

SICP 里面用的 Scheme 语言让我见识到了世界上还有一类叫 “函数式” 的程序设计语言。搜索相关资料,都说 λ 演算是函数式语言的基础。好奇心害死猫啊,我自然是要看看 λ 演算是怎么一回事的。一番了解之后,我觉得如此形式简单而内涵丰富的东西,就是美的体现啊。尤其当我看到 Church 数的时候,那种 “啊哈” 的惊艳感真是永生难忘。

后来我学了越来越多的关于计算机,特别是程序设计语言的理论,发现 λ 演算从可计算性理论到形式语义到类型理论,无处不在,自己目前所知不过皮毛罢了。刚好我最近因为学习 Haskell 语言,实现了一个简单的无类型纯 λ 演算解释器。于是我决定写篇初步介绍无类型纯 λ 演算的文章,有个解释器的好处是能一边介绍一边通过解释器演示规约求值的过程,让相对抽象的 λ 演算更加直观和易于理解。

1 历览前贤

一切要从上个世纪初,也就是 1900 年左右开始说起了。那时候数学界的气象是这样的:希尔伯特刚刚提出他的 23 个问题,其中第 2 个问题问的是公理系统的相容性;1901 年提出的,动摇了集合论基础的罗素悖论还是个很流行的话题;至于哥德尔不完备性定理呢,大家都还不知道。所以在这之后的几十年很多数学家和逻辑学家都在致力于给整个数学建立一个一致的公理基础。比如罗素和怀特海德写了 “数学原理”(Principia Mathematica[2]);比如约翰·冯·诺伊曼 [1]写了关于公理化集合论的博士论文 [3];再比如阿隆佐·邱奇(Alonzo Church)提出了 λ 演算 [4]。

邱奇是个美国逻辑学家,生于 1903 年。他在 1928 年开始构造的一个形式系统中包含了纯 λ 演算。他发明这一形式系统的初衷是为了给逻辑学提供一个基础,能代替罗素的类型理论和恩斯特·策梅洛(Ernst Zermelo)[2]的集合理论。这个系统 1932 年发表后不久就被发现有矛盾,于是一年后邱奇修正了一番重新发表。当时的人对不完备性定理威力有多大还缺乏清醒广泛的认识。所以他还希望那时发现不久的哥德尔关于 “数学原理” 一书的不完备性定理不会扩展到他的系统上。

愿望是良好的,结果是残酷的。到 1935 年,邱奇的两个学生,Stephen Kleene 和 Barkley Rosser 发现邱奇的逻辑系统是不一致的。不过柳暗花明又一村,他们发现系统包含的纯 λ 演算则具有一系列良好的性质,再后来更是证明用 λ 演算可以等价的定义出可计算函数,邱奇觉得能有效计算的函数等价于 λ 可定义性,这就是著名的 “邱奇 –图灵论题” 的一部分了。更一进步的,邱奇用 λ 演算证明了一

[1]对,你没有看错,他就是那个定下了计算机基本结构 “冯·诺伊曼结构” 的那个冯·诺伊曼。
[2]他就是 ZFC 公理化集合论的那个 “Z”。

阶逻辑不存在递归判定过程,这是对希尔伯特提出的判定性问题(Entscheidungsproblem)的第一个否定性答案,这比用图灵证明停机问题不可判定还要早几个月。

不过二十世纪三十年代在 λ 演算方面的成果差不多也就这些了,再接下来的 20 年都没有太多研究和进展。直到六十年代,那时有了计算机,有了程序设计语言,有了计算机科学家。在 1965 年,英国计算机科学家 Peter Landin 发现可以通过把复杂的程序语言转化成简单的 λ 演算,来理解程序语言的行为。这个洞见可以让我们把 λ 演算本身看成一种程序设计语言。而众所周知的 John McCarthy 的 Lisp 语言,更是让 λ 演算广为传播。现在不论是各种实际的程序设计语言还是理论上的研究工作,λ 演算都是一个绕不过去的基本工具了。

λ 演算之所以这么重要,用 Benjamin C. Pierce 的话说在于它具有某种 "二象性":它既可以被看作一种简单的程序设计语言,用于描述计算过程,也可以被看作一个数学对象,用于推导证明一些命题。在这篇介绍文章里,我也打算从两个方面来讲,先讲它作为数学理论这方面的内容。

2 数学理论

这一节的内容主要参考了这本书 [5]。让我们先从数学开始说起。函数是数学中一个非常基本的概念,我们很容易就能写出一个计算平方和的函数如下:

$$f(x,y) = x \times x + y \times y. \tag{1}$$

上面这个平方和函数有个名字叫 f,有名字的好处是能方便的表示后续的计算,比如:

$$f(3,4) = 3 \times 3 + 4 \times 4 = 25.$$

但再想想,名字对一个函数来讲是必须的么?当然不是,下面这个映射也表示了平方和函数:

$$(x,y) \mapsto x \times x + y \times y \tag{2}$$

把 (2) 整体当成函数的名字,同样可以表示计算平方和过程:

$$((x,y) \mapsto x \times x + y \times y)(3,4) = 3 \times 3 + 4 \times 4 = 25.$$

我们把 (2) 称为匿名函数,它有两个参数。那么再问,单参数函数和多参数函数的区分有必要吗?可以换个角度来看,我们把 (2) 改写成下面这个样子:

$$x \mapsto (y \mapsto x \times x + y \times y). \tag{3}$$

这个映射是什么意思呢?x 被映射成了 $y \mapsto x \times x + y \times y$,后者是一个以 y 为参数的函数。换句话说,(3) 表示的是把一个数映射成函数的函数,这就是所谓的 "高阶函数"(High-Order Function)了。注意,(3) 现在是个单参数的函数,我们可以把它先应用于参数 3,得到一个新的函数:

$$(x \mapsto (y \mapsto x \times x + y \times y))(3) = y \mapsto 3 \times 3 + y \times y = y \mapsto 9 + y \times y.$$

再把这个函数应用于参数 4:

$$(y \mapsto 9 + y \times y)(4) = 9 + 4 \times 4 = 25.$$

这样就清楚了,(3) 同样可以进行平方和的计算。用类似的方法,可以把任意多参数函数都转换成单参数的高阶函数,这个转换又叫做柯里化(Currying),这是以数学家 Haskell Brooks Curry 命名的。

匿名函数和柯里化,是 λ 演算为简化函数概念而采取的方法,上面这段说明可以看作是 λ 演算的非正式介绍。事实上,大多数程序语言里的所谓 Lambda 表达式,也就是这两个概念组合在一起罢了。下面正式介绍严格的,形式化的 λ 演算。

2.1 形式化定义

定义 1. **(λ 项)** 假设我们有一个无穷的字符串集合，里面的元素被称为变量(和程序语言中变量概念不同，这里就是指字符串本身)。那么 λ 项定义如下：

1. 所有的变量都是 λ 项(名为原子)；
2. 若 M 和 N 是 λ 项，那么 (MN) 也是 λ 项(名为应用)
3. 若 M 是 λ 项而 ϕ 是一个变量，那么 $(\lambda\phi.M)$ 也是 λ 项(名为抽象)。

上面定义的变量集合是可以任意指定的，比如在我写的解释器(见附录 A)中，这个集合就是一般程序语言中可以用作变量名的字符串[3]。在这篇文章里我们规定，变量集合是所有的小写英文字母及形如 x', x'', x''' ... 的字符串。

示例 1. **(一些 λ 项)** 下面这些都是 λ 项：

$$(\lambda x.(x\,y)) \qquad (x(\lambda x.(\lambda x.x))) \qquad ((((a\,b)\,c)\,d)\,e)$$
$$(((\lambda x.(\lambda y.(y\,x)))\,a)\,b) \qquad ((\lambda y.y)\,(\lambda x.(x\,y))) \qquad (\lambda x.(y\,z))$$

λ 项是一种形式语言，换句话说，就是一类特殊形式的字符串罢了，没有任何内在的意义，只是个“形式”。通常情况下，当讨论一个形式语言的时候，我们需要用另一种元语言来指称形式语言里的元素。就如讨论自然数时，我们经常说“对任意自然数 n”，这里的 n 本身并不是自然数，用来指称自然数罢了。我们也需要一些“n”来表示 λ 项中的元素。因此，我们作如下符号约定：

符号约定 1. 本文中我们用大写英文字母表示任意 λ 项，用除 λ 以外的小写希腊字母如 ϕ, ψ 等表示任意 λ 项中的变量。

对于括号，则有如下的省略规定：

1. λ 项中最外层的括号可以省略，如 $(\lambda x.x)$ 可以省略表示为 $\lambda x.x$；
2. 左结合的应用型的 λ 项，如 $(((M\,N)\,P)\,Q)$，括号可以省略，表示为 $M\,N\,P\,Q$；
3. 抽象型的 λ 项 $(\lambda\phi.M)$ 中，M 最外层的括号可以省略，如 $\lambda x.(y\,z)$ 可以省略为 $\lambda x.y\,z$。

也就是说，我们把省略形式视同定义 1 中的 λ 项。

示例 2. **(省略表示)** 下面给出了一些省略表示的 λ 项。

省略表示	完整的 λ 项
$\lambda x.\lambda y.y\,x\,a\,b$	$(\lambda x.(\lambda y.(((y\,x)\,a)\,b)))$
$(\lambda x.\lambda y.y\,x)\,a\,b$	$(((\lambda x.(\lambda y.(y\,x)))\,a)\,b)$
$\lambda g.(\lambda x.g\,(x\,x))\,\lambda x.g\,(x\,x)$	$(\lambda g.((\lambda x.(g\,(x\,x)))\,(\lambda x.(g\,(x\,x)))))$
$\lambda x.\lambda y.a\,b\,\lambda z.z$	$(\lambda x.(\lambda y.((a\,b)\,(\lambda z.z))))$

示例 3. 用 λ 演算解释器可以方便的查看一个 λ 项的省略表示和完整表示。解释器用字符 \ 代表 λ。(更多关于这个解释器的用法和各个命令的意义，请参考附录 A。)

```
Lambda> :set +hold
Lambda> (((\x.(\y.(y x))) a) b)
(\x.\y.y x) a b
Lambda> :set +fullform
Lambda> (\x.\y.y x) a b
(((\x.(\y.(y x))) a) b)
```

[3]就是以英文字母开头，后面跟着有限多个字母或数字的字符串。

2.2 替换

只定义了一个形式语言，那是没什么用处的。这一小节将形式化的定义什么是 λ 项的替换操作。直观的来讲，就是把 λ 项中不被 $\lambda\phi$ 约束的变量 ϕ 替换成另一个 λ 项罢了。但这么一个操作的精确定义却不是直接和易于理解的。我觉得这块的定义因严谨而优美，但有的人看来可能就是琐碎而无聊了。没有兴趣的读者可以跳过这一小节。只要记住 λ 项的替换，是有精确严格定义的就行，反正具体计算可以让解释器代劳嘛。

定义 2.（语法全等） 我们用恒等号 "$\equiv$" 表示两个 λ 项完全相同。换句话说

$$M \equiv N$$

表示 M 和 N 有完全相同的结构，且对应位置上的变量也完全相同。这意味着若 $MN \equiv PQ$ 则 $M \equiv P$ 且 $N \equiv Q$，若 $\lambda\phi.M \equiv \lambda\psi.P$ 则 $\phi \equiv \psi$ 且 $M \equiv P$。

定义 3.（自由变量） 对一个 λ 项 P，我们可以定义 P 中自由变量的集合 $\mathrm{FV}(P)$ 如下：

1. $\mathrm{FV}(\phi) = \{\phi\}$
2. $\mathrm{FV}(\lambda\phi.M) = \mathrm{FV}(M) \setminus \{\phi\}$
3. $\mathrm{FV}(MN) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$

从第 2 可以看出抽象 $\lambda\phi.M$ 中的变量 ϕ 是要从 M 中被排除出自由变量这个集合的。若 M 中有 ϕ，我们可以说它是被约束的。据此可以进一步定义约束变量集合。值得注意的是，对同一个 λ 项来说，这两个集合的交集未必为空。

示例 4.（自由变量）

λ 项 P	自由变量集合 $\mathrm{FV}(P)$
$\lambda x.\lambda y.x\,y\,a\,b$	$\{a, b\}$
$a\,b\,c\,d$	$\{a, b, c, d\}$
$x\,y\,\lambda y.\lambda x.x$	$\{x, y\}$

上面最后一个例子里，最左边的 x, y 是自由变量，而最右侧的 x 则是约束变量。若对 λ 项 P 有 $\mathrm{FV}(P) = \varnothing$，则称 P 是封闭的，这样的 P 又称为组合子。

定义 4.（出现） 对于 λ 项 P 和 Q，可以定义一个二元关系出现。我们说 P 出现在 Q 中，是这样定义的：

1. P 出现在 P 中；
2. 若 P 出现在 M 中或 N 中，则 P 出现在 (MN) 中；
3. 若 P 出现在 M 中或 $P \equiv \phi$，则 P 出现在 $(\lambda\phi.M)$ 中。

有了上面这些定义，我们终于可以定义什么叫 λ 项的替换操作了：

定义 5.（替换） 对任意 M, N, ϕ，定义 $[N/\phi]M$ 是把 M 中出现的自由变量 ϕ 替换成 N 后得到的结果，这个替换有可能改变部分约束变量名称以避免冲突。具体精确定义是一个对 M 的归纳定义：

1. $[N/\phi]\,\phi \equiv N$
2. $[N/\phi]\,\alpha \equiv \alpha$ 对所有满足 $\alpha \not\equiv \phi$ 的原子 α
3. $[N/\phi]\,(PQ) \equiv ([N/\phi]\,P\ [N/\phi]\,Q)$
4. $[N/\phi]\,(\lambda\phi.P) \equiv \lambda\phi.P$
5. $[N/\phi]\,(\lambda\psi.P) \equiv \lambda\psi.P$ 若 $\phi \notin \mathrm{FV}(P)$

6. $[N/\phi]\,(\lambda\psi.P) \equiv \lambda\psi.[N/\phi]\,P$ 若 $\phi \in \mathrm{FV}(P)$ 且 $\psi \notin \mathrm{FV}(N)$
7. $[N/\phi]\,(\lambda\psi.P) \equiv \lambda\eta.[N/\phi][\eta/\psi]\,P$ 若 $\phi \in \mathrm{FV}(P)$ 且 $\psi \in \mathrm{FV}(N)$

对其中从 5 到 7 的各条来说，$\phi \neq \psi$；而对 7，η 是满足 $\eta \notin \mathrm{FV}(N\,P)$ 的任意变量。

下面解释一下这个看上去很长很恐怖的定义，其实只要带着“替换 λ 项中的自由变量 ϕ 为 N”这个直观去看，就不难理解。前两条无非是说，要是一个原子刚好是要被替换的变量，那就替换，不然就不动。第 3 条也好说，遇到应用了，那就替换各子项。第 4 条，P 中的 ϕ 是被约束的，不能替换所以结果不变。同样的情况发生在第 5 条，P 中没有自由变量 ϕ，结果也不变。顺着这个思路往下想，P 中要是有自由变量 ϕ，是不是就可以替换了呢？是的，这就是第 6 条的内容了。不过这条还有个附加条件，$\psi \notin \mathrm{FV}(N)$，为什么呢？因为 P 是被 $\lambda\psi$ 约束着的，把 N 替换进去的时候，要是 $\mathrm{FV}(N)$ 里面有 ψ，那替换进去的 N 中的 ψ 就从自由变量变成约束变量了。这多少有点让人不放心，所以我们先把这个条件加上，这样替换不会把自由变量变成约束变量，这就是第 6 了。但要是不如意事长八九，N 中偏偏有 ψ 怎么办呢？为了避免这个冲突，干脆把原先约束的变量先换成绝对不会起冲突的吧，找一个 $\eta \notin \mathrm{FV}(N\,P)$ 换掉 ψ，然后再按第 6 条来办，这就是第 7 的内容了。特别注意这一条中，不仅把 P 中的 ψ 换成了 η，连最前面的 $\lambda\psi$ 都被换成了 $\lambda\eta$。

上面的解释只是为了让读者理解一下定义 5 的合理性，其实还是有一些没有解释清楚的疑问，比如为什么从自由变量变成约束变量就不好了？第 7 条中 η 有无数种选择，这种不确定性会不会有什么问题？这些问题的答案是：没什么为什么，定义如此。定义不是定理，它说了算，就算有它内蕴的合理性，也并不需要在这个时候证明。

定义 5 的第 7 条里，替换的不确定性暗示了可以任意替换约束变量而不改变“意义”。这个可以类比的理解，例如定积分

$$\int_a^b f(x)\mathrm{d}x$$

里的 x 是所谓的哑变量，换成别的如 $f(t)\mathrm{d}t$ 不会改变积分式的值。又比如正弦函数是写成 $\sin x$ 还是 $\sin t$ 都不影响它的意义。不过这只是我为了能直观理解而加上的一种解释。实际上 λ 项是形式系统，只是形式，一堆字符串罢了，哪里有什么“意义”。

可以说为了补救任意替换约束变量给人带来的不安，我们有了下面的定义：

定义 6.（**α 变换和 α 等价**）设 $\lambda\phi.M$ 出现在一个 λ 项 P 中，且设 $\psi \notin \mathrm{FV}(M)$，那么把 $\lambda\phi.M$ 替换成

$$\lambda\psi.[\psi/\phi]\,M$$

的操作被称为 P 的 α 变换。当且仅当若 P 经过有限步（包括零步）α 变换后，得到新的 λ 项 Q，则我们可以称 P 与 Q 是 α 等价的，又写作

$$P \quad \equiv_\alpha \quad Q$$

示例 5.

$$\lambda x.\lambda y.x(x\,y) \equiv_\alpha \lambda x.\lambda v.x(x\,v) \equiv_\alpha \lambda u.\lambda v.u(u\,v)$$

容易证明，$\equiv_\alpha$ 满足自反性、对称性和传递性，所以确实是等价关系。有了 α 等价，定义 5 就显得更加合理了，第 7 带来的疑问也得到解答了：不是担心任意替换不妥当么？那么我们可以把 α 等价的 λ 项都看成相同的，这不就“补救”了由于不确定替换带来的问题嘛。事实上，有更好的结果：

定理 1. *若 $M \equiv_\alpha M'$ 且 $N \equiv_\alpha N'$，则 $[N/x]\,M \equiv_\alpha [N'/x]\,M'$*

强调一下，这个小节里定义的 $[N/\phi]\,M$ 并不是 λ 项这个形式语言里的东西，它只是一种记号，用来指称替换后的 λ 项而已，切记切记。而我各种解释说明中的意义和类比都只是为了方便直观的理解，并不是说 λ 项有这些“意义”。它始终只是“名”，和“实”不相关。作来体现。

2.3 规约

现在我们有了 λ 项，有了替换的规则，那么什么时候需要进行替换呢？这就是本小节要讨论的话题：规约。有了规约，整个关于 λ 项的形式系统才算完整，才可以说是 λ 演算。

先介绍 β 规约，β 规约可以这么直观的理解：我们可以把 $(\lambda\phi.M)$ 看成是参数为 ϕ，函数体为 M 的一个函数；把 $(M\,N)$ 看成是函数 M 作用到实际参数 N 上[4]。平时我们要是定义了函数 $f(x) = x + 5$，那么函数应用 $f(6)$ 就是把 $x + 5$ 中的 x 替换成 6，得到 $f(6) = 6 + 5 = 11$。替换是函数应用的实质啊。

定义 7.（β **规约**）形如

$$(\lambda\phi.M)\,N$$

的 λ 项被称为 β *可约式*，对应的项

$$[N/\phi]\,M$$

则称为 β *缩减项*。当 P 中含有 $(\lambda\phi.M)\,N$ 时，我们可以把 P 中的 $(\lambda\phi.M)\,N$ 整体替换成 $[N/\phi]\,M$，用 R 指称替换后的得到的项，那么我们说 P 被 β 缩减为 R，写做：

$$P \rhd_{1\beta} R$$

当 P 经过有限步（包括零步）的 β 缩减后得到 Q，则称 P 被 β *规约*到 Q，写做：

$$P \rhd_{\beta} Q$$

示例 6.（β **缩减**）

$$\begin{array}{rlll}
(\lambda x.x\,(x\,y))\,m & \rhd_{1\beta} & m(m\,y) & \\
(\lambda x.y)\,n & \rhd_{1\beta} & y & \\
(\lambda x.(\lambda y.y\,x)z)\,v & \rhd_{1\beta} & [v/x]\,((\lambda y.y\,x)z) & \equiv \quad (\lambda y.y\,v)z \\
 & \rhd_{1\beta} & [z/y]\,(y\,v) & \equiv \quad z\,v \\
(\lambda x.x\,x)\,(\lambda x.x\,x) & \rhd_{1\beta} & [(\lambda x.x\,x)/x]\,(x\,x) & \equiv \quad (\lambda x.x\,x)\,(\lambda x.x\,x) \\
 & \rhd_{1\beta} & [(\lambda x.x\,x)/x]\,(x\,x) & \equiv \quad (\lambda x.x\,x)\,(\lambda x.x\,x) \\
 & \cdots & \text{etc.} & \\
(\lambda x.x\,x\,y)\,(\lambda x.x\,x\,y) & \rhd_{1\beta} & (\lambda x.x\,x\,y)\,(\lambda x.x\,x\,y)\,y & \\
 & \rhd_{1\beta} & (\lambda x.x\,x\,y)\,(\lambda x.x\,x\,y)\,y\,y & \\
 & \cdots & \text{etc.} &
\end{array}$$

从上面的例子可以看出，β 缩减虽然名为缩减，但实际情况是复杂的。像最后这两个例子，一个永远缩减为自身，一个甚至更糟糕，越来越长，非但不是缩减反而是增长了。

定义 8.（β **范式**）若一个 λ 项 Q 不含有 β 可约式，则称 Q 为 β *范式*。若有 P 可被 β 规约到 Q，则称 Q 是 P 的 β 范式。

从示例 6 可以看出，并不是所有的 λ 项都能 β 规约到 β 范式。这其实还不是最让人担心的，有一个之前读者也许没注意到的问题：定义 7 里关于 β 规约的定义非常“狡猾”，或者叫非常含糊，它只要求存在一个有限长的 β 缩减链就行，当 λ 项中含有不止一个 β 可约式的时候，可以有不同顺序的缩减方式。

[4]还是再强调一下，这只是直观的理解，并不是说两种 λ 项一个是函数，一个是函数应用。λ 项不是这些意义，只是字符串。

示例 7. 让我们重新看一下示例 6 里的 $(\lambda x.(\lambda y.y\,x)z)\,v$，最外层的 $(\lambda \ldots)\,v$ 是一个 β 可约式，里层的 $(\lambda y.y\,x)\,z$ 也是一个 β 可约式。那么就有两种缩减方式：

$$\begin{array}{lll} (\lambda x.(\lambda y.y\,x)z)\,v & \rhd_{1\beta} \quad (\lambda y.y\,v)z & \text{缩减 } (\lambda x.(\lambda y.y\,x)z)\,v \\ & \rhd_{1\beta} \quad z\,v & \text{缩减 } (\lambda y.y\,v)\,z \\ (\lambda x.(\lambda y.y\,x)z)\,v & \rhd_{1\beta} \quad (\lambda x.z\,x)\,v & \text{缩减 } (\lambda y.y\,x)\,z \\ & \rhd_{1\beta} \quad z\,v & \end{array}$$

从示例 7 可以看到，虽然缩减顺序不一样，但最后得到了同样的结果。这个是不是对所有的 λ 项都成立呢？要是不成立，也就说顺序会影响规约结果，就不好了。这个问题的答案可以说既让人满意也让人不满意。

定理 2.（***Church–Rosser* 定理**）*若 $P \rhd_\beta M$ 且 $P \rhd_\beta N$，则存在一个 λ 项 T 使得*

$$M \rhd_\beta T \quad \text{且} \quad N \rhd_\beta T.$$

Church–Rosser 定理最重要的一个应用就是可以用来证明如下重要结果：

定理 3. *若 P 有 β 范式，则该范式在模 $\equiv_\alpha$ 的意义下唯一；也就是说若 P 有 β 范式 M 和 N，则 $M \equiv_\alpha N$。*

有了这个定理，我们可以放一半的心了：如果你对一个 λ 项做 β 规约，最后得到个范式，可以确定这个范式某种程度上是唯一的，不用担心你因为规约顺序上的随意性导致这个范式和别人不一样。但为啥说是放一半的心呢？*因为，这个定理并没有说：若 P 有 β 范式，那么你随便按任意顺序规约，都能得到 β 范式。*

示例 8. 让我们看下面有 β 范式的 λ 项的例子：

$$\begin{array}{rl} (\lambda x.\lambda y.x)\,a\,((\lambda x.x\,x)\lambda x.x\,x) \rhd_{1\beta} & ([a/x]\,(\lambda y.x))\,((\lambda x.x\,x)\lambda x.x\,x) \equiv (\lambda y.a)\,((\lambda x.x\,x)\lambda x.x\,x) \\ \rhd_{1\beta} & [((\lambda x.x\,x)\lambda x.x\,x)/y]\,a \equiv a \end{array}$$

这个 λ 项其实包含三项，我们要是先归约前两项 (λ…) a 就能得到一个 β 范式。但注意第三项也是一个 β 可约式，完全可以先对它进行规约。但如我们在示例 6 里的倒数第二项看到的，$((\lambda x.x\,x)\lambda x.x\,x)$ 会不断规约到自身，所以要是执着于把它先规约了，就永远得不到 β 范式了。

于是我们就要问了，若 P 有 β 范式，怎么才能规约得到它？按任意顺序规约就有可能出现示例 8 中的问题。可要是全部穷举，含有 n 个 β 可约式的话就有 $n!$ 种规约顺序，一一列举也太费事了吧。万幸，1958 年 Curry 证明了这么一个定理：

定理 4. *对 P 的总是先 β 缩减最左侧最外侧的 β 可约式，若这个过程能无限进行下去，那么对 P 的所有任意顺序的规约都能无穷进行下去。*

换而言之，要是 P 有 β 范式，那么这种最左最外侧优先的规约方式总能保证得到那个 β 范式，这种规约顺序又称为*正则序*。我的解释器实现的规约顺序就是正则序，虽然它规约需要的步数可能较多，但抵消不了这个巨大的优势啊。

到目前为止，我们得到的结果都是满意的：一个 λ 项若是有 β 范式，那么这个 β 范式某种程度上是唯一的，而且我们有确定的规约顺序保证得到这个范式。下面就只有一个问题了：能否有一个通用算法，判定一个 λ 项是否有 β 范式？非常不幸，答案是否定的：

定理 5. *λ 项是否有 β 范式是不可判定的。*

到这里我们算是粗粗的把 λ 演算的主要内容：β 规约介绍了一下。还有个 η 规约有兴趣的读者可以自行查找相关资料，这里就不多做介绍了。这个 β 规约就已经足够把 λ 演算看成一个完整的程序设计语言了。下面我将结合 λ 演算解释器向读者展现这一点。

3 程序设计语言

这一节的主要内容参考了维基百科 [6]。前面为了能指称一般的 λ 项引入了一些符号。本节中，需要指称具体的组合子，所以也做如下约定：

符号约定 2. 本文中，我们用粗体的大写字母及由它们组成的字符串代表具体的组合子，不同的粗体字母字符串如不做特殊说明，一般表示不同的组合子。当它们出现在 λ 项中时，视同对应的组合子整体出现在 λ 项中。

用粗体大写字母及其字符串代表组合子，可用等号 "=" 表示。比如想用 $\mathbf{M}$ 代表 $\lambda x.x$，可写作：$\mathbf{M} = \lambda x.x$。

示例 9. 随便举些例子好了：

1. 定义 $\mathbf{I} = \lambda x.x$，则 $\mathbf{I}\,a \equiv (\lambda x.x)\,a \rhd_\beta a$。
2. 定义 $\mathbf{SWAP} = \lambda x.\lambda y.y\,x$，则 $\mathbf{SWAP}\,a\,b \equiv (\lambda x.\lambda y.y\,x)\,a\,b \rhd_\beta ba$
3. 定义 $\mathbf{S} = \lambda x.\lambda y.\lambda z.xz(yz)$，则 $\mathbf{S}\,a\,b\,c \equiv (\lambda x.\lambda y.\lambda z.xz(yz))\,a\,b\,c \rhd_\beta ac(bc)$

从例子可以看出，组合子命名并没有改变 β 规约的规则。不过是在规约前先做简单替换罢了，和 C 语言中的宏一个道理。在解释器里面，也可以定义定义组合子，也是用等号，但不再限于用大写字母，任何合法的变量名都可以，但这种名称和组合子的绑定建立，就不能重新定义了。解释器里等于维护了一张表，在进行 β 规约前，会先查表把素所有绑定的名称都替换成对应的组合子。实在需要重定义的话，只能把整张表重置或清空。具体用法请参考附录 A。

示例 10.

```
Lambda> I = \x.x
Lambda> I a
a
Lambda> SWAP = \x.\y.y x
Lambda> SWAP a b
b a
Lambda> S = \x.\y.\z.x z(y z)
Lambda> S a b c
a c (b c)
Lambda> l = S I
Lambda> l m n
n (m n)
```

有了这个约定和基本工具，我们就可以好好感受一下 λ 演算的强大了。

3.1 算术计算

想想自然数的定义，我们只要有一个 0，有一个一元的后继函数 S 就可以了。事实上这就是皮亚诺算术公理的一部分。虽然 λ 演算里，规约前规约后一切都是 λ 项，没有什么数字，但仿照这个自然数定义，我们可以 "模拟" 自然数运算的，只用 β 规约即可。

我们定义

$$\begin{aligned}
\mathbf{ZERO} &= \lambda f.\lambda x.x \\
\mathbf{SUCC} &= \lambda n.\lambda f.\lambda x.f\ (n\ f\ x) \\
\mathbf{PLUS} &= \lambda m.\lambda n.m\ \mathbf{SUCC}\ n \\
\mathbf{MULT} &= \lambda m.\lambda n.\lambda f.m\ (n\ f) \\
\mathbf{POW} &= \lambda b.\lambda e.e\ b \\
\mathbf{PRED} &= \lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u) \\
\mathbf{SUB} &= \lambda m.\lambda n.n\ \mathbf{PRED}\ m
\end{aligned}$$

如它们名字所暗示的那样,这些函数分别代表了 0、后继、加法、乘法、乘方、前趋以及减法函数。这到底什么意思我们还是在解释器里看个明白。先看看零以外的数字是什么样子的:

```
Lambda> ZERO = \f.\x.x
Lambda> SUCC = \n.\f.\x.f (n f x)
Lambda> SUCC ZERO
\f.\x.f x
Lambda> SUCC (SUCC ZERO)
\f.\x.f (f x)
Lambda> SUCC(SUCC (SUCC ZERO))
\f.\x.f (f (f x))
Lambda> SUCC(SUCC( SUCC (SUCC ZERO)))
\f.\x.f (f (f (f x)))
```

看出来了吗,有了零和后继函数,数字 n 的定义就是

$$\lambda f.\lambda x.\underbrace{f\ (f\ \ldots\ (f\ x)\ \ldots)}_{n\text{重}f}$$

这种在 λ 演算中编码自然数的方法最早是邱奇发明的,又称为邱奇数。下面就可以尽情的在解释器里验证前面那些定义的组合子确实如名字那样暗示了可以对邱奇数进行计算吧(读者可以自行输入上面的定义,也可以用解释器已经预定义的如 `plus` 这样小写的组合子,详见附录 B)。

```
Lambda> ONE = SUCC ZERO
Lambda> TWO = SUCC ONE
Lambda> THREE = SUCC TWO
Lambda> FOUR = SUCC THREE
Lambda> PLUS TWO THREE
\f.\x.f (f (f (f (f x))))
Lambda> POW TWO FOUR
\x.\a.x (x (x (x (x (x (x (x (x (x (x (x (x (x (x (x a))))))))))))))))
Lambda> MULT THREE TWO
\f.\x.f (f (f (f (f (f x)))))
Lambda> SUB FOUR TWO
\f.\x.f (f x)
Lambda> PRED ONE
\f.\x.x
```

3.2 逻辑与谓词

一门程序语言光能做数学计算是远远不够的,它还必须能够处理逻辑判断,也就是有分支语句。光有分支语句也不够,还得能处理“与或非”等逻辑连接词。当然,这一切的基础还在于你首先得有布尔值真和假。下面我将把定义直接在解释器里给出,并演示它的确实能处理布尔值和逻辑连接词:

```
Lambda> TRUE = \x.\y.x
Lambda> FALSE = \x.\y.y
Lambda> AND = \p.\q.p q p
Lambda> OR = \p.\q.p p q
Lambda> NOT = \p.\a.\b.p b a
Lambda> IF = \p.\a.\b.p a b
Lambda> AND TRUE FALSE
\x.\y.y
Lambda> AND TRUE TRUE
\x.\y.x
Lambda> OR TRUE FALSE
\x.\y.x
Lambda> NOT TRUE
\a.\b.b
Lambda> NOT (NOT TRUE)
\a.\b.a
Lambda> IF TRUE a b
a
Lambda> IF FALSE a b
b
Lambda> IF (OR FALSE FALSE) a b
b
```

更进一步的,可以结合前面的算术运算定义一些谓词:

```
Lambda> ISZERO = \n.n (\x.FALSE) TRUE
Lambda> LEQ = \m.\n.ISZERO (SUB m n)
Lambda> EQ = \m.\n. AND (LEQ m n) (LEQ n m)
Lambda> ISZERO TWO
\x.\y.y
Lambda> ISZERO ZERO
\x.\y.x
Lambda> LEQ ONE ONE
\x.\y.x
Lambda> LEQ TWO ONE
\x.\y.y
Lambda> IF (EQ ONE TWO) a b
b
```

`LEQ` 表示自然数之间的 $\leq$,`EQ` 则表示自然数的等于。

3.3 函数与递归

从前面两个小节的例子可以看出，确实可以把抽象 $(\lambda\phi.M)$ 看成函数，把 $(M\,N)$ 函数调用，记住这两点的话，就可以定义一些简单的函数，多元函数就用柯里化解决。比如有了 **IF** 这个组合子，其实我们可以自行定义逻辑连接词。

```
Lambda> myNOT = \x.IF x FALSE TRUE
Lambda> myAND = \x.\y.IF x y FALSE
Lambda> myOR = \x.\y.IF x TRUE y
Lambda> myNOT(myNOT TRUE)
\x.\y.x
Lambda> myAND TRUE FALSE
\x.\y.y
Lambda> myAND TRUE TRUE
\x.\y.x
Lambda> myOR TRUE FALSE
\x.\y.x
```

组合子可以当作函数来用，说得更明白些，$\lambda\phi.M$ 中的 ϕ 就相当于参数，要是两个参数呢？那就定义形如 $\lambda\phi\psi.M$ 这样的组合子就行了。比如最大值和最小值函数可以这么定义：

```
Lambda> MAX = \m.\n.IF (LEQ m n) n m
Lambda> MAX ONE TWO
\f.\x.f (f x)
Lambda> MAX FOUR TWO
\f.\x.f (f (f (f x)))
Lambda> MIN = \m.\n.IF (LEQ m n) m n
Lambda> MIN TWO THREE
\f.\x.f (f x)
```

至此，把 λ 演算当作一门程序设计语言已经初见雏形了：有算术运算，有布尔运算，有条件分支，是不是差个循环语句就 OK 了？不过实际上，对一门程序设计语言而言，循环语句不是必须的，如 Scheme 语言就没有循环这个语法。对一种程序设计语言来讲，只要能定义递归函数就行了，所有的循环都可以转换成递归函数，不过这是另外一个话题了，有兴趣的可以自己查找相关资料。

还是举例子来说明简单些，比如我们要定义阶乘函数，它貌似很好定义：

$$\mathbf{FACT} = \lambda n.\mathbf{IF}\,(\mathbf{ISZERO}\,n)\,\mathbf{ONE}\,(\mathbf{MULT}\,n\,(\mathbf{FACT}\,(\mathbf{PRED}\,n))) \tag{4}$$

但这里有个问题，等号右边的那一堆定义里有 **FACT**，它指向了这个定义自身。根据符号约定 2，等号的右边必须是个组合子，现在 **FACT** 是个待定义的东西，不是个 λ 项。所以这个定义是非法的。符号约定 2 只是为了能够不用时时都写出完整的 λ 项而给它起个名称，方便使用罢了，本身不是 λ 项演算里的东西，不能用来做递归定义。$\lambda\phi.M$ 能看成函数不假，但这个函数没有“名字”，是不可以在函数体 M 里面自指的。解释器里也不行：

```
Lambda> FACT = \n.IF (ISZERO n) ONE (MULT ONE (FACT (PRED n)))
FACT can't be recursively defined!
```

乍看上去，似乎就没办法在 λ 演算里面定义阶乘，定义递归函数了，不允许自指那还怎么定义递归？让人惊叹不已的地方就在这里了，看看下面这个定义：

```
Lambda> FACT1 = \f.\n.IF (ISZERO n) ONE (MULT n (f f (PRED n)))
Lambda> FACT = FACT1 FACT1
Lambda> FACT THREE
\f.\x.f (f (f (f (f x))))
Lambda> FACT FOUR
\f.\x.f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f
(f (f x)))))))))))))))))))))))
```

数数看上面结果里的 f 是不是分别作用了 6 层和 24 层，确实是 3! 和 4!，我们真的定义出了阶乘函数。这是怎么做到的呢？仔细看看 FACT1 的定义，它和之前 (4) 的定义区别在于多了一个参数 f，并用 f f 代替了递归自指的 **FACT**。而在 FACT 的定义中，我们将 FACT1 作用于自身。这样规约的时候 f f x 就是 FACT1 FACT1 x 就是 FACT x。没有自指，通过两重应用这个技巧，就可以定义递归函数。进一步的，也不需要把 FACT1 写两遍，可以定义一个组合子 $\mathbf{W} = \lambda x.x\,x$。这样可以更简单的定义阶乘如下：

```
Lambda> W = \x.x x
Lambda> FACTD = W (\f.\n.IF (ISZERO n) ONE (MULT n (f f (PRED n))))
Lambda> FACTD THREE
\f.\x.f (f (f (f (f x))))
```

两重应用这个技巧是普适的，可以用来定义一般的递归函数。比如我们重新定义一下加法函数：

```
Lambda> ADD = W (\f.\n.\m.IF (ISZERO m) n (f f (SUCC n) (PRED m)))
Lambda> ADD TWO FOUR
\f.\x.f (f (f (f (f (f x)))))
Lambda> ADD FOUR FOUR
\f.\x.f (f (f (f (f (f (f (f x)))))))
```

到了这一步，一般人如我，就觉得 λ 演算定义递归函数已经完美解决，天下太平。但当年的逻辑学家们，计算机科学家们不是这么想的，他们觉得两重作用 f f 直接写在函数体里面，明明白白告诉别人我要自指，还是不够通用。能不能找到一个更加一般解决办法，在函数体里面连两重应用都不用，只用一重，照样能定义递归函数呢？答案是能，他们找到了不动点组合子。

在我写的解释器里，已经预定义了一个不动点组合子 **Y**，我们可以看看它是什么样子的：

```
Lambda> :set +hold
Lambda> Y
\g.(\x.g (x x)) \x.g (x x)
Lambda> :set -hold
Lambda> Y
Y seems can't be reduced!
```

可一看到前面命令 :set +hold 使解释器不自动进行规约是有道理的：**Y** 没有 β 范式。为什么叫 **Y** 不动点组合子呢，组合子好理解，不动点从何说起呢？这其实是个数学概念

定义 9.（不动点）对函数 f 而言，我们称 x 是 f 的不动点，当且仅当 x 满足

$$f(x) = x.$$

既然 λ 项中的抽象 $\lambda\phi.M$ 可以看作函数，自然也可以问问什么样的函数有不动点。不过我们到目前为止，只有 β 规约，还没有定义除了语法相等之外的 “等于” 是什么意思。β 规约下的等于定义是十分自然的：

定义 10. (**β 等于**) 我们说 P 是 β 等于 Q, 当且仅当 P 可以在有限步(包括零步)内经过 β 缩减或者反 β 缩减或者 α 变换得到 Q。也即 $P =_\beta Q$ 当且仅当存在 $P_0, P_1 \ldots P_n\,(n \geq 0)$, 其中 $P_0 \equiv P, P_n \equiv Q$ 并满足

$$(\forall i \leq n-1)\ (P_i \rhd_{1\beta} P_{i+1} \quad 或 \quad P_{i+1} \rhd_{1\beta} P_i \quad 或 \quad P_i \equiv_\alpha P_{i+1})$$

然后我们就可以回答 λ 演算中的不动点问题了:在 λ 演算中, 把 $\mathbf{Y}$ 应用于任意 F, 即 $\mathbf{Y}F$ 就是 F 的不动点

$$\begin{aligned}
&\mathbf{Y}F\\
&\equiv (\lambda g.(\lambda x.g\,(x\,x))\,\lambda x.g\,(x\,x))\,F\\
&=_\beta (\lambda x.F\,(x\,x))\,\lambda x.F\,(x\,x)\\
&=_\beta F\,((\lambda x.F\,(x\,x))\,\lambda x.F\,(x\,x))\\
&=_\beta F\,(\mathbf{Y}F)
\end{aligned}$$

解释了关于不动点组合子的疑问, 我们可以言归正传, 看它如何解决不用两重应用就定义递归函数的问题:

```
Lambda> FACT2 = \f.\n.IF (ISZERO n) ONE (MULT n (f (PRED n)))
Lambda> FACTY = Y FACT2
Lambda> FACTY THREE
\f.\x.f (f (f (f (f (f x)))))
Lambda> FACTY FOUR
\f.\x.f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f
(f (f x))))))))))))))))))))))))
```

很神奇吧? 这是什么原理呢? 牢牢记住前面证明不动点时的那个等式, 然后就可以逐步推断: $\mathbf{Y}F =_\beta F\,(\mathbf{Y}F)$。所以

$$\begin{aligned}
&\mathbf{FACT\ THREE}\\
=_\beta\ &\mathbf{Y\ FACT2\ THREE} \qquad (由定义)\\
=_\beta\ &\mathbf{FACT2\ (Y\ FACT2)\ THREE} \qquad (因为\ \mathbf{Y}F =_\beta F\,(\mathbf{Y}F))\\
=_\beta\ &\mathbf{IF\ (ISZERO\ THREE)\ ONE\ (MULT\ THREE\ (Y\ FACT2\ TWO))}\\
=_\beta\ &\mathbf{MULT\ THREE\ (Y\ FACT2\ TWO)}
\end{aligned}$$

这样 $\mathbf{Y\,FACT2}$ 再次出现, 但面对的参数已经被减去了 1, 依次类推, 可见确实定义了阶乘函数而 **FACT2** 也没有两重应用。

也不要以为不动点组合子只有一个, 事实上有很多很多, 比如图灵就发明过一个图灵不动点组合子, 定义如下:

$$\mathbf{T} = (\lambda x.\lambda y.y\,(x\,x\,y))\,\lambda x.\lambda y.y\,(x\,x\,y)$$

它和 $\mathbf{Y}$ 组合子一样可以用来定义递归函数, 有解释器为证:

```
Lambda> T = (\x.\y.y (x x y)) \x.\y.y (x x y)
Lambda> FACTT = T FACT2
Lambda> FACTT FOUR
\f.\x.f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f
(f (f x))))))))))))))))))))))))
```

3.4 数据结构

我相信前面已经展示了 λ 演算可以作为程序语言的很多功能，已经很完备了，但美中还有不足。作为控制结构来讲，有分支有函数有递归，已经足够了。但数据类型还是不够丰富：只有正整数类型和布尔类型。学过 Lisp 的读者应该清楚，只要程序语言能够定义有序对 (a,b) 这样的，那基本的数据结构列表也好，树也罢都可以用有序对来构造。在 λ 演算里，这也不是问题：

```
Lambda> CONS = \x.\y.\f. f x y
Lambda> CAR = \p.p TRUE
Lambda> CDR = \p.p FALSE
Lambda> NIL = \x. TRUE
Lambda> NULL = \p.p (\x.\y.FALSE)
```

这里面 CONS 是构造有序对的，CAR 是取有序对的头，CDR 是取有序头以外的尾，NIL 表示一个空的有序对，而 NULL 是谓词，用于判断一个有序对是否为空。可以验证这些基本定义的语义没有问题：

```
Lambda> CONS a (CONS b (CONS c NIL))
\f.f a \f.f b \f.f c \x.\x.\y.x
Lambda> CAR (CONS a (CONS b (CONS c NIL)))
a
Lambda> CDR (CONS a (CONS b (CONS c NIL)))
\f.f b \f.f c \x.\x.\y.x
Lambda> CAR (CDR (CONS a (CONS b (CONS c NIL))))
b
Lambda> NULL (CDR (CONS a (CONS b (CONS c NIL))))
\x.\y.y
Lambda> NULL NIL
\x.\y.x
```

可以看到连续的应用 CONS 可以得到列表。我们甚至可以定义长度函数：

```
Lambda> LENGTH = Y (\g.\c.\x. NULL x c (g (SUCC c) (CDR x))) ZERO
Lambda> LENGTH NIL
\f.\x.x
Lambda> LENGTH (CONS a (CONS b (CONS c NIL)))
\f.\x.f (f (f x))
Lambda> LENGTH (CONS a (CONS b (CONS c (CONS d NIL))))
\f.\x.f (f (f (f x)))
```

用 λ 演算可以定义的更多的东西，比如除法，有兴趣的读者可以自己用关键词 “lambda calculus division” 搜索看看，网上相关的资料汗牛充栋。

通过这一系列的演示可以看到，虽然可定义的东西如此之多，从基本数据类型到复合的数据结构，从分支判断到递归函数。他们表面看上去千差万别，但实际上都是 λ 项，都可以用 β 规约计算，都可以得到同为 λ 项的结果。λ 项的定义很简单，β 规约直观理解起来也不复杂。从如此简单的定义出发，可以得到如此丰富、美妙，不可思议的结果。我觉得这就是 λ 演算被认为是程序语言，特别是函数式程序设计语言基石的原因：所有的语义都可以用 λ 演算来解释，一切皆 λ！

A 解释器简要使用说明

在学习了解 λ 演算的过程中,为了加深自己对 λ 演算的感性认识,我实现了一个简单的解释器。这个解释器只实现了两个简单的功能:对无类型纯 λ 演算进行 β 规约,允许命名绑定组合子。有了这两个简单的功能,就可以把 λ 演算当作一种程序设计语言,真实不虚的 "运行" 这篇文章里介绍的各种 λ 项,看到归约后的结果。用户还可以对解释器做一些设置,决定输出形式,是否逐步给出规约过程等等。

解释器可以从这里下载:`https://github.com/txyyss/Lambda-Calculus/releases`。这个解释器中用字符 \ 代表 λ,其余语法和 2.1 中定义的一样。下面介绍一下用户可以设定的参数。

默认设置下,解释器会自动对用户输入的 λ 项不断进行 β 规约,直到得到一个范式(如果存在的话)。可用命令 `:set +hold` 让解释器进入 "原样输出" 模式,命令 `:set -hold` 则恢复立即执行的模式。

```
Lambda> (\x.\y.y x) a b
b a
Lambda> :set +hold
Lambda> (\x.\y.y x) a b
(\x.\y.y x) a b
```

解释器默认的输出样式采用了省略表示,去掉了不必要的括号,可以用命令 `:set +fullform` 和 `:set -fullform` 控制是否输出全部带括号的结果。

```
Lambda> (a b) c d e f g
a b c d e f g
Lambda> :set +fullform
Lambda> a b c d e f g
((((((a b) c) d) e) f) g)
Lambda> \x.\y.y x a b
(\x.(\y.(((y x) a) b)))
```

解释器默认情况下,只输出最终的规约结果,可以用命令 `:set +trace` 和 `:set -trace` 控制是否输出中间结果。

```
Lambda> (\x.\y.y x) a b
b a
Lambda> :set +trace
Lambda> (\x.\y.y x) a b
==> (\x.\y.y x) a b
==> (\y.y a) b
==> b a
```

由于一个 λ 项是否一定可以被 β 规约规约到 β 范式是不可判定的,所以为了折衷解决这个问题,解释器设置了一个阈值,当规约步数超过 "λ 项长度 $\times$ 阈值" 时,解释器就输出说疑似不可规约。这个阈值可以用 `:set steps` 非负整数 来修改,默认是 100。用户若是觉得阈值及前述三个控制选项经多次修改之后太乱了,可以用命令 `:reset settings` 来恢复默认设置。

```
Lambda> (\x.\y.y y x) a b
b b a
Lambda> :set steps 0
Lambda> (\x.\y.y y x) a b
(\x.\y.y y x) a b seems can't be reduced!
Lambda> :reset settings
Lambda> (\x.\y.y y x) a b
b b a
```

解释器支持用户绑定名称到组合子,注意,只能是组合子,也就是没有自由变量的 λ 项。组合子中可以出现已经绑定的名称,但不允许递归定义。已经绑定的名称不能再次绑定。

```
Lambda> swap = \x.\y.y x
Lambda> swap a b
b a
Lambda> foo = swap \x.x
Lambda> foo \x.y
y
Lambda> id = id \x.x
id can't be recursively defined!
Lambda> foo = \x.x
foo has been defined already!
```

解释器已经默认绑定了一些这篇文章里出现的组合子,全部列表可见 B。用户在添加了自己的绑定后,可以用 `:reset state` 来恢复默认的绑定,清空自定义的绑定。也可以用命令 `:clear state` 来清空全部的绑定。

```
Lambda> plus
\m.m \n.\f.\x.f (n f x)
Lambda> plus four four
\f.\x.f (f (f (f (f (f (f (f x)))))))
Lambda> id = \x.x
Lambda> id a
a
Lambda> :reset state
Lambda> id a
id a
Lambda> :clear state
Lambda> plus four four
plus four four
Lambda> :reset state
Lambda> plus four four
\f.\x.f (f (f (f (f (f (f (f x)))))))
```

以上就是用户在 λ 演算解释器里可以做的全部设置了。要退出解释器,输入 `:q` 命令即可。

B 解释器默认绑定的组合子和名称

其实就是依次运行如下命令后，解释器内部状态中绑定的组合子。

```
zero   = \f.\x.x
succ   = \n.\f.\x.f (n f x)
plus   = \m.\n.m succ n
mult   = \m.\n.\f.m (n f)
pow    = \b.\e.e b
pred   = \n.\f.\x.n (\g.\h.h (g f)) (\u.x) (\u.u)
sub    = \m.\n.n pred m
one    = succ zero
two    = succ one
three  = succ two
four   = succ three
true   = \x.\y.x
false  = \x.\y.y
and    = \p.\q.p q p
or     = \p.\q.p p q
not    = \p.\a.\b.p b a
if     = \p.\a.\b.p a b
iszero = \n.n (\x.false) true
leq    = \m.\n.iszero (sub m n)
eq     = \m.\n. and (leq m n) (leq n m)
Y      = \g.(\x.g (x x)) (\x.g (x x))
```

参考文献

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. Electrical engineering and computer science series. MIT Press, 1996.

[2] A.N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1927.

[3] John von Neumann. Eine Axiomatisierung der Mengenlehre. *Journal für die reine und angewandte Mathematik*, 154, page:219–240, 1938.

[4] Alonzo Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 33(2):346–366, 1932.

[5] J.R. Hindley and J.P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2008.

[6] Wikipedia. Lambda calculus — wikipedia, the free encyclopedia, 2013. [Online; accessed 6-August-2013].