第 1 章　λ-演算

1.1　λ-演算的起源

20 世纪 30 年代，可计算性（computability）问题得到一些数学家和逻辑学家的关注。通俗而言，一个自然数集合上的函数 $f : \mathbb{N} \to \mathbb{N}$ 称为可计算的，是指对任何一个给定的自然数 n，可以给一位受过良好数学训练的人足够用的纸和笔，不管花费多长时间，最终他总能用一组基本的计算操作构造出函数 f 在 n 上的值 $f(n)$。当然，这只是对可计算函数的直觉描述，不是一个数学定义。为了严格地刻画可计算函数，前面这个非形式化（informal）的定义显然不方便，因此有必要对可计算性给出形式化（formal）的定义。当时有多位学者在这方面进行了尝试，其中最有名的三位是英国数学家、计算机科学家阿兰 • 图灵（Alan Turing），奥地利逻辑学家、数学家库特 • 哥德尔（Kurt Gödel）和美国数学家、逻辑学家阿隆佐 · 邱奇（Alonzo Church，见图 1.1 [①]）。

Alan Turing
(1912—1954)

Kurt Gödel
(1906—1978)

Alonzo Church
(1903—1995)

图 1.1　研究可计算性问题的三位先驱

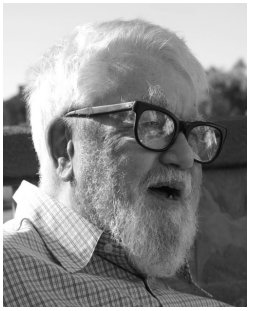
- 图灵提出一种抽象的理想化的计算模型，现在通称之图灵机（Turing machine）。他假定一个函数 f 在直觉意义上是可计算的，当且仅当有一个图灵机可计算出该函数在任何给定自然数 n 上的值 $f(n)$。
- 哥德尔定义了一类一般递归函数（general recursive function），由常函数、后继函数等基本函数在函数复合和递归等算子组合下形成。他假定一个函数在直觉意义上是可计算的，当且仅当它是一般递归的。
- 邱奇[②]定义了一个称为λ-演算（λ-calculus）的形式化语言并假定一个函数在直觉意义上是可计算的，当且仅当它可用一个 λ-表达式写出来。

① 本书中出现的人物图片均来自 https://www.wikipedia.org 或其工作单位的网站。

② 邱奇是图灵的博士论文导师。此外，邱奇还有许多其他优秀的学生，例如 1976 年图灵奖获得者迈克尔 · 拉宾（Michael O. Rabin）和达纳 · 斯科特（Dana Scott）均出自他的门下。

后来，邱奇等证明了上面三个模型是互相等价的，即它们定义的是同一类（class）可计算函数。至于它们是否等价于直觉意义上的可计算函数类，则是一个没法回答的问题，因为我们没有一个描述直觉意义上的可计算函数类。著名的邱奇-图灵论题（Church-Turing thesis）断言上面三个模型表达的恰好正是直觉意义上的可计算函数类。

本章主要关注 λ-演算，因为函数式编程语言最初是从它发展而来的。早在 20 世纪 50 年代，美国计算机科学家约翰 · 麦卡锡（John McCarthy，见图 1.2）为 IBM 700/7000 系列机器创造了第一个函数式编程语言 Lisp。现在很多人知道麦卡锡是人工智能领域的奠基人之一并获得 1971 年图灵奖，殊不知他对函数式编程也有重要贡献。现在我们经常用不带类型的 λ-演算（untyped λ-calculus）指代邱奇当初提出的那个形式化语言。后来又出现这个语言的各种更精细的版本，例如带类型的 λ-演算（typed λ-calculus）。1.2节和 1.3节将分别介绍不带类型的 λ-演算和简单类型的 λ-演算；更深入、全面的讨论可以参考文献 [2,3]。

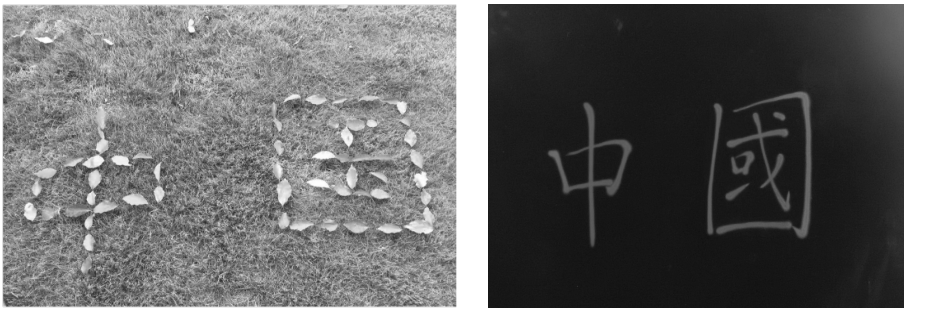


John McCarthy
(1927—2011)

图 1.2 Lisp 的发明人

1.2 不带类型的 λ-演算

在介绍一种语言时，通常会给出它的语法和语义。初学者可能不熟悉这两个术语之间的区别，因此有必要简单解释一下。通俗而言，语法是某一文本的外在呈现形式，语义是被传达的意思。例如，图 1.3 中展示的两个图案，图 1.3(a) 用树叶拼成，图 1.3(b) 是在液晶手写板上写的。虽然语法呈现形式不同（树叶拼的图案或是手写的繁体字），但表达相同的语义（“中国” 这两个汉字）。



(a) 语法 (b) 语义

图 1.3 语法与语义的直观解释

不带类型的 λ-演算是一种形式化语言，它的语法和语义可以形式化地给出。语法用于规定这个语言中允许出现的表达式，称为λ-项（λ-term），是如何被构造出来的，而语义则规定 λ-项演化的方式，用于描述我们直觉上的计算过程。

1.2.1　语法

不带类型的 λ-演算中的表达式可以通过极其简单的规则构造出来。

定义 1.1　假设给定一个可数无穷的变量集合 $\mathcal{V}$，其中的元素用 x，y，z 等表示。我们用如下的巴克斯-诺尔范式（Backus-Naur Form，BNF）生成 λ-项：

$$M,\ N\ ::=\ x \mid (MN) \mid (\lambda x.M)$$

在上述的巴克斯-诺尔范式中，被定义的对象出现在符号 ::= 左边，即用大写字母 M、N 等代表 λ-项；在 ::= 右边列出其允许的形式，两条竖线隔开三种情况，说明用三条规则生成 λ-项。换句话说，所有 λ-项的集合记为 Λ，是由下面三条规则生成的最小集合。

（1）如果 $x \in \mathcal{V}$，那么 $x \in \Lambda$；

（2）如果 $M, N \in \Lambda$，那么 $(MN) \in \Lambda$；

（3）如果 $x \in \mathcal{V}$ 且 $M \in \Lambda$，那么 $(\lambda x.M) \in \Lambda$。

这些规则会产生三种形式的 λ-项，依次把它们称为变量（variable）、作用（application），以及λ-抽象（lambda abstraction）。例如，下面三个例子展示的就是这三种类型的 λ-项：

$$y \qquad (\lambda x.(xx))(\lambda y.(yy)) \qquad (\lambda f.(\lambda x.x))$$

注意　为方便读者记住上面的语法规则，不妨把每个 λ-项看成一个函数，其功能是把输入值变换成输出值。变量 x 是一个基本的 λ-项，相当于一个常函数，不管输入是什么，总是输出当前 x 的值；项 (MN) 表示把函数 M 作用到它的参数 N 上；项 $(\lambda x.M)$ 表示构造一个新函数，对于输入 x，输出函数体 M 表示的值。需要注意的是，这种直观理解实际上是不准确的，比如在项 (xx) 中，一个函数 x 的参数是其自身，与常规直觉不符合，但对初学者而言，这种直觉很大程度上有利于理解抽象的数学符号。

需要注意的是，在定义 1.1 中强制性地加入括号以便把一个项和它的子项严格区分开。所谓一个项 M 的子项（subterm），是指构成 M 的字符串的一个子串，它根据定义 1.1也组成一个合法的项。例如，对于 $(\lambda f.(\lambda x.x))y$ 这个项，它有 x、$(\lambda x.x)$、y 等子项。

为了少写一些括号但又不引起歧义，我们采用下面的约定（convention）：

- 忽略最外层的括号，例如将 $(\lambda x.M)$ 写成 $\lambda x.M$；
- λ-项的作用满足左结合性，即 $MNOP$ 等同于 $((MN)O)P$；
- λ-抽象的主体部分，即点号右边的部分，包含右边尽可能多的项，例如 $\lambda x.MN$ 表示 $\lambda x.(MN)$，而不是 $(\lambda x.M)N$；
- 当有连续多个 λ-抽象时，只写最左边的 λ 符号，例如把 $\lambda f.\lambda x.f(fx)$ 写成 $\lambda fx.f(fx)$。

练习 1.1　(1) 根据上面的约定，尽可能减少下面这些 λ-项中的括号，但不改变这些项的结构：

(i) $\lambda f.(\lambda x.x)$

(ii) $\lambda f.(\lambda x.(fx))$

(iii) $x(y(\lambda z.z))$

(2) 尽量补全下面这些 λ-项中的括号，但不改变这些项的结构:

(i) $\lambda xy.y$

(ii) $\lambda ab.(\lambda ab.b)$

(iii) $\lambda nmfx.nf(mfx)$

1.2.2 α-等价

之前我们说过，λ-演算是一种形式化的语言。可以粗略地认为，用这种语言写出来的句子应该是给计算机识别的。对机器而言，$\lambda x.x$ 和 $\lambda y.y$ 是两个不同的字符串。但在语义层面上，希望把这两个 λ-项看成同一个函数，对输入的值不做任何改变而直接输出，即恒等函数。这种想法其实并不陌生，在数学上会写 $f(x) = x$ 或者 $f(y) = y$ 来表示恒等函数，而且不假思索地认为这两种写法没有任何区别。为了把 $\lambda x.x$ 和 $\lambda y.y$ 这样的项等同起来，本节引入 α-等价的概念。在此之前，需要介绍自由变量和受限变量的概念。

定义 1.2 对形如 $\lambda x.M$ 的项，称 λx 为一个绑定子（binder），子项 M 为这个绑定子的范围（scope）。变量 x 的一次出现（occurrence）如果在 M 中，则称这次出现为受限的（bound），否则称为自由的（free）。

注意，绑定子 λx 中的 x 不算作变量 x 的一次出现。

例 1.1 对于项 $(\lambda x.yx)(\lambda y.zy)$，从左往右看，$y$ 的第一次出现是自由的，第二次出现是受限的；变量 x 和 z 分别出现一次，前者是受限的，后者是自由的。

例 1.2 在项 $\lambda x.(\lambda y.\lambda x.xy)xy$ 中，变量 x 出现两次。从左往右看，x 的第一次出现受限于内层的绑定子 λx，第二次出现受限于外层的绑定子 λx。变量 y 也出现两次，第一次出现受限于绑定子 λy，而第二次出现是自由的。

下面形式化地定义任何一个给定项 M 中自由出现的变量的集合，记为 $FV(M)$。

$$\begin{aligned} FV(x) &\stackrel{\text{def}}{=} \{x\} \\ FV(MN) &\stackrel{\text{def}}{=} FV(M) \cup FV(N) \\ FV(\lambda x.M) &\stackrel{\text{def}}{=} FV(M) \backslash \{x\} \end{aligned}$$

注意 以上对函数 $FV(\cdot)$ 的定义是一种典型的归纳定义，称为结构归纳（structural induction）。在同一行中，如果 $FV(M)$ 出现在定义符号 $\stackrel{\text{def}}{=}$ 左边，某个 $FV(N)$ 出现在右边，那么 N 一定是 M 的子项，即 N 的结构比 M 简单，以确保 $FV(\cdot)$ 是良定义的（well defined）。

练习 1.2 令 $V(M)$ 为 λ-项 M 中出现的所有变量的集合。利用结构归纳的方式定义函数 $V(\cdot)$。

练习 1.3　令 $ST(M)$ 为 λ-项 M 的所有子项的集合，并假设 M 是其自身的一个子项。利用结构归纳的方式定义函数 $ST(\cdot)$。例如，

$$ST(\lambda x.xy) = \{\lambda x.xy,\ xy,\ x,\ y\}.$$

有时候希望给一个项 M 中的变量换名字，或者说重命名（rename）一个变量。我们用记号 $M\{y/x\}$ 表示把项 M 中的所有变量 x 换成变量 y 之后得到的项。同样，用结构归纳的方式定义这个重命名函数。

$$\begin{aligned}
x\{y/x\} &\stackrel{\text{def}}{=} y \\
z\{y/x\} &\stackrel{\text{def}}{=} z \qquad \text{若} x \neq z \\
(MN)\{y/x\} &\stackrel{\text{def}}{=} (M\{y/x\})(N\{y/x\}) \\
(\lambda x.M)\{y/x\} &\stackrel{\text{def}}{=} \lambda y.(M\{y/x\}) \\
(\lambda z.M)\{y/x\} &\stackrel{\text{def}}{=} \lambda z.(M\{y/x\}) \qquad \text{若} x \neq z
\end{aligned}$$

从上面的定义可以看出，在重命名时把所有 x 的出现替换为 y 的出现，不管这些出现是自由的还是受限的。

有了前面这些准备工作，现在可以给出α-等价（α-equivalence）的严格定义。

定义 1.3　在 λ-项上满足表 1.1中所有规则的最小二元关系称为 α-等价，记为 $=_\alpha$。

表 1.1　α-等价的定义规则

规则	关系式	规则	关系式
(R_r)	$\dfrac{}{M = M}$	(R_{ap})	$\dfrac{M = M' \qquad N = N'}{MN = M'N'}$
(R_s)	$\dfrac{M = N}{N = M}$	(R_{ab})	$\dfrac{M = M'}{\lambda x.M = \lambda x.M'}$
(R_t)	$\dfrac{M = N \qquad N = P}{M = P}$	(R_α)	$\dfrac{y \notin V(M)}{\lambda x.M = \lambda y.(M\{y/x\})}$

表 1.1 中共有 6 条规则，其中（R_r）、（R_s）、（R_t）分别表示自反、对称、传递性质，满足这三条规则的二元关系是一个等价关系；（R_{ap}）和（R_{ab}）分别表示在作用和 λ-抽象两个语法构造下的封闭性，在 λ-项上满足这两条规则的二元关系具有同余性（congruence）。最核心的规则是（R_α），其意义是可以把一个受限变量 x 重命名为任何一个目前没出现过的新变量 y。定义 1.3说明 α-等价是在 λ-项上满足（R_α）的等价且同余的最小二元关系。

例 1.3　若两个项 M 和 N 不满足 α-等价关系，则记为 $M \neq_\alpha N$。

- $\lambda x.\lambda x.x \ =_\alpha \ \lambda x.\lambda y.y \ \neq_\alpha \ \lambda x.\lambda y.x$
- $\lambda x.M(\lambda y.y) \ \neq_\alpha \ \lambda x.M(\lambda x.y)$
- $\lambda x.xy \ \neq_\alpha \ \lambda y.yx \ \neq_\alpha \ \lambda x.xx$

练习 1.4 证明如下性质：任何一个 λ-项 M 都 α-等价于另一个项 M'，使得 M' 中任何一个受限变量都与其他的受限或自由变量不重名。

从现在开始，将不再区分 α-等价的 λ-项，就像我们不希望区分数学表达式 $\int x \, \mathrm{d}x$ 和 $\int y \, \mathrm{d}y$ 一样。

1.2.3 替换

与重命名相近但略复杂的一个概念是替换（substitution），它允许把一个变量替换为一个 λ-项。通常用记号 $M[N/x]$ 表示把项 M 中的 x 替换为 N 的结果。不过，这样的替换需要满足下面两个条件。

（1）只能把自由变量替换为其他项。1.2.2 节介绍的 α-等价允许把受限变量重命名为任何新的变量。如果对受限变量进行替换，会破坏 α-等价。例如，$(\lambda x.x)[yy/x]$ 应该等于 $(\lambda x.x)$ 而不是 $(\lambda x.yy)$，否则，本来 $(\lambda x.x)$ 与 $(\lambda z.z)$ 是 α-等价的，替换之后两者将不等价，这不是我们期望的结果。

（2）项 N 中的自由变量不能被 M 中的绑定子捕获（capture）。例如，假设有 $M \stackrel{\text{def}}{=} \lambda x.xy$，$N \stackrel{\text{def}}{=} yx$。注意，变量 x 在 N 中是自由的，但在 M 中是受限的。如果允许如下替换：

$$M[N/y] = (\lambda x.xy)[yx/y] = \lambda x.x(yx)$$

则把原来 N 中自由的 x 捕获了，而变成和 M 中受限的 x 视为相同的变量。这不是我们期望的结果，因为后者可以重命名为任何一个新的变量但前者不可以。遇到这种情况，正确的做法应该是在替换之前就对 M 中的受限变量 x 重命名：

$$M[N/y] = (\lambda x'.x'y)[yx/y] = \lambda x'.x'(yx)$$

为满足第二个条件，有时需要把一个受限变量重命名为一个新鲜的（在目前考虑的项中未曾使用过的）变量。在定义 1.1中，假定变量集合 $\mathcal{V}$ 是可数无穷的，这样可以保证任何时候都能取到新变量。

定义 1.4 把项 M 中自由出现的 x 改写成项 N 的免捕获（capture-avoiding）替换定义如下：

$$\begin{aligned}
x[N/x] &\stackrel{\text{def}}{=} N \\
y[N/x] &\stackrel{\text{def}}{=} y \qquad \text{若} x \neq y \\
(MP)[N/x] &\stackrel{\text{def}}{=} (M[N/x])(P[N/x]) \\
(\lambda x.M)[N/x] &\stackrel{\text{def}}{=} \lambda x.M \\
(\lambda y.M)[N/x] &\stackrel{\text{def}}{=} \lambda y.(M[N/x]) \qquad \text{若 } x \neq y \text{ 且 } y \notin FV(N) \\
(\lambda y.M)[N/x] &\stackrel{\text{def}}{=} \lambda y'.(M\{y'/y\}[N/x]) \qquad \text{若 } x \neq y,\ y \in FV(N) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{且 } y' \text{ 是新鲜的}
\end{aligned}$$

在这个定义的最后一行，我们只说 y' 是新鲜的（fresh），但没有明确具体如何选择这样的变量。不失一般性，不妨假设集合 $\mathcal{V}$ 中的元素有一个枚举：$y_1, y_2, \cdots$ 每当需要一个新鲜变量时，就选一个下标最小并且在 M 和 N 中未使用过的那个变量 y_i。

例 1.4　下面三个替换的结果不同，分别对应定义 1.4中的最后三种情况。

（1）$(\lambda x.xyy)[xy/x] = \lambda x.xyy$

（2）$(\lambda x.xyy)[zy/y] = \lambda x.x(zy)(zy)$

（3）$(\lambda x.xyy)[xy/y] = \lambda x'.x'(xy)(xy)$

练习 1.5　证明对任何变量 x，任何三个项 M、N 和 P，如果 $M =_\alpha P$，那么 $M[N/x] =_\alpha P[N/x]$。

1.2.4　β-归约

给定两个简单的算术表达式，比如 $1+2\times 3$ 和 $3+2+2$，很容易看出它们表示相同的结果。究其原因，原来我们可以运用乘法和加法的等式公理做递等式计算，这个过程是对数学表达式的化简，最后两个等式都可以化简到数字 7。

$$\begin{array}{rl} & 1+2\times 3 \\ = & 1+6 \\ = & 7 \end{array} \qquad\qquad \begin{array}{rl} & 3+2+2 \\ = & 5+2 \\ = & 7 \end{array}$$

在上面的递等式计算过程中，用到好几条等式公理，比如 $2\times 3 = 6$，$1+6=7$ 等。

给定 λ-演算中的两个表达式，即两个 λ-项 M 和 N，也希望对它们进行某种转换或者化简，看最后能否变成相同的项。在 λ-演算中，只需要一条化简规则，称为 β-归约，其直观想法就是把函数的参数代入函数体。一个形如 $(\lambda x.M)N$ 的项称为一个β-可约项（β-redex），它把一个 λ-抽象 $(\lambda x.M)$ 作用到另一个项 N 上。我们把它归约（reduce）到 $M[N/x]$，后面这个项称为规约结果项（reduct）。对 λ-项的这一步转换，可写成下面的形式：

$$(\lambda x.M)N \ \longrightarrow_\beta \ M[N/x]$$

对 λ-项归约的过程就是不断寻找可约项，换成它的规约结果项的过程。如果一个项中没有可约项，就称这个项是一个β 范式（β-normal form）。

例 1.5　对项 $(\lambda x.x((\lambda y.y)z))(\lambda x.y)$ 进行 β-归约。在每一步中，对即将归约的可约项用下画线标示出来。

$$\begin{array}{rcl} \underline{(\lambda x.x((\lambda y.y)z))(\lambda x.y)} & \longrightarrow_\beta & (\lambda x.y)(\underline{(\lambda y.y)z}) \\ & \longrightarrow_\beta & \underline{(\lambda x.y)z} \\ & \longrightarrow_\beta & y \end{array} \tag{1.2.1}$$

最后这个项 y 没有可约项，是一个范式。实际上，在第二步可以选择另外一个可约项进行归约：

$$\begin{array}{lll} \underline{(\lambda x.x((\lambda y.y)z))(\lambda x.y)} & \longrightarrow_\beta & \underline{(\lambda x.y)((\lambda y.y)z)} \\ & \longrightarrow_\beta & y \end{array} \tag{1.2.2}$$

通过上面的例子可以观察到如下几点：

- 对一个可约项进行归约可以创建出新的可约项。例如，在式 (1.2.1) 的第一行右边，项 $(\lambda x.y)((\lambda y.y)z)$ 是一个新出现的可约项。
- 对一个可约项进行归约可以删除其他的可约项。例如，在式 (1.2.2) 的第一行右边，原来存在的可约项 $((\lambda y.y)z)$ 被下一步的归约删除了。
- 把一个项归约为一个范式的步骤数目可以随归约顺序的不同而变化。例如，在式 (1.2.1) 和式 (1.2.2) 中，虽然从相同的一个项开始归约，但经过的步骤数分别为 3 和 2。

从上面的例子中可以看到，不同的归约顺序虽然经历不同长度的归约路径，但最后都到达同一个范式 y。实际上，这并非偶然，而是一个一般结论，将在 1.2.8 节介绍。

存在一些项，从它们出发进行归约，最后可能不能到达任何范式。例如，

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta \cdots$$

项 $(\lambda x.xx)(\lambda x.xx)$ 只有唯一的一个可约项，归约到的规约结果项是其自身。

练习 1.6 判断下面四个项是否都能归约到范式:

(1) $(\lambda xy.x)(\lambda x.xx)(\lambda x.xx)$

(2) $(\lambda xy.x)((\lambda x.xx)(\lambda x.xx))$

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

(4) $(\lambda fx.fxx)(\lambda xy.y)y$

练习 1.7 给出一个项 M，每归约一步就到达一个更复杂的项，从而无法最终到达一个范式。

练习 1.8 给出一个项 M，它经过一系列的归约能到达一个范式，但是若经过另一系列的归约，则无法到达一个范式。

下面给出 β-归约的形式化定义。

定义 1.5 单步 β-归约，记作 $\longrightarrow_\beta$，是满足下列规则的最小关系:

$$(B_\beta)\quad \frac{}{(\lambda x.M)N \longrightarrow_\beta M[N/x]} \qquad (B_{ap1})\quad \frac{M \longrightarrow_\beta M'}{MN \longrightarrow_\beta M'N}$$

$$(B_{ab})\quad \frac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'} \qquad (B_{ap2})\quad \frac{N \longrightarrow_\beta N'}{MN \longrightarrow_\beta MN'}$$

可以看出，存在单步归约 $M \longrightarrow_\beta M'$ 当且仅当 M' 是通过归约 M 中的一个可约项得到的。我们用 $\longrightarrow_\beta^*$ 表示 $\longrightarrow_\beta$ 的自反传递闭包，即包含 $\longrightarrow_\beta$ 的自反和传递的最小关系。如果 $M \longrightarrow_\beta^* M'$，那么 M' 是从 M 出发经过零步或多步 β-归约得到的项。

定义 1.6 令 $\longleftarrow_\beta$ 为 $\longrightarrow_\beta$ 的逆关系，即 $M' \longleftarrow_\beta M$ 当且仅当 $M \longrightarrow_\beta M'$。定

义 $=_\beta$ 为 $\longrightarrow_\beta$ 及其逆关系的自反传递闭包，即 $(\longrightarrow_\beta \cup \longleftarrow_\beta)^*$。因此，$M =_\beta N$ 表示可以从 M 出发通过反复地正向或逆向使用 β-归约而得到 N。

例 1.6　令 $M \stackrel{\text{def}}{=} (\lambda x.x((\lambda y.y)z))(\lambda x.y)$ 和 $N \stackrel{\text{def}}{=} (\lambda a.\lambda b.b)xy$。由例 1.5 可以知道 $M \longrightarrow_\beta^* y$。从项 N 出发有 $N \longrightarrow_\beta (\lambda b.b)y \longrightarrow_\beta y$，于是

$$M \longrightarrow_\beta^* y \longleftarrow_\beta^* N$$

也就是说，有 $M =_\beta N$。

练习 1.9　证明如下性质：如果 $M \longrightarrow_\beta N$，那么 $FV(N) \subseteq FV(M)$。为什么 $FV(N) = FV(M)$ 不一定成立？

练习 1.10　证明如下性质：

（1）如果 $N \longrightarrow_\beta^* N'$，那么 $M[N/x] \longrightarrow_\beta^* M[N'/x]$。

（2）如果 $N \longrightarrow_\beta^* N'$，那么 $M[N/x] \longrightarrow_\beta^* M[N'/x]$。如果 $M \longrightarrow_\beta M'$，那么 $M[N/x] \longrightarrow_\beta M'[N/x]$。

（3）如果 $N \longrightarrow_\beta^* N'$，那么 $M[N/x] \longrightarrow_\beta^* M[N'/x]$。如果 $M \longrightarrow_\beta^* M'$ 且 $N \longrightarrow_\beta^* N'$，那么 $M[N/x] \longrightarrow_\beta^* M'[N'/x]$。

1.2.5　表达能力

虽然 λ-演算的语法和归约语义非常简单，但是可以用它编码各种数据值，如布尔值和自然数，以及操纵这些数据的常见运算。事实上，我们并不特意区分数据和数据上的运算，因为它们都可用 λ-项表达。为了给一些项命名，下面用粗体字母作为名字。

布尔值　定义两个 λ-项 **T** 和 **F** 来编码布尔类型中的真和假两个值：

$$\begin{aligned} \mathbf{T} &\stackrel{\text{def}}{=} \lambda xy.x \\ \mathbf{F} &\stackrel{\text{def}}{=} \lambda xy.y \end{aligned}$$

有了布尔值之后，接下来定义对布尔值的运算：与、或、非。and 表示“与”运算，若令 $\mathbf{and} \stackrel{\text{def}}{=} \lambda ab.aba$，则下面四条归约性质成立：

$$\begin{aligned} \mathbf{and\ TT} &\longrightarrow_\beta^* \mathbf{T} \\ \mathbf{and\ TF} &\longrightarrow_\beta^* \mathbf{F} \\ \mathbf{and\ FT} &\longrightarrow_\beta^* \mathbf{F} \\ \mathbf{and\ FF} &\longrightarrow_\beta^* \mathbf{F} \end{aligned}$$

例 1.7　作为一个例子，我们验证第一条性质，其他三条性质类似。

$$\begin{aligned} \mathbf{and\ TT} &\equiv (\lambda ab.aba)(\lambda xy.x)(\lambda xy.x) \\ &\longrightarrow_\beta (\lambda b.(\lambda xy.x)b(\lambda xy.x))(\lambda xy.x) \\ &\longrightarrow_\beta (\lambda xy.x)(\lambda xy.x)(\lambda xy.x) \\ &\longrightarrow_\beta (\lambda y.(\lambda xy.x))(\lambda xy.x) \\ &\longrightarrow_\beta (\lambda xy.x) \\ &\equiv \mathbf{T} \end{aligned}$$

在上面的推导过程中，用符号 $\equiv$ 表示语法等价（syntactic equivalence），即如果按定义展开，该符号左右两边的表达式在语法上一模一样。

由于 **T** 和 **F** 是范式，因此可以说 **and TT** 求值（evaluate）到 **T**。针对上面定义的 **T** 和 **F**，可见 **and** 能完成所需要的运算，因此是对“与”运算的一个有效编码。这里需要注意两点：

- 项 **and** 构成对“与”运算有效编码的前提是“真”“假”值分别用 **T**、**F** 编码。如果布尔值用其他项 M、N 表示，就不能保证 **and** MN 求值到合适的项。
- 即使对于上面定义的 **T** 和 **F**，“与”运算的编码也不唯一，比如项 $\lambda ab.bab$ 也符合要求。

练习 1.11 把“否定”“或”“异或”运算定义成下面的项：

$$\begin{aligned} \mathbf{not} &\stackrel{\text{def}}{=} \lambda a.a\mathbf{FT} \\ \mathbf{or} &\stackrel{\text{def}}{=} \lambda ab.aab \\ \mathbf{xor} &\stackrel{\text{def}}{=} \lambda ab.a(b\mathbf{FT})b \end{aligned}$$

验证这几个编码的有效性。另外，对这几个运算尝试给出和上面不一样的编码方式。

练习 1.12 布尔值还可用在条件判断中，对于条件判断，定义

$$\mathbf{if\_then\_else} \stackrel{\text{def}}{=} \lambda x.x$$

验证这个项具有如下所期望的行为，即对任何项 M 和 N，有

$$\begin{aligned} \mathbf{if\_then\_else}\ \mathbf{T}MN &\longrightarrow_\beta^* M \\ \mathbf{if\_then\_else}\ \mathbf{F}MN &\longrightarrow_\beta^* N \end{aligned}$$

自然数 如果 f 和 x 是两个 λ-项，n 是一个自然数，用记号 $f^n x$ 代表把 f 作用 n 次到 x 上所得到的项。例如 $f^0 x = x$，$f^1 x = fx$，$f^2 x = f(fx)$ 等。对每个自然数 n，定义第 n 个邱奇数（Church numeral）为一个 λ-项，$\overline{n} \stackrel{\text{def}}{=} \lambda fx.f^n x$。从 0 开始的前几个邱奇数如下：

$$\begin{aligned} \overline{0} &\stackrel{\text{def}}{=} \lambda fx.x \\ \overline{1} &\stackrel{\text{def}}{=} \lambda fx.fx \\ \overline{2} &\stackrel{\text{def}}{=} \lambda fx.f(fx) \\ &\vdots \end{aligned}$$

邱奇数是在 λ-演算中对自然数的编码表示，这种表示方法是邱奇提出来的。这里，$\overline{0}$ 和之前定义的 **F** 是 α-等价的。接下来需要编码自然数集合上的一些常用函数。我们把后继函数编码为项 **succ**，其中

$$\mathbf{succ} \stackrel{\text{def}}{=} \lambda nfx.f(nfx)$$