Lambda 微积分教程介绍

Raul Rojas* 柏林

自由大学 ¨

2.0 版，2015 年

摘要

本文简明扼要地介绍了 λ 微积分。这种形式主义是由 Alonzo Church 开发的，是研究有效可计算函数数学特性的工具。该形式主义广受欢迎，为函数式编程语言系列奠定了坚实的理论基础。本教程介绍了如何使用 λ *微积分*执行算术和逻辑运算，以及如何定义递归函数，尽管 λ *微积分*函数没有名称，因此无法明确指向自身。

1 定义

λ 微积分可谓*世界上最小的通用编程语言*。λ 微积分由一条转换规则（变量替换，也称为 β 转换）和一个函数定义组成

*请将更正或建议发送至 rojas@inf.fu-berlin.de

方案。它是 20 世纪 30 年代由阿朗佐-丘奇提出的，作为一种将有效可计算性概念具体化的方法。从任何可计算函数都可以用这种形式主义来表达和评估的意义上讲，*λ计算*是通用的。因此，它等同于图灵机。然而，*λ算元*强调使用符号转换规则，并不关心实际的机器实现。它是一种与软件而非硬件更为相关的方法。

λ-微积分的核心概念是 "表达式"。名称 "是一个标识符，就我们的目的而言，它可以是 a、b、c 等字母中的任何一个。表达式可以是一个名称，也可以是一个函数。函数使用希腊字母 λ 来标记函数参数的名称。函数的 "主体 "指定如何重新排列参数。例如，身份函数用字符串 (λx.x) 表示。片段 "λx "告诉我们函数的参数是 x，函数返回的 参数 "x "不变。

函数可以应用于其他函数。例如，将函数 A 应用于函数 B，就可以写成 AB。本教程使用大写字母表示函数。事实上，在 λ- 微积分中，任何感兴趣的东西都是函数。即使是数字或逻辑值，也可以用函数来表示，这些函数可以互相作用，将一串符号转换成另一串符号。λ- 微 积 分 中没有类型：任何函数都可以作用于任何其他函数。程序员负责保持计算的合理性。

表达式的递归定义如下

<表情 >	:=	<名称> | < 功能> | < 应用 >
<功能 >	:=	λ< 名称> .< 表达式 >
<应 用 >	:=	<表达 ><>

为了清晰起见，表达式可以用括号括起来，也就是说，如果 E 是一个表达式，(E) 就是同一个表达式。否则，语言中使用的关键词只有 λ 和点。为了避免用括号使表达式杂乱无章，我们采用了这样的约定：函数应用关联

从左边开始，即综合表达式

$$E_{(1)}\, E_{(2)}\, E_{(3)} \ldots E_n$$

应用连续表达式进行计算，如下所示

$$\ldots(E_{(1)}\, E_{2)}\, E_3 \ldots\ldots E_{(n})$$

从 λ-*expression* 的定义中可以看出，前面提到的字符串（无论是否用括号括起来）就是一个形式良好的函数示例：

$$\lambda x.x \equiv (\lambda x.x)$$

我们使用等价符号 "≡ " 表示当 A≡ B 时，A 只是 B 的同义词。如上所述，λ 后面的名称是该函数参数的标识符。点后面的表达式（此处为单个 x）称为函数定义的 "主体"。

函数可以应用于表达式。一个简单的应用例子是

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

这是应用于变量 y 的标识函数。在函数定义的正文中，通过替换参数 x 的 "值"（本例中为被处理的 y）来评估函数应用。图 1 显示了变量 y 是如何被函数 "吸收 " 的（红线），同时也显示了变量 y 被用来替代 x 的地方（绿线）。结果是右箭头所示的还原，最终结果为 y。

由于我们不可能总是拥有图 1 ，因此中的图片使用符号 $[y/x]$来表示函数体中所有出现的 x 都被 y 取代。例如，我们写道：$(\lambda x.x)y \rightarrow [y/x]x \rightarrow y$。函数定义中的参数名称本身没有任何意义。它们只是 "占位符"，也就是说，它们用来表示函数求值时如何重新排列参数。因此，下面所有字符串代表的是同一个函数：

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

这种纯字母替换也称为 α 还原。

$$(\lambda x.x)y \qquad (\lambda\ .\nabla\nabla\ )y$$

$$\to y \qquad \to y$$

图 1: 两次显示的是同一个还原过程。函数参数的符号只是一个占位符。

1.1 自由变量和绑定变量

如果我们只有 λ 表达式管道的图片，就不必关心变量的名称了。由于我们使用字母作为符号，因此在重复使用时必须小心谨慎，如本节所示。

在 λ-calculus 中，所有名称都是定义的局部名称（就像在大多数编程语言中一样）。在函数 λx.x 中，我们说 x 是 "绑定 "的，因为它在定义体中出现时前面有 λx。前面没有 λ 的名称称为 "自由变量"。在表达式

$$(\boldsymbol{\lambda}\mathbf{x}.\mathbf{x})(\boldsymbol{\lambda}\mathbf{y}.yx)$$

左起第一个表达式正文中的 x 与第一个 λ 绑定，第二个表达式正文中的 y 与第二个 λ 绑定，后面的 x 是自由的。绑定变量以粗 体 显示。值得注意的是，第二个表达式中的 x 与第一个表达式中的 x 完全无关。如图 2 。所示，如果我们画出函数应用的 "管道 "以及随后的还原，就能更容易地看出这一点

在图 2 中，我们可以看到符号表达式（第一行）如何被解释为一种电路，其中绑定参数被移动到函数体内部的一个新位置。第一个函数（标识函数）"消耗 "了第二个函数。第二个函数中的符号 x 与表达式的其他部分没有任何联系，它在函数定义中是自由浮动的。

$$(\lambda x.x)(\lambda y.yx)$$

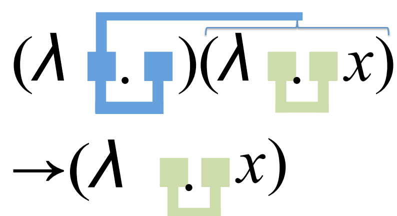

图 2： 连续行：函数应用、符号表达式的 "管道 "以及由此产生的缩减。

从形式上说，如果以下三种情况之一成立，我们就说表达式中的 变量< 名> 是自由的
：

- <名称> 在< 名称> 中是自由
  的。(例如：a 在 a 中是空闲的
  ）。

- <> 在 λ<name$_1$> 中是空闲的。< exp> 如果标识符< name />=< name$_{(1)}$ >
  和< 名称> 在< exp> 中是自由的（例如：
  y 在 λx.y 中是自由的）。

- <名称> 在 $E_{(1)}E_2$中是自由的，如果< 名称> 在 E_1中是自由的，或者在 E_2
  中是自由的。（例如：x 在 (λx.x)x 中是自由的）。

如果两种情况之一成立，变量< 名称> 将被绑定：

- <name> 被绑定在 λ< name$_1$> .< exp> 如果标识符< name>=< name($_1$) >
  或如果< 名称> 绑定在< exp> 中。（例如
  ：x 绑定在 (λy.(λx.x)) 中）。

- <如果< 名称> 绑定在 E(1) E(2) 中，则> 名称 绑定在 $E_1E_{(2)}$ 中。
  (例如：x 在 (λx.x)x 中受约束）。

需要强调的是，在同一个表达式中，同一个标识符既可以是自由标识符，也可以是绑定标识符。在表达式

$$(\boldsymbol{\lambda}\mathbf{x}.xy)(\boldsymbol{\lambda}\mathbf{y}.\mathbf{y})$$

第一个 y 在左边的括号子表达式中是自由变量，但在右边的子表达式中是绑定变量。因此，它在整个表达式中既是自由变量，也是绑定变量（绑定变量用粗体表示）。

1.2 替换

初次接触标准 λ *微积分*时，比较令人困惑的部分是我们不给函数命名。当我们要应用一个函数时，我们只需写出完整的函数定义，然后对其进行求值。不过，为了简化符号，我们会使用大写字母、数字和其他符号（san serif）作为某些函数的同义词。例如，同一函数可以用字母 I 表示，将其作为 $(\lambda\mathrm{x}.x)$ 的简称。

应用于自身的同一函数是

$$\mathsf{II} \equiv (\lambda\mathrm{x}.x)(\lambda\mathrm{x}.x).$$

在这个表达式中，括号内第一个函数体中的第一个 x 与第二个函数体中的 x 无关（请记住，"管道 "是局部的）。为了强调两者的区别，我们实际上可以将上述表达式重写为

$$\mathsf{II} \equiv (\lambda\mathrm{x}.x)(\lambda\mathrm{z}.z).$$

应用于自身的同一函数

$$\mathsf{II} \equiv (\lambda\mathrm{x}.x)(\lambda\mathrm{z}.z)$$

因此

$$[(\lambda\mathrm{z}.z)/x]x \rightarrow \lambda z.z \equiv \mathsf{I}、$$

也就是同一函数。

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

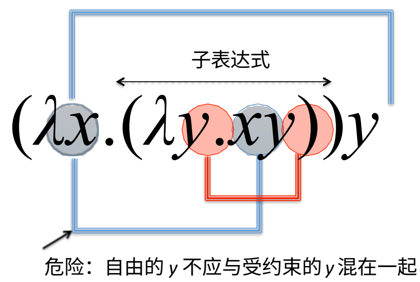

图 3：自由变量不应被替换到绑定它的子表达式中，否则会产生一个与原来不同的新 "管道"。

在进行替换时，我们应注意避免将标识符的自由出现与绑定出现混淆。在表达式

$$\mathbf{\lambda x.(\lambda y.xy)}\ y$$

左边的函数包含一个绑定*的* y，而右边的 y 是自由的。错误的替换会将这两个标识符混合在一起，产生错误的结果

$$(\mathbf{\lambda y.yy}).$$

只需将边界 y 重命名为 t，我们就可以得到

$$\lambda x.(\lambda t.xt)\ y \rightarrow (\lambda t.yt)$$

这是一个完全不同的结果，但却是正确的。因此，如果将函数 λx.< exp> 应用于

E，我们就可以将所有*自由的*

< exp> 中 x 的出现次数与 E。

在一个表达式中，如果 E 的变量出现绑定，我们将其重命名为

在执行替换之前，必须先删除绑定变量。例如，在表达式

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

我们将第一个 x 与 y 关联起来。

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

只有第一个 x 是自由的，可以被替换。不过，在替换之前，我们必须重新命名变量 y，以避免将其绑定与自由出现混淆：

$$[y/x]\ (\lambda t.(x(\lambda x.xt))) \rightarrow (\lambda t(y(\lambda x.xt)))$$

在正序还原法中，我们总是先还原一系列应用中最左边的表达式。直到无法继续还原为止。

2 算术

编程语言应该能够指定算术运算。在 λ *微积分*中，数字可以从 0 开始表示，并用 "0 的后继数"，即 "suc(zero) "来表示 1，用 "suc(suc(zero)) "来表示 2，以此类推。由于在 λ *微积分*中我们只能定义新的函数，因此将采用以下方法把数定义为函数：零可以定义为

$$\lambda s.(\lambda z.z)$$

这是一个包含两个参数 s 和 z 的函数，我们将把这种包含多个参数的表达式缩写为

$$\lambda sz.z$$

这里的理解是，s 是求值时要替换的第一个参数，z 是第二个参数。使用这种符号，第一个自然数可以定义为

$$\begin{aligned} 0 &\equiv \lambda sz.z \\ 1 &\equiv \lambda sz.s(z) \\ 2 &\equiv \lambda sz.s(s(z)) \\ 3 &\equiv \lambda sz.s(s(s(z))) \end{aligned}$$

等等。

用这种方法定义数字的最大好处是，我们现在可以对参数 a 应用任意次数的函数 f。例如，如果我们想把函数 f 应用于 a 三次，我们就可以把函数 3 应用于参数 f 和 a，从而得到：

$$3fa \rightarrow (\lambda sz.s(s(sz)))fa \rightarrow f(f(fa))。$$

这种定义数字的方式为我们提供了一种语言结构，类似于其他语言中的 "FOR i=1 to 3 "指令。将数字 0 应用于参数 f 和 a 可以得到 $0fa \equiv (\lambda sz.z)fa \rightarrow a$。也就是说，将函数 f 应用于参数 a 0 次后，参数 a 保持不变。

在定义了自然数之后，我们第一个感兴趣的函数是后继函数。它可以定义为

$$S \equiv \lambda nab.a(nab)。$$

这个定义看起来很别扭，但却行之有效。例如，应用于我们对零的表示的后继函数是表达式：

$$S0 \equiv (\lambda nab.a(nab))0$$

在第一个表达式的正文中，我们用 0 代替出现的 n
这样就得到了简化的表达式：

$$\lambda ab.a(0ab) \rightarrow \lambda ab.a(b) \equiv 1$$

也就是说，结果就是数字 1 的表示（请记住，绑定的变量名是 "假名"，可以更改）。

继承人适用于 1 个孳息：

$$S1 \equiv (\lambda nab.a(nab))1 \rightarrow \lambda ab.a(1ab) \rightarrow \lambda ab.a(ab) \equiv 2$$

请注意，将数字 $1 \equiv (\lambda sz.sz)$ 应用于参数 a 和 b 的唯一目的，是 "重命名 "数字定义中内部使用的变量。

2.1 加法

如果我们想把 2 和 3 相加，只需在 3 上应用两次后继函数即可。

让我们试着用下面的方法计算 2+ 3：

$$2S3 \equiv (\lambda sz.s(sz)))S3 \rightarrow S(S3) \rightarrow \text{- - } \rightarrow 5$$

一般来说，m 加 n 可以通过表达式 mSn 计算出来。

2.2 乘法

两个数字 x 和 y 的乘法运算可以用下面的函数计算：

$$(\lambda xya.x(ya))$$

那么 3 乘 3 的积是

$$(\lambda xya.x(ya))33$$

这还原为 $$(\lambda a.3(3a))$$

利用数字 3 的定义，我们可以将上述表达式进一步简化为

$$(\lambda a.(\lambda sb.s(s(sb)))(3a)) \rightarrow (\lambda ab.(3a)((3a)((3a)b)))$$

为了理解为什么这个函数真正计算的是 3 乘 3 的乘积，让我们来看一些示意图。第一次应用 $(3a)$ 是在图 4 的左边计算的。 注意，将 3 应用于 a 的效果是产生了一个新函数，它将 a 三次应用于函数参数。

现在，将函数 3 应用于 $(3a)$ 的结果，会产生图 4 中得到的函数的三个副本，如图 4 右侧所示进行连接

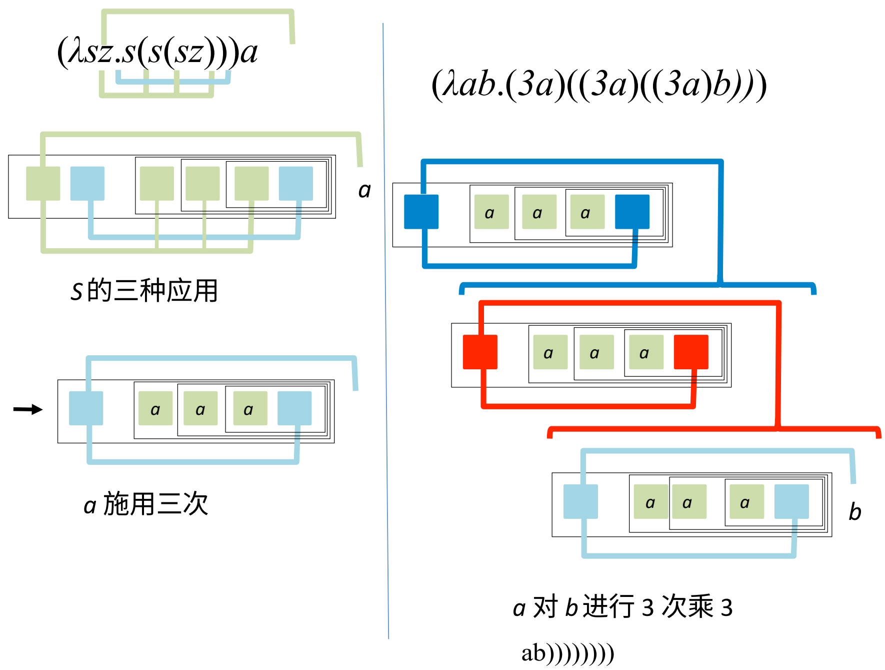

图 4： 左：数字 3 应用于参数 a 会产生一个新函数。右图将函数 3 的管道应用于 3a，并将结果应用于 b。

(为清晰起见，结果应用于 b）。请注意，我们有一个由三次相同函数组成的 "塔" ，每一次都吸收了下一次函数作为参数，对函数 a 进行了三次应用，总共九次应用。

3 条件式

我们引入以下两个函数，并将其值称为 "true"

$$\mathrm{T} \equiv \lambda xy.x$$

和 "假"

$$F \equiv \lambda xy.y$$

第一个函数接收两个参数，并返回第一个参数。第二个函数返回两个参数中的第二个参数。

3.1 逻辑运算

现在，我们可以使用这种真值表示法来定义逻辑运算。

两个参数的 AND 函数可定义为

$$\wedge \equiv \lambda xy\ .xyF$$

如果 x 为假（因 此 选择了 xyF 中的第二个参数），则无论 y 的值如何，完整的 AND 运算都是假的。

两个参数的 OR 函数可定义为

$$\vee \equiv \lambda xy\ .xTy$$

在这里，如果 x 为真，则 OR 为真。如果 x 为假，它会选择第二个参数
一个参数的否定可以定义为

$$\neg \equiv \lambda x\ .xFT$$

例如，应用于 "true "的否定函数是

$$\neg T \equiv (\lambda x.xFT)T$$

这还原为

$$TFT \equiv (\lambda cd.c)FT \rightarrow F$$

即真值为 "假"。

有了这三个逻辑函数，我们就可以对任何其他逻辑函数进行编码，并在没有反馈的情况下重现任何给定的电路（我们将在处理递归时讨论反馈）。

3.2 条件测试

在编程语言中，如果有一个函数在数字为零时为真，否则为假，那就非常方便了。下面的函数 Z 就具有这种功能：

$$Z \equiv \lambda x.xF\ F\neg$$

要了解该函数的工作原理，请记住

$$0fa \equiv (\lambda sz.z)fa = a$$

也就是说，将函数 f 应用于参数 a 的次数为零，会得到 a。另一方面，将 F 应用于任何参数，都会得到同一函数

$$Fa \equiv (\lambda xy.y)a \rightarrow \lambda y.y \equiv I$$

现在我们可以测试函数 Z 是否正常工作。将函数应用于零，可以得到

$$Z0 \equiv (\lambda x.xF\neg\ F)0 \rightarrow 0F\ F\neg \rightarrow \neg\ \ F \rightarrow T$$

因为 F 对¬ 施加 0 次，得到¬ 。将函数 Z 应用于任何其他数 N 都会得到

$$ZN \equiv (\lambda x.xF\neg\ F)N \rightarrow NF\ F\neg$$

然后将函数 F 应用 N 次到¬ 上。但 F 应用到任何地方都是等式（如前所述），因此，对于任何大于零的数字 N，上述表达式都简化为

$$IF \rightarrow F$$

3.3 前置函数

现在我们可以结合上面介绍的一些函数来定义前置函数。在查找 n 的前代时，一般的策略是创建一对（$n, n-1$），然后选取其 中 的第二个元素作为结果。

在 λ 微积分中，一对 (a, b) 可以用以下函数表示

$$(\lambda z.zab)$$

我们可以从对 T 应用该函数的表达式中提取出一对元素中的第一个元素

$$(\lambda z.zab)\mathrm{T}\rightarrow Tab\rightarrow a、$$

而第二个函数则将该函数应用于 F

$$(\lambda z.zab)\mathrm{F}\rightarrow Fab\rightarrow b。$$

下面的函数从一对（$n, n-1$）（即函数中的参数 p）生成一对（$n+1, n$）：

$$\Phi\equiv(\lambda pz.z(\mathrm{S}(\mathrm{pT}))(\mathrm{pT}))$$

子表达式 pT 从数据对 p 中提取第一个元素，并用这个元素组成一个新的数据对，新数据对的第一个位置上的元素会递增，新数据对的第二个位置上的元素会被复制。

将函数 Φ 应用于数对（λ.z00）n 次，然后选择新数对的第二个成员，就得到了一个数 n 的前置数：

$$\mathrm{P}\equiv(\lambda n.(n\Phi(\lambda z.\mathrm{z}00))\mathrm{F})$$

请注意，使用这种方法时，零的前置数为零。这一特性对于定义其他函数非常有用。

3.4 平等与不平等

有了前置函数这个基础模块，我们现在就可以定义一个函数，测试数字 x 是否大于或等于数字 y：

$$\mathrm{G}\equiv(\lambda xy.\mathrm{Z}(xPy))$$

如果对 y 应用 x 次前置函数，结果为零，那么 $x \geq y$ *是*真的。

如果 $x \geq y$，$y \geq x$，那么 $x = $ y。由此可以得出检验两个数是否相等的函数 E 的定义如下：

$$E \equiv (\lambda xy. \wedge (Z(xPy))(Z(yPx)))$$

同样，我们也可以定义函数来测试 $x > y$、$x < y$ 或
$x \neq y$。

4 递归

递归函数可以在 λ *微积分*中使用调用函数 y 然后再生自身的函数来定义。考虑下面的函数 Y，可以更好地理解这一点：

$$Y \equiv (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$ 将此

函数应用于函数 R 即可得到

$$YR \equiv (\lambda x.R(xx))(\lambda x.R(xx))$$ 进

一步缩小得到

$$R((\lambda x.R(xx))(\lambda x.R(xx))$$

但这意味着 YR→ R(YR)，也就是说，函数 R 是以递归调用 YR 作为第一个参数来求值的。

例如，一个无限循环可编程为 YI，因为它可还原为 I(YI)，然后还原为 YI，如此无限循环。

更有用的函数是将前 n 个自然数相加。我们
可以使用递归定义，因为 $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$。让我们使用
R 的定义如下

$$R \equiv (\lambda rn.Zn0(nS(r(Pn))))$$

这个定义告诉我们，要对数字 n 进行测试：如果 n 为零，求和的结果就是零。如果 n 不为零，那么后继函数将对 n 的前代函数的递归调用（参数 r）应用 n 次。

由于 λ 微积分中的函数没有名称，我们如何知道上面表达式中的 r 是对 R 的递归调用呢？我们不知道，而这正是我们必须使用递归操作符 Y 的原因：

yr3→ r(yr)3→ z30(3s(yr(p3)))

由于 3 不等于零，因此计算结果相当于

3S(YR(P3))

也就是说，从 0 到 3 的数字之和等于 3 加上从 0 到 3 的前位数（即 2）的数字之和。对 YR 的连续递归求值将导致正确的最终结果。

注意，在上面定义的函数中，当参数变为 0 时，递归将停止。 最终结果将是

3S(2S(1S0))

即数字 6。

5 读者项目

1. 定义两个数字参数的 "小于 "和 "大于 "函数。
2. 用自然数对定义正整数和负整数。
3. 定义整数的加法和减法。
4. 递归定义正整数的除法。

5. 递归定义函数 $n! = n \cdot (n-1) \cdot \cdot \cdot 1$。

6. 将有理数定义为一对整数。

7. 定义有理数加、减、乘、除的函数。

8. 定义表示数字列表的数据结构。

9. 定义一个从列表中提取第一个元素的函数。

10. 定义一个递归函数，计算列表中元素的个数。

11. 你能用λ演算法模拟图灵机吗?

参考资料

[1] P.M. Kogge，《*符号计算机体系结构*》，麦格劳-希尔，纽约，1991 年，第 4 章。

[2] G. Michaelson, *An Introduction to Functional Programming through λ- calculus*, Addison-Wesley, Wokingham, 1988.

[3] G. Revesz，《*Lambda-Calculus 组合器与函数式编程*》，剑桥大学出版社，剑桥，1988 年，第 1-3 章。