# $\LaTeX$ Document Test Page

duangsuse

December 15, 2018

## 1 Made by duangsuse with love and $\LaTeX 2_\varepsilon$

Definition for the Y Combinator

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \tag{1}$$

Sample Matrix

$$Matrix_0 = \begin{Bmatrix} 1 & 9 & 3 & 5 & 10 \\ 3 & \mathbf{5} & 9 & 4 & 71 \\ 2 & 3 & 1 & 9 & 34 \\ 9 & 4 & 3 & 2 & 29 \\ 2 & 8 & 4 & 3 & 12 \end{Bmatrix} \tag{2}$$

$$Matrix_{022} \equiv 5 \tag{3}$$

$$\sum_{i=2}^{4} \sum_{j=1}^{4} Matrix_{0ij} \tag{4}$$

## 2 Made by others with unbelievable IQ

$$x = -b \pm \sqrt{b^2 - 4ac}\frac{2}{a}. \tag{5}$$

$$d_i = \sum_{j=1}^{n} a_{ij} \tag{6}$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{7}$$

$$(\nabla_X Y)^k = X^i (\nabla_i Y)^k = X^i \left( \frac{\partial Y^k}{\partial x^i} + \Gamma^k_{im} Y^m \right) \tag{8}$$

$$\vec{\nabla} \times \vec{F} = \left( \frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \mathbf{i} + \left( \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \mathbf{j} + \left( \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \mathbf{k} \tag{9}$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ \frac{100-x}{100} & 0 \leq x \leq 100 \\ 0 & 100 \leq x \end{cases} \tag{10}$$

$$z = \overbrace{\underbrace{x}_{\text{real}} + i \underbrace{y}_{\text{imaginary}}}^{\text{complex number}} \tag{11}$$

$$C_n^i = \frac{n!}{i!(n-i)!} \tag{12}$$

$$B_{i,n}(t) = C_n^i (1-t)^{n-i} t^i \tag{13}$$

$$R(t) = \sum_{i=0}^{n} R_i B_{i,n}(t), \quad 0 \leq t \leq 1 \tag{14}$$

## 3 $\lambda$ - 算子

早在上世纪 30 年代, 理论上早已出现了许多种不能跑在当下计算机中的 "编程语言", 甚至编程语言这个概念还尚未成熟, 当时人们主要研究 formal system (形式系统) 这一领域. 而这其中最为突出的成果为图灵的导师 Alonzo Church 的 $\lambda$-算子, 它是 FP 中非常重要的理论基石, 之后的文章也都全部围绕这一系统进行讨论.

在讨论 $\lambda$ 表达式之前, 我们需要一个 context (上下文), 它类似于我们所说的集合论中的 universe 集合 (全集) $U$, 它保证了我们的论域中拥有独一无二的元素, 写作 $\Gamma$. $\Gamma$ 可以是一个空集 $\emptyset$, 也可以是引入了某元素后的集合, 以此类推. 类比一般的高级编程语言, 它就像符号表, 像一个装有独特变量的相关信息的 哈希表, 甚至也可以是放有变量名的 (数据结构里的) 集合, 当然这些只是比喻.

我们定义一个 $\lambda$ 函数, 在 $\Gamma$ 下引入, 形为

$$\Gamma \vdash \lambda x.x$$

它类似数学中常用的函数 $f(x) = x$, 但这种表示引入了函数名称 $f$, 而 $\lambda$ 函数是 匿名的, 两者显然并不等同. 这种最基本的 输入即输出 的函数, 被称作 identity function.

和数学中常用的函数表示更加不同的是, 一个 $\lambda$ 函数可以返回另一个 $\lambda$ 函数, 而且对上一个函数的参数是可见的, 即以下函数表示是合法的

$$\lambda x.\lambda y.x$$

简记为

$$\lambda xy.x$$

这一种特性叫做 lexical scoping. 有了这一种特性, 我们便可以做到多参数输入, 单结果输出了. 那具体是怎么做的呢? 首先我们先看一个简单的场景

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

这句话的意思是, 给定一个 context 叫 $\Gamma$, 我们将两个 $\lambda$ 函数添加到 $\Gamma$ 之中. 接着我们用第一个函数的参数 $x$ 代替第二个函数的整体, 按照第一个函数的 "执行规则", 返回一个新的 term (项), 即

$$(\lambda x.x)(\lambda y.y) =_\beta \lambda y.y$$

这一句话的 "执行结果", 即第二个函数本身. 前面将函数加入到 context 中的过程, 我们简称 eval (即 evaluate), 而实际执行这些函数的过程, 简称 apply, 而之前所提到的 "执行规则", 称作 $\beta$-reduction, 它属于一种 term rewriting rules (项的重写规则). 可见, 在 $\lambda$-算子 这个系统中, 最基本的操作即 eval/apply 两个过程的循环.

至于刚刚提到的 多参数输入, 单结果输出, 考虑以下的例子

$$\Gamma \vdash (\lambda xy.x)(\lambda a.a)(\lambda b.b)$$

其 apply 的过程即

$$(\lambda xy.x)(\lambda a.a)(\lambda b.b) \equiv (\lambda x.\lambda y.x)(\lambda a.a)(\lambda b.b) =_\beta (\lambda y.(\lambda a.a))(\lambda b.b) =_\beta \lambda a.a$$

这么一个过程. 要注意的是, 在习惯中, 项的 apply 是 left recursive (左递归) 的, 即 $abc$ 是 $((ab)c)$ 的习惯表达, 而不是 右递归的 $(a(bc))$.

有 $\beta$-reduction 的规则, 是否还存在 $\alpha$-X 这样的规则呢? 这里补充下, 这套系统中还存在有 $\alpha$-conversion 的规则, 在形如

$$\Gamma \vdash (\lambda x.x)(\lambda x.x)$$

的情况下, 这两个项在 $\Gamma$ 中是独一无二的, 但是为了区分二者, 我们可以用如 $y$ 等其他符号替代, 这一种 conversion (转换) 即称为 $\alpha$-conversion, 转换的过程为

$$(\lambda x.x)(\lambda x.x) =_\alpha (\lambda x.x)(\lambda y.y)$$