Lambda Calculus

2022 年 6 月 16 日

1 Lambda Calculus for Humans

1.1 前言

这是一个写给正常人学习的 演算教程，如果你搜索「Lambda 演算」，维基百科上的说明长这样：

```
0 =  f. x.x
1 =  f. x.f x
2 =  f. x.f (f x)
3 =  f. x.f (f (f x))
```

然后你搜索「A Tutorial Introduction to the Lambda Calculus」，得到的结果更抽象了：

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

研究表明，长时间观看这些抽象的算符会造成心理上的不适，并容易产生智商上的挫败感，这对正常人类的心智是有害的。

因此我肩负着拯救人类的使命，写下了这一份「给正常人看的 -calculus 教程」。

1.2 函数式编程

首先，让我们摒弃掉那些 -conversion、-reduction、currying 这些让人 san 值狂掉的术语，先来介绍一个更广为人知的术语「函数式编程」(Functional Calculus)，它的核心思想很简单：**用函数来表**

达一切。

传统编程范式中，我们面向的对象主要是变量，而在函数式编程中，我们面向的对象是函数，举个例子，我们定义一个 now 函数打印日期

[1]:
```python
def now(data):
    print('Today is: ' + data)

data = '2022-6-11'
now(data)
```

```
Today is: 2022-6-11
```

这就是一个非常传统的面向变量的函数：输入一个变量，然后对变量进行操作。

函数式编程输入的就是一个函数，然后对函数进行操作，比如我们定义一个 log 函数打印输入函数的函数名：

[2]:
```python
def log(func):
    print(func.__name__)

log(now)
```

```
now
```

但是这样有一点不好，那就是我们没法在调用 now 函数的同时也打印出函数名，因此我们将 log 函数稍微修改一下：

[3]:
```python
def log(func):
    def F(data):
        print('call %s():' % func.__name__)
        return func(data)
    return F

log(now)(data)
```

```
call now():
Today is: 2022-6-11
```

这样将 log 函数变成了一个二阶函数，于是一切就 OK 了。

PS：其实在 Python 中有一个更优雅的写法，那就是「装饰器」(Decorator)，本质就是额外执行了

一个 now = log(now)

[4]:

```
@log
def now(data):
    print('Today is: ' + data)

now(data)
```

```
call now():
Today is: 2022-6-11
```

1.3 逻辑运算

1.3.1 数字编码

在计算机中，我们可以用数字来编码 `False` 和 `True`

$$
\begin{aligned}
\text{False} &\quad\longleftrightarrow\quad 0 \\
\text{True} &\quad\longleftrightarrow\quad 1
\end{aligned}
$$

逻辑运算 `not`、`and` 和 `or` 也可以基于数字来实现

$$
\begin{aligned}
\text{not } x &\quad\longleftrightarrow\quad 1-x \\
x \text{ and } y &\quad\longleftrightarrow\quad xy \\
x \text{ or } y &\quad\longleftrightarrow\quad x+y-xy
\end{aligned}
$$

那么问题就来了，我们可不可以用函数来编码 `True` 和 `False` 呢？

1.3.2 True, False 的定义

在传统函数思想中，True 和 False 都是 Bool 变量，但在 -演算中，所有的对象都是一个函数。

因此我们使用两个函数来表示 True 和 False：

$$
\text{TRUE}(x,y) \;=\; x, \qquad \text{FALSE}(x,y) \;=\; y
$$

为什么要这样定义呢？我们暂且不纠结这个问题，先来看下面这段代码

[5]:
```
def TRUE(x,y):
    return x

def FALSE(x,y):
    return y

print(TRUE(TRUE,FALSE).__name__)
print(TRUE(FALSE,TRUE).__name__)
print(FALSE(TRUE,FALSE).__name__)
print(FALSE(FALSE,TRUE).__name__)
```

```
TRUE
FALSE
FALSE
TRUE
```

我们看到了什么？`TRUE(TRUE,FALSE)` 居然返回了函数 `TRUE` 本身！

是不是有点感觉了，我们写一个 `show` 函数来作为解码器

[6]:
```
def show(f):
    print(f(TRUE,FALSE).__name__)

show(TRUE)
show(FALSE)
```

```
TRUE
FALSE
```

1.3.3 Not, And, Or

根据 True 和 False 的定义，我们将两个变量的位置进行交换，便可定义出 NOT 函数

$$\mathrm{NOT}(f) = f(\mathrm{FALSE}, \mathrm{TRUE})$$

[7]:
```
def NOT(f):
    return f(FALSE,TRUE)

show(NOT(TRUE))
```

```
show(NOT(FALSE))
```

FALSE

TRUE

NOT(TRUE) 返回了函数 FALSE，而 NOT(FALSE) 返回了函数 TRUE！

那么你先不往下看，能自己想象出怎么定义 AND 函数吗？

[8]:
```
def AND(f,g):
    pass
```

AND(f,g)==TRUE 函数要求 f 和 g 都是 TRUE，当 f 为 FALSE 时，会返回第二个参数，因为我们可以返回 f(?,f)。

如果 f 为 TRUE，返回第一个参数中的?，我们可以将? 设为 g 本身，即

$$\mathrm{AND}(f,g) \;=\; f(\,g,f)$$

[9]:
```
def AND(f,g):
    return f(g,f)


show(AND(TRUE,TRUE))
show(AND(TRUE,FALSE))
show(AND(FALSE,TRUE))
show(AND(FALSE,FALSE))
```

TRUE

FALSE

FALSE

FALSE

OR 函数也是类似的：

$$\mathrm{OR}(f,g) \;=\; f(f,g)$$

[10]:
```
def OR(f,g):
    return f(f,g)
```

```
assert OR(TRUE,TRUE) is TRUE
assert OR(TRUE,FALSE) is TRUE
assert OR(FALSE,TRUE) is TRUE
assert OR(FALSE,FALSE) is FALSE
```

1.3.4 柯里化 (Currying)

柯里化就是将所有的函数变成只有一个参数，例如一个双参数函数 $f(x,y)$，我们将其变成一个二阶函数 $F(x)(y)$

[11]:
```
def f(x,y):
    return x+y

def F(x):
    def Fx(y):
        return x+y
    return Fx

print(f('x','y'))
print(F('x')('y'))
```

```
xy
xy
```

我们可以将之前的逻辑函数都写成柯里化的形式

$$N(f,x,y) \;=\; f(y,x)$$

$$A(f,g,x,y) \;=\; f(\,g(x,y),y)$$

$$O(f,g,x,y) \;=\; f(x,\,g(x,y))$$

[12]:
```
def T(x):
    return lambda y: x
```

```
def F(x):
    return lambda y: y

def N(f):
    return f(F)(T)

def A(f):
    return lambda g: f(g)(f)

def O(f):
    return lambda g: f(f)(g)

assert T(True)(False) is True
assert F(True)(False) is False
assert N(T) is F
assert N(F) is T
assert A(T)(T) is T
assert A(T)(F) is F
assert O(T)(F) is T
assert O(F)(F) is F
```

我们也可以写一个 `curry` 函数将 `f(x,y)` 转化成柯里化的版本 `F(x)(y)`

[13]:
```
def curry(f):
    return lambda x: lambda y: f(x,y)

F = curry(f)
print(F('x')('y'))
```

xy

在经过非平凡的思考（参考 https://zh.javascript.info/currying-partials ）之后，我们可以写出柯里化函数 `curry` 的通用形式：

[14]:
```
def curry(f):
    def curried(*args):
#         print('---',*args,'---')
        if (len(args) == f.__code__.co_argcount):
```

```
            return f(*args)
        else:
            return lambda *args2: curried(*args,*args2)
    return curried
```

这里的 f.__code__.co_argcount 返回 f 定义时的参数个数，例如 f3.__code__.co_argcount=3。

这段代码的思想就是当 f 的输入参数个数 len(args) 与定义参数个数 f.__code__.co_argcount 一致时，就直接输出 f(*args)。

当参数不够时，就先把当前参数 *args 传进去，再递归调用外面一层的参数 *args2。

[15]:
```
f1 = lambda x: x
F1 = curry(f1)
print(F1('x'))

f2 = lambda x,y: x+y
F2 = curry(f2)
print(F2('x')('y'))

f3 = lambda x,y,z: x+y+z
F3 = curry(f3)
print(F3('x')('y')('z'))

f4 = lambda x,y,z,w: x+y+z+w
F4 = curry(f4)
print(F4(1)(2)(3)(4))
print(F4(1,2)(3)(4))
print(F4(1,2,3,4))
```

```
x
xy
xyz
10
10
10
```

如果你难以理解的话，可以取消掉 curry 函数中的注释，观察每次调用时传入的参数。

我们可以直接用 `curry` 函数将逻辑函数柯里化，当然，这和我们在上面直接定义的柯里化逻辑函数有着细微的区别，不过这并不重要。

[16]:
```
T = curry(TRUE)
F = curry(FALSE)
N = curry(NOT)
A = curry(AND)
O = curry(OR)

assert T(True)(False) is True
assert F(True)(False) is False
assert N(T) is FALSE
assert N(F) is TRUE
assert A(T)(T) is T
assert A(T)(F) is F
assert O(T)(F) is T
assert O(F)(F) is F

def show(f):
    print(f(TRUE,FALSE).__name__)

show(T)
show(F)
show(N(T))
show(A(T)(F))
show(O(T)(F))
```

```
TRUE
FALSE
FALSE
FALSE
TRUE
```

1.4 对自然数编码

既然能用函数对逻辑运算编码，当然也可以对自然数编码，我们通过如下方式定义：

- 每个自然数都是一个函数，它的输入和输出都是函数

- 函数 0 对于任何输入 $f(\cdot)$，输出都是一个恒等函数 $x \to x$
- 函数 1 对于输入 $f(\cdot)$，返回 f 本身
- 函数 2 对于输入 $f(\cdot)$，返回 f 的二阶函数 $f(f(\cdot))$

[17]:
```
def  ZERO(f): return lambda x: x
def   ONE(f): return lambda x: f(x)
def   TWO(f): return lambda x: f(f(x))
def THREE(f): return lambda x: f(f(f(x)))
def  FOUR(f): return lambda x: f(f(f(f(x))))
def  FIVE(f): return lambda x: f(f(f(f(f(x)))))
def   SIX(f): return lambda x: f(f(f(f(f(f(x))))))
```

[18]:
```
hi = lambda x: x + ' World'
x = 'Hello,'

print(ZERO(hi)(x))
print(ONE(hi)(x))
print(TWO(hi)(x))
print(THREE(hi)(x))
print(FOUR(hi)(x))
print(FIVE(hi)(x))
print(SIX(hi)(x))
```

```
Hello,
Hello, World
Hello, World World
Hello, World World World
Hello, World World World World
Hello, World World World World World
Hello, World World World World World World
```

如果我们定义零元为 0，每一个后继就是前驱元素 +1，那么就能得到常规形式的自然数：

[19]:
```
def Num(x):
    return x + 1
x = 0
print(ZERO(Num)(x))
print(ONE(Num)(x))
```

```
print(TWO(Num)(x))
print(THREE(Num)(x))
```

```
0
1
2
3
```

我们借此将函数版的自然数转化成常规形式的自然数

[20]:
```
def show(x):
    print(x(lambda x: x+1)(0))


show(ZERO)
show(ONE)
show(TWO)
show(THREE)
```

```
0
1
2
3
```

1.4.1 Next Number

显然，我们不可能手动定义所有的自然数函数，因此我们需要定义一个” 后继函数”：对于任意一个自然数 `A`，我们定义它的后继为 `NEXT(A)`。

我们可以先写出一个非柯里化版本：

[21]:
```
def NEXT(A,f,x):
    return f(A(f)(x))


fish = lambda x: x + ' fish'
x = 'I want:'
print(NEXT(THREE,fish,x))
```

```
I want: fish fish fish fish
```

如果要柯里化，那么我们可以使用前面定义的 `curry` 函数：

[22]:
```
NEXT_c = curry(NEXT)
print(NEXT_c(THREE)(fish)(x))
show(NEXT_c(THREE))
```

```
I want: fish fish fish fish
4
```

手写柯里化版本也很简单:

[23]:
```
def NEXT(A):
    return lambda f: lambda x: f(A(f)(x))


show(NEXT(THREE))
```

```
4
```

1.4.2 加法

我们要计算 A+B，以 A 为起点，计算 B 次 NEXT 即可

[24]:
```
def ADD(A,B):
    return B(NEXT)(A)


show(ADD(TWO,THREE))
```

```
5
```

显然我们的加法是满足交换律的

[25]:
```
show(ADD(ZERO,SIX))
show(ADD(ONE,FIVE))
show(ADD(TWO,FOUR))
show(ADD(THREE,THREE))
show(ADD(FOUR,TWO))
show(ADD(FIVE,ONE))
```

```
6
6
6
6
```

```
6
6
```

1.4.3 乘法

要计算 A*B，那么只需累加 A 次 B(f) 即可

[26]:
```
def MULTIPLY(A,B):
    return lambda f: A(B(f))

show(MULTIPLY(FOUR,THREE))
show(MULTIPLY(THREE,FOUR))
```

```
12
12
```

另一种思路是执行 A 次 ADD(0,B)

[27]:
```
def MULTIPLY(A,B):
    def ADD_B(C):
        return ADD(C,B)
    return A(ADD_B)(ZERO)

show(MULTIPLY(FOUR,THREE))
show(MULTIPLY(THREE,FOUR))
```

```
12
12
```

1.4.4 指数

A**B 就是将 A 重复执行 B 次

[28]:
```
def POWER(A,B):
    return B(A)

show(POWER(FOUR,THREE))
show(POWER(THREE,FOUR))
```

```
64
81
```

1.4.5 减法

关于减法，这并非一件简单的事情，我们首先要考虑的问题是怎么计算 `N-ONE`，也就是找到 A 的前驱，除非我们知道 `f` 的逆函数，并且逆是唯一的。

但对于一般的情况，我们的做法是定义一个函数

$$\Phi \ : \ (a,b) \ \to \ (b,b+1)$$

对于 `(ZERO,ZERO)` 作用 N 次就得到了

$$\Phi^N \ : \ (0,0) \ \to \ (N-1,N)$$

我们取第一个元素就得到了 `N-ONE`

[29]:
```
def PAIR(A,B):
    return lambda P: P(A,B)

def PHI(P):
    B = P(FALSE)
    return PAIR(B,NEXT(B))

def PRIOR(N):
    PHI_N = POWER(PHI,N)
    return PHI_N(PAIR(ZERO,ZERO))(TRUE)


show(PRIOR(FOUR))
```

```
3
```

在定义了前驱之后，就可以类似于加法一样定义减法了

[30]:
```
def SUBTRACT(A,B):
    return B(PRIOR)(A)


show(SUBTRACT(SIX,TWO))
show(SUBTRACT(POWER(TWO,FOUR),THREE))
```

```
4
13
```

当然，这也存在着一些问题，因为我们是从零元开始的，并没有定义零元之前的”负数”，因此用小数减大数得到的依然为零元

[31]:
```python
show(SUBTRACT(TWO,SIX))
```

```
0
```

1.4.6 判断零元

由零函数的定义可知，对于任何 `f`，都有 `ZERO(f)(x)==x`

[32]:
```python
f = lambda x: x + ' world'
x = 'Hello,'
print(ZERO(f)(x))
print(ONE(f)(x))
print(TWO(f)(x))
print(THREE(f)(x))
```

```
Hello,
Hello, world
Hello, world world
Hello, world world world
```

因此我们可以让 `f` 总是返回 FALSE，让 x 为 TRUE，那么只有零函数会返回 TRUE，其他自然数都返回 FALSE

[33]:
```python
f = lambda x: FALSE
x = TRUE
print(ZERO(f)(x))
print(ONE(f)(x))
print(TWO(f)(x))
print(THREE(f)(x))
```

```
<function TRUE at 0x1064e5550>
<function FALSE at 0x1064e5310>
<function FALSE at 0x1064e5310>
<function FALSE at 0x1064e5310>
```

据此可写出判断零元的函数

[34]:
```
def ISZERO(N):
    f = lambda x: FALSE
    return N(f)(TRUE)

print(ISZERO(ZERO).__name__)
print(ISZERO(ONE).__name__)
print(ISZERO(TWO).__name__)
print(ISZERO(THREE).__name__)
```

```
TRUE
FALSE
FALSE
FALSE
```

1.4.7 递归

假如我们要写一个阶乘函数

[35]:
```
from math import factorial
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)

print(fact(3))
assert fact(9) == factorial(9)
```

```
6
```

如果要写成函数形式，首先要考虑的是怎么实现 if else 这样的流程控制。

如果 `N==ZERO`，那么 `ISZERO(N)` 就会等价于 `TRUE`，而 `TRUE(ONE,n*f(n-1))` 会返回 `ONE`。

因此 `ISZERO(N)(ONE,n*f(n-1))` 就完成了流程控制。

[36]:
```python
def FACT(N):
    return ISZERO(N)(ONE,MULTIPLY(N,FACT(SUBTRACT(N,ONE))))

try:
    show(FACT(THREE))
except Exception as e:
    print(repr(e))
```

```
RecursionError('maximum recursion depth exceeded')
```

但在实际运行时，可以看到我们的函数出现了无限循环的问题，这是因为 Python 并不是惰性求值 (lazy evaluation) 的，它会先计算参数，再把参数的值代入函数

[37]:
```python
def f(a,b):
    return a

try:
    f(1,1/0)
except Exception as e:
    print(repr(e))
```

```
ZeroDivisionError('division by zero')
```

在上面这个例子中，虽然我们并不需要参数 b，但 Python 仍会先计算 b=1/0，此时就返回了错误。

避免这种情况出现的方法就是，不直接传入参数，而是传入返回参数的函数:

[38]:
```python
def f(a,b):
    return a()

f(lambda: 1, lambda: 1/0)
```

[38]: 1

此时 Python 计算的步骤为:

1. `a=lambda: 1`
2. `b=lambda: 1/0`
3. `a()=1`
4. `return 1`

使用类似的方法对我们的原函数进行一些小改造，最终就能正常进行了

[39]:
```
LAZY_TRUE = lambda x,y: x()
LAZY_FALSE = lambda x,y: y()
LAZY_ISZERO = lambda N: N(lambda _: LAZY_FALSE)(LAZY_TRUE)

def FACT(N):
    return LAZY_ISZERO(N)(lambda: ONE,lambda: MULTIPLY(N,FACT(SUBTRACT(N,ONE))))

show(FACT(THREE))
```

6

1.5 Y combinator

如果我告诉你 Y combinator 的定义如下

$$Y = \lambda f.(\lambda x.f(x(x)))(\lambda x.f(x(x)))$$

那你肯定就开始怀疑下面的内容是否能看懂，别担心，其实很简单，忘记上面这个公式继续往下看吧。

1.5.1 依然是阶乘函数

在上面的例子中，我们是这样定义阶乘函数的：

[40]:
```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)

fact(3)
```

[40]: 6

对于熟练掌握递归的人来说，这样貌似是非常理所当然的，但实际想一想，多少有点不可思议，因为我们在定义 `fact` 函数的过程中，居然用到了 `fact` 本身！

熟悉数理逻辑的人都会对自指（self-reference）满怀敬畏之心：- 罗素使用了自指，引发了第三次数学危机 - 哥德尔使用了自指，得出了哥德尔不完备定理 - 图灵使用了自指，提出了停机测试悖论

为了更好地理解递归，我们现在要避免在完整定义 `fact` 之前调用 `fact`。

那么我们将函数内部的 `fact` 替换成 `f`，这里的 `f` 就是我们需要的阶乘函数：

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*f(n-1)
```

但乍一看，这个 `f` 是凭空出现的，因此为了提供 `f`，我们给 `fact` 再添加一个参数：

[41]:
```
def fact(f,n):
    if n == 0:
        return 1
    else:
        return n*f(n-1)
```

这样的定义是可行的吗？

如果我们调用 `fact(3)` 肯定不对，因为 `fact` 有两个参数要输入。

调用 `fact(f,3)` 也不对，`f` 还是没有定义，既然 `f` 就是 `fact`，那么我们尝试调用 `fact(fact,3)`

[42]:
```
fact(fact,3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/1y/8ypw_bc55x5d69n0rnzpwxjr0000gn/T/ipykernel_25288/2401241230.py␣
↪in <module>
----> 1 fact(fact,3)

/var/folders/1y/8ypw_bc55x5d69n0rnzpwxjr0000gn/T/ipykernel_25288/901191219.py i␣
↪fact(f, n)
      3         return 1
      4     else:
----> 5         return n*f(n-1)
```

```
TypeError: fact() missing 1 required positional argument: 'n'
```

可以看到我们的第 5 行中，`f(n-1)` 出现了问题，因为此时其实是 `fact(n-1)`，而 `fact` 是需要两个参数的，因此我们将其改为 `f(f,n-1)`

[43]:
```
def fact(f,n):
    if n == 0:
        return 1
    else:
        return n*f(f,n-1)
```

[44]:
```
fact(fact,3)
```

[44]: 6

这回终于正确了，但其实我们想要的阶乘函数应该是 `fact(3)=6`，于是我们可以稍微修改一下：

[45]:
```
def F(f,n):
    if n == 0:
        return 1
    else:
        return n*f(f,n-1)
fact = lambda n: F(F,n)
fact(3)
```

[45]: 6

将其柯里化得到

[46]:
```
F = lambda f: lambda n: 1 if n==0 else n*f(f)(n-1)
fact = F(F)
fact(3)
assert fact(9) == factorial(9)
```

让我们回到最初的定义：

[47]:
```
fact = (lambda f: lambda n: 1 if n==0 else n*f(n-1))(fact)
fact(4)
```

[47]: 24

如果我们用 `R` 来代替 `lambda f: lambda n: 1 if n==0 else n*f(n-1)`，那么就有 `fact = R(fact)`，`fact` 就是 `R` 的一个不动点

[48]:
```python
R = lambda f: lambda n: 1 if n==0 else n*f(n-1)
fact = R(fact)
fact(4)
```

[48]: 24

假设我们有一个函数 `Y(R)`，可以计算出 `R` 的不动点，即 `Y(R) = R(Y(R))`

```
F    = lambda f: lambda n: 1 if n==0 else n*f(f)(n-1)
F(F) =           lambda n: 1 if n==0 else n*F(F)(n-1)
fact =           lambda n: 1 if n==0 else n*fact(n-1)
<==> fact = F(F)

fact = (lambda f: lambda n: 1 if n==0 else n*f(n-1))(fact)
R    = lambda f: lambda n: 1 if n==0 else n*f(n-1)
<==> fact = R(fact)

F(x) = lambda f: lambda n: 1 if n==0 else n*f(f)(n-1) (x)
     =           lambda n: 1 if n==0 else n*x(x)(n-1)
     = R(x(x))
<==> F = lambda x: R(x(x))

Y(R) = fact
     = R(fact)
     = R(F(F))
<==> Y(R) = R(F(F))
```

通过这些恒等式，我们可以通过 `R(F(F))` 来构建 `Y(R)=fact`

[49]:
```python
R = lambda f: lambda n: 1 if n==0 else n*f(n-1)
F = lambda x: R(x(x))
Y = lambda R: R(F(F))


try:
    fact = Y(R)
except Exception as e:
```

```
    print(repr(e))
```

```
RecursionError('maximum recursion depth exceeded')
```

依然是因为惰性求值的问题，我们把 `x(x)` 换成 `lambda z: x(x)(z)`，延后计算 `x` 的值。

[50]:
```
R = lambda f: lambda n: 1 if n==0 else n*f(n-1)
F = lambda x: R(lambda z: x(x)(z))
Y = lambda R: R(F(F))
fact = Y(R)

fact(4)
```

[50]: 24

1.5.2 一般形式

对于一般的情况，我们怎么计算 f 的不动点呢?

首先定义一个

$$F(x) = f(x(x))$$

于是有 $F(F) = f(F(F))$，然后如下定义即可

$$Y(f) = F(F)$$

证明:

$$\begin{aligned} Y(f) &= F(F) \\ &= f(F(F)) \\ &= f(Y(f)) \end{aligned}$$

于是可知 $Y(f)$ 是 f 的不动点。

[51]:
```
def Y(f):
    F = lambda x: f(lambda z: x(x)(z))
    return F(F)
```

我们还可以试着计算一下斐波拉契数列:

[52]:
```
def Fib(n):
    if n==1 or n==2:
        return 1
```

```python
    else:
        return Fib(n-1)+Fib(n-2)

for i in range(1,10):
    print(Fib(i))
```

```
1
1
2
3
5
8
13
21
34
```

[53]:

```python
R = lambda f: lambda n: 1 if n==1 or n==2 else f(n-1)+f(n-2)
fib = Y(R)
for i in range(1,10):
    print(fib(i))
```

```
1
1
2
3
5
8
13
21
34
```

1.6 Reference

1. 康托尔、哥德尔、图灵——永恒的金色对角线 (rev#2) – 刘未鹏 | Mind Hacks
2. A Tutorial Introduction to the Lambda Calculus
3. 从零开始的 演算 | weirane's blog
4. David Beazley - Lambda Calculus from the Ground Up - PyCon 2019 - YouTube

5. GitHub - orsinium-labs/python-lambda-calculus: Lambda Calculus things implemented on Python