Y-Combinator简介

先不提 Y-Combinator. 我对这个问题的兴趣最初来自于 Friedman 的 *The Little Schemer* 第九章，下面的推导过程也参考了此书。我曾经在使用 JavaScript 编程时一度对 first-class function 和匿名函数这样的特性很着迷，并沉醉于将所有的函数定义都写成这样：

```
let someFunction = function(args) {
    ...
};
```

这样定义函数直接带来一个显而易见的问题：如何实现函数的递归定义?

JavaScript的问题

无论是否像我上述的风格写 JavaScript 代码，也就是说，尽管你写成这样：

```
function someFunction(args) {
    ...
}
```

在 JavaScript 中，问题都一样存在：如果你在函数内部递归调用了你定义的函数本身，就有可能导致 bug.

```
function fact(n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}

let someFunction = fact;
fact = null;
console.log(someFunction(5));
```

显然，fact 函数是用来计算阶乘的，但在像上面这样的情况下，由于 fact 自身只是一个变量——或者说，一个对象引用——someFunction 的功能随着 fact 的置空就丧失了。对此，JS 程序员会很自然地用 arguments.collee 完成一般的递归工作：

```
function fact(n) {
    if (n ≤ 1) {
    return 1;
  } else {
    return n * arguments.collee(n - 1);
  }
}

let someFunction = fact;
fact = null;
console.log(someFunction(5));
```

但是我们今天不会满足于此！因为 arguments.collee 更像是一种由语言本身带来的破解技术，而非我们动脑后得到的解决方案。

lambda演算基本知识

在解决上述递归函数的难题之前，有必要介绍一些 lambda 演算的基本知识。考虑到读者更有可能对现代编程语言更熟悉，我将不使用文献中通常会使用的数学语言，而是编程语言来表达下文的内容。个人认为编程语言相比之下有不少优点，例如它们极少情况下会产生歧义（大部分语言语法没有二义性），以及读者可以随时 copy 文中的代码在自己的机器上进行验证。

如果你没有 Racket 解释器，可以尝试安装一个 DrRacket, 或者 mit-scheme, 因为我将要使用 Racket 语言。如果你本来就熟悉 Racket 或者 Scheme, 那么不借助机器也是可以验证下面代码的。

在 Racket 中，定义一个阶乘函数可以像这样做：

```
(define (fact-1 n)
  (if (= n 0)
      1
      (* n (fact-1 (- n 1)))))
```

现在我们完全抛弃 Racket 的函数定义语法，只允许自己使用 lambda 表达式，并且手动对函数柯里化（每个 lambda 表达式只有一个参数）。也就是说，把加法函数写成如下形式：

```
(define add
  (λ (a)
    (λ (b)
      (+ a b))))
```

我在这里用到了希腊字母 λ, 仅仅只是为了让代码看起来短小一些。如果你的环境不支持希腊字母，你可以把它换成单词 "lambda".

这些代码在 Racket/Scheme 中都是良定义的，有编程基础的人很容易就能理解函数的功能，进而理解 lambda 表达式。但 lambda 演算本身是一个数学系统，lambda 表达式本身和 lambda 表达式的各种转换都需要很严密的定义。由于这不是我们要讨论的主要内容，我在这里只简要地说明一下，对于 Racket 中的一个 lambda 表达式而言：

```
(λ (<arg>) <expression>)
```

它满足 lambda 演算理论中所谓的α转换和β转换，在不涉及其它转换规则时，你所知的有关 JavaScript 的匿名函数或者 Scheme 的 lambda 知识都是够用的。

α转换的例子

lambda 表达式中的约束变量被替换时，替换后的表达式与原表达式等价。

```
(λ (a) (* 2 a))
```

和

```
(λ (b) (* 2 b))
```

是等价的。

β转换的例子

lambda 表达式应用于另一个表达式时

```
((λ (a) (* 2 a)) 4)
```

等价于将被应用表达式代入应用式后的表达式：

```
(* 2 4)
```

这两个转换也是 Lisp 系统的函数代换模型的根本。

再次重申，以上说明只是例子，而非标准定义。真正的严密定义还需要关注很多细节问题，此处不再讨论。

无法约简的lambda表达式

并不是所有的 lambda 表达式都是可以通过转换约简的，例如以下这个：

```
((λ (a) (a a))(λ (a) (a a)))
```

无论进行多少次β转换，这个 lambda 表达式都会保持自身。这个例子本质上是一个无限递归，而且它是一个简单且直观的递归典型。下面的递归函数求解过程中我们将会用到它。

为无名者起名字

回到我们的问题上来，我们想要把一个没有名字的函数定义成递归函数。一个很直接的想法是给这个函数起个名字——函数参数，或者说，约束变量能完成这个工作。如果我们有这样一个函数就好了：

```
(define fact-gen
  (λ (g)
    (λ (n)
      (if (= n 0) 1 (* n (g (- n 1)))))))
```

我把它叫作 fact-gen, 它专门用来生成阶乘函数 fact.

如果把我们的想法表示成数学方程，大概是这个样子：

```
fact(n) = fact-gen(fact)(n)
```

根据我们之前提到的，左右式作用于同样变量 n 时得到相同的结果，就可以认为它们本身是相同的。我们可以把上式抽象地写成：

```
fact = fact-gen(fact)
```

这一步将会为最后的解带来些许不同，对此我们后面再讨论。显然，这是一个不动点方程。现在我们要寻找该方程的解，以便于把 fact 写成：

```
fact = magic(fact-gen)
```

这样的形式，其中 magic 函数和 fact 无关，这样的话我们就能得到 fact 作为递归函数的非递归定义了。现在我们把语言切换回 Racket，参数 g 这里需要填充的就是 fact-gen，所以可以把 fact-gen 作为自己的参数，得到了一个不带递归的阶乘函数。

```
(define fact (fact-gen fact-gen))
```

但这个写法显然是错的！因为 fact-gen 定义内的 g 只接受一个参数，且类型为数字。我们可以在原先的 fact-gen 上做一点修改，让它满足上面这个 fact 函数的定义。

```
(define fact-part
  (λ (g)
     (λ (n)
       (if (= n 0) 1 (* n ((g g) (- n 1)))))))

(define fact (fact-part fact-part))
```

这个重复的 `(fact-part fact-part)` 可以写作：

```
((λ (f) (f f)) fact-part)
```

看起来还是在重复，但至少重复的部分是一个通用函数了。为了方便我们可以为这个通用函数命名：

```
(define recurs ((λ (f) (f f))))
```

这样一来：

```
(define fact (recurs fact-part))
```

现在，fact-part 和我们最初追寻的 fact-gen 之间仍然有些差距，问题在于 fact-part 内部有诸如 `(f f)` 这样糟糕的重复部分。有了 recurs, 我们可以把它们自然地约简。

抽象重复过程

首先将 `((f f) n)` 这样的调用抽象成类似于 recurs 的通用函数：

```
(define wrap
  (λ (h)
    (recurs (λ (f) (h (λ (n) ((f f) n)))))))
```

注意到 wrap 其实就是一个对带有一个参数的 lambda 表达式的 recurs. 这样 fact 就可以写成如下形式：

```
(define fact (wrap fact-gen))
```

现在我们的最终程序就像这个样子：

```
(define fact-gen
  (λ (g)
```

```
    (λ (n)
      (if (= n 0) 1 (* n (g (- n 1)))))))

(define recurs (λ (f) (f f)))

(define wrap
  (λ (h)
    (recurs (λ (f) (h (λ (n) ((f f) n)))))))

(define fact (wrap fact-gen))
```

这个程序里 recurs 和 wrap 是通用过程，fact-gen 是中间过程，fact 是我们要的结果。它已经满足了我们要解的方程，并得到了 fact 的最终解：

```
fact = magic(fact-gen)
```

Y 就是我们要的 magic 函数。由于 fact-gen 只用到了一次，recurs 只在 wrap 中使用过，我们将程序稍作整理：

```
(define Y
  (λ (h)
    ((λ (g) (g g))
     (λ (f)
       (λ (n) ((h (f f)) n))))))

(define fact (Y
              (λ (g)
                (λ (n)
                  (if (= n 0) 1 (* n (g (- n 1)))))))))
```

现在我们终于得到了 fact 的最终表达形式，并且附带得到了一个很有用的函数 Y, 它可以用来产生诸如 fact 这样带有一个参数的递归函数。它就是本文题目中提到的 Y-Combinator.

遗留问题

我们提到，上面得到的这个 Y 可以用来产生带一个参数的递归函数，那么没有参数或者有一个以上参数的递归函数要怎么做呢？如果按照上面的过程，仅仅只是对这个方程求解：

```
fact = fact-gen(fact)
```

可能会得到这样的 Y:

```
(λ (h)
  ((λ (g) (g g))
   (λ (f)
     (h (f f)))))
```

它的确是满足不动点方程 `fact = fact-gen(fact)` 的， 但是它在 Racket 语言中不满足 `fact(n) = fact-gen(fact)(n)` . 从后一个方程到前一个方程是一个单纯的数学过程，而非 Racket 解释器能明白的。根本问题在于 Racket 系统的函数调用是值调用规则，而非名调用规则。这使得对于任何函数 g, `(Y g)` 都将发散。

同样的道理，如果需要定义的递归函数带有两个参数，也就是要解这样的方程：

```
func(m)(n) = func-gen(func)(m)(n)
```

得到的 Y 可能就是如下形式：

```
(define Y
  (λ (h)
    ((λ (g) (g g))
     (λ (f)
       (λ (n)
         (λ (m) (((h (f f)) n) m)))))))
```

网络上与 Y-Combinator 相关的文章还有很多，解决递归问题也是不错的思维训练。如果这篇文章解答了你的困惑，或者让你对 Y-Combinator 或 lambda 演算产生了兴趣，请......

（本文定价1元，多谢支持！！！）