[读书笔记]SICP：3[B]用高阶函数做抽象

本章对应于书中的1.3。

我们看到了过程是一种层次上的抽象，用来描述一些对数字的复合操作，但又不依赖于特定的数值。这里将介绍一种操作过程的过程，它们以过程为参数或以过程为返回值，称为**高阶过程**，我们将探索高阶过程作为更进一层的抽象，如何增强语言的表达能力。

1 过程作为参数

1.1 高阶过程表示“概念”

比如对于序列求和操作 $\sum_{n=a}^{b} f(n) = f(a) + \ldots + f(b)$，我们关注的是“求和”概念本身，而不是特定的求和操作，如果通过一个高阶过程来表示“求和”概念后，通过传入特定的求和过程作为参数，就能得到针对特定求和的过程。这里我们可以定义以下表示“求和”概念的高阶过程

```
(define (sum a b opt next)
  (if (> a b)
      0
      (+ (opt a)
         (sum (next a) b opt next))))
```

这里的`a`和`b`分别表示求和的上下界，而`opt`表示特定的求和过程`f`，`next`表示计算下一个值的操作。对应的迭代计算过程版本为

```
(define (sum a b opt next)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ result (opt a)))))
  (iter a 0))
```

由此我们就能通过传入不同的`opt`和`next`过程来得到针对特定求和操作的过程，比如

```
; 普通求和
(define (sum-intergers a b)
  (define (identity x) x)
  (define (inc x) (+ x 1))
  (sum a b identity inc))
; 立方求和
(define (sum-cube a b)
  (define (cube x) (* x x x))
  (define (inc x) (+ x 1))
  (sum a b cube inc)))
```

此外，不仅可以定义高阶过程来抽象序列求和概念，还可以用一个高阶过程来抽象序列乘积概念，对应的代码为

```
;; 递归计算过程
```

```scheme
(define (product1 a b opt next)
  (if (> a b)
      1
      (* (opt a)
         (product1 (next a) b opt next))))

;; 迭代计算过程
(define (product2 a b opt next)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* result (opt a)))))
  (iter a 1))
```

更进一步，可以发现其实`sum`和`product`都有更一般的"累积"概念，我们可以定义一个更加抽象的高阶过程来表示"累积"概念，对应的代码为

```scheme
 ;; 递归计算过程
(define (accumulate1 combiner null-value a b opt next)
  (if (> a b)
      null-value
      (combiner (opt a) (accumulate1 combiner null-value (next a) b opt next))))
;; 迭代计算过程
(define (accumulate2 combiner null-value a b opt next)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (combiner result (opt a)))))
  (iter a null-value))
```

这里的`combiner`是一个两个参数的过程，用来描述如何将当前项与前面各项的积累结果组合起来，`null-value`是基本值。通过这个高阶函数，我们可以容易就得到`sum`和`product`高阶过程

```scheme
 ;; 定义sum
(define (sum a b opt next)
  (accumulate1 + 0 a b opt next))

;; 定义product
(define (product a b opt next)
  (accumulate1 * 1 a b opt next))
```

综上：通过将过程作为高阶过程的参数，使得高阶过程能注重于概念本身。

1.2 `lambda`和`let`

可以发现我们为了将过程作为高阶过程的参数，定义了很多简单的参数，我们这里可以用`lambda`来创建这些简单的过程

```scheme
(lambda (<formal-parameters>) <body>)
```

通过`lambda`创建的过程没有名字，可以直接将该过程作为参数，也可以用作组合式的运算符

```scheme
((lambda (<v1> <v2> ... <vn>)
    <body>)
<exp1> <exp2> ... <expn>)
```

而这种形式下，其实就是在`内创建了若干个局部变量`、`......`，并将其赋值为`、`......`` `，这里提供 ``let`来包装局部变量

```
(let ((<v1> <exp1>)
      (<v2> <exp2>)
       ...
      (<vn> <expn>))
  <body>)
```

它的本质就是上面的`lambda`，可以一一对应来探索`let`**的特点**：

- `let`表达式描述的局部变量的作用域只在`中，我们可以在尽可能近的地方创建我们要的局部变量。比如当前`x`为5，此时用`let`表达式将`x`设置为3，要注意它的作用域仅在`中，所以只有第二行的`x`才是3，而第三行的`x`不在``中，所以是5，最终运算结果为38。

```
(+ (let ((x 3))
       (+ x (* x 10)))
   x)
```

- 变量的值是在`let`之外计算的。通过查看`lambda`表达式，可以发现`的计算不在`中，所以不受局部变量定义的影响。比如当前`x`为2，此时第二行使用的`x`并不是第一行定义的值，而是外界的2，所以最终运算结果为12。

```
(let ((x 3)
      (y (+ x 2)))
  (* x y))
```

1.3 例子

首先**对比复合过程和高阶过程**：

- 复合过程是为了作为一种将若干操作的模式抽象出来的机制，使所描述的计算不再依赖于所涉及的特定数值
- 高阶过程是用来描述计算的一般性过程，与其中所涉及的特定函数无关。

接下来将用两个例子来介绍如何通过过程去直接描述一般性方法。

1.3.1 寻找函数零点

可以使用**折半查找方法**来寻找函数零点，如果对于给定的两点具有 $f(a) < 0 < f(b)$ ，则该函数的零点就在`a`和`b`之间，基本步骤为：

- 取`a`和`b`的中间点`mid`，判断`f(mid)`的正负性
-
    - 如果`f(mid)`为正，则函数零点一定在`a`和`mid`之间，用这两个点再执行该步骤
    - 如果`f(mid)`为正，则函数零点一定在`mid`和`b`之间，用这两个点再执行该步骤
- 当`a`和`b`很接近时，就将`mid`作为最终结果

这样我们可以得到以下过程

```scheme
(define (search f neg-point pos-point)
  (define (close-enough? x y) ;判断a和b是否足够靠近的过程
    (< (abs (- x y)) 0.001))
  (let ((midpoint (average neg-point pos-point))) ; 定义mid局部变量
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond
           ((positive? test-value) (search f neg-point midpoint))
           ((negative? test-value) (search f midpoint pos-point))
           (else midpoint))))))
```

此外，我们还需对其进行封装

```scheme
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value)) (search f a b))
          ((and (negative? b-value) (positive? a-value)) (search f b a))
          (else (error "Values are not of opposite sign" a b)))))
```

这样，我们就用一个高阶过程`half-interval-method`描述了通过折半查找方法求解函数`f`的零点的一般性过程，此时只要将函数`f`作为参数传入该方法就能求解。

1.3.2 寻找函数不动点

函数不动点就是`f(x)=x`，可以通过反复调用`f`来近似，基本步骤为：

- 给定一个初始猜测值`x`
- 调用`f(x)`得到一个新的值，判断`x`和`f(x)`的差距
-
  - 如果差距较大，则将`f(x)`作为新的`x`，重新调用该过程
  - 如果差距较小，则将`f(x)`作为函数不动点

这样我们可以得到以下过程

```scheme
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

其实可以用这个迭代不动点方法来求解之前的平方根，对于$y^2 = x$可以转化为$y = x/y$，则求解平方根`y`其实就是求解`x/y`的不动点（这里的`y`是变量）。但是假设我们以`y0`作为初始点，则下一个迭代的值为`x/y0`，再下一个迭代的值为`x/(x/y0)=y0`，使得计算过程不断在`y0`和`x/y0`之间来回震荡，这里可以使用**平均阻尼**的技术，我们知道不动点是在`y0`和`x/y0`之间，而震荡的原因是每次变化的幅度太大，所以我们可以将函数转化为$y = \dfrac{1}{2}(y + x/y)$，使得下次迭代的值是在`y0`和`x/y0`之间，减小了变化幅度。对应的代码为

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))
```

注意：平均阻尼技术常常用于帮助不动点搜索收敛。（平均阻尼技术就是将当前的值`x`和`f(x)`取平均，作为下一时刻的值）

2 过程作为返回值

同样的我们也可以将过程作为高阶过程的返回值，比如上文使用的平均阻尼技术，就是返回`x`和`f(x)`的平均值，作为一个通用技术，可以用一个高阶过程进行封装

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

通过传入一个过程`f`到`average-damp`高阶过程，就能得到使用了平均阻尼的过程。

这样我们就能进一步对求平方根的过程修改为

```
 (define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

由此将三种思想组合到了一个方法中：不动点、平均阻尼和求平方根的`x/y`。与之前的求平方根过程相比

```
 (define sqrt x)(
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (average y)
    (/ (+ x y) 2))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
      guess
      (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

- 原始方法是完全针对求平方根这个任务来写的，不具有扩展性
- 现在的方法，首先是基于寻找不动点的高阶过程`fixed-point`，然后使用了能返回应用平均阻尼技术的过程的高阶过程`average-damp`，是基于一些通用的高阶过程来扩展到求平方根这个任务的，思路会更加清晰，且具有扩展性。

我们同样能很简单地扩展到求立方根的方法

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
               1.0))
```

精华：将一个计算过程形式化为一个过程，有很多方法，最好将该计算过程中有用的元素能表现为一些相互分离的个体，并使它们还可能重新用于其他的应用。也就是用高阶过程来抽象“通用概念”，而具体应用的实现就是将任务特定的过程传入高阶过程中。

3 牛顿法

我们想要求解某个函数`f`的零点时，可通过牛顿迭代法，根据牛顿法的推导结果可知，它就是反复迭代以下公式

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

而其迭代的过程其实可以看成是求以下函数的不动点

$$g(x) = x - \frac{f(x)}{f'(x)}$$

由此我们可以根据求解函数不动点的高阶过程`fixed-point`来将牛顿法实现为一个过程。

首先，对`f`求导的结果可以表示为

$$f'(x) = \frac{f(x+dx) - f(x)}{dx}$$

我们可以将函数`f`对应的求导函数作为高阶函数的返回值，对应的代码为

```
(define dx 0.00001)

(define (deriv f)
  (lambda (x) (/ (- (f (+ x dx)) (f x))
                 dx)))
```

然后封装得到一个`g`函数

```
(define (newton-transform f)
  (lambda (x) (- x
               (/ (f x)
                  ((deriv f) x)))))
```

由此我们就可以通过`fixed-point`获得通用的牛顿迭代法过程了

```
(define (newton-method f guess)
  (fixed-point (newton-transform f) guess))
```

由此通过这个牛顿迭代法的高阶过程，我们也可以很容易得到求平方根的过程，对应需要求解零点的函数为 $f(y) = y^2 - x$，则对应的代码为

```
(define (sqrt x)
  (newton-method (lambda (y) (- (square y) x))
                 1.0))
```

我们跟进一步可以发现，以上介绍的两种求平方根的方法其实都是基于求解函数不动点的`fixed-point`，只是一个使用了平均阻尼法对`x/y`进行变换，另一个使用牛顿法对 $y^2 - x$ 进行变换。我们可以抽象出更加一般的方法

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

这里会使用`transform`对`g`进行变换后，使用`fixed-point`求解函数的不动点。这样我们可以对以上两种求平方根的方法进一步修改为

```scheme
; 使用平均阻尼技术的求函数不动点的方法
(define (sqrt x)
  fixed-point-of-transform (lambda (y) (/ x y))
                            average-damp
                            1.0)
; 使用牛顿迭代法
(define (sqrt x)
  fixed-point-of-transform (lambda (y) (- (square y) x))
                            newton-method
                            1.0)
```

4 总结

我们需要培养出识别程序里基本抽象的能力，基于它们去进一步构造，并进行推广来创建更高层次的抽象，应该根据工作需要，选择合适的抽象层次，使得能在新的上下文中去应用它们。