[读书笔记]SICP：4[B]数据抽象引导

本章对应于书中的2.1。

- **将抽象作为克服复杂性的一种技术。**
- 每个抽象数据都要创建一层构造函数和选择函数，再创建一层调用该抽象数据的基本操作，其中，构造函数就是构造抽象对象实例，而选择函数就是能返回该抽象对象成分的过程，而基本操作就是通过选择函数实现的一系列该抽象对象的基本操作。
- 我们基于对抽象数据的概念，来操作它提供的构造函数和选择函数，希望能获得满足我们预期的效果。其实这个抽象数据是存在于我们脑海中的，该构造数据和选择函数背后是如何实现的，是数据还是基于过程，都不会影响我们对该抽象数据的操作。所以我们认为的抽象数据，本质上可以是数据也可以是过程。

之前的文章介绍的是对过程的抽象，主要包含以下内容：

- 如何使用基本数据和基本操作
- 如何通过复合、条件以及参数的使用将一些过程组合起来，形成复合过程
- 如果通过 `define` 做过程抽象
- 如何通过高阶过程来提升抽象层次，使得我们能操作通用的计算方法

而接下来我们将关注对数据的抽象，谈论将数据对象组合起来形成**复合数据**的方式，由此来提升我们在设计程序时所位于的概念层次，提高设计的模块性，增强语言的表达能力。并且也能把处理复合数据的程序部分和与复合数据如何表示的细节分隔开来，这种称为**数据抽象**。此外，现在有了复合数据这一层抽象后，过程的设计应该更关注“概念”本身，使其在各种数据中更加通用，比如线性组合 `ax+by`，可以表示为如下形式

```
(define (linear-combination a b x y)
  (+ (* a x) (* b y)))
```

但是该过程就假设了输入的 `a`、`b`、`x` 和 `y` 是基本数据，如果这些参数是复合数据，则无法简单地通过 `+` 和 `*` 进行运算，所以该程序应该写成如下形式

```
 (define (linear-combination a b x y)
  (add (mul a x) (mul b y)))
```

这样该过程就更加注重“线性组合”这一概念，并通过传入特定的 `add` 和 `mul` 过程就能在不同复合数据中实现“线性组合”这一概念。

1 数据抽象引导

数据抽象使得我们能将一个复合数据对象的使用，与该数据对象是如何通过更基本的数据对象构造起来的细节隔离开来。基本思想就是设法构造一些使用符合数据对象的程序，使它们像在“抽象数据”上操作一样，不对所传入的数据作任何多余的假设，而“具体数据”的表示也应该与程序中使用数据的方式无关，这里通过**选择函数**和**构造函数**来实现。

我们这里要来实现有理数的一些基本算数运算。

首先，我们还未定义构造函数（如何通过分子和分母构造一个有理数）和选择函数（如何选择出有理数的分子和分母）的具体实现，只希望具有以下过程：

- `(make-rat )`：构造一个有理数
- `(numer )`：选择分子

- `(denom )`：选择分母

则根据有理数的运算规则，我们就可以实现以下有理数的操作了

```
; 有理数加法
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
; 有理数减法
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
; 有理数乘法
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
; 有理数除法
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
; 有理数相等判断
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

而有理数的具体实现，可以使用Lisp的序对`cons`过程来实现（可参考这里），它返回一个包含两个输入参数作为成分的复合数据对象，可以通过基本过程`car`和`cdr`来取出这两部分，比如

```
(define x (cons 1 2)) ; 构建一个序对(1 2)
(car x) ; 取出第一个成分1
(cdr x) ; 取出第二个成分2
```

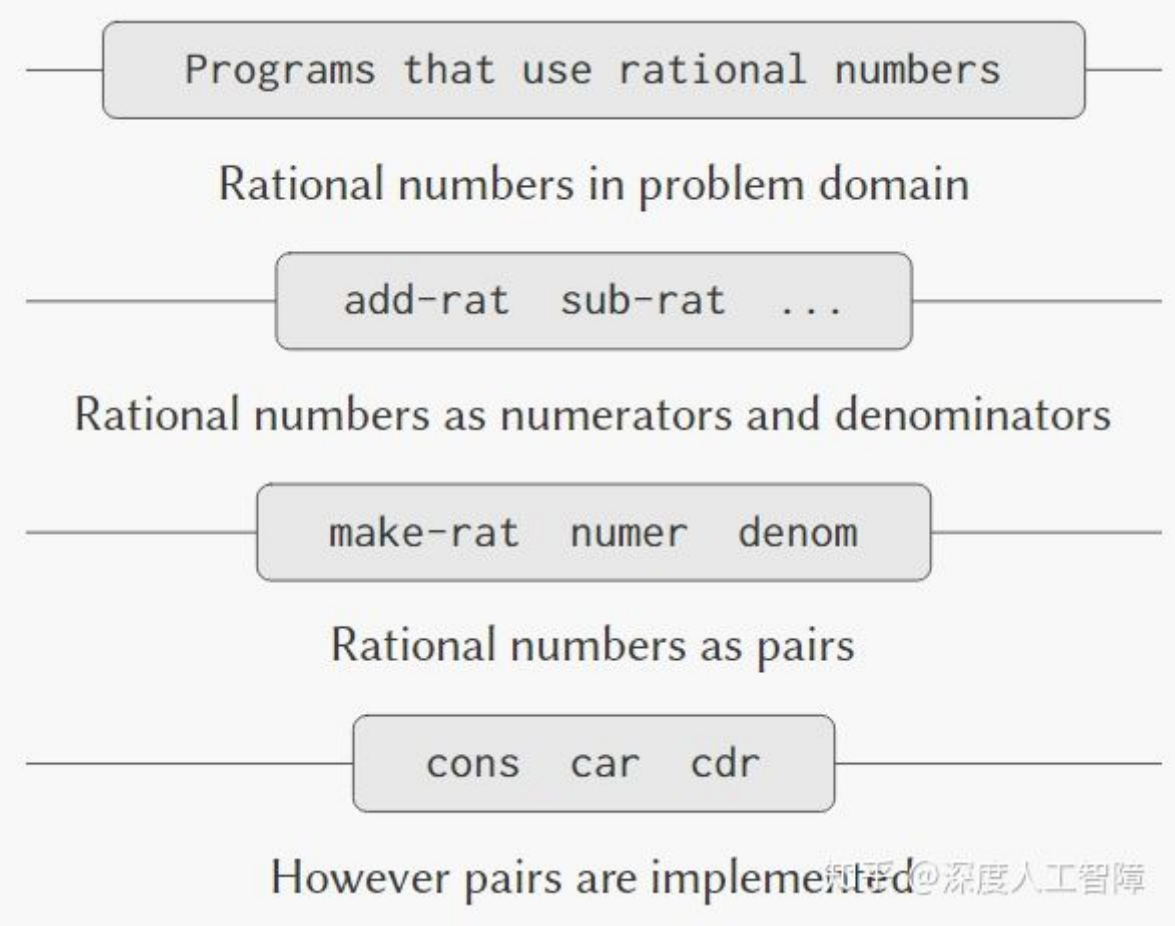
通过序对构建的数据对象也称为**表结构数据**，由此基于`cons`我们可以实现有理数的构造函数

```
(define (make-rat n d)
  (let ((g ((if (< d 0) - +) (gcd n d))))
    (cons (/ n g) (/ d g))))
```

对应的也能实现有理数的选择函数

```
(define (numer x) (car x))
(define (denom x) (cdr x))
```

我们可以将该有理数系统的结果表示为如下形式



可以看到具有不同的层次结构：

- 使用有理数的程序仅仅使用有理数包提供给“公众使用”的那些过程
- 有理数的这些过程又是完全基于构造函数和选择函数的实现
- 构造函数和选择函数又是基于序对实现的

可以看出每一层都表示一个**抽象屏障，**将使用数据抽象的程序（上层）和实现数据抽象的程序（下层）分隔开来，而每一层中的过程构成了所定义的抽象屏障的界面，联系系统中的不同层次。使得程序很容易维护和修改，因为对抽象数据的具体标识方式和操作方式限制在少数几个**界面过程**，而我们在使用时就无需考虑这些界面过程的变化，且界面过程的修改也十分容易。比如我们通过`add-rat`或`sub-rat`这些界面过程来操作有理数时，无需考虑它们是如何实现的，而它们的实现也可以进行修改，而不会影响调用它们的过程。

数据抽象的基本思想：为每一类数据对象标识处一类基本操作，使得对这类数据对象的所有操作都可以基于这些基本操作。

> 创建一层构造函数和选择函数，再创建一层调用该抽象数据的基本操作

2 数据是什么

我们可以看到，对有理数的操作`add-rat`、`sub-rat`等等实现都是基于数据对象有理数定义的，是基于我们对“有理数”的概念进行定义的，该概念要求如果从一对整数`n`和`d`构造出的有理数`x`，则抽取出`x`的`numer`和`denom`的相除结果与`n`和`d`相除结果相同。为了满足这个“有理数概念”，我们通过定义构造函数和选择函数来得到一个抽象数据来表示有理数，因为该抽象数据的行为是由满足“有理数概念”的构造函数和选择函数刻画的，所以可以认为该抽象数据表示有理数，而对应的构造函数和选择函数能作为有理数实现的基础。

> **综上：**数据可以定义为一组满足特定概念的选择函数和构造函数，可作为抽象数据实现的基础。

更进一步来看，其实抽象数据只要满足我们对它的构造函数和选择函数的假设就行，与它究竟是什么无关。比如我们对“序对”的假设就是我们能通过`cons`创建一个数据对，然后通过`car`和`cdr`来获取该数据对的两个数据。基于这个假设，我们可以通过以下方式来实现“序对”

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else
           (error "Argument not 0 or 1:
                   CONS" m))))
  dispatch)
(define (car z) (z 0))
(define (cdr z) (z 1))
```

可以发现，我们这里完全没有用任何的数据结构，而是直接通过过程就实现了“序对”，我们能直接通过该`cons`、`car`和`cdr`来操作该“序对”抽象数据，但是其实它只是一个过程。

> **综上：**通过构造函数和选择函数可以操作我们概念中的抽象数据，但是该抽象数据实际如何实现的，以及它本身是什么没有影响，这其实也模糊了过程和抽象数据的间隔，可以将过程作为对象去操作。

我们基于对抽象数据的概念，来操作它提供的构造函数和选择函数，希望能获得满足我们预期的效果。其实这个抽象数据是存在于我们脑海中的，该构造数据和选择函数背后是如何实现的，是数据还是基于过程，都不会影响我们对该抽象数据的操作。所以我们认为的抽象数据，本质上可以是数据也可以是过程。

我们也可以以书上的练习2.6为例，完全不用数字的情况下来表示数，比如0和加一可以表示为以下形式：

```
(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))

(define zero
  (lambda (f) (lambda (x) x)))
```

而我们可以通过对`(add-1 zero)`的代换求值得到表示1的形式：

```
 ; (add-1 zero)
; (lambda (f) (lambda (x) (f ((zero f) x))))
; (lambda (f) (lambda (x) (f (((lambda (f) (lambda (x) x)) f) x))))
; (lambda (f) (lambda (x) (f ((lambda (x) x) x))))
; (lambda (f) (lambda (x) (f x)))

(define one
  (lambda (f) (lambda (x) (f x))))
```

同理可以通过对`(add-1 one)`的代换求值得到表示2的形式：

```
 ; (add-1 one)
; (lambda (f) (lambda (x) (f ((one f) x))))
; (lambda (f) (lambda (x) (f (((lambda (f) (lambda (x) (f x))) f) x))))
; (lambda (f) (lambda (x) (f ((lambda (x) (f x)) x))))
; (lambda (f) (lambda (x) (f (f x))))

(define two
  (lambda (f) (lambda (x) (f (f x)))))
```

可以发现最内侧有和我们主观认为的数值相同个数的`f`，则我们认为的3可以表示为：

```
 (define three
  (lambda (f) (lambda (x) (f (f (f x))))))
```

并且观察 `add-1` 过程和后面代换求值的计算过程，可以发现内部的 `(f ((n f) x))` 是关键，首先通过最内侧的 `(n f)` 将传入进来的“数值”最外侧的 `lambda (f)` 去掉，再通过 `((n f) x)` 将内侧的 `lambda (x)` 去掉，最终在外侧包上一个 `f`，从而达到添加一个 `f` 的效果，也就是我们主观上认为的加1。由此，我们可以定义一个加法操作

```
(define (+ a b)
  (lambda (f) (lambda (x) ((b f) ((a f) x)))))

; (+ one two)
; (lambda (f) (lambda (x) ((two f) ((one f) x))))
; (lambda (f) (lambda (x) ((two f) (((lambda (f) (lambda (x) (f x))) f) x))))
; (lambda (f) (lambda (x) ((two f) ((lambda (x) (f x)) x))))
; (lambda (f) (lambda (x) ((two f) (f x))))
; (lambda (f) (lambda (x) (((lambda (f) (lambda (x) (f (f x)))) f) (f x))))
; (lambda (f) (lambda (x) ((lambda (x) (f (f x))) (f x))))
; (lambda (f) (lambda (x) (f (f (f x)))))
```

所以，虽然这些定义的“数值”都是过程，但是它满足我们对数的假设，所以也可以认为是抽象数据。