计算机科学导论-SICP
第二章 构造过程抽象
2.1小节 构造数据抽象

王超

Center for Research and Innovation in Software Engineering (RISE), Southwest University

2022 年 9 月 28 日

第二章讲了什么故事

本章讲述如何构造和使用数据对象

- 程序中使用数据有着必要性
- 数据需要包装起来：数据抽象思想，合约（规约）思想
- 数据需要组织起来
- 本章只讲如何构造一个数据对象，不讲怎么修改构造出的数据对象。每次操作往往新构造一个数据对象

本章涉及的数据对象演变过程如下

- 第一种数据：序对，包含两个元素
- 第二种数据：表，包含一列元素
- 第三种数据：树，以嵌套的表的形式实现

涉及的LISP语法

- 序对：cons，car，cdr
- 表：list，nil
- 符号：’
- 谓词：null?，pair?，number?，eq?

序对

- 构造数据需要一种机制把数据粘合起来
- 序对可以作为序对的元素，即闭包性质
- 基于序对，可以构造各种复杂的数据，如表和树

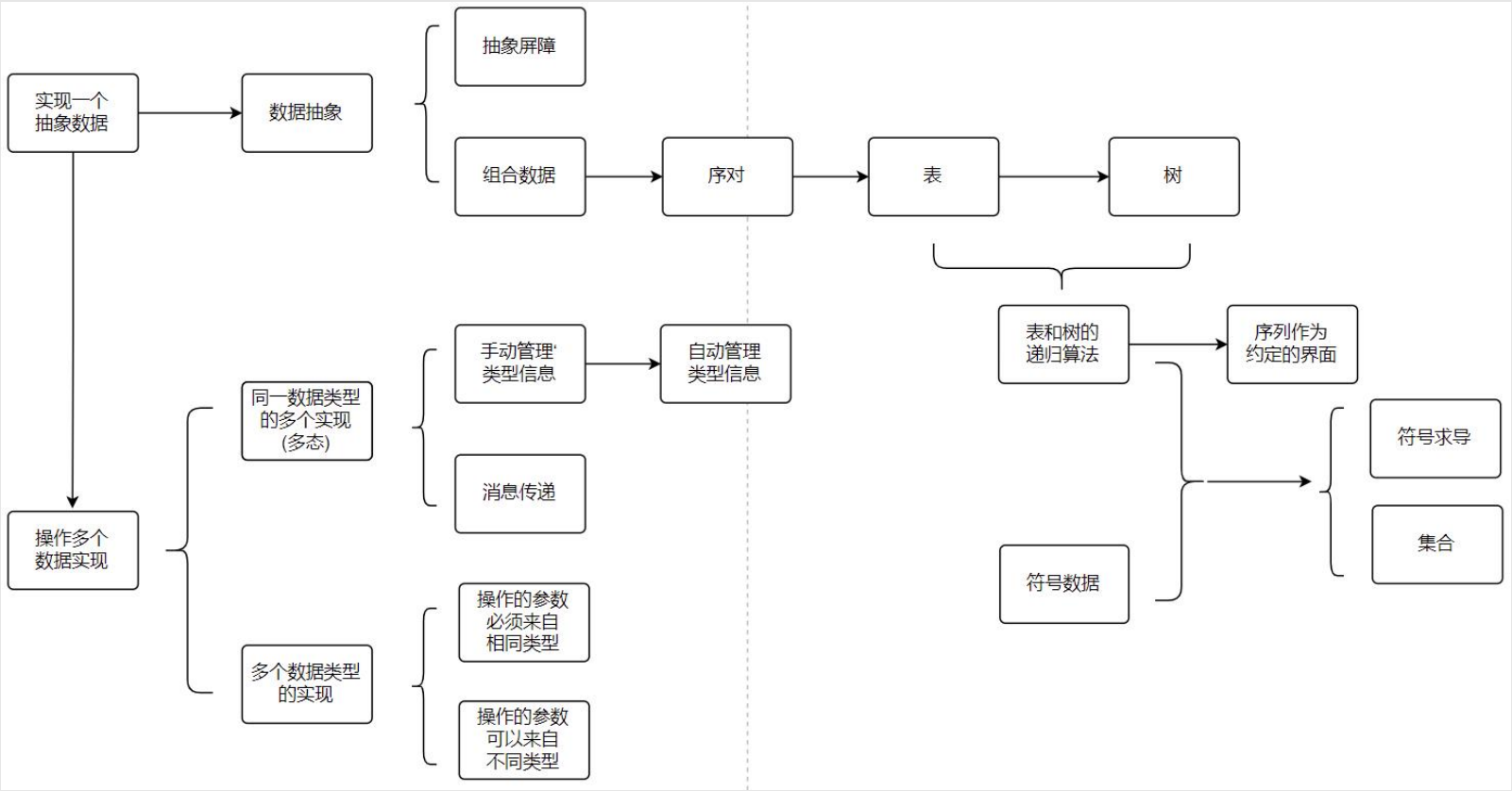
表和树

- 常见的数据组织方式
- 可以以表为核心编写数据处理程序

系统里包含多数据实现

- 一些操作只涉及同一数据类型的不同实现
- 需要解决如何以对应的方式操作对应实现
- 一些操作涉及多个数据类型
- 需要解决如何进行数据类型的转换

第二章知识间的关系


实现一个
抽象数据
数据抽象
抽象屏障
组合数据
序对
表
树
表和树的
递归算法
序列作为
约定的界面
符号求导
集合
符号数据
操作多个
数据实现
同一数据类型
的多个实现
(多态)
手动管理
类型信息
自动管理
类型信息
消息传递
多个数据类型
的实现
操作的参数
必须来自
相同类型
操作的参数
可以来自
不同类型

2.1节讲了什么故事

- 数据抽象：分离数据使用和数据表示
- 抽象屏障：分层的实现
- 合约（规约）：什么是一个正确的实现
- 数据的过程性表示：数据实现了功能就是正确的，其实现可以采取多种形式

2.1节的PPT涉及哪些内容

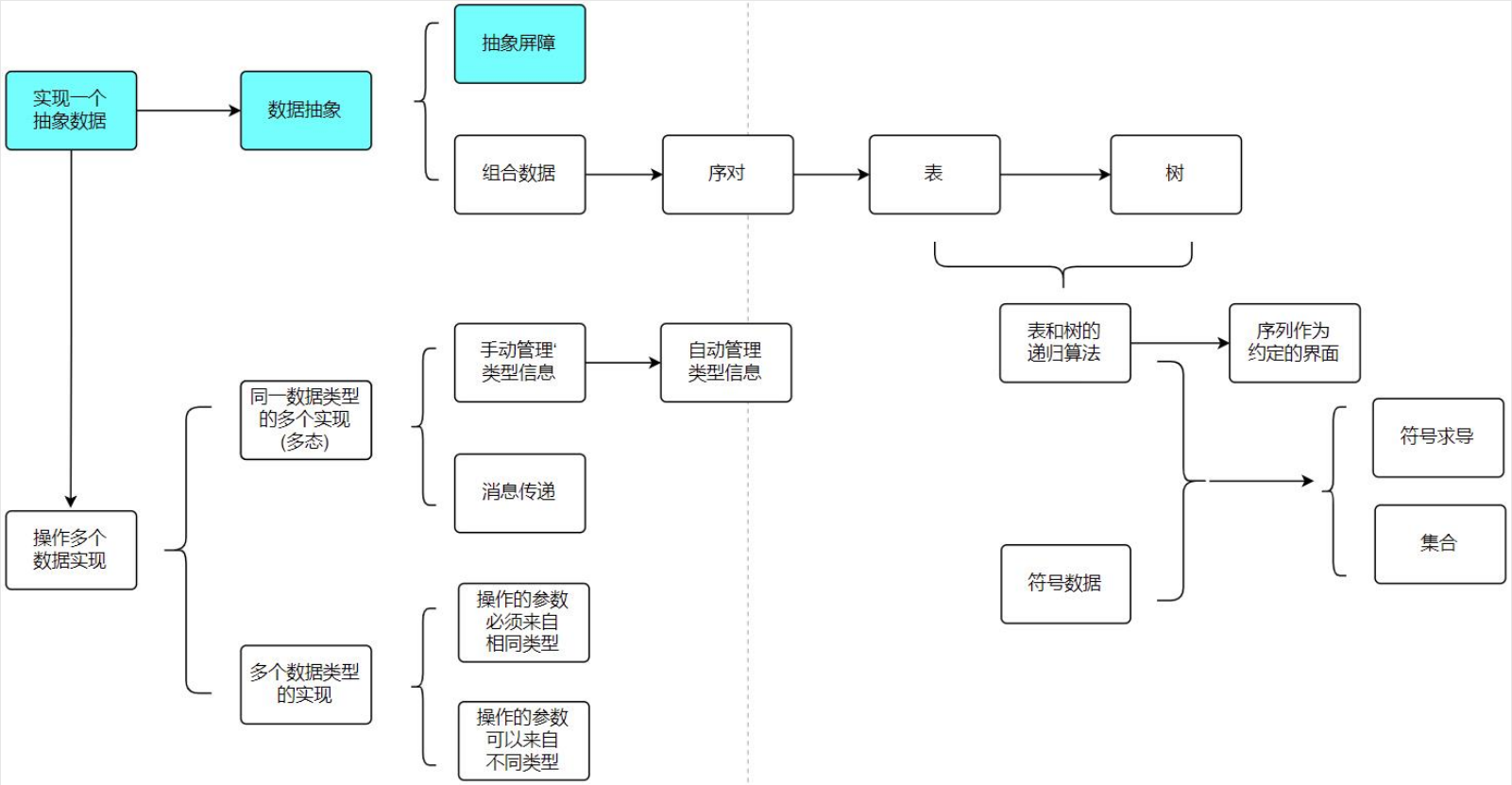
- 数据抽象
- 抽象屏障
- 合约（规约）
- 数据的过程性表示

内容调整

- 2.1.4节不讲

2.1节的内容

蓝色为本次要讲解的内容。



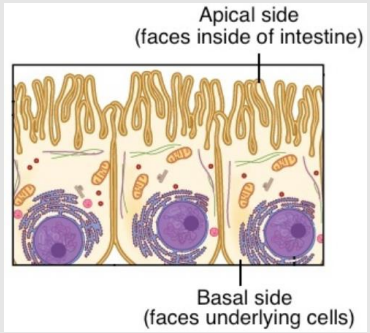
大纲

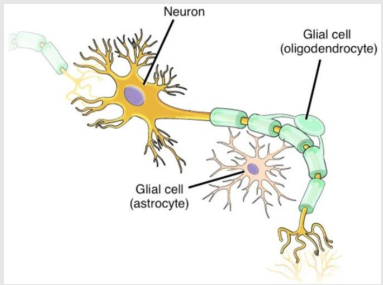
引入

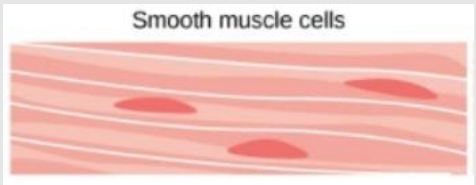
- 通过第一章的学习，我们可以完成数字计算，但这是不够的
- 数字计算通过参数传递数据
- 有时计算需要涉及大量的数据，且编程时无法确切知道数据的规模
- 例如，电子商务网站，涉及大量商品和用户信息
- 如果数据有1T，那么不可能在代码中使用1T个参数
- 我们需要一种机制，把数据堆积起来，并提供访问方法

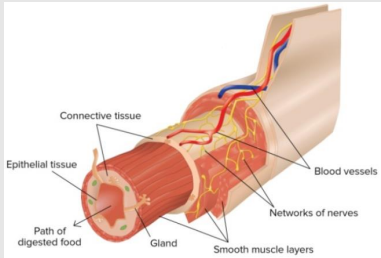
数据需要被有层次地组织起来

- 数据不是被单纯堆积在一起，而是以特定方式分层次组织在一起
- 为了便于理解，让我们把目光转向生物
- 细胞构成组织，组织构成器官，器官构成器官系统，由器官系统构成生物
- 需要特定的细胞以特定的方式工作和协调，才能完成功能
- 例子：细胞 $\Rightarrow$ 上皮组织、神经组织和平滑肌（肌肉组织） $\Rightarrow$ 小肠









数据结构

- 软件系统也是如此
- 数据以特定的方式被组织起来，每一堆被组织起的数据被视作一个抽象的单元，被称为一个数据结构
- 数据结构提供给外界操作接口，操作数据结构时无需知道数据结构的具体实现方式
- 使用简单的数据结构实现复杂的程序和数据结构

例子

- 集合：2.3节
- 树：2.2节
- 二维表格：第三章

数据需要被包装起来

- 用户只能通过数据结构提供的过程来读和修改数据结构，而不被允许绕过数据结构直接读写其存储的内容
- 为什么要有这样的限制

这不是限制，而是保护

- 保证数据不会以设想之外的方式修改
- 原因一：数据结构提供的接口实现更为优化
- 原因二：本就不应该让用户知道接口的实现细节

为什么用户无需知道接口的实现细节呢

仍然以生物为例

- 打字时，大脑通过神经指挥手部肌肉。指挥并非事无巨细
- 大脑只会指挥某块肌肉做收缩或拉伸，不会直接指挥具体肌肉细胞
- 当大脑给每块肌肉下达指令后，由这块肌肉负责指挥自己的肌肉细胞具体完成行动
- 这是幸运的，否则，想象一下每次打字都需要指挥到具体细胞

上一章的过程抽象就是类似的思想，它也同样出现在数据的构造中

大纲

- 数据的必要性
- 数据抽象
- 抽象屏障
- 合约（规约），以及数据的过程性表示
- 结束

引入

有理数的定义和计算

- 计算机无法准确地保存诸如$\frac{1}{3}$这样的循环小数
- 计算机也无法准确地计算0.1+0.2的结果，进而无法比较0.1+0.2和0.5-0.2是否相等
- 让我们实现一个可以完成这些任务的有理数类型

工作步骤

- 构造过程来生成有理数
- 构造过程执行有理数计算

每个有理数被视为一个分数$\frac{x}{y}$

- (make-rat n d)：构造有理数$\frac{n}{d}$
- (numer x)：有理数x的分子部分
- (denom x)：有理数x的分母部分
- (add-rat x y)：求值结果是有理数x和y的和
- (sub-rat x y)：求值结果是有理数x和y的差
- (mul-rat x y)：求值结果是有理数x和y的积
- (div-rat x y)：求值结果是有理数x和y的商

按愿望思维

- 基于make-rat、numer和denom三个过程，实现其他过程
- 以加法为例，给定$x = \frac{a}{b}$和$y = \frac{c}{d}$，则(make-rat x y)的计算结果为
- $\frac{a}{b} + \frac{c}{d} = \frac{a*d+c*b}{b*d}$
- 求x的分子a和求y的分子c，可以通过numer完成
- 求x的分母b和求y的分母d，可以通过denom完成
- 使用数字运算完成a*d+c*b和b*d计算
- 通过make-rat，构造分子为a*d+c*b，分母为b*d的有理数

我们甚至还不知道make-rat、numer和denom到底如何实现

- 我们只是假定有人已经实现好了(make-rat n d)，其求值结果是某种东西(比如一朵云)，里面包含了n和d的信息，并且可以分别使用numer和denom取出这些信息
- 云计算？

这实际上并不是一件难以理解的事情

- 多人合作开发复杂软件时，每个人负责实现一些部分
- 一个人负责的部分可能无法独立运作，需要另一个人实现的部件才能真正工作
- 他可以假定别人已经实现了一个部件，并基于这个还未诞生的部件设计自己的程序
- 他既无法知道别人如何实现，也控制不了别人如何实现，只能假定别人的实现是对的

所谓的按愿望思维

- 把系统划分为不同的部分
- 其中一些部分更为基本，作为基础
- 在设计系统的其他部分时，假定这些基础部分已经被实现，通过基础部分构造其他部分
- 基础部分往往足够抽象，不暴露实现细节

是数据抽象的一部分

有理数的加减乘除，以及判断相等

- 给定$x = \frac{a}{b}$和$y = \frac{c}{d}$
- (add-rat x y)：$\frac{a}{b} + \frac{c}{d} = \frac{a*d+c*b}{b*d}$
- (sub-rat x y)：$\frac{a}{b} - \frac{c}{d} = \frac{a*d-b*c}{b*d}$
- (mul-rat x y)：$\frac{a}{b} * \frac{c}{d} = \frac{a*c}{b*d}$
- (div-rat x y)：$\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a*d}{b*c}$
- (equal-rat? x y)：$\frac{a}{b} = \frac{c}{d}$，如果$a * d = b * c$

代码实现

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (sub-rat x y)
    (make-rat (- (* (numer x) (denom y))
                 (* (numer y) (denom x)))
              (* (denom x) (denom y))))

(define (mul-rat x y)
    (make-rat (* (numer x) (numer y))
              (* (denom x) (denom y))))

(define (div-rat x y)
    (make-rat (* (numer x) (denom y))
              (* (denom x) (numer y))))

(define (equal-rat? x y)
    (= (* (numer x) (denom y))
       (* (numer y) (denom x))))
```

这个设计体现了数据抽象的思想

数据抽象

- 分隔数据的使用和表示
- 程序员可以在无需知道数据如何表示的情况下使用数据
- 可以认为自己在与接口（方法+参数）打交道，而无需关注接口是如何实现的

以有理数为例

- make-rat，numer和denom的代码，是有理数的具体定义
- make-rat，numer和denom作为接口，被add-rat等过程使用
- 这两者彼此是独立的

现在我们回看make-rat，numer和denom如何实现

序对

- 由Scheme提供
- 一种粘合数据的机制
- 可以把两个数据粘合在一起，或者说，存储一对数据

序对提供给程序员的接口

- (cons a b)：构造由a和b组成的序对，序对的第一个元素为a，第二个元素为b
- (car x)：返回序对x的第一个元素
- (cdr x)：返回序对x的第二个元素

序对的一个元素可以取自

- 数字
- 另一个序对
- 过程

由于序对的元素可能是序对，因此一个序对中最终存储的数字或函数可能超过两个

- 伏笔：2.2节我们关注嵌套的序对

- a是由两个数字构成的序对，b是由序对a和数字构成的序对，c是由两个过程构成的序对
- 序对b包含了三个数字
- 这种由构造方法（序对）构造出的结果（a）可以进一步用在构造（b）中的现象，正是2.2节中要介绍的闭包性质

```
(define (square a)(* a a))
(define a (cons 1 2))
(define b (cons a 3))
(define c (cons inc square))
a
(car a)
(cdr a)
b
c
(car c)
((cdr c) 10)
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
(1 . 2)
1
2
((1 . 2) . 3)
(#<procedure:inc> . #<procedure:square>)
#<procedure:inc>
100
```

使用序对构造有理数

- 一个有理数包括一个分子和一个分母，适合用序对存储
- cons过程：一个分子为a，分母为b的有理数被构造为一个序对(cons a b)
- numer过程：(numer x)过程实现为(car x)，返回序对的首元素
- denom过程：(denom x)过程实现为(cdr x)，返回序对中第二个元素

使用概念（接口），而不是细节

目前，下面的两个有理数加法实现都是正确的

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (add-rat-strange x y)
  (make-rat (+ (* (car x) (cdr y))
               (* (car y) (cdr x)))
            (* (cdr x) (cdr y))))
```

我们推荐add-rat，而不是add-rat-strange

- add-rat是基于有理数的概念写出的代码，不管有理数以何种方式实现，add-rat的实现都是正确的
- add-strange是基于当前实现写出的代码
- 如果未来有理数$\frac{x}{y}$不是实现为(cons x y)，而是(cons y x)，则add-rat代码无需任何修改，而add-strange则必须找到所有本质上在求分子或分母的代码，并一一修改

更重要的是

- 计算机程序是为了实现人的思想
- 人不擅长思考冗长的步骤，而善于抽象概念，搭建逻辑层次
- 所以，计算机代码也应该多使用概念，少暴露具体实现细节

通过print-rat过程，我们实现对有理数的打印

```
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))

(define one-five (make-rat 1 5))
(define three-ten (make-rat 3 10))
(define five-six (make-rat 5 6))

(print-rat one-five)
(print-rat three-ten)
(print-rat five-six)
(print-rat (add-rat one-five three-ten))
(print-rat (mul-rat three-ten five-six))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.

1/5
3/10
5/6
25/50
15/60
>
```

- 相比打印为序对，这种方式显示更为清晰
- 我们可以看到，有理数并没有写成最简形式

改进

- 在每次使用make-rat构造有理数时，分子分母同时约去最大公约数
- 课堂思考题：(print-rat (add-rat one-five three-ten))的求值结果从25/50变为了1/2，为什么

```
;有理数,在定义时除以最大公约数
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))

(define (make-rat n d)
  (let ( (g (gcd n d)) )
    (cons (/ n g) (/ d g))))

(define (numer x) (car x))

(define (denom x) (cdr x))
```

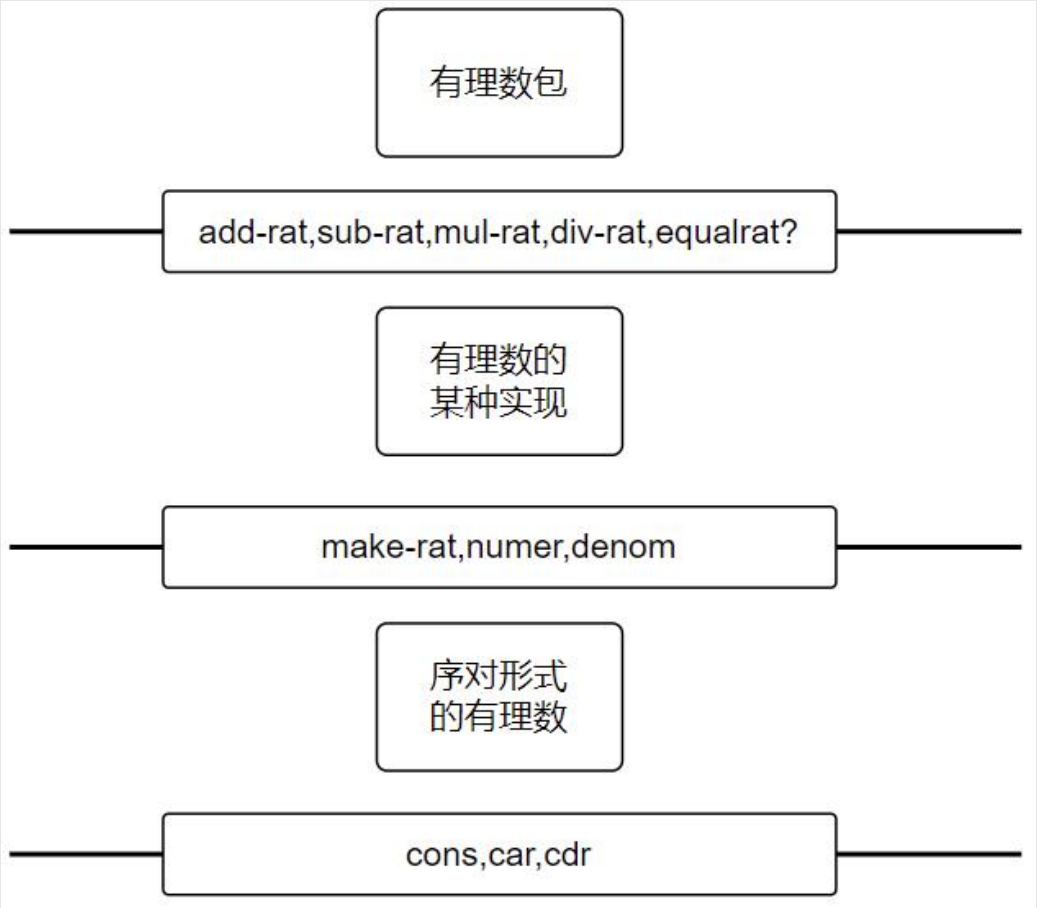
```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.

1/5
3/10
5/6
1/2
1/4
>
```

大纲

- 数据的必要性
- 数据抽象
- 抽象屏障
- 合约（规约），以及数据的过程性表示
- 结束

抽象屏障

有理数的实现展现出一种层次



数据抽象

- 为数据定义接口
- 只基于接口操作数据

抽象屏障

- 一些接口，隔离了系统
- 分隔开使用数据抽象的程序（上面）和实现数据抽象的程序（下层）
- 在有理数的实现中，我们可以找到多层抽象屏障

第一层抽象屏障

- 有理数的加减乘除和相等
- 上面：这些构成了有理数的包
- 下面：这些操作的实现，需要有理数的某种实现并给出构造方法

第二层抽象屏障

- make-rat，numer和denom
- 上面：通过这三个过程，可以构造出以某种方式实现的有理数
- 下面：实现这三个过程，需要使用序对

第三层抽象屏障

- cons，car和cdr
- 上面：通过这三个过程，以序对形式实现有理数
- 下面：序对的具体实现

- 抽象屏障使得程序更容易维护和修改
- 当修改一个过程的时候，不会影响到上层代码
- 如果是重大修改，也许下层实现会被更换
- 需要新实现的代码是“对的”，下一小节叙述这到底意味着什么
- 例如，当分子和分母可约去时，我们可以选择不在构造时约分，而仅在显示时显示约去后的结果

```
;有理数,在numer和denom时除以最大公约数
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))

(define (make-rat n d)
  (cons n d))

(define (numer x)
  (let ( (g (gcd (car x) (cdr x))) )
    (/ (car x) g)))

(define (denom x)
  (let ( (g (gcd (car x) (cdr x))) )
    (/ (cdr x) g)))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.

1/5
3/10
5/6
1/2
1/4
>
```

想象一下

- 实现某种复杂数据（有理数）时，一些预处理（约分）可以在构造时执行，也可以在需要使用时执行
- 如果预测大多数数据会被多次使用，则构造时执行更好
- 如果预测大多数数据只被存储，少数数据才用于计算，则在需要使用时处理更好

数据抽象给了我们一种在设计阶段无需考察此细节的能力

大纲

- 数据的必要性
- 数据抽象
- 抽象屏障
- 合约（规约），以及数据的过程性表示
- 结束

- 我们有了三个不同的有理数实现，区别在于是否执行约分，以及何时执行约分
- 站在使用者的角度，无法区分后两个实现
- 本质上，我们有了关于有理数的两个“正确但不同的实现”
- 这个说法并不严格，因为我们还没有定义什么是正确的实现
- 对（计算结果为数字的）过程，这件事是简单的，只要对相同输入计算出相同输出即可。此时可以指定一个过程是正确的
- 数据的正确性参照类似的思想

抽象数据

- 我们的算法可以视为对抽象数据的实现
- 抽象数据包含构造函数和选择函数
- 构造函数：生成实例
- 选择函数：返回属性

合约（规约 specification）

- 定义了构造函数和选择函数之间的关系
- 用来判断构造函数和选择函数是否被正确地实现

序对的规约

- (car (cons a b))=a
- (cdr (cons a b))=b

有理数的规约

- $\frac{(numer(make-rat\ n\ d))}{(denom(make-rat\ n\ d))} = \frac{n}{d}$

一个数据结构实现是正确的，如果它的所有执行都执行符合规约

伏笔

- 规约这个概念在未来的课程（如数据结构）中将会遇到
- 在一些课程中，将使用一致性这个词代替正确
- 当系统变得更加复杂时（例如在并发环境下），一致性会较为复杂

数据的过程性表示

- 一个抽象数据的实现是正确的，只要其满足规约即可
- 本小节将以令人惊奇的方法实现序对，且满足规约
- 合约的实现只使用第一章内容

序对就是构造出的dispatch方法

- dispatch接收一个参数，依据参数的不同给出不同值
- (car z)时，z是一个这样的序对，即一个dispatch
- 把(car z)，即求序对中的第一个元素，实现为对 dispatch的应用
- (cdr z)时，z也是一个dispatch

以(car (cons 1 2))为例

- (car (cons 1 2)) $\Rightarrow$ (car dispatch) $\Rightarrow$ (dispatch 0)
- $\Rightarrow$ (cond (0=0 $\rightarrow$ 1) (0=1 $\rightarrow$ 2) (else ...)) $\Rightarrow$ 1

```
;数据的过程性实现
(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))
```

到底发生了什么

为什么一个过程dispatch可以像数据一样，作为cons过程的求值结果

- 在1.3节中高阶过程中我们已经看到过类似的现象
- Scheme允许把过程视作某种程度上的数据，作为参数或作为结果返回，即所谓的第一级元素

如果a定义为(cons 1 2)，先后执行(car a)和(cdr a)，可以正常工作吗

- 可以
- (car (cons 1 2)) $\Rightarrow$ (car dispatch) $\Rightarrow$ (dispatch 0)
- $\Rightarrow$ (cond (0=0 $\rightarrow$ 1) (0=1 $\rightarrow$ 2) (else ...)) $\Rightarrow$ 1
- (cdr (cons 1 2)) $\Rightarrow$ (cdr dispatch) $\Rightarrow$ (dispatch 1)
- $\Rightarrow$ (cond (1=0 $\rightarrow$ 1) (1=1 $\rightarrow$ 2) (else ...)) $\Rightarrow$ 1
- 这里要注意的是，a在被替换为(cons 1 2)之后，其内部的dispatch就是使用1和2的

dispatch只有一个，但序对可以有很多。那么a=(cons 1 2)和b=(cons 3 4)在这种实现下是不同的序对吗

- 根据刚才的例子，使用a和b时，会被自动替换为带有不同参数的dispatch
- 因此，不同的序对对应参数不同的dispatch
- 在某种情况下，系统里a和b同时生效。准确来说，可以认为dispatch不是“只有一个”。每次定义序对时就定义了一个“不同”的dispatch

有些微妙的问题

- 跳出代换模型，如果从实现角度出发，dispatch在系统里“有几个实例”？
- 每次执行(cons a b)时，其内部过程dispatch可以视为临时创建的一个全新的过程
- 可以看出，代换模型足以让dispatch具有这样的性质

既然(cons 1 2)和(cons 3 4)是不同的序对，那么写在代码不同地方的两个(cons 1 2)是两个序对还是一个序对

- 嗯，我也不清楚

课堂思考题：给出(car (cons (cons 1 2) 3))的执行过程
伏笔：我们称使用数据的过程性表示的程序设计风格为消息传递。第三章中，我们会遇到更多的消息传递。

理解数据的过程性表示

- 使用数据的过程性表示，你拿到了一个存储了相关信息的过程
- 这一点和我们一般认为的数据是不同的

类比：如果一个数据结构应该包含一颗玉米和一个苹果

- 普通的数据结构：一个盒子，里面装着一颗玉米和一个苹果
- 数据的过程性表示：一个盒子，里面是一个机器，上面有两个按钮
- 按第一个按钮，出现一颗玉米
- 按第二个按钮，出现一个苹果
- 对数据结构的访问被转化为了按这两个按钮

- 站在抽象数据的角度，数据的过程性表示是一个正确的实现
- 如果只能使用cons、car和cdr过程，则无法把新序对和老序对的实现区分开

"On the Internet, nobody knows you're a dog."

- 如果能够考察更多方面，可以发现不同实现的不同，例如执行时间、占用内存，或者其他可以被捕获的特征

- 为了展示新序对实现的正确性，我们在之前的有理数实现中使用新序对的实现

```
;数据的过程性实现
(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))



;有理数,在numer和denom时除以最大公约数
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b)) ))

(define (make-rat n d)
  (cons n d))
```

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.

1/5
3/10
5/6
1/2
1/4
>
```

教材练习2.4

- 另一种数据的过程性表示

```
;2.4
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (a b) a)))
```

课堂思考题

- 解释(car (cons (cons 1 2) 3))的执行过程
- 给出cdr的实现代码

大纲

- 数据的必要性
- 数据抽象
- 抽象屏障
- 合约（规约），以及数据的过程性表示
- 结束

课堂总结

- 数据抽象：分离数据的使用和实现
- 抽象屏障：分层
- 合约（规约）：什么是正确的实现
- 数据的过程性表示：一种风格很奇特的实现

Scheme语法

- cons，car，cdr

本节内容与其他知识的联系

在后续章节中会进一步进展的

- 第二章后面章节中，我们会见到当系统里包含同一抽象数据的多个实现时，怎样处理
- 第三章我们会见到更多消息传递代码

2.1节涉及一些课程

数据结构

- 计算机程序设计的核心课程之一
- 计算机早期：程序=算法+数据结构
- 事实上整个第二章和第三章都和数据结构有着关联

内容（本科）

- 一些经典的抽象数据：队列，栈，优先级队列，链表，树
- 每个抽象数据都有若干种实现

内容（研究生）

- 如何严格定义数据结构的规约和一致性
- 多核和分布式体系下的数据结构算法

λ-演算

- λ-演算是计算机的理论模型之一
- λ-演算非常简洁：简单的运算（抽象和应用）+几条转换规则。细节不予叙述
- 教材练习2.6涉及到λ-演算中的Church计数

Church计数

- λ-演算中并不包含自然数系统
- 通过Church计数，我们可以在λ-演算中编码自然数及自然数计算。进而，可以编码计算机中的计算

预告

- 目前的序对主要用来处理两个数字，而不是多个数字
- 在2.2节，我们通过表来存储多个元素，进一步可以存储树
- 我们可以实现矩阵运算，以及计算8皇后问题

2.1节练习题

练习级

- 教材练习2.1
- 教材练习2.2
- 教材练习2.3

挑战级

- 教材练习2.5
- 教材练习2.6

- 开始学习数据是好事，可以实现更强的算法
- 要习惯过程抽象和数据抽象
- 有问题及时提问，以及和老师交流

下课