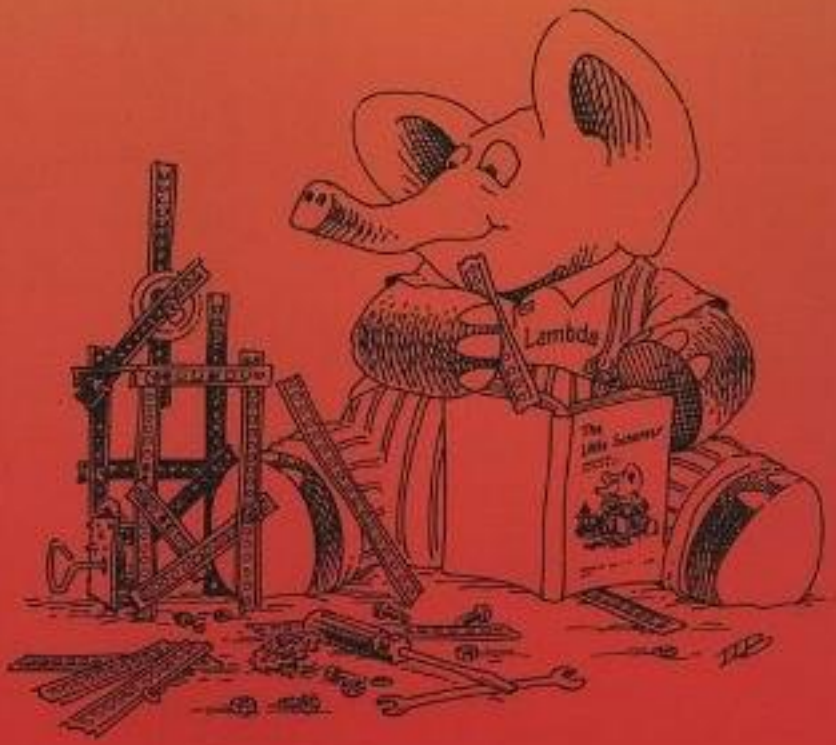
The Little Schemer

Fourth Edition



Daniel P. Friedman and Matthias Felleisen

Foreword by Gerald J. Sussman

The Little Schemer

第四版

Daniel P. Friedman

Indianna University

Bloomington, Indiana

Matthias Felleisen

Rice University

Houston Texas

The MIT Press

Cambridge, Massachusetts

London, England

前言

此前言是第三版The Little Lisper 的前言。经过作者的允许，我们把它放在这里。

1976年，我在学习摄影入门。大多数学生（包括我）学习这们课程是为了变得跟有创造力——拍摄出我敬仰的艺术家 Edward Weston 那样的艺术作品。第一天，老师耐心的讲解了我们这学期将要学习到的技术技巧。重点是Ansel Adams的“Zone System”，用来预可视化冲印照片的值（打印的亮度）和在场景中获得他们的光照强度。为了获得这种技术，我们必须学会使用曝光计测量光的强度和曝光时间的使用和发展的时间来控制的黑色层次和图像的对比度。而这需要耕地层次的技术比如装底片，开发及冲印，混合化学品。摄影师必须学会仪式化敏感材料制定的过程中，使 多年的工作，得到一致的结果。第一次进实验室，满是滑湿的手感，定影液臭气熏天。

但是创意成分呢？为了有创造力，你必须要能控制媒介。没有这些必备的技能是不可能拍摄出伟大的摄影作品的。工程技术，其他的创造性的艺术，我们必须学习分析来支持综合的成果。一个人不可能没有钢铁泥土的知识和计算结构的数学技术去建造一座华丽和实用的桥梁。同样一个人不可能不深入理解如何“预可视化”他/她写的程序产生的进程而构造出精巧的的计算机系统。

一些摄影师选择8×10格而其他人选择35mm的片，这各有各的有缺点。如同摄影，编程序需要选择媒介。Lisp是一种自由和富有弹性的媒介。Lisp 是为递归论和符号代数而构造的理论体系。它已经发展成为一个独一无二的强大和灵活的软件开发工具的家庭体系，Lisp 为快速原型软件系统提供 wrap-around 支持。与其他语言一起，Lisp 为了使用用户社区提交的各种巨大的库当作胶水语言。Lisp 中，procedures是一等数据，被当作参数，返回值，存储在数据中。这种弹性是有价值的，并且更重要的是它提供了通用模式使用必不可少的工程设计——idioms——的正式设立，命名和简化的机制。另外，Lisp 程序可以很容易地操控Lisp程序。这是一个推动程序合成和分析工具大力发展的功能，例如交叉引用。The Little Lisper 方法独特以培养读者在 Lisp 中基础的创意的编程技能。它循序渐进的包含了学习构造递归和操作递归数据结构所需要的训练和练习。The Little LISPer 对与学生学习 Lisp 程序犹如 Hanon 的指法练习或者 Czerny 的练习曲对于学生学习钢琴弹奏一样。

Gerald J. Sussman

Gambridge, Massachusetts

Preface

为了庆祝Scheme的20周年纪念，我们修订The Little LISPer第三次，给了一个更加准确的名字The Little Schemer，和续集The Seasoned Schemer.

程序接受数据和产生数据。设计程序需要对数据有透彻的理解；好的程序反映了数据的结构。大多数数据和大多数程序是递归的。递归是定义一个对象或者靠自己解决解决一个问题的行为。

本书的目的是教会读者递归。第一个问题是使用何种程序来交流这种概念，三种选择：自然语言，如英语；正式数学；或者程序语言。自然语言时含混的不精确的，有时还笨拙冗长。这些是一般交流的优势，但是有时这对交流一些概念，如递归很不简洁。数学与自然语言相反：使用极少的符号就可以表达出强大且正式的想法。不幸的是，数学语言常常是隐蔽的，没有特殊训练几乎不能运用。技术和数学的联姻给我们第三中选择，一个几乎完美的选择：程序语言。我们认为程序语言时是最好的描述递归的方法。他们分享了数学给符号以正式意义的能力。但是与数学不同的是，程序语言是可以直接体验的——你可以拿着本书中的程序，观察他们的行为，修改他们，体验这些修改的效果。

也许教授递归的最好程序语言就是Scheme。Scheme语言本质上是符号性的——程序员不需要考虑他的自己的语言的符号之间的关系以及计算机中表示的关系。递归是Scheme天生的运行机制；程序的初始行为是（潜在的）创建递归定义。Scheme语言的实现主要是交互的，程序员可以立即参与和观察他的程序的行为。另外，这本书中最重要的是，Scheme程序的结构和这些程序操作的数据之间有着直接的对应。

尽管Scheme可以被描述的很正式，理解Scheme却不需要特别的数学角度。实际上，The Little Schemer 是基于向没有编程经历和承认不喜欢任何数学的学生在两周内介绍Scheme快速入门的演讲稿。许多这些同学准备公共事业作为职业。我们认为写Scheme的递归程序本质上就是简单的模式匹配。既然我们唯一考虑的是递归编程，我们仅仅限定处理Scheme特性的一些“为什么”：car, cdr, cons, eq?, null?, zero?, add1, sub1, number?, and, or, quote, lambda, define, 和cond。确实，我们的语言是理想化的Scheme。

The Little Schemer 和 The Seasoned Schemer将不会引导你实际的编程，但是掌握书中的这些概念会让你理解计算的本质。

阅读本书需要读者：认得字，认识数，会算术。

感谢***(略了)

导读

不要快速略读本书。要细细的读；有意义的提示散布在课文中。阅读本书不要少于3次settings。系统地读。如果你没有完全读懂一章，下一章你懂得更少。问题难度渐增；前边的问题不能解决，遇见后边的将束手无策。本书是以对话的形式套路你有趣的Scheme程序例子。如果你可以，读书的时候做作习题。Scheme是可用的。尽管Scheme的不同实现有微小的语法变化（主要是特殊函数的名称和domain的拼写），所有的Scheme都是一样的。为了使用Scheme，你需要定义atom?, sub1, 和add1。我们在The Little Schemer中介绍：

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))
```

为了找出你的Scheme是否定义了正确的atom?, 尝试(atom? (quote ()))然后确保它返回#f。实际上，这在现代Lisp，如Common Lisp中是自带的。为了使用Lisp，你还需要添加函数atom?:

```
(defun atom? (x)
    (not (listp x)))
```

另外你需要做一些程序的修改。特别的，只需要微小的修改。如何尝试修改这些程序的建议在framenotes中。Framenotes中有“S：”的是Scheme相关的，那些有”L:”的是Lisp相关的。

第四章中我们将会从add1，sub1和zero？三种操作中获得基本的算术。Scheme本身没有提供add1和aub1，你必须用内建的基元定义加法和减法。为了避免circularity我们的算术加法和减法我们必须分别使用不同的符号：+和-。

在本书中我们没有给出任何的正式定义。我们认为你得到你自己的定义然后理解记忆他们比我们写给你的好。但是要确信使用前你得透彻理解规则。学习Scheme的关键是“模式匹配“，戒律指出了你看到过的模式。本书中初期，概念简化了；后期，概念得以拓展和限定。记住本树种所有的东西都是Scheme。Scheme本身具有一般普遍性，它包涵了远远多于我们书本中介绍的内容。你掌握了这本书后，就可以阅读更加综合性的Scheme书了。

我们整个的文本中使用了一些符号约定，主要是对不同类别的符号有字体的变化。变量和基本操作符的名称使用斜体。基本数据包括数和真假值是用sans serif。如define, lambda, cond, else, and, or和quote是粗体。当你尝试程序时注意framenotes而应忽略字体。为了突出字体的作用，framenotes中的程序使用打字机字体。当到第十章我们忽略这种字体的区分，在那里我们把程序和代码一视同仁。

最后，Webster把“punctuation”定义为“断句”行为；特别是，为了分隔语句和语句元素使其易懂明了，而使用标准标记来书写和打印的行为。我们已经从字面上采取了这一定义。为了使意思更清晰，我们已经放弃了一些熟悉的使用标点符号。具体来说，当在程序中时，我们丢弃在左栏标点符号的使用。

食品在我们的例子当中出现了很多次。首先，食物比起抽象符号更加容易形象化。（这书作为在消化的时候的读物可不适合 。可别吐一地：)我们希望食物可以帮助你理解我们使用的例子和概念。第二，我们想给你分分心。我们知道这门课程让人多沮丧，分心帮你保持理智。

你可以开始了。祝好运！我们希望你能够享受接下来的挑战。

Bon appetit !

Daniel P. Friedman

Matthias Felleisen

五项规则

> ### *car的规则*
>
> *car只对非空列表有定义。*

> ### *cdr的规则*
>
> *cdr只对非空列表有定义。任何非空列表的cdr是另外一个列表。*

> ### *cons的规则*
>
> *cons有两个参数。cons的第二个参数必须是一个list，结果也是一个list。*

> ### *Null的规则*

> > *Null仅为list定义。*

> ### *eq的规则*
>
> > *eq有两个参数。每一个参数都是非数值atom*

目录

> *内容来源*
>
> *https://github.com/uternet/uternet.github.io*
>
> *https://www.cnblogs.com/Z-X-L/tag/scheme/*
>
> *http://uternet.github.io/TLS/*

这是原子（atom）吗?

`atom`

> 是的，因为atom是一个字母a开头的字符串。

这是原子吗?

`turkey`

> 是的，因为atom是字母开头的字符串。

这是原子吗?

`1492`

> 是的，因为1492是数字的字符串。

这是原子吗?

`u`

> 是的，因为u是字母开头的字符串，仅仅一个字符。

这是原子吗?

`*abc$`

> 是的，因为atom是字母或者除了括号"("和")"外的特殊字符开头的字符串。

这是列表（list）吗?

`(atom)`

> 是的，因为(atom)是一个atom原子外加括号构成。

这是列表吗?

`(atom turkey or)`

> 是的，因为它是由一组原子外加括号构成。

这是列表吗?

`(atom turkey) or`

> 不是，这是两个没有括起来的两个S-表达式表达式，第一个是含有两个原子的列表，第二个是一个原子。

这是列表吗?

`((atom turkey) or)`

> 是的，因为这是两个S-表达式表达式外加括号构成。

这是S-表达式吗?

`xyz`

> 是的，因为所有的atom原子都是S-表达式表达式。

这是S-表达式吗?

`(x y z)`

> 是的，因为它是一个列表。

这是S-表达式吗?

`((x y) z)`

> 是的，因为所有的列表都是S-表达式表达式。

这是list吗?

`(how are you doing so far)`

> 是的，因为它是S-表达式表达式外加括号组成的。

列表中有多少个 S-表达式?

`(how are you doing so far)`

> 六个，how, are, you, doing, so, 和far。

这是list吗?

`(((how) are) ((you) (doing so)) far)`

> 是的，因为它是S-表达式表达式外加括号组成的。

列表中有多少个 S-表达式?

`(((how) are) ((you) (doing so)) far)`

> 三个，((how) are), ((you) (doing so)), 和 far

这是list吗?

`()`

> 是的，因为它是0个S-表达式表达式外加括号组成的。这个特殊的S-表达式称为null(或者empty)列表

这是atom原子吗?

`()`

> 不是，因为()仅仅是一个列表。

这是list吗?

`(() () () ())`

> 是的，因为它是S-表达式表达式外加括号组成的。

下面的声明中的列表 l 的car是什么
`(a b c)`

> 是a，因为a是列表的第一个atom原子

下面的声明中的列表 l 的car是什么
l 是`((a b c) x y z)`

> `(a b c)`, 因为(a b c)是这个非空列表的第一个 S-表达式表达式。

下面的声明中的列表 l 的car是什么
l 是`hotdog`

> 没有答案，atom原子没有car

下面的声明中的列表 l 的car是什么
l 是`()`

> 没有答案，空表没有car。
> 注脚：L：nil

Car的规则

基本操作符car仅为非空列表定义。

下面的声明中的列表 l 的car是什么
l 是`(((hotdogs)) (and) (picle) relish)`

> `((hotdogs))`,叫做“hotdogs的列表的列表。”((hotdog))是l的第一个 S-表达式.

下面的声明中的列表 l ，取`(car l)`是什么
l 是`(((hotdogs)) (and) (picle) relish)`

> `((hotdogs))`, (car l)是另一种“l的car”的写法。

下面的声明中的列表 l 的`(car (car l))`是什么
l 是`(((hotdogs)) (and))`

> (hotdogs)

下面的声明中的列表 l 的cdr是什么
l 是`(a b c)`

> `(b c)`， 因为(b c)是 l 中除了(car l)的部分。
> 注意：“cdr”发音为“could-er”。

下面的声明中的列表 l 的cdr是什么
l 是`((a b c) x y z)`

> `(x y z)`.

下面的声明中的列表 l 的cdr是什么
l 是`(hamburger)`

> `()`

下面的声明中的列表 l ， 取 (cdr l) 是什么
l 是`((x) t r)`

> `(t r)`，因为(cdr l) 仅仅"列表 l 的cdr"的另一种表示方法。

下面的声明中a的cdr是什么
`(cdr a)`
假设 a 是 `hotdogs`

> 没有答案，atom原子没有cdr。

下面的声明中的列表 l 的cdr是什么
`(cdr l)`
假设 l 是()

> 没有答案，空表null没有cdr。

---

cdr的规则

基本操作符cdr仅为非空列表定义。
任何非空list的cdr都是另一个list。

下面的声明中的列表 l ， 取`(car (cdr l))`是什么
假设 l 是`((b) (x y) ((x)))`

> `(x y)`， 因为((x y) ((c)))是(cdr l), (x y) 是(cdr l)的car。

下面的声明中的列表 l ， 取`(cdr (cdr l))`是什么
假设 l 是`((b) (x y) ((c)))`

> `(((c)))`, 因为((x y) ((c)))是(cdr l)，而(x y)是(cdr l)的car。

下面的声明中的列表 l ， 取`(cdr (car l))`
l 是`(a (b (c)) d)`

> 没有答案，因为(cdr l)是一个atom原子，它不能取cdr了。

---

car 的参数可以是什么

> 任何非空列表。

---

cdr 的参数可以是什么

> 任何非空列表。

假设 a 是 `peanut`，l 是 `(butter and jelly)`
那么a和l的cons(这通常写作"(cons a l)"
读作把原子a cons到列表 l 上)是什么

> `(peanut butter and jelly)`
> 因为cons把一个atom原子追加到列表最前头。

假设 s 是 `(banana and)`，l 是 `(peanut butter and jelly)`
那么把s cons到l上是什么

> `((banana and) peanut butter and jelly)`
> 因为cons任意 S-表达式追加到列表最前头。

假设 s 是 `((help) this)`，l 是 `(is very ((hard) to learn))`
那么 `(cons s l)` 是什么

> `(((help) this) is very ((hard) to learn))`。

---

cons 的参数可以是什么

> cons带两个参数，第一个可以是任何 S-表达式，第二个是任意列表。

假设 s 是 `(a b (c))`，l 是 `()`，那么 `(cons s l)` 是什么

> `((a b (c)))`

假设 s 是 `a`，l 是 `()`，那么 `(cons s l)` 是什么

> `(a)`

假设 s 是 `((a b c))`，l 是 `b`，那么 `(cons s l)` 是什么

> 没有答案，因为 l 不是list不能cons。

---

cons的规则

```
<p>基本操作符cons有两个参数。
cons的第二个参数必须是列表，
结果返回一个列表。</p>
```

假设 s 是`a`，l 是`((b) c d)`
那么`(cons s (car l))`是什么

> `(a b)`

假设 s 是`a`，l 是`((b) c d)`
那么`(cons s (cdr l))`是什么

> `(a c d)`。

假设 l 是`()`，那么l 是null 列表，对么

> 是的，因为它是一个含有0个 S-表达式表达式的列表。

`(null? (quote ()))`是什么

> `true`真。因为`(quote())`是空表null的写法。

假设 l 是`(a b c)`，那么`(null? l)`是真还是假

> 假。l 是非空集。

假设 a 是`spaghetti`，那么`(null? a)`是真还是假

> 没有答案，atom原子没有真假。

Null的规则

基本操作符null?仅仅为列表定义

假如 s 是`harry`，那么 s 是否是atom原子

> 真。因为harry是字母开始的字符串。

如果 s 是`harrry`那么`(atom? s)`是什么

> 真。因为`(atom? s)`是另外一种表达"s是否是一个atom原子"的方式。
> 注:

```
L:(defun atom? (x)
    (not (listp x)))
S:(define atom?
    (lambda (x)
        (and (not (pair? x)) (not （null? x))))
```

假如 s 是`(Harry had a heap of apples)`，那么`(atom? s)`是真是假

> 假。因为s是一个列表。

---

atom? 有几个参数

> 一个。这个参数可以是任意 S-表达式。

假如 s 是 `(Harry had a heap of apples)`
那么 `(atom? (car l))` 是真是假

> 真。

假如 s 是 `(Harry had a heap of apples)`
那么 `(atom? (cdr l))` 是真是假

> 假。

假如l 是 `(Harry)`
那么 `(atom? (cdr l))` 是真还是假

> 假，因为空列表不是一个原子

假如 s 是 `(swing low sweet cherry oat)`
那么 `(atom? (car (cdr l)))` 是真是假

> 真。

假如 s 是 `(swing (low sweet) cherry oat)`
那么 `(atom? (car (cdr l)))` 是真是假

> 假。

假如 a1 是 `Harry`，a2是 `Harry`
那么 a1 和 a2 是否相等

> 真。因为a1和a2完全相同。

假如 a1 是 `Harry`，a2是 `Harry`
那么 `(eq? a1 a2)` 是真还是假

> 真。因为 `(eq? a1 a2)` 是另外一种表达"a1和a2是否是同一个非数值atom原子"的方式。
> 注：Lisp：`eq`

假如 a1 是 `margarine`，a2是 `butter`
那么 `(eq? a1 a2)` 是真还是假

> 假。a1和a2不同。

`eq?` 需要多少参数？各是什么？

> 两个参数，都是非数值atom原子。

假设 l1 是`()`，l2是`(strawberry)`
那么`(eq? l1 l2)`是真还是假

> 没有答案，`()`和`(strawberry)`是列表。
> 注：实际当中是可以作为eq？的参数的。

假设 n1 是`6`，n2是`7`，那么`(eq? n1 n2)`是真还是假

> 没有答案，因为两个参数是数字。
> 注：实际当中是可以作为eq？的参数的。

---

eq？的规则

基本操作符eq？带有两个参数，
两个都是非空数值的atom原子

假设 l 是`(Mary had a little lamb chop)`
a 是`Mary`
那么`(eq? (car l) a)`是真还是假

> 真。

假设 l 是`(soured milk)`，
a 是`Mary`，
那么`(eq? (cdr l) a)`是真还是假

> 没有答案。

假设 l 是`(beans beans we need jelly beans)`，
`(eq? (car l) (car (cdr l)))`
是真还是假

> 真。因为这是在比较列表中的第一个和第二个atom原子。

现在去，做出自己的花生酱果冻三明治！

假设 l 是 `(Jack Sprat could eat no chicken fat)`
那么 `(lat? l)` 是真还是假

> 真。因为每个list表l中的 S-expression 都是atom原子。

假设 l 是 `((Jack) Sprat could eat no chicken fat)`
那么 `(lat? l)` 是真还是假

> 假。因为(car l)是一个list。

假设 l 是 `(Jack (Sprat could) eat no chicken fat)`
那么 `(lat? l)` 是真还是假

> 假。因为list表l的其中一个 S-expression 不是list。

假设 l 是`()`，(lat? l)是真还是假

> 真。因为list表l中没有list表。

请问：lat是一个单纯由atom原子构成的列表,对吗?

> 对。每个lat都是一个list的atom原子。

用以下函数写一个lat?函数
car cdr cons null? atom? 和eq?

> 你现在还不能完成这个问题，因为你还有些东西没有学到。看下面一个问题。

休息好了吗?

```
(define lat?
  (lambda (l)
    (cond
     ((null? l) #t)
     ((atom? (car l)) (lat? (cdr l)))
     (else #f))))
```

假设 l 是 `(bacon and eggs)`，那么`(lat? l)`是什么

> #t——真——因为l是一个lat
> 注：Lisp中：

```
(defun lat? (l)
  (cond
    ((null l) t)
    ((atom? (car l)) (lat? (cdr l)))
    (t nil)))
```

我们怎么推知 `(lat? l)` 的答案是 `#t`

> 现在这个问题你也回答不了。这个问题由lat?的查询决定。
> 提示：写下lat?的定义然后看下面的系列问答。

`(lat? l)` 的第一个查询是什么

> (null? l)
> 注意：
> (cond ...) 提问
> (lambda ...) 产生函数
> (define ...) 给出函数名

---

cond 行
`((null?l) #t)`
是什么意思
其中 l 是 `(bacon and eggs)`

> (null? l) 查询参数l是否是null。如果是，则是真；否则为假，cond执行下一个查询。

下一个查询是什么

> `(atom? (car l))`

下面这行是什么意思
其中 l 是 `(bacon and eggs)`
`((atom? (car l)) (lat? (cdr l)))`

> (atom? (car l))询问list表l的第一个 S-expression 是否是atom。如果(car l)是一个atom原子，那么我们要知道l的其余组成部分是否是atom原子。如果(car l)不是atom原子，我们询问下一个查询。在当前情况下(car l)是一个atom原子，所以值由函数(lat? (cdr l))决定。

`(lat? (cdr l))` 是什么意思

> (lat? (cdr l))通过lat?查询新的参数发现是否list表l的剩余组成部分都是是atom原子。

现在lat?的参数l是什么

> 现在的 l 是(cdr l)，也就是 (and eggs)

下一个查询是什么

> (null? l)

下面这行是什么意思，其中l现在是 `(and eggs)`
`((null? l) #t)`

> (null? l) 查询参数l是否是null。如果是，则是#t真；否则cond执行下一个查询。

下一个查询是什么

> `(atom? (car l))`

下面这行是什么意思，其中 l 是 `(and eggs)`
`((atom? (car l)) (lat? (cdr l)))`

> `(atom? (car l))` 询问list表l的第一个 S-expression 是否是atom。如果(car l)是一个atom原子，那么我们要知道l的其余组成部分是否是atom原子。如果(car l)不是atom原子，我们询问下一个查询。在当前情况下(car l)是一个atom原子。

`(lat? (cdr l))` 是什么意思

> (lat? (cdr l))通过lat?查询新的参数发现是否list表l的剩余组成部分都是是atom原子，这一次(cdr l)是(eggs)。

下一个查询是什么

> `(null? l)`

下面这行是什么意思，其中l现在是 (eggs)
((null? l) #t)

> (null? l) 查询参数l是否是null。如果是，则是#t真；否则cond执行下一个查询。此情况下，l不是null，所以查询下一个问题。

下一个查询是什么

> `(atom? (car l))`

下面这行是什么意思，其中 l 是 `(and eggs)`
`((atom? (car l)) (lat? (cdr l)))`

> (atom? (car l))询问list表l的第一个 S-expression 是否是atom。r如果(car l)是一个atom原子，那么我们要知道l的其余组成部分是否是atom原子。如果(car l)不是atom原子，我们询问下一个查询。在当前情况下(car l)是一个atom原子，所以再一次我们查询(lat? (cdr l))。

`(lat? (cdr l))` 是什么意思

> (lat? (cdr l))通过lat?查询新的参数发现是否list表l的剩余组成部分都是是atom原子，其中新查询的参数l变为(cdr l)。

现在lat?的参数是什么

> ()

下面这行是什么意思，其中l现在是 `()`
`((null? l) #t)`

> (null? l) 查询参数l是否是null。如果是，则是#t真；否则cond执行下一个查询。此情况下，l是空表()，所以(lat? l)是#t真，其中 l 是(bacon and eggs)。

还记得 (lat? l) 的查询吗

> 记不得了吧?
> 当l为(bacon and eggs)时，操作(lat? l)的值为#t

你可以用自己的语言描述一下函数lat?做了什么吗

> 这是我们的描述
> “lat? 查找列表中的每一个 S-expression表达式，看每一个 S-expression表达式是否是atom原子，直到没有 S-expression了。如果从头至尾没有遇到一个list表，那么返回#t，否则返回#f——false”

下面再写一次lat?的定义

```
(define lat?
  (lambda (l)
    (cond
      ((null? l) #t)
      ((atom? (car l)) (lat? (cdr l)))
      (else #f))))
```

`(lat? l)` 的值是多少，其中l现在是`(bacom (and eggs))`

> `#f`

第一个查询是什么

> `(null? l)`

`((null? l) #t)` 是什么意思
其中 l 是 `(bacon (and eggs))`

> (null? l) 查询参数l是否是null。如果是，则是真；否则为假，cond执行下一个查询。此情况下，l非空，进行下一个查询。

下一个查询是什么

> `(atom? (car l))`

下面这行是什么意思，其中 l 是`(bacon (and eggs))`
`((atom? (car l)) (lat? (cdr l)))`

> (atom? (car l))询问list表l的第一个 S-expression 是否是atom。如果(car l)是一个atom原子，那么我们要知道l的其余组成部分是否是atom原子。如果(car l)不是atom原子，我们询问下一个查询。在当前情况下(car l)是一个atom原子，所以我们查询list表l的剩余部分看是否都是原子。

`(lat? (cdr l))`
是什么意思

> (lat? (cdr l))通过lat?查询新的参数发现是否list表l的剩余组成部分都是是atom原子，其中把l替换为(cdr l)。

下面这行是什么意思，其中l现在是`((and eggs))`
`((null? l) #t)`

> (null? l) 查询参数l是否是null。如果是，则是#t真；否则cond执行下一个查询。

下一个查询是什么

> `(atom? (car l))`

下面这行是什么意思，其中 l 是`((and eggs))`
`((atom? (car l)) (lat? (cdr l)))`

> (atom? (car l))询问list表l的第一个 S-expression 是否是atom。如果(car l)是一个atom原子，那么我们要知道l的其余组成部分是否是atom原子。如果(car l)不是atom原子，我们询问下一个查询。在当前情况下(car l)不是一个atom原子，进行下一个查询。

下一个查询是什么

> `else`

---

else查询是什么意思

> else查询是否为真

---

else是真吗

> 是的，else总是真。

---

else

> 当然of course

为什么else是最后一个查询

> 因为我们不需要其它查询

为什么我们不需要其它查询

> 因为list可以是空表，也可以在第一个位置是atom原子，也可以在第一个位置是list表

`(else #f)` 是什么意思

> else查询else是否是真。else本身是真查询#f，得到#f假。

解释下
)))

> 匹配前边的(cond ..,(lambda...和(define...括号

你能描述一下我们如何得出 `(lat? l)` 是 #f的，其中 l 是 `(baco (and eggs))`

> 一种解释是
> “(lat? l)查询参数中的每一项是否是原子。如果在发现一个list之前就结束，那么值是#t。如果如这个例子(bacon (and eggs))中它找到一个list，那么(lat? l)的值就是#f”

问 `(or (null? l1) (atom? l2))` 是真还是假， 其中 l1 是 `()`， l2 是 `(d e f g)`

> 真，因为(null? l1)是真。

问 `(or (null? l1) (null? l2))` 是真还是假， 其中 l1 是 `(a b c)`， l2 是 `()`

> 真，因为(null? l2)是真。

问(or (null? l1) (null? l2))是真还是假， 其中 l1 是(a b c)， l2 是(atom)

> 假，因为无论(null? l1)还是(null? l2))是都是假。

`or ...)` 是做什么的

> (or ...)查询两个问题，每次一个。如果第一个是真那么停止并回答真。否则查询第二个问题并以第二个问题的回答作为答案。

问 a 是否是lat的一个成员，其中 a 是 `tea`，lat是 `(coffee tea or milk)`

> 真。

问 `(member? a lat)` 是真还是假，其中 a 是 `poached`，lat是 `(fried eggs and scrambled eggs)`

> 假。

下面是函数member?的定义

```
(define member?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      (else (or (eq? (car lat) a)
                (member? a (cdr lat)))))))
```

问 `(member? a lat)` 是什么 其中 a 是 `meat`，lat是 `(mashed potatoes and meat gravy)`

> 真 #t,因为atom原子meat是lat中的一个原子。

我们怎么推知是#t

> 值由查询(member? a lat)得到。提示：写出member?函数然后对照下面的问题。

`(member? a lat)` 的第一个查询是什么

> `(null? lat)`。这也是lat?的第一个查询。

第一戒

```
<p>(开端)</p>
<p>任何函数的第一个查询表达式重视null?</p>
```

`((null? lat) #f)`
其中lat是 `(mashed potatoes and meat gravy)`

> (null? lat) 查询参数l是否是null。如果是，则值是#f，
> 否则为假，cond执行下一个查询。

下一个查询是什么

> `else`。

为什么else是下一个查询

> 因为我们不需要其它查询。

---

else真的是一个查询吗

> 是的，else总是真。

下面这行是什么意思

```
(else (or (eq? (car lat) a)
                (member? a (cdr lat))))
```

> 现在我们知道lat不是null，我们必须找出lat的car是不是a，或者lat的其他地方是不是a。

问 a 是 `meat`，lat是 `(mashed potatoes and meat gravy)` 时

```
(or (eq? (car lat) a)
    (member? a (cdr lat))))
```

是真还是假

> 下面一点点分析

问 a 是`meat`，lat是`(mashed potatoes and meat gravy)`时
`(eq? (car lat) a)`是真还是假

> 假。

`(or ...)`的第二个查询是什么

> `(member? a (cdr lat))`。递归查询函数`menber?`，参数lat变为`(cdr lat)`

现在member?的参数是什么

> lat现在是`(cdr lat)`,即`(potatoes and meat gravy)`

下一个查询是什么

> `(null? lat)`。记得第一戒。

当lat是(potatoes and meat gravy)时，(null? lat) 真还是假

> 假#f。

下面做什么

> 下一个查询。

下面这行是什么意思

```
(or (eq? (car lat) a)
    (member? a (cdr lat))))
```

> 找出 a 是否与lat的car是eq?相同或者递归查找 a 是否是lat的cdr中的成员。

`(eq? a (car lat))`

> No,因为 a 是 `meat`, 而(car lat)等于`potatoes`

下面做什么

> 查询`(member? a (cdr lat)

现在member的参数是什么

> a 是`meat`，lat是`(and meat gravy)`

下一个查询是什么

> (null? lat)`

下面做什么

> 因为`(null? lat)`是假，下一个查询

下一个查询是什么

> `else`

下面的代码是什么值

```
(or (eq?  (car lat) a)
    (member? a (cdr lat)))
```

> #t。因为(car lat)即meat与a相同都是原子meat，所以(or ...)是真。

`(member? a lat)`的值是什么，其中 a 是`meat`，lat是`(meat gravy)`

> #t

`(member? a lat)`的值是什么，其中 a 是`meat`，lat是`(and meat gravy)`

> #t

`(member? a lat)`的值是什么，其中 a 是`meat`，lat是`(potatoes and meat gravy)`

> #t

`(member? a lat)`的值是什么，其中 a 是`meat`，lat是`(mashed potatoes and meat gravy)`

> #t

为了确保你的确会了，让我们快速的再来一次
`(member? a lat)`的值是什么，

其中 a 是`meat`，
lat是`(mashed potatoes and meat gravy)`

> #t
> 提示：写下member?函数的定义然后参考下面的问题。

`(null? lat)`

> 否,移动到下一行。

`else`

> 是

```
(or (eq? (car lat) a)
    (member? a (cdr lat))))
```

> 或许

`(eq? (car lat) a)`

> 否，下个查询

下个查询是什么

> 函数递归参数 a 和(cdr lat)， a 是`meat`，(cdr lat)是`(potatoes and meat gravy)`。

`(null? lat)`

> 否。下一行。

`else`

> 是，但是`(eq? (car lat) a)`为否。函数递归参数 a 和(cdr lat)，a为`meat`，(cdr lat)为`(and meat gravy)`

`(null? lat)`

> 否。下一行。

`else`

> 是，但是`(eq? (car lat) a)`为否。函数递归参数 a 和(cdr lat)，
> a 是`meat`，lat是`(meat gravy)`。

`(null? lat)`

> 否，下个查询。

`(eq? (car lat) a)`

> 是，值为#t。

```
(or (eq? (car lat) a)
    (member? a (cdr lat))))
```

> #t
> 回溯上去知道a为`meat`，lat为`(mashed potatoes and meat gravy)`时为#t真。

`(member? a lat)` 的值是什么，其中 a 是 `liver`，lat是 `(bagels and lox)`

> `#f`

让我们看看为什么是#f。第一个查询是什么

> `(null? lat)`

`(null? lat)`

> 否。下一行。

`else`

> 是，但是 `(eq? (car lat) a)` 是false。函数递归参数 a 和(cdr lat)，其中 a 是liver，(cdr lat)是(and lox)。

`(null? lat)`

> 否，下个查询

`else`

> 是，但是(eq? (car lat) a)是false。函数递归参数 a 和(cdr lat)，其中 a 是liver，(cdr lat)是(lox)。

`(null? lat)`

> 否，下个查询

`else`

> 是，但是(eq? (car lat) a)是false。函数递归参数 a 和(cdr lat)，其中 a 是liver，(cdr lat)是()。

`(null? lat)`

> 是。

`(member? a lat)` 的值是什么，其中 a 是 `liver`，lat是 `()`

> `#f`

下面的代码是什么值，其中 a 是 `liver`，lat是 `(lox)`

```
(or (eq? (car lat) a)
  (member? a (cdr lat)))
```

> `#f`

`(member? a lat)` 的值是什么，其中 a 是 `liver`，lat是 `(lox)`

> `#f`

下面的代码是什么值，其中 a 是`liver`，lat是`(and lox)`

```
(or (eq? (car lat) a)
  (member? a (cdr lat)))
```

> `#f`

`(member? a lat)`的值是什么，其中 a 是`liver`，lat是`(and lox)`

> `#f`

下面的代码是什么值，其中 a 是`liver`，lat是`(bagels and lox)`

```
(or (eq? (car lat) a)
  (member? a (cdr lat)))
```

> `#f`

`(member? a lat)`的值是什么，其中 a 是`liver`，lat是`(bagels and lox)`

> `#f`

相信这些吗？你可以休息下了！

`(rember a lat)` 是什么，其中
a 是 `mint`，
lat 是 `(lamb chops and mint jelly)`

> `(lamb chops and jelly)`
> "rember"表示remove a member 删除一个成员。

`(rember a lat)`
是什么，其中
a 是 `mint`
lat 是 `(lamb chops and mint flavored mint jelly)`

> `(lamb chops and flavored mint jelly)`

`(rember a lat)`
是什么，其中
a 是 `toast`
lat 是 `(bacon lettuce and tomato)`

> `(bacon lettuce and tomato)`

`(rember a lat)`
是什么，其中
a 是 `cup`
lat 是 `(coffee cup tea cup and hick cup)`

> `(coffee tea cup and hick cup)`

`(rember a lat)` 做什么

> 删除lat中的第一个和参数a相同的原子

那么我们应该做哪些步骤

> 首先测试 `(null? lat)`——这是第一戒

如果 `(null? lat)` 是真呢

> 返回 ()。

如果 `(null? lat)` 是假呢

> 至少lat中有一个atom原子

需要其它查询吗

> 不需要。要么 lat 是空要么至少有一个原子。

如果至少有一个atom原子呢

> 查询 a 是否与`(car lat)`相同。

怎么查询

> 使用

```
(cond
  (_____ _______)
  (_____ _______)
```

怎么查询 a 是否与(car lat)相同

> `(eq? (car lat) a)`

如果 a 与`(car lat)`相同那么`(rember a lat)`的值是什么

> `(cdr lat)`

如果 a 与`(car lat)`不相同那么`(rember a lat)`的值是什么

> 保留`(car lat)`，但是寻找lat的其他部分是否有a

怎么删去 lat 其余部分的第一个 a

> `(rember a (cdr lat)`

还有查询吗

> 没了

现在让我们写出刚才想到的函数rember

```
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) '())
      (else (cond
              ((eq? a (car lat)) (cdr lat))
              (else (rember a (cdr lat))))))))
```

a 是`bacon`
lat 是`(bacon lettuce and tomato)`
那么`(rember a lat)`的值是什么?

> `(lettuce and tomato)`
> 提示:写下rember函数和参数，然后参考下面的系列问题。

看看函数是否正常。第一个查询是什么

> `(null? lat)`

现在呢

> 移动到下一行。

else

> 是。

然后呢

> 移动到下一行。

`(eq? (car lat) a)`

> 是，所以值是(cdr lat)，在此例子中为(lettuce and tomato)。

这是正确的吗

> 是的，所以值是`(cdr lat)`。此例中为`(lettuce and tomato)`

但是，我们的用例对么

> 谁知道啊?但是，证明甜点是用吃，让我们试试其它例子。

rember是做什么的

> 它的参数为一个atom原子和一个lat，rember在第二个参数中寻找出第一个出现的第一个参数并把它删除。

现在呢

> 把 a 对比lat中的每一个atom原子，如果没有找到，那么我们把刚才对比的原子放入新的list中。

`(rember a lat)`
是什么值，其中
a 是`and`
lat 是`(bacon lettuce and tomato)`

> `(bacon lettuce tomato)`

让我们看看我们的rember函数是否正确。rember的第一个查询是什么

> `(null? lat)`

下面呢

> 移动到下一行。

`else`

> OK，下一个问题

`(eq? (car lat)a)`

> 否，移动到下一行

`(else (rember a (cdr lat)))`
是什么意思

> `else`询问真假。下一行用参数a和`(cdr lat)`递归查询， a 是`and`，`(cdr lat)`是`(lettuce  and tomato)`

`(null? lat)`

> 否，移动到下一行

`else`

> 确定

`(eq? (car lat) a)`

> 否，移动到下一行

`(rember a (cdr lat))`
的意思是什么

> 递归查询， a 是`and`，`(cdr lat)`是`(and tomato)`

`(null? lat)`

> 否，移动到下一行

---

else

> 当然of course

`(eq? (car lat) a)`

> 是。

所以结果是什么

> `(cdr lat)` --> `(tomato)`

正确吗

> 不对，正确的应该是`(bacom lettuce and tomato)`

我们做错什么了

> 我们把 and，以及 and 以前的东西都丢弃了。

我们怎么才能留住bacon和lettuce

> 使用cons，还记得第一章的cons吗

第二戒

使用cons构建list表

让我们看看使用了cons后的rember函数

```
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) '())
      (else (cond
              ((eq? (car lat) a) (cdr lat))
              (else (cons (car lat)
                (rember a (cdr lat)))))))))
```

假设 a 是 `and`
lat 是 `(bacon lettuce and tomato)`
`(rember a lat)` 的值是什么

> `(bacon lettuce tomato)`

第一个问题是什么?

> `(null? lat)`

现在做什么?

> 第二个问题

---

else

> 是的，当然。

`(eq? (car lat) a)`

> No, 移动到下一行

这是什么意思：

```
(cons (car lat)
  (rember a (cdr lat)))
```

现在 a 是 `and`
lat 是 `(bacon lettuce and tomato)`

> 它表示，把 lat 的 car ——bacon ， cons 到 `(rember a (cdr lat))` 的结果上。
> 但是现在我们还不知道具体的结果。

`(rember a (cdr lat))`
是什么意思

> 重新执行函数rember，现在参数lat被替换成了(cdr lat) ——`(lettuce and tomato)`了。

`(null? lat)

> No, 移动到下一行

---

else

> 是的，下个问题

`(eq? (car lat) a)`

> No, 移动到下一行

这是什么意思：

```
(cons (car lat)
  (rember a (cdr lat)))
```

> 它表示，把 lat 的 car ——lettuce ， cons 到 `(rember a (cdr lat))` 的结果上。
> 但是现在我们还不知道具体的结果。

`(rember a (cdr lat))`
是什么意思

> 重新执行函数rember，现在参数lat被替换成了(cdr lat) ——`(and tomato)`了。

`(null? lat)`

> No，下个问题

---

else

> 仍然

`(eq? (car lat) a)`

> Yes.

`((eq? (car lat) a) (cdr lat))`
的值是什么

> (cdr lat) —— `(tomato)`

完成了吗?

> 没有!
> 我们现在知道当 lat等于`(and tomato)`时，`(rember a lat)`的结果。
> 但是我们还不知道当 lat等于
> `(lettuce and tomato)`
> 以及
> `(bacon lettuce and tomato)`
> 时，`(rember a lat)`的结果

现在我们得到了`(rember a (cdr lat))`的值
当 a 是`and`
`(cdr lat)`是`(and tomato)`
这个值就是`(tomato)`

那么接下来呢?

> 回忆下刚才，我们试图把lettuce cons 到`(rember a (cdr lat))`的结果上，现在我们有了这个结果——`(tomato)`，我们可以把lettuce cons 进去了。

把 lettuce cons 到 (tomato)的结果是什么

> (lteeuce tomato)

`(lettuce tomato)`代表了什么

> 它代表了
>
> ```
> 1 (cons (car lat)
> 2 (rember a (cdr lat)))
> ```
>
> 的值，当时，lat 是`(lettuce and tomato)`
> 而`(rember a (cdr lat))`的值是`(tomato)`

完成了吗?

> 还没有。我们知道了当
> lat 是 `(lettuce and tomato)` 时
> `(rember a lat)` 的值，但是，我们还不知道当
> lat 是 `(bacon lettuce and tomato)` 时它的值是什么

现在我们得到了 `(rember a (cdr lat))` 的值，
当 a 是 `and`
且 `(cdr lat)` 是 `(lettuce and tomato)`
这个值就是 `(lettuce tomato)`

接下来我们必须再次做什么?

> 再回忆下，我们曾试图把 `bacon` cons 到 `(rember a (cdr lat))` 的结果中去。
> 当时，a 是 `and`
> `(cdr lat)` 是 (lettuce and tomato) 现在，我们有了这个值，它就是 (lettuce tomato)`
> 我们可以cons了。

把 `bacon` cons 到 `(lettuce tomato)` 的结果是什么?

> `(bacon lettuce tomato)`

`(bacon lettuce tomato)` 代表了什么?

> 它代表了
>
> ```
> 1 (cons (car lat)
> 2 (rember a (cdr lat)))
> ```
>
> 的值，lat 是 `(bacon lettuce and tomato)`
> `(rember a (cdr lat))` 是 `(lettuce tomato)`

现在我们全都完成了吗?

> 是的。

你能用你自己的话描述一下最终得到的这个值:
`(bacon lettuce tomato)` 吗

> 函数 `rember` 依次检查 `lat` 中的每一个原子是否与 `and` 相同，如果 `car lat` 与 `and` 并不相同，我们将它保存下来，稍后再cons到最终的值上面去。如果rember在lat中找到了 `and` 原子，则丢弃它。返回lat的剩余部分，然后再将先前暂存的原子cons到最终结果上去。（乱译）

你可以重写rember函数，使它更能反映上面的描述吗?

> 是的，我们可以简化它

```
(define rember
(lambda (a lat)
(cond
((null? lat) '())
((eq? (car lat) a) (cdr lat))
(else (cons (car lat)
            (rember a (cdr lat)))))))
```

(验证重写的rember函数是否正确，略过。)

`(firsts l)`是什么，其中l是

```
((apple peach pumpkin)
 (plum pear cherry)
 (grape rasin pea)
 (bean carrot eggplant))
```

> `(apple plum grape bean)`

`(firsts l)`
是什么，其中l是`((a b) (c d) (e f))`

> `(a c e)`

`(firsts l)`是什么，其中l是`()`

> `()`

`(firsts l)`是什么，其中l是

```
((five plums)
 (four)
 (eleven green oranges))
```

> `(five four eleven)`

`(firsts l)`是什么，其中l是

```
(((five plums) four)
 (eleven green oranges)
 ((no) more))
```

> `((five plums) eleven (no))`

现在，用你自己的话说说(firsts l)做什么

> 下面是我们的尝试
> “firsts函数输入一个lists表参数，null空表，或者只包含非空表。它抽取list表中每一个成员的第一个 S-expression表达式构建新的list表。”

试试你能不能写出函数firsts，记得戒律。

```
(define firsts
(lambda (l)
(cond
 ((null? l) ...)
 (else (cons ... (firsts (cdr l)))))))
```

为什么要

```
(define firsts
  (lambda (l)
    ...))
```

> 因为要定义函数名`firsts`和参数(l)

为什么`(cond ...)`

> 因为我们对参数需要查询问题

为什么要
`((null? l) ...)`

> 第一戒

为什么`(else`

> 因为我们对参数l只有两个查询。

为什么`(else`

> 看上面。另外最后一个查询必须总是else

为什么`(cons`

> 因为我们需要创建list表——第二戒

为什么需要`(firsts (cdr l))`

> 因为我们每次只能看到一个S-表达式，余下的必须递归。

为什么`)))`

> 与(cond, (lambda, (define, 的括号匹配。

回忆下firsts的定义。`(firsts l)`的典型元素是什么
当l是`((a b) (c d) (e f))`

> a

剩下的典型元素是什么

> c，还有后边的e

考虑一下seconds的定义。
`(seconds l)`的元素是什么，其中l是
`((a b) (c d) (e f))`

> b, d, 或者 f。

我们怎么描述`(firsts l)`的典型元素

> 参照第一章，那么l的一个元素的car为`(car (car l))`

找到一个典型的`(firsts l)`后，我们该做什么

> 把它cons到递归`(firsts (cdr l))`中。

第三戒

当构建list表时，描述第一个典型元素，然后把它cons到自然递归中去。

有了第三戒，我们可以添入firsts函数更多部分

```
(else (cons (car (car l)) (firsts (cdr l)) ))
            ~~~~~~~~~~~~  ^^^^^^^^^^^^^^^^
              典型元素        自然递归
```

(firsts l)结果是什么，其中l是`((a b) (c d) (e f))`

```
(define firsts
  (lambda (l)
    (cond
      ((null? l) ...)
      (else (cons (car (car l)) (firsts (cdr l)))))))
```

> 什么也没有。我们的菜谱中还缺点重要的成分。不过，我们现在可以就可以执行它。

`(null? l)`为真吗
其中l是`((a b) (c d) (e f))`

> 否，下个查询

下面的代码是什么意思
`(cons (car (car l)) (firsts (cdr l)))`

> 把`(car (car l))`cons到`(firsts (cdr l))`。为了找出`(firsts (cdr l))`，我们用新参数`(cdr l)`递归函数

`(null? l)`
其中l是`((c d) (e f))`

> 否，移动到下一行

下面的代码是什么意思
`(cons (car (car l)) (firsts (cdr l)))`

> 保存`(car (car l))`，递归`(firsts (cdr l))`

`(null? l)`为真吗
其中l是`((e f))`

> 否，移动到下一行

下面的代码是什么意思
`(cons (car (car l)) (firsts (cdr l)))`

> 把`(car (car l))`cons到`(firsts (cdr l))`。为了找出`(firsts (cdr ))`，我们用新参数`(cdr l)递归函数

`(null? l)`为真吗

> 是。

现在`((null? l) ...)`的值是什么

> 没有值，缺了点的东西。

我们应该把原子cons到什么上?

> 一个list。还记得cons的规则。

为了达到cons，当`(null? l)`为真时，应该是什么值。

> 既然最后的值必须是一个list，#t或#f都不对。让我们试试`(quote ())`

我们需要依次cons三次到一个()中。我们可以:

```
I.这样:
1. 把e cons 到()
2. 把c cons 到1的输出
3. 把a cosn 到2的输出
```

```
II.或者这样：
1. 把a cons 到2的输出
2. 把c cons 到3的输出
3. 把e cons 到()

III.或者这样：
把 a cons 到
  把 c cons 到
    把e cons 到
      ()
```

> 不论如何`(firsts l)`结果都是`(a c e)`

以上三种哪种是最舒服的呢

> 对啊，就选那个

`(insertR new old lat)`
是什么，其中
new是`topping`
old是`fudge`
lat是`(ice cream with fudge for dessert)`

> `(ice ceame with fudge topping for dessert)`

`(insertR new old lat)`
是什么，其中
new是`jalapeno`
old是`and`
lat是`(tacos tamales and salsa)`

> `(tacos tamales and jalapeno salsa)`

`(insertR new old lat)`
是什么，其中
new是`e`
old是`d`
lat是`(a b c d f g d h)`

`(a b c d e f g d h)`

用你的话描述一下`(insertR new old lat)`是做什么的

> 这里是我们的叙述：
> “它有三个参数：atom原子new和old，还有一个列表lat。函数insertR把new插入到第一个old后边创建一个lat。”

试试看你能不能写出insertR的开始三行

```
(define insertR
  (lambda (new old lat)
    (cond ...)))
```

当递归insertR时，哪个参数变化了

> lat，因为我们次只能查询一个atom原子。

我们要查询多少次lat

> 两次。lat要么是null空表，要么是非空表。

我们首先要问什么?

> 首先，我们问`(null? lat)`。第二else，因为else是做后一个查询。

如果`(null? lat)`假呢

> 那么至少lat有一个成员。

我们对第一个成员第一个查询问什么

> 首先，`(eq? (car lat) old)`。然后是else，没有其他情况了。

现在试试看你能不能写出整个insertR函数

> 这是我们的第一次尝试

```
(define insertR
(lambda (new old lat)
(cond
 ((null? lat) (quote()))
 (else (cond
         ((eq? (car lat) old) (cdr lat))
         (else (cons (car lat)
                 (insertR new old
                   (cdr lat)))))))))
```

当 new 是 `topping`
old 是 `fudge`
lat 是 `(ice cream with fudge for dessert)`

`(insertR new old lat)`的值是什么

> `(ice cream with for dessert)`

可以看到new的地方被删除了，仅此而已。这和rember一样了。当`(eq? (car lat) old)`是真时该做什么

> 在old的右侧插入new

怎么做

> 把new cons到`(cdr lat)`上

现在的insertR的函数定义为

```
(define insertR
  (lambda (new old lat)
    (cond
      ((null? lat) (quote()))
      (else (cond
              ((eq? old (car lat))
               (cons new (cdr lat)))
              (else (cons (car lat)
                      (insertR new old (cdr lat))))))))))
```

> Yes

现在,`(insertR new old lat)`的结果是什么?
当 new 是 `topping`
old 是 `fudge`
lat 是 `(ice cream with fudge for dessert)`

> `(ice cream with topping for dessert)`

这是我们想要的吗

> 不是，这是把new替换了old

还要做什么

> 需要把old cons到刚刚cons到new前边。

怎么把old加到new前边

> 把new cons到`(cdr lat)`上，再把old cons上即
> `(cons old (cons new (cdr lat)`

现在写出insertR函数

```
(define insertR
(lambda (new old lat)
(cond
 ((null? lat) (quote()))
 (else (cond
         ((eq? old (car lat))
          (cons old (cons new (cdr lat))))
         (else (cons (car lat)
                 (insertR new old (cdr lat))))))))))
```

```
(insertR 'topping 'fudge '(ice cream with fudge for dessert))
=>(ice cream with fudge topping for dessert)
```

OK了，这次

现在试一试insertL函数
提示：insertL把new原子插入到lat中的第一个old的左边。

这个容易多了，对吗

```
(define insertL
(lambda (new old lat)
(cond
((null? lat) (quote ()))
(else (cond
       ((eq? (car lat) old)
        (cons new
              (cons old (cdr lat))))
       (else (cons (car lat)
                   (insertL new old
                            (cdr lat)))))))))
```

你还能想到别的方法吗?

比如:

```
((eq? (car lat) old)
(cons new (cons old (cdr lat))))
```

可以改成

```
((eq? (car lat) old)
(cons new lat))
```

当 old 等于 (car lat) 的时候， (cons old (cdr lat)) 等同于 lat.

现在试试看你能不能写出subst函数(把old替换成new)
提示：(subst new old lat)用new替换lat中的第一个old

当然就是

```
(define subst
(lambda (new old lat)
(cond
 ((null? lat) (quote()))
 (else (cond
        ((eq? old (car lat))
         (cons new (cdr lat)))
        (else (cons (car lat)
                    (subst new old (cdr lat)))))))))
```

这明明就是刚刚我们没写对的insertR嘛。

来，cons一块蛋糕到嘴里吧。

现在试试看你能不能写出subst2函数
提示：(subst2 new o1 o2 lat)
把lat中第一个出现的o1或者第一个出现的o2替换为new
例如，
new是`vanilla`
o1是`chocolate`
o2是`banana`
lat是`(banana ice cream with chocolate topping)`

值是`(vanilla ice cream with chocolate topping)`

> 可以这样

```
(define subst2
(lambda (new o1 o2 lat)
(cond
((null? lat) (quote()))
(else (cond
       ((eq? (car lat) o1)
        (cons new (cdr lat)))
       ((eq? (car lat) o2)
        (cons new (cdr lat)))
       (else (cons (cdr lat)
                   (subst2 new o1 o2
                           (cdr lat)))))))))
```

也可以这样，把o1和o2直接用or表示

```
(define subst2
  (lambda (new o1 o2 lat)
    (cond
      ((null? lat) (quote()))
      (else (cond
              ((or (eq? o1 (car lat))
                   (eq? o2 (car lat)))
               (cons new (cdr lat)))
              (else (cons (car lat)
                          (subst new old (cdr lat)))))))))
```

如果你直接写出最后一种，再cons一块蛋糕到嘴里吧。

还记得rember是做什么的吗

> rember查找lat中的每一个atom看是否与a相同。如果不相同，rember留下这个atom。当第一次遇到a时，它停止并给出(cdr lat),即lat的剩余部分。所以最终仅仅删除了lat中的a。

写出函数 multirember ，它把lat中所有的a都删除

```
(define multirember
  (lambda (a lat)
    (cond
      (_____ _____)
      (else
        (cond
          (_____ _____)
          (_____ _____))))))
```

提示：当(eq? (car lat) a)是真时该做什么
考虑一下例子， a 是cup， lat 是(cup tea cup and hick cup)

```
(define multirember
(lambda (a lat)
(cond
((null? lat) (quote ()))
(else
 (cond
  ((eq? (car lat) a)
   (multirember a (cdr lat)))
  (else (cons (car lat)
              (multirember a (cdr lat))))))))

(multirember 'cup '(coffee cup tea cup and hick cup))

=> (coffee tea and hick)
```

你知道multirember如何工作的吗

> 不知道没关系，下边一步步的看。

`(null? lat)`

> 否，移动到下一行

---

else

> Yes

`(eq? (car lat) a)`

> 否，移动到下一行

下面的代码是什么意思
`(cons (car lat) (multirember a (cdr lat)))`

> 保存`(car lat)`即`coffee`，cons到一会计算的`(multirember a (cdr lat))`上。现在看`(multirember a (cdr lat))`

`(null? lat)`

> 否，移动到下一行

---

else

> 当然

`(eq? (car lat) a)`

> 是，所以忘记`(car lat)`，直接看`(multirember a (cdr lat))`

`(null? lat)`

> 否，移动到下一行

---

else

> Yes!

`(eq? (car lat) a)`

> 否，移动到下一行

下面的代码是什么意思
`(cons (car lat) (multirember a (cdr lat)))`

> 保存`(car lat)`即`tea`，cons到一会计算的`(multirember a (cdr lat))`上。现在看`(multirember a (cdr lat))`

`(null? lat)`

> 否，下个查询

---

else

> OK，下一个问题

`(eq? (car lat) a)`

> 是，所以忘记`(car lat)`，直接看`(multirember a (cdr lat))`

`(null? lat)`

> 否，移动到下一行

`(eq? (car lat) a)`

> 否，移动到下一行

下面的代码是什么意思

```
(cons (car lat) (multirember a (cdr lat)))
```

> 保存`(car lat)`即`and`，cons到一会计算的`(multirember a (cdr lat))`上。现在看`(multirember a (cdr lat))`

`(null? lat)`

> 否，下个查询

`(eq? (car lat) a)`

> 否，下个查询

下面的代码是什么意思

```
(cons (car lat) (multirember a (cdr lat)))
```

> 保存`(car lat)`即`hick`，cons到一会计算的`(multirember a (cdr lat))`上。现在看`(multirember a (cdr lat))`

`(null? lat)`

> 否，下个查询

`(eq? (car lat) a)`

> 是，所以忘记`(car lat)`，直接看`(multirember a (cdr lat))`

`(null? lat)`

> 是，所以值是`()`

结束了吗

> 没有，还有几个cons没执行，需要回溯递归。

下面呢

> 把最近一次的`(car lat)`即`hick` cons 到`()`上。

下面呢

> 把`and` cons到`(hick)`

下面呢

> 把`tea` cons 到 `(and hick)`

下面呢

> 把`coffee` cons到`(tea and hick)`

结束了吗

> yes

现在写一下函数 multiinsertR

```
(define multiinsertR
  (lambda (new old lat)
    (cond
      (_____ _____)
      (else
        (cond
          (_____ _____)
          (_____ _____))))))
```

```
(define multiinsertR
(lambda (new old lat)
(cond
((null? lat) (quote ()))
(else
 (cond
  ((eq? (car lat) old)
   (cons (car lat)
         (cons new
               (multiinsertR new old
                             (cdr lat)))))
  (else (cons (car lat)
              (multiinsertR new old
                            (cdr lat)))))))))
```

下面这个函数正确吗

```
(define multiinsertL
  (lambda (new old lat)
    (cond
     ((null? lat) (quote ()))
     (else
      (cond
       ((eq? (car lat) old)
        (cons new
              (cons old
                    (multiinsertL new old lat))))
       (else (cons (car lat)
                   (multiinsertL new old (cdr lat)))))))))
```

这个程序能结束吗

> No，因为遇到第一个old后一直不停的往它左边 `(cons new (cons old ...` 直到内存耗尽。

现在试试更改的 multiinsertL 函数

```
(define multiinsertL
(lambda (new old lat)
(cond
((null? lat) (quote ()))
(else
 (cond
  ((eq? (car lat) old)
   (cons new
         (cons old
               (multiinsertL new old
                             (cdr lat)))))
  (else (cons (car lat)
              (multiinsertL new old
                            (cdr lat)))))))))
```

第四戒 (初版)

递归时至少要有一个参数变化，并且向终止条件方向变化。变化的参数必须有终止测试条件：当时用cdr时，用null?测试终止。

现在我们写个 multisubst 函数

```
(define multisubst
  (lambda (new old lat)
    (cond
      (_____ ______)
      (else
        (cond
          (_____ _____)
          (_____ _____))))))
```

```
(define multisubst
(lambda (new old lat)
(cond
((null? lat) (quote ()))
(else (cond
       ((eq? (car lat) old)
        (cons new
              (multisubst new old
                          (cdr lat))))
       (else (cons (car lat)
                   (multisubst new old
                               (cdr lat)))))))))
```

`14` 是原子吗

> 是的，数都是原子

`(atom? n)` 是真还是假，其中n是 `14`

> 真，14 是原子

`-3` 是数吗

> 是的，不过我们暂不考虑负数

`3.14159` 是数吗

> 是的，不过我们仅仅考虑整数

`(add1 n)` 是多少，其中n是 `67`

> 68
>
> 注：
> Lisp中：1+
> Scmeme：

```
(define add1
(lambda (n)
(+ n 1)))
```

`(add1 67)` 是多少

> 68。同上

`(sub1 n)` 是多少，其中n是5

> 4
> 注：
> Lisp中：1-
> Scmeme中：

```
(define sub1
(lambda (n)
(- n 1)))
```

`(sub1 0)` 是多少

> 没有答案
> 注：
> 我们仅仅考虑非负数。实际是-1。

`(zero? 0)` 是真还是假

> 真
> 注：
> Lisp中：zerop

`(zero? 1492)` 是真是假

> 假

`(+ 46 12)` 是多少

> 58

试试写下函数+
提示：使用zero? add1 和sub1

```
(define +
(lambda (n m)
(cond
 ((zero? m) n)
 (else (add1 (+ n (sub1 m)))))))
```

> 很简单不是吗
> 注：
> Lisp，Scheme中: 这个有点像+。应写成o+(参看preface)

但是我们不是没有遵循第一戒吗

> 是的，但是我们可以把zero?看作是null?一样，因为zero?可以查询一个数是否是空 就如同null?查询一个list是否是空。

如果zero?像null?,那么add1像cons

> 对！cons构建list表，add1构建数

`(- 14 3)` 是多少

> 11

`(- 17 9)` 是多少

> 8

`(-18 25)` 是多少

> 没有答案。未考虑负数。

试试写下函数-
提示：使用sub1

> 下面这个如何
>
> ```
> (define -
> (lambda (n m)
> (cond
>  ((zero? m) n)
>  (else (sub1 (- n (sub1 m)))))))
> ```
>
> 注：
> Lisp，Scheme中:这个有点像-。应写成o-(参看preface)

你能描述一下`(- a b)`是如何工作的吗

> 输入两个参数，把第二个参数减一直到0，同时每次把第一个参数减一。

`(2 11 3 79 47 6)`这是一个tup吗

> 是的，tuple是tuple的简写。

`(8 55 5 555)`这是一个tup吗

> 是的，是一个数字组成的list

`(1 3 8 apple 4 3)`这是一个tup吗

> 不是，这只是一个原子组成的list表。

`(3 (7 4) 13 9)`这是一个tup吗

> 不是，(7 4)不是一个数

`()`这是一个tup吗

> 是的，这是一个特殊的空tup

---

tup 是`(3 5 2 8)`那么`(addtup tup)`是什么

> 18

---

tup 是`(15 6 7 12 3)`
`(addtup tup)`是什么

> 43

---

addtup做什么了

> 返回一个tuple参数的所有数成员的总和。

怎样从一个list构建数

> 用`+`替代`cons`：+构建数就像cons构建list表

当用cons构建list表，终止条件是`()`，那么+的终止条件呢

> 0

---

list 的自然终止条件

> `(null? l)`

---

tup 的自然终止条件

> `(null? tup)`

当从一个list表的数创建一个数，终止条件是啥样

> `((null? tup) 0)`
> 如同`((null? l) (quote ()))`是从list创建list的终止条件

---

addtup的终止条件

> `((null? tup) 0)`

---

lat 是怎么定义的

> 空表，或者成员都是原子的list表。

---

tup 是怎么定义的

> 空表，或者成员都是数的list表。

怎样对一个 list 进行递归

> (cdr lat)

怎么对一个 tup 进行递归

> (cdr tup)

为什么

> 因为剩余的非空表是表，剩余的非空tup是一个tup

---

list 需要几个查询

> 两个

---

tup 需要几个查询

> 两个。空tup的情况，以及非空tup的情况。

数是怎么定义的

> 0，或者从一个数增加一递归得到。

数的 自然递归终止条件是什么

> `(zero? n)`

怎样对数字进行递归

> `(sub1 n)`

数需要多少个查询

> 两个。

第五戒 (第一次修订)

当递归atom原子，lat时，两个查询：(null? lat) 和else

当递归数n时,两个查询：(zero? n) 和else

---

cons 是做什么的

> 构建list表

---

addtup 是做什么的

> 从tup的所有数成员构建一个总和数。

---

addtup 的终止条件

> `((null? tup) 0)`

---

addtup 的自然递归是什么

> `(addtup (cdr tup))`

---

addtup 用什么构建数

> 用`+`，因为+也构建数字

填写下面的定义

```
(define addtup
  (lambda (tup)
    (cond
      ((null? tup) 0)
      (else ...))))
```

> 下面这是我们填写的
> `(+ (car tup) (addtup (cdr tup)))`
> 注意这行与rember函数的这行非常相近：
> `(cons (car lat) (rember a (cdr lat)))`

`(x 5 3)` 是多少

> 15

`(x 13 4)` 是多少

> 64

`(x n m)` 做什么

> 做乘法，求n和m的乘积

---

x 的终止条件是什么

> `((zero? m) 0)`，因为 n x 0 = 0

既然 `(zero? m)` 是终止条件，m必须最终减到一，那么怎样做到

> 用sub1

第四戒 (第一次修订)

递归时至少要有一个参数变化，并且向终止条件方向变化。变化的参数必须有终止测试条件：当时用cdr时，用null?当使用sub1，用zero?

`(x n (sub1 m))` 是啥

> x的自然递归。

试试写下函数x

```
(define x
(lambda n m
(cond
 ((zero? m) 0)
 (else (+ n (x n (sub1 m)))))))
```

注：
Lisp，Scheme中：有点像*

`(x 12 3)` 是多少

> 36。让我们一步步看函数是怎么得到值的。

`(zero? m)`

> 否。

`(+ m (x m (sub1 n)))` 是什么意思

> 把n(n = 12)加到自然递归上。如果x正确的话，那么`(x 12 (sub1 3))`应该是24

`(x n m)` 的新参数是什么

> n是12，m是2

`(zero? m)`

> 否。

`(+ m (x m (sub1 n)))` 是什么意思

> 把n(n=12)加到`(x n (sub1 m))`

`(x n m)` 的新参数是什么

> n是12，m是1

`(zero? m)`

> 否。

`(+ m (x m (sub1 n)))` 是什么意思

> 把n(n=12)加到(x n (sub1 m))

下面这行是多少
`((zero? m) 0)`

> 0，因为(zero? 0)现在为true

完成了吗

> 没呢

为什么

> 还有3个`+`没计算呢

原始应用的值是多少

> 12加到12加到12加到0上，得到36。n加了m次。

用等式表示就是：

```
(x 12 3) = 12 + (x 12 2)
         = 12 + 12 +( x 12 1)
         = 12 + 12 + 12 (x 12 0)
         = 12 + 12 + 12 + 0
         = 36
```

为什么0是x的终止条件

> 因为0不影响+,n + 0 = n

第五戒

当用+构建一个值时，使用0作为终止值，因为0不改变加法的值。

当用x构建一个值时，使用1作为终止值，因为1不改变乘法的值。

当用cons构建list表，终止条件是()

---

tup1是`(3 6 9 11 4)`，tup2是`(8 5 2 0 7)`，问`(tup+ tup1 tup2)`是什么

> `(11 11 11 11 11)`

---

tup1是`(2 3)`，tup2是`(4 6)`，问`(tup+ tup1 tup2)`是什么

> (6 9)

`(tup+ tup1 tup2)`做什么

> 把所有tup1和tup2的元素一一对应加，tup的长度相同。

---

tup+ 有什么特殊之处

> 每次查找两个tup的元素，即每次递归都是处理两个元素。

递归tup需要多少个查询。

> 两个。`(null? tup)`和`else`

当对两个tup递归时，需要查询多少个问题。

> 四个。第一个tup是空或者非空。第一个tup是空或者非空。

就是这几个吗

```
(and (null? tup1) (null? tup2))
(null? tup1)
(null? tup2)
else
```

> 没错

当第一个tup是()时，第二个tup不是()可以吗

> 否，因为两个tup必须一样长。

也就是仅仅需要
`(and (null? tup1) (null? tup2))`
`else`
对吗

> 是的，因为(null? tup1)真时，(null? tup2)也是真。

写出函数tup+

> ```
> ;;tup1 和 tup2 必须是相同长度
> (define tup+
> (lambda (tup1 tup2)
> (cond
> ((and (null? tup1) (null? tup2))
>  (quote ()))
> (else
>  (cons (+ (car tup1) (car tup2))
>        (tup+
>         (cdr tup1) (cdr tup2)))))))
> ```
>
> ```
> (tup+ '(1 2 3)'(4 5 6))
> =>(5 7 9)
> ```

比较简单就不详细解释了。

这是另一个版本的tup+函数，与上面的版本不同之处在于可以接受两个长度不同的tup作为参数。（比较简单，我也不解释了-by GeekPig）

```
(define tup+
  (lambda (tup1 tup2)
    (cond
     ((null? tup1) tup2)
     ((null? tup2) tup1)
     (else
      (cons (+ (car tup1) (car tup2))
            (tup+
             (cdr tup1) (cdr tup2)))))))
```

`(> 12 133)`

> #f假

`(> 120 11)`

> #t真

需要多少个数来迭代。

> 两个，n 和 m

怎么递归

> `(sub1 n)`和`(sub1 m)`

什么时候递归

> 当有一个数都不是零时。

---

n 和 m 需要多少个查询

> 三个`(zero? n)`, `(zero? m)`,以及`else`

现在你能用zero?和sub1写出函数>吗

> 下面这个对吗

```
(define >
(lambda (n m)
(cond
 ((zero? m) #t)
 ((zero? n) #f)
 (else (> (sub1 n) (sub1 m))))))
```

不对，没有考虑到m与n相等时的情形，m与n最后同时递归到0。结果取了第一种情况(zero? m) #t了。可以把两种情况交换下顺序。

```
(define >
(lambda (n m)
(cond
 ((zero? n) #f)
 ((zero? m) #t)
 (else (> (sub1 n) (sub1 m)))))
```

下面尝试写出<函数

```
(define <
(lambda (n m)
(cond
 ((zero? m) #f)
 ((zero? n) #t)
 (else (< (sub1 n) (sub1 m)))))
```

下面是=的定义

```
(define =
  (lambda (n m)
    (cond
      ((zero? m) (zero? n))
      ((zero? n) #f)
      (else (= (sub1 n) (sub1 m))))))
```

用函数<和函数>重写函数=

```
(define =
(lambda (n m)
(cond
 ((> n m) #f)
 ((< n m) #f)
 (else #t))))
```

就是说我们有两个函数来比较atom原子是否有相等对么

> 对。用 = 查询数，用 eq? 查询其它的。

`(^ 1 1)`

> 1

`(^ 2 3)`

> 8

`(^ 5 3)`

> 125

现在，写出函数^
提示：记得第一戒和第五戒

```
(define ^
(lambda (n m)
(cond
 ((zero? m) 1)
 (else (x n (^ n (sub1 m)))))))
```

> 注：在Lisp和scheme中，类似于expt函数

这个函数的名字怎样才好

```
(define ???
  (lambda (n m)
    (cond
      ((< n m) 0)
        (else (add1 (??? (- n m) m))))))
```

> 我们从来没有见过这样的定义；自然递归看起来很奇怪

第一个查询是什么

> 看第一个参数是否小于第二个参数

第二行呢

> 我们递归函数，第一个参数变为原来两个参数的差。函数返回时加一。

那么函数做了什么

> 数出第一个参数能减掉多少个第二个参数。

这叫什么

> 除法

```
(define ÷
(lambda (n m)
(cond
 ((< n m) 0)
   (else (add1 (÷ (- n m) m))))))
```

来一份(ham and cheese on rye)怎么样，别忘了芥末哦！

`(length lat)` 的值是什么，其中lat是 `(hotdogs with mustard sauerkraut and pickles)`

> 6

`(length lat)` 的值是什么，其中lat是 `(ham and cheese on rye)`

> 5

现在写一个函数length

```
(define length
(lambda (lat)
(cond
 ((null? lat) 0)
 (else (add1 (length (cdr lat)))))))
```

`(pick n lat)` 是什么
其中 n 是 `4`
lat 是 `(lasagna spaghetti ravioli macaroni meatball)`

> `macaroni`

`(pick 0 lat)` 是什么，其中lat是 `(a)`

> 没有答案。

写出函数pick

```
(define pick
(lambda (n lat)
(cond
 ((zero? (sub1 n)) (car lat))
 (else (pick (sub1 n) (cdr lat))))))
```

`(rempick n lat)` 是什么，其中
n 是 3
lat 是 `(hotdogs with hot mustard)`

> `(hotdogs with mustard)`

现在写一个函数rempick

```
(define rempick
(lambda (n lat)
(cond
 ((zero? (sub1 n)) (cdr lat))
 (else (cons (car lat)
             (rempick (sub1 n) (cdr lat)))))))
```

`(number? a)` 是真还是假
当
a 是 `tomato`

> 假

`(number? 76)` 是真还是假

> 真

你能写出number?函数吗

> 不能。number? 如同add1, sub1, zero?, car, cdr, cons, null?, eq?, 以及atom?, 都是元函数

现在使用number?函数写一个no-nums，它删除一个lat中的所有出现的数。例如，lat是(5 pears 6 prunes 9 dates)，那么(no-nums lat)的值是(pears prunes dates)

```
(defin no-nums
(lambda (lat)
(cond
((null? lat) (quote ()))
(else (cond
       ((number? (car lat))
        (no-nums (cdr lat)))
       (else (cons (car lat)
                   (no-nums (cdr lat)))))))))
```

现在写一个函数 all-nums，它从一个lat中抽取出所有的数构成一个tup。

```
(define all-nums
(lambda (lat)
(cond
((null? lat) (quote ()))
(else
 (cond
  ((number? (car lat))
   (cons (car lat)
         (all-nums (cdr lat))))
  (else
   (all-nums (cdr lat)))))))))
```

写一个函数eqan?，当两个参数a1和a2是一样的原子时为真。

```
(define eqan?
(lambda (a1 a2)
(cond
 ((and (number? a1) (number? a2)) (= a1 a2))
 ((or (number? a1) (number? a2)) #f)
 (else (eq? a1 a2)))))
```

那么可以把所有用eq?的地方都可以替换为eqan?吗

> 可以当然了。

现在写一个函数occur描述一个lat中出现原子a的次数

```
(define occur
(lambda (a lat)
(cond
((null? lat) 0)
(else
 (cond
  ((eq? (car lat) a)
   (add1 (occur a (cdr lat))))
  (else
   (occur a (cdr lat))))))))
```

写一个函数one?仅当n为1时(one? n)为#t真，否则为#f假

```
(define one?
(lambda (n)
(cond
 ((zero? n) #f)
 (else (zero? (sub1 n))))))
```

或者

```
(define one?
(lambda (n)
(cond
 (else (= 1 n)))))
```

又或者直接

```
(define one?
(lambda (n)
(= 1 n)))
```

现在重写rempick，它删去 lat 中的第 n 个atom原子。例如 n为`3` lat是`(lemon meringue salty pie)` 那么`(rempick n lat)`的值是 `(lemon meringue pie)` 你可以使用你自己的one?函数。

```
(define rempick
(lambda (n lat)
(cond
 ((one? n) (cdr lat))
 (else (cons (car lat)
             (rempick (sub1 n) (cdr lat)))))))
```

`(rember* a l)` 是什么
其中 a 是 `cup`
l 是 `((coffee) cup ((tea) cup) (and (hick)) cup)`
rember*发音为rember-star

> `((coffee ((tea)) (and (hick)))`

`(rember* a l)` 是什么
其中
a 是 `suace`
l 是 `(((tomato sauce)) ((bean) sauce) (and ((flying)) sauce))`

> `(((tomato)) ((bean)) (and ((flying))))`

现在写出函数rember*，下面是框架

```
(define rember*
  (lambda (a l)
    (cond
      (_____ _____)
      (_____ _____)
      (_____ _____))))
```

```
(define rember*
(lambda (a l)
(cond
((null? l) (quote ()))
((atom? (car l))
 (cond
  ((eq? (car l) a)
   (rember* a (cdr l)))
  (else (cons (car l)
              (rember* a (cdr l))))))
(else (cons (rember* a (car l))
            (rember* a (cdr l)))))))
```

`(lat? l)` 是什么值，其中
l 是

```
(((tomato sauce))
 ((bean) sauce)
  (and ((flying)) sauce))
```

> `#f`

`(car l)` 是 atom 吗
当 l 是

```
(((tomato sauce))
 ((bean) sauce)
  (and ((flying)) sauce))
```

> 不是

`(insertR* new old l)` 是什么值
其中
new 是 `roast`
old 是 `chuck`
l 是

```
   ((how much (wood))
    could
   ((a (wood) chuck))
   (((chuck)))
   (if (a) ((wood chuck)))
    could chuck wood)
```

```
((how much (wood))
could
((a (wood) chuck roast))
(((chuck roast)))
(if (a) ((wood roast)))
could chuck roast wood)
```

现在写出函数 `insertR*`，下面是框架

```
(define insertR*
  (lambda (new old l)
    (cond
      (_____ _____)
      (_____ _____)
      (_____ _____))))
```

```
(define insertR*
(lambda (new old l)
(cond
((null? l) (quote ()))
((atom? (car l))
 (cond
  ((eq? (car l) old)
   (cons old
         (cons new
               (insertR* new old
                         (cdr l)))))
  (else (cons (car l)
              (insertR* new old
                        (cdr l))))))
(else (cons (insertR* new old
                      (car l))
            (insertR* new old
```

```
18                    (cdr l)))))))
```

`insertR*` 和 `rember*` 有什么相似之处

> 当 car 是一个 list 时，他们都有 car 的递归。

第一戒

最终版

当在一个由原子构成的列表(lat)上递归时，问两个问题：(null? lat) 还有 else.

当在一个数字上递归时，问两个问题：(zero? n) 还有 else.

当在一个由S-表达式构成的列表（复合列表）上递归时，问三个问题：(null? l), (atom? (car l)),还有 else.

How are `insertR*` and `rember*` similar?

> Each function recurs on the car of its argument when it finds out that the argument's car is a list.

How are `rember*` and `multirember` different?

> The function multirember does not recur with the car. The function rember* recurs with the car as well as with the cdr. It recurs with the car when it finds out that the car is a list.

How are `insertR*` and `rember*` similar?

> They both recur with the car, whenever the car is a list, as well as with the cdr.

所有的*-函数都有什么相似之处

> 都有三个分支查询，当car是一个list时，car和cdr一样也要递归。

为什么

> 因为所有的*-函数的list参数都是要么空,要么里边有原子被cons到list中，要么有list被cons到list中。

第四戒

(最终版)

递归时至少要有一个参数变化，并且向终止条件方向变化。变化的参数必须有终止测试条件：当递归原子(lat)时使用(cdr lat)。当递归数n时使用(sub1 n)。当递归一个 S-expression的列表l,当(null? l)或者(atom? (car l))使用(car l)和(cdr l)。

递归时参数要向终止条件方向变化：当使用cdr时，用null?测试终止；

当使用sub1时，用zero?测试终止。

`(occursomething a l)`
其中 a 是 `banana`
l 是

```
((banana)
 (split ((((banana ice)))
         (cream (banana))
         sherbet))
 (banana)
 (bread)
 (banana brandy))
```

> 5

那么 occursomething 的好点的名字是什么

> occur*

写一个函数`occur*`

```
(define insertR*
  (lambda (a l)
    (cond
      (_____ _____)
      (_____ _____)
      (_____ _____))))
```

> ```
> (define occur*
> (lambda (a l)
> (cond
>  ((null? l) 0)
>  ((atom? (car l))
>   (cond
>     ((eq? a (car l))
>      (add1 (occur* a (cdr l))))
>     (else (occur* a (cdr l)))))
>  (else (+ (occur* a (car l))
>           (occur* a (cdr l))))))
> ```

`(subst* new old l)`
其中 new 是`orange`
old 是`banana`
l 是

```
((banana)
 (split ((((banana ice)))
        (cream (banana))
        sherbet))
(banana)
(bread)
(banana brandy))
```

```
((orange)
(split ((((orange ice)))
    (cream (orange))
    sherbet))
(orange)
(bread)
(orange brandy))
```

写一个函数`subst*`

```
(define subst*
  (lambda (new old l)
    (cond
      (_____ _____)
      (_____ _____)
      (_____ _____))))
```

```
(define subst*
(lambda (new old l)
(cond
 ((null? l) (quote ()))
 ((atom? (car l))
  (cond
    ((eq? (car l) old)
     (cons new
           (subst* new old (cdr l))))
    (else (cons (car l)
                (subst* new old (cdr l))))))
 (else
     (cons (subst* new old (car l))
           (subst* new old (cdr l)))))))
```

`(insertL* new old l)`
是什么
其中 new 是`pecker`
old 是`chuck`
l 是

```
((how much (wood))
 could
 ((a (wood) chuck))
 (((chuck)))
 (if (a) ((wood chuck)))
 could chuck wood)
```

> ```
> ((how much (wood))
> could
> ((a (wood) pecker chuck))
> (((pecker chuck)))
> (if (a) ((wood pecker chuck)))
> could pecker chuck wood)
> ```

写一个函数 `insertL*`

```
(define insertL*
  (lambda (new old l)
    (cond
      (_____ _____)
      (_____ _____)
      (_____ _____))))
```

> ```
> (define insertL*
> (lambda (new old l)
> (cond
> ((null? l) (quote ()))
> ((atom? (car l))
>  (cond
>   ((eq? (car l) old)
>    (cons new
>          (cons old
>                (insertL* new old (cdr l)))))
>   (else (cons (car l)
>               (insertL* new old (cdr l))))))
> (else (cons (insertL* new old (car l))
>             (insertL* new old (cdr l))))))
> ```

`(nember* a l)`
a 是 `chips`
l 是 `((potato) (chips (( with) fish) (chips)))`

> #t，因为原子chips在列表l中。

写出函数 `member*`

```
(define member*
  (lambda (a l)
    (cond
      (_____ _____)
      (_____ _____)
      (_____ _____))))
```

```
(define member*
(lambda (a l)
(cond
((null? l) #f)
((atom? (car l))
 (or (eq? (car l) a)
     (member* a (cdr l))))
(else (or (member* a (car l))
          (member* a (cdr l))))))
```

`(member* 'chips '((potato) (chips (( with) fish) (chips))))`
=>#t

`(leftmost l)`
是什么，其中
l 是`((potatoe) (chips ((with) fish) (chips)))`

> potato

`(leftmost l)`
是什么，其中
l 是`(((hot) (tuna (and))) cheese)`

> hot

`(leftmost l)`是什么，其中 l 是`((((() four)) 17 (seventeen))`

> 没有答案

`(leftmost (quote()))`是什么

> 没有答案

你能描述 leftmost 是什么

> 这是我们的描述
> “函数 leftmost 查找 S-expression 表达式中非空 list 中的的最左边的一个原子。

---

leftmost 是一个*-函数吗

> 它的参数为 S-expression构成的表达式，但是仅在car上递归。

---

leftmost 需要所有的三种分支查询吗

> 不，仅需要两个。我们已经确认 leftmost的参数是非空表并且没有空表成员。

现在看看你能不能写出 leftmost函数

```
(define leftmost
  (lambda (l)
    (cond
      (_____ _____)
      (_____ _____))))
```

```
(define leftmost
(lambda (l)
(cond
 ((atom? (car l)) (car l))
 (else (leftmost (car l))))))
```

还记得(or ...)是做什么的吗

> (or ...) 一次查询一个直到出现真然后停下来，返回真。如果找不到真，那么(or ...)的值是假。

假设 x 是`pizza`
l 是`(mozzarella pizza)`
那么`(and (atom? (car l)) (eq? (car l) x))`的值是多少

> #f

为什么是false假

> 因为(and ...)查询(atom? (car l))，是真，但是(eq? (car l) x)是假。于是为假。

假设 x 是`pizza`，
l 是`((mozzarella) pizza)`
那么`(and (atom? (car l)) (eq? (car l) x))`的值是什么

> #f

为什么

> 因为(and ...)查询(atom? (car l))，但是(car l)不是一个atom。于是为#f

举个 x 和 l 的例子让`(and (atom? (car l)) (eq? (car l) x))`为真

> 下面是一个例子: x 是 pizza，l 是 (pizza (tastes good))

请用自己的话描述一下(and ...)的作用

> 下面是我们的描述
> "(and ...)一次查询一个，直到出现否，然后返回假。如果总找不到假的表达式，(and ...)的值为真。

判断真假：(and ...)和(or ...)的有些参数没有查询

> 是的。(and ...)当找到#f时就停止了。(or ...)当找到t时就停止了。
> 注：(cond ...)也有不查询参数的时候。因为这个特性，(and ...) 和(or ...)也可以由(cond ...)定义：
> (and alpha beta)=(cond (alpha beta) (else #f))
> (or alpha beta)=(cond (alpha #t) else beta)

当:
l1 是`(strawberry ice cream)`
l2 是`(strawberry ice cream)`
`(eqlist? l1 l2)`是什么

> #t

当:
l1 是`(strawberry ice cream)`
l2 是`(strawberry cream ice)`
`(eqlist? l1 l2)`是什么

> #f

当:
l1 是`(banana ((split)))`
l2 是`((banana) (split))`
`(eqlist? l1 l2)`是什么

> #f

当:
l1 是`(beef ((sausage)) (and (soda)))`
l2 是`(beef ((salami)) (and (soda)))`
`(eqlist? l1 l2)`是什么

> #f，差一点就是#t真

当:
l1 是`(beef ((sausage)) (and (soda)))`
l2 是`(beef ((sausage)) (and (soda)))`
`(eqlist? l1 l2)`是什么

> #t

---

eqlist? 是什么

> 查询两个list是否完全相同

`eqlist?`需要多少个查询

> 九个

为什么是九个查询

> 下面是我们的解释
> ”每个参数都可能
> ——是空表
> ——是原子cons出的list表
> ——是list表cons出的list表
> 例如，当第一个参数可能是空表，第二个参数可以有三种情况，那么3乘3共九种情况。

用eqan?写函数 eqlist?

```
(define eqlist?
(lambda (l1 l2)
(cond
((and (null? l1) (null? l2)) #t)
((and (null? l1) (atom? (car l2))) #f)
((null? l1) #f)
((and (atom? (car l1)) (null? l2)) #f)
((and (atom? (car l1))
      (atom? (car l2)))
 (and (eqan? (car l1) (car l2))
      (eqlist? (cdr l1) (cdr l2))))
((atom? (car l1)) #f)
((null? l2) #f)
((atom? (car l2)) #f)
(else
 (and (eqlist? (car l1) (car l2))
      (eqlist? (cdr l1) (cdr l2)))))))
```

第二个查询 `(atom? (car l2))` OK吗

> 是的，第二个list不能为空，否则就是第一个查询条件了。

第三个查询为什么是 `(null? l1)`

> 通过前两个查询知道，第一个参数是空表时，第二个参数既不是空表也不是第一个元素是原子的list表。如果(null? l1)是真，那么第二个参数必须是其中第一个元素是list表的一个list表。

判断真假：如果第一个参数是 `()`， `eqlist?` 得到 #t 对吗

> 对。因为 `(eqlist? (quote ()) l2)` 为真，l2必须为空表。

这就是说 `(and (null? l1) (null? l2))` 和 `(or (null? l1) (null? l2))` 满足前三种查询的情况

> 是的。如果第一个查询是真，那么eqlist?得到#t；否则为#f。

重写eqlist?

```
(define eqlist?
```

```
(lambda (l1 l2)
(cond
((and (null? l1) (null? l2)) #t)
((or (null? l1) (null? l2)) #f)
((and (atom? (car l1))
      (atom? (car l2)))
 (and (eqan? (car l1) (car l2))
      (eqlist? (cdr l1) (cdr l2))))
((or (atom? (car l1))
     (atom? (car l2)))
 #f)
(else
 (and (eqlist? (car l1) (car l2))
      (eqlist? (cdr l1) (cdr l2))))))
```

什么是 S-expression表达式

> 原子或者由list构成的 S-expression表达式

为得到两个 S-expression表达式是否相同，equal?需要多少个查询

四个。第一个参数可能是空表或者list表构成的 S-expression表达式，第二个参数可能是空表或者list表构成的 S-expression表达式。

写出函数equal?

```
(define equal?
(lambda (s1 s2)
(cond
((and (atom? s1) (atom? s2))
 (eqan? s1 s2))
((atom? s1) #f)
((atom? s2) #f)
(else (eqlist? s1 s2)))))
```

第二个问题为什么是`(atom? s1)`

> 如果为真，我们知道第一个参数是atom原子，第二个参数是list表。

第二个问题为什么是`(atom? s2)`

> 到第三个查询时，我们知道第一个参数不是atom原子，所以我们唯一需要区分的时第二个参数数是否是atom原子。第一个参数必然是一个list。

我们可以把第二个和第三个查询写成这样吗
`(or (atom? s1) (atom? s2))`

> 当然可以

简化equal?

```
(define equal?
(lambda (s1 s2)
(cond
((and (atom? s1) (atom? s2))
 (eqan? s1 s2))
((or (atom? s1) (atom? s2))
 #f)
(else (eqlist? s1 s2)))))
```

---

equal?询问了足够的问题了吗

> 是的，问题覆盖了所有的四种可能性

现在，使用equal?重写eqlist?

```
(define eqlist?
(lambda (l1 l2)
(cond
((and (null? l1) (null? l2)) #t)
((or (null? l1) (null? l2)) #f)
(else
 (and (equal? (car l1) (car l2))
      (eqlist? (cdr l1) (cdr l2)))))))
```

第六戒

仅当函数正确后再简化

下面是把表lat替换为 S-expression表达式, 把 a 替换为任何 S-expression表达式后得到的新的rember

```
(define rember
  (lambda (s l)
    (cond
     ((null? l) (quote ()))
     ((atom? (car l))
      (cond
       ((equal? (car l) s) (cdr l))
       (else (cons (car l)
                   (rember s (cdr l))))))
     (else (cond
            ((equal? (car l) s) (cdr l))
            (else (cons (car l)
                        (rember s (cdr l)))))))))
```

可以简化吗

> 当然

```
(define rember
(lambda (s l)
(cond
((null? l) (quote ()))
(else (cond
       ((equal? (car l) s) (cdr l))
       (else (cons (car l)
                   (rember s (cdr l)))))))))
```

笔记：这个 S-expression表达式的rember有问题。如果(car l)本身不等于 s 但是 l 的成员或者 l 的成员的成员等等含有s的话那岂不是跳过去了?
`(rember 'a '((a) b c))`
=>'((a) b c)

列表内的 a 并没有被删除

---

rember 是一个"star"函数吗

> 不是

为什么

> 因为 rember 仅仅递归 l 的 cdr

---

rember 还能被简化吗

> 能，内层的(cond ...)可以提到外边的conde中。

做做看

```
(define rember
(lambda (s l)
(cond
((null? l) (quote ()))
((equal? (car l) s) (cdr l))
(else (cons (car l)
            (rember s (cdr l)))))))
```

这个能正常工作么

> 当然，所有的分支和递归和化简之前一样。

简化 insertL*

> 不能。在查询`(eq? (car l) old)`我们必须知道`(car l)`是否是atom原子。

当函数设计的正确，我们就能够更好的思考它们。

> 而这比起错误的函数节约时间。

所有用 eq? 和 = 的地方和广义 eq? 都可以用函数 equal? 吗

> 不能。eqan? 的地方就不能，其它的都可以。实际上，不考虑 eqan? 例子的细节，就是我们的假设。

`1` 是算术表达式吗

> 是

`3` 是算术表达式吗

> 是的

`1 + 3` 是算术表达式吗

> 是的

`1 + 3 × 4` 是算术表达式吗

> 当然是

cookie 是算术表达式吗

> 是啊，你需要来一块吗

那么 `3 ^ y + 5`

> 是的

你来说说什么是算术表达式

> 我们这样描述
> “对于这一章，算术表达式可以是atom原子(包括数)，或者由+，×，或者^连接的两个算术表达式。”

`(quote a)` 是什么

> a

`(quote +)` 是什么

> 原子 +，而不是操作 +

`(quote ×)` 代表什么

> 代表原子 ×，而不是操作 ×

`(eq? (quote a) y)` 是真还是假，其中 y 是 a

> 真

`(eq? x y)` 是真还是假，其中 x 是 a，y 是 a

> 这同前边的那个问题一样。真。

`(n + 3)` 是算术表达式吗

> 不是啊，括号括着 n + 3。我们的算术定义没有提到括号。

我们可以认为 `(n + 3)` 是算术表达式吗吗

> 可以，只要我们假定括号不存在。
> (原文没看明白：Yes, if we keep in mind that the parentheses are not really there.)

那 `(n + 3)` 怎么称呼

> 称作(n + 3)的表示法(representation)

为什么 `(n + 3)` 是一个好的表示法

> 因为
>
> 1. (n + 3) 是一个 S-expression表达式，它可以作为函数参数
> 2. 它就像 n + 3

`(numbered? x)` 是真还是假，其中 x 是 1

> 真

`3 + 4 × 5` 的表示是什么

> (3 + (4 × 5))

`(numbered? y)` 是真还是假，其中 y 是 `(3 + (4 ^ 5))`

> 真

`(numbered? z)` 是真还是假，其中 z 是 `(2 × sausage)`

> 假，因为sausage不是一个数。

`numbered?` 是什么

> 一个函数，查询一个算术表达式是否只包含有在 + ， × ， 和 ^ ， 及其后边的数。

写个 numbered?函数的框架试试

```
(define numbered?
(lambda (aexp)
(cond
 (_____ _____)
 (_____ _____)
 (_____ _____)
 (_____ _____))))
```

第一个问题是什么

> `(atom? aexp)`

`(eq? (car (cdr aexp)) (quote +))` 是什么

> 这是第二个问题

你能猜出来第三个问题吗

> `(eq? (car (cdr aexp)) (quote ×))`

那第四个查询呢

> `(eq? (car (cdr aexp)) (quote ^))`

我们还需要查询aexp吗

> 不需要了，我们可以用else把前一个查询替换掉

为什么我们查询算术表达式是四个，而不是两个。毕竟像(1 + 3)这样的算术表达式是lats（即原子构成的list表)

> 因为我们把(1 + 3)表达看作是lsit表组成的算术表达式，而不是它本身的那样。而且算术表达式可以是数，或者有算术表达式和 + ， × ，或者 ^ 连接组成。

现在你能写出函数 numbered?吗

```
(define numbered?
(lambda (aexp)
(cond
 ((atom? aexp) (number? aexp))
 ((eq? (car (cdr aexp)) (quote +)) ...)
 ((eq? (car (cdr aexp)) (quote ×)) ...)
 ((eq? (car (cdr aexp)) (quote ^)) ...))))
```

为什么当aexp是atom原子时查询(number? aexp)

> 因为我们想知道算术表达式中的原子是否是数

为什么我们需要知道是否aexp中 + 连接的东西是否是两个算术表达式

> 我们需要查找出来两个子表达式是否是numbered

第一个 子表达式在哪儿

> 就是aexp的car

第二个 子表达式在哪儿

> 就是aexp的cdr的cdr的car(即aexp的第三个成员)

所以我们需要查询什么

> `(numbered? (car aexp))` 和 `(numbered? (car (cdr (cdr aexp))))` 必须都是真。

第二个回答是什么

> `(and (numbered? (car aexp)) (numbered? (car (cdr (cdr aexp)))))`

再试试写出函数 numbered?

```
(define numbered?
(lambda (aexp)
(cond
 ((atom? aexp) (number? aexp))
 ((eq? (car (cdr aexp)) (quote +))
  (and (numbered? (car aexp))
       (numbered? (car (cdr (cdr aexp))))))
 ((eq? (car (cdr aexp)) (quote x))
  (and (numbered? (car aexp))
       (numbered? (car (cdr (cdr aexp))))))
 ((eq? (car (cdr aexp)) (quote ^))
  (and (numbered? (car aexp))
       (numbered? (car (cdr (cdr aexp)))))))))
```

既然aexp已经被认为是算术表达式，我们可以把 numbered?写得更简单些

```
(define numbered?
(lambda (aexp)
(cond
 ((atom? aexp) (number? aexp))
 (else (and (numbered? (car aexp))
            (numbered? (car (cdr (cdr aexp)))))))))
```

为什么可以简化

> 因为我们知道函数是正确的

`(value u)` 的值是多少，其中 u 是 `13`

> 13

`(value x)` 的值是多少，其中 x 是 `(1 + 3)`

> 4

`(value y)` 的值是多少，其中 x 是 `(1 + (3 ^ 4))`

> 82

`(value z)` 的值是多少，其中 z 是 `cookie`

> 没有答案

(value nexp) 返回我们认为的那样的数学算术表达式的值

> 期待如此

value 对nexp要几个查询

> 4个

现在，让我们试着写出函数 value

```
(define value
(lambda (nexp)
(cond
 ((atom? nexp) ...)
 ((eq? (car (cdr nexp)) (quote +)) ...)
 ((eq? (car (cdr nexp)) (quote ×)) ...)
 (else ...))))
```

由 + 把两个算术表达式相连而构成的算术表达式的值是多少

> 如果我们知道两个子表达式的值，我们把它们求和就可以了。

`(1 + (3 × 4))` 中的那个子表达式的值是什么

> 当然是用value计算 1 ，再用value计算 (3 × 4) 就可以了

总的来说呢

> 在子表达式上递归value

第七戒

在相同本性的东西上递归子组成部分：

*list表
*算术表达式

再试试函数value

```
(define value
(lambda (nexp)
(cond
((atom? nexp) nexp)
((eq? (car (cdr nexp)) (quote +))
 (+ (value (car nexp))
    (value (car (cdr (cdr nexp))))))
((eq? (car (cdr nexp)) (quote x))
 (x (value (car nexp))
    (value (car (cdr (cdr nexp))))))
(else
 (^ (value (car nexp))
    (value (car (cdr (cdr nexp)))))))))
```

你能想出算术表达式的不同表示吗

> 有好几种呢

`(3 4 +)`可以表示 3 + 4吗

> 可以啊

`(+ 3 4)` 可以吗

> 可以吗

或者`(plus 3 4)`

> 可以

`(+ ( 3 6) (^ 8 2))`是一个算术表达式的表达吗

> 是的。

试着写写函数value来处理一种新的算术表达式，可以是：
——是数
——是 + 后边跟着两个算术表达式
——是 × 后边跟着两个算术表达式
——是 ^ 后边跟着两个算术表达式

> 这个怎么样

```
(define value
(lambda (nexp)
(cond
((atom? nexp) nexp)
((eq? (car nexp) (quote +))
 (+ (value (cdr nexp))
    (value (cdr (cdr nexp)))))
((eq? (car nexp) (quote x))
 (x (value (cdr nexp))
    (value (cdr (cdr nexp)))))
(else
 (^ (value (cdr nexp))
    (value (cdr (cdr nexp)))))))
```

你猜的没错

> 这个是错的

让我们试一试

> (+ 1 3)

`(atom ? nexp)`，其中 nexp 是`(+ 1 3)`

> 否

`(eq? (car nexp) (quote +))` 其中 nexp 是 `(+ 1 3)`

> 是

现在递归

> 是的。

`(cdr nexp)` 是什么 其中 nexp 是 `(+ 1 3)`

> (1 3)

(1 3) 现在不是我们定义的合法的算术表达式了

> 不，我们违反了第七戒。(1 3)子部分不是一个算术表达式的表达。我们对一个list表做递归。但是不是所有的list都是算术表达式的表达。我们必须是对子表达式做递归。

怎样得到算术表达式的第一个子表达式

> 取cdr再去car。就是第二个list成员。

`(cdr (cdr nexp))` 是一个算术表达式吗

> 不是，取cdr再取cdr就是(3)了，(3)不是算术表达式

再一次，我们之前把(+ 1 3)考虑的是表而不是算术表达式的表达。

> 取cdr取cdr再去car就得到了第二个子表达式，就是第三个list成员。

nexp取cdr再取nexp是什么

> 算术表达式的表达的第一个子表达式

我们来为算术表达式写一个函数 `1st-sub-exp`

```
(define 1st-sub-exp
(lambda (aexp)
(cond
 (else (car (cdr aexp))))))
```

为什么else

> 因为第一个查询也是最后一个查询

我们可以把(cond ...)去掉吗，既然不需要分支查询

> 当然，第四章的rember一行版本的就是这样子的

```
(define 1st-sub-exp
(lambda (aexp)
(car (cdr aexp))))
```

为算术表达式写函数 2nd-sub-exp

```
(define 2nd-sub-exp
(lambda (aexp)
(car (cdr (cdr aexp)))))
```

最后我们把(car nexp)替换为(operator nexp)

```
(define  operator
(lambda (aexp)
(car aexp)))
```

现在再一次重写函数value

```
(define value
(lambda (nexp)
(cond
((atom? nexp) nexp)
((eq? (operator nexp) (quote +))
 (+ (value (1st-sub-exp nexp))
    (value (2nd-sub-exp nexp))))
((eq? (operator nexp) (quote x))
 (x (value (1st-sub-exp nexp))
    (value (2nd-sub-exp nexp))))
(else
 (^ (value (1st-sub-exp nexp))
    (value (2nd-sub-exp nexp)))))))
```

我们可以对这一章的算术表达式的表达使用value函数吗

> 可以，通用改为使用 1st-sub-exp和 operator

试试看

```
(define 1st-sub-exp
  (lambda (aexp)
    (car (cdr aexp))))

(define operator
  (lambda (aexp)
    (car aexp)))
```

很简单不是吗

> 是的，因为我们用辅助函数来隐藏表达。

第八戒

```
<p>使用辅助函数来抽象表达</p>
```

我们之前见到过表达没

> 是的，只不过我们没告诉你

我们用过哪写表达

> 真值！数！

数是表达?

> 是的。比如说4是概念“4”。我们使用符号因为我们习惯阿拉伯数字表达。

我们还可以用什么表示呢

> `(() () () ())`也可以表示。`((((()))))`如何？`(I V)`怎么样？

还记得我们有多少个对数使用的元函数吗

> 四个：`numbers?` `zero?` `add1`和`sub1`

让我们再试试数的其它表示，零怎么表示

> 我们选择`()`

---

1 怎么表示

> `(())`

---

2 怎么表示

> `(() ())`

知道了？那3呢

> `(() () ())`

写一个函数来测试是否是zero零

> ```
> 1 (define sero?
> 2 (lambda (n)
> 3 (null? n)))
> ```

写一个像add1函数的函数

> ```
> 1 (define edd1
> 2 (lambda (n)
> 3 (cons (quote ()) n)))
> ```

那sub1呢

> ```
> 1 (define zub1
> 2 (lambda (n)
> 3 (cdr n)))
> ```

这样对吗

> 让我们瞧瞧看

当 n 是`()`时`(zub1 n)`是多少

> 没有答案，但是没有关系。——回忆cdr的规则。

使用表达重写 +

```
(define +
(lambda (n m)
(cond
 ((sero? m) n)
 (else (edd1 (+ n (zub1 m)))))))
```

+变了吗

> Yes and no.它变化了，但是只是一点点。

回忆下lat?

> 很简单：
>
> ```
> (define lat?
> (lambda (l)
> (cond
>  ((null?) #t)
>  ((atom? (car l)) (lat? (cdr l)))
>  (else #f))))
> ```
>
> 但是，你问这个干吗？

还记得当 ls 是 `(1 2 3)` 时，(lat? ls)是什么吗

> 真，当然了。

对于我们的新数，`(1 2 3)` 是什么

> `((()) (() ()) (() () ()))`

那当 ls 是 `((()) (() ()) (() () ()))` 时，`(lat? ls)` 是什么

> 假

这样有什么坏的地方吗

> 你必须得提防shadows了。

这是一个set集合吗

`(apple peaches apple plum)`

> 不是，apple出现了不止一次

`(set? lat)` 是真还是假，其中lat是 `(apples peaches peaches plums)`

> #t，因为没有重复出现的原子

那么 `(set? lat)` 呢，其中lat是 `()`

> #t，因为没有重复出现的原子

试试看写出函数set?

```
(define set?
(lambda (lat)
(cond
 ((null? lat) #t)
 (else (cond
         ((member? (car lat) (cdr lat)) #f)
         (else (set? (cdr lat))))))))
```

简化set?

```
(define set?
(lambda (lat)
(cond
 ((null? lat) #t)
 ((member? (car lat) (cdr lat)) #f)
 (else (set? (cdr lat))))))
```

这个函数对
`(apple 3 pear 4 9 apple 3 4)`
有用吗

> 是的，member?中现在用equal?替代了eq?

看到 `member?` 出现在 `set?` 的定义中你惊讶吗

> 别这样，我们已经写过 `member?` 了，现在我们想用就用。

`(makeset lat)` 是什么，其中lat是
`(apple peach pear peach plum apple lemon peach)`

> `(apple peach pear plum lemon)`

试试用member?写出函数makeset

```
(define makeset
(lambda (lat)
(cond
 ((null? lat) (quote ()))
 ((member? (car lat) (cdr lat))
  (makeset (cdr lat)))
 (else (cons (car lat)
             (makeset (cdr lat)))))))
```

函数短得让你惊讶吧

> 希望如此。别怕：这个是正确的。

使用上面的函数定义，那么`(makeset lat)`是什么，其中`(apple peach pear peach plum apple lemon peach)`

> `(pear plum apple lemon peach)`

试试用 multirember 重写 makeset 函数

```
(define makeset
(lambda (lat)
(cond
((null? lat) (quote ()))
(else (cons (car lat)
            (makeset
             (multirember (car lat)
                          (cdr lat))))))))
```

使用第二个定义，那么(makeset lat)是什么，其中lat是`(apple peach pear peach plum apple lemon peach)`

> `(apple peach pear plum lemon)`

用自己的话描述及喜爱第二个 makeset 函数定义是如何工作的。

> 下面是我们的描述：
> “函数 makeset 在用删除再次出现与第一个原子相同的原子成员后，把第一个原子cons到自然递归上。”

这个函数对`(apple 3 pear 4 9 apple 3 4)`有用吗

> 是的，multirember中现在用equal?替代了eq?

`(subset? set1 set2)`
是什么，其中
set1 是`(5 chicken wings)`
set2 是`(5 hamburgers 2 pieces fried chicken and light duckling wings)`

> #t，因为每一个set1中的原子也在set2中。

写出函数 subset?

```
(define subset?
(lambda (set1 set2)
(cond
 ((null? set1) #t)
 (else (cond
         ((member? (car set1) set2)
          (subset? (cdr set1) set2))
         (else #f))))))
```

你能写出一个更简单的吗

```
(define subset?
(lambda (set1 set2)
(cond
 ((null? set1) #t)
 ((member? (car set1) set2)
  (subset? (cdr set1) set2))
 (else #f))))
```

试试用(and ...)重写 subset?

```
(define subset?
(lambda (set1 set2)
(cond
((null? set1) #t)
(else
 (and (member? (car set1) set2)
      (subset? (cdr set1) set2))))))
```

`(eqset? set1 set2)`
是什么，其中
set1 是`(6 large chickens with wings)`,
set2 是`(6 chickens with large wings)`

> #t

写出函数eqset?

```
(define eqset?
(lambda (set1 set2)
(cond
 ((subset? set1 set2)
  (subset? set2 set1))
 (else #f))))
```

你能用一行的cond-写出 eqset吗

```
(defin eqset?
(lambda (set1 set2)
(cond
(else (and (subset? set1 set2)
           (subset? set2 set1))))))
```

写出一行版

```
(define eqset?
(lambda (set1 set2)
(and (subset? set1 set2)
    (subset? set1 set2))))
```

`(intersect? set1 set2)`
是什么，其中
set1 是 `(stewed tomatoes and macaroni)`
set2 是 `(macaroni and cheese)`

> #t，因为set1至少有一个原子出现在set2中。交集

定义函数 intersect?

```
(define intersect?
(lambda (set1 set2)
(cond
 ((null? set1) #f)
 (else (cond
         ((member? (car set1)  set2) #t)
         (else (intersect? (cdr set1) set2)))))))
```

写出更简洁的版本

```
(define intersect?
(lambda (set1 set2)
(cond
 ((null? set1) #f)
 ((member? (car set1) set2) #t)
 (else (intersect? (cdr set1) set2)))))
```

试试使用(or ...)写 intersect?函数

```
(define intersect?
(lambda (set1 set2)
(cond
 ((null? set1) #f)
 (else (or (member? (car set1)  set2)
           (intersect? (cdr set1) set2))))))
```

对比`subset?`和`interset?`

`(intersect set1 set2)`是什么，其中
set1 是`(stewed tomatoes and macaroni)`
set2 是`(macaroni and cheese)`

> `(and macaroni)`

现在你能写出简短版的函数 intersect

```
(define intersect
(lambda (set1 set2)
(cond
 ((null? set1) (quote ()))
 ((member? (car set1) set2)
  (cons (car set1)
        (intersect (cdr set1) set2)))
 (else (intersect (cdr set1) set2)))))
```

`(union set1 set2)`
是什么，其中
set1 是`(stewed tomatoes and macaroni casserole)`
set2 是`(macaroni and cheese)`

> `(stewed tomatoes casserole macaroni and cheese)`

写出函数union

```
(define union
(lambda (set1 set2)
(cond
 ((null? set1) set2)
 ((member? (car set1) set2)
  (union (cdr set1) set2))
 (else (cons (car set1)
             (union (cdr set1) set2))))))
```

这个函数是什么

```
(define xxx
  (lambda (set1 set2)
    (cond
     ((null? set1) '())
     ((member? (car set1) set2)
      (xxx (cdr set1) set2))
     (else
      (cons (car set1)
            (xxx (cdr set1) set2))))))
```

> 我们的解释是：
> “这个函数返回的是set1中有，但是set2中没有的原子。

`(intersectall l-set)`
是什么，其中
l-set 是 `((a b c) (c a d e) (e f g h a b))`

> (a)
> 它返回在l-set的所有子set中都有的原子。

`(intersectall l-set)`
是什么，其中 l-set 是

```
((6 pears and)
 (3 peaches and 6 peepers)
 (8 pears and 6 plums)
 (and 6 prunes with some apples))
```

> `(6 and)`
> 6 和 and 在上面4个列表中都出现了

现在使用任何你需要的的辅助函数写出 intersectall ， 假定所有的集合表都非空。

```
(define intersectall
(lambda (l-set)
(cond
((null? (cdr l-set)) (car l-set))
(else
 (intersect (car l-set)
            (intersectall (cdr l-set)))))))
```

这是一对 pair吗
`(pear pear)`

> 是的，列表中只有两个原子。

这是一对 pair吗
`(3 7)`

> 是

这是一对 pair吗
`((2) (pair))`

> 是。 因为列表中只有两个表达式。

`(a-pair? l)`
其中 l 是
`(full (house))`

> #t。 因为列表中只有两个表达式。

定义函数 a-pair?

```
(define a-pair?
(lambda (x)
(cond
 ((atom? x) #f)
 ((null? x) #f)
 ((null? (cdr x)) #f)
 ((null? (cdr (cdr x))) #t)
 (else #f))))
```

怎样取得一个piar的第一个 S-expression表达式

> 对pair取car

怎样取得一个piar的第二个 S-expression表达式

> 对pair取cdr再取car

怎样用两个原子构建一个 pair

> 把两个原子依次cons到一个`()`里，即`(cons x1 (cons x2 (quote ())))`

怎样用两个S-expression表达式构建一个 pair

> 把两个S-expression表达式依次cons到一个`()`里，即`(cons x1 (cons x2 (quote ())))`

觉得上边最后两个问题的回答有什么区别吗

> 没什么区别

```
(define first
  (lambda (p)
    (cond
      (else (car p)))))

(define second
  (lambda (p)
```

```
    (cond
      (else (car (cdr p))))))

(define build
  (lambda (s1 s2)
    (cond
      (else (cons s1 (cons s2 (quote ()))))))))
```

这三个函数会有什么用

> 它们被用来表示pair对和获得pair对的成员。详见第六章。他们用来提高可读性，见下文。

请把这三个函数用一行重定义。

```
(define first
  (lambda (p)
    (car p)))

(define second
  (lambda (p)
    (car (cdr p))))

(define build
  (lambda (s1 s2)
    (cons s1 (cons s2 (quote ())))))
```

你能写出third的一行定义吗

```
(define third
(lambda (p)
(car (cdr (cdr p)))))
```

---

l 是 rel 吗，其中，l 是 `(apple peaches pumpkin pie)`

> 不是，因为 l 不是全由 pair 对构成。我们使用 rel 来表示relation。

---

l 是 rel 吗，其中，l 是 `((apples peaches) (pumpkin pie) (apples peaches))`

> 不是，因为 l 不是pair对的一个的集合，有重复的。

---

l 是 rel 吗，其中，l 是 `((apples peaches) (pumpkin pie))`

> 是

---

l 是 rel 吗，其中，l 是 `((4 3) (4 2) (7 6) (6 2) (3 4))`

> 是

rel 是 fun 吗，其中 rel 是
`((4 3) (4 2) (7 6) (6 2) (3 4))`

> 不是，我们用fun表示function函数。

`(fun? rel)`
的值是什么，其中rel是
`((8 3) (4 2) (7 6) (6 2) (3 4))`

> #t，因为`(firsts rel)`是一个 set。 详见第三章。
> firsts取出各子列表的第一个元素：
> `(8 4 7 6 3)`
> 里边没有重复的原子

`(fun? rel)`
是什么，其中rel是
((d 4) (b 0) (b 9) (e 5) (g 4))

> #f，因为 b 重复了。

用 set? 和 firsts 写出函数fun?

```
(define fun?
(lambda (rel)
(set? (firsts rel))))
```

函数fun?是简单的one-liner吗

> 当然是啊

我们怎么定义有限函数

> 对于我们，有限函数是pair对组成的list表，其中每个pair的第一个元素没有相同的。

`(revrel rel)`
是什么，其中 l 是
`(8 a) (pumpkin pie) (got sick))`

> `((a 8) (pie pumpkin) (sick got))`

现在能写出revrel函数

```
(define revrel
(lambda (rel)
(cond
((null? rel) '())
(else
 (cons (build
        (second (car rel))
        (first (car rel)))
       (revrel (cdr rel)))))))
```

下面的这个对吗

```
(define revrel
  (lambda (rel)
    (cond
      ((null? rel) (quote ()))
      (else (cons (cons (car (cdr (car rel)))
                        (cons (car (car rel))
                              (quote ())))
                  (revrel (cdr rel)))))))
```

> 是的，现在你看出“representation表达”对可读性的益处了吧。

假设我们有如下函数revpair可以反转一个piar对的两个成员。

```
(define revpair
  (lambda (pair)
    (build (second pair) (first pair))))
```

我们如何用这个辅助函数重写revrel

> 没问题，而且更易读

```
(define revrel
(lambda (rel)
(cond
 ((null? rel) (quote ()))
 (else (cons (revpair (car rel))
             (revrel (cdr rel)))))))
```

猜猜看为什么fun不是fullfun，其中fun是
`((8 3) (4 2) (7 6) (6 2) (3 4))`

> fun不是fullfun，因为所有pair的第二项中2出现了不止一次。

为什么`(fullfun? fun)`是 #t，其中fun是
`((8 3) (4 8) (7 6) (6 2) (3 4))`

> 因为`(3 8 6 2 4)`是一个set。

`(fullfun? fun)`
是什么，其中fun是
`((grape raisin) (plum prune) (stewed prune))`

> #f

`(fullfun? fun)`
是什么，其中fun是
`((grape raisin) (plum prune) (stewed grape))`

> #t，因为`(raisin prune grape)`是一个集合。

定义fullfun?

```
(define fullfun?
(lambda (fun)
(set? (seconds fun))))
```

你能定义seconds吗

> 如同firsts

---

fullfun?还可以称作什么

> one-to-one一对一

你能想出第二种one-to-one的写法吗

```
(define one-to-one?
(lambda (fun)
(fun? (revrel fun))))
```

问
`((chocolate chip) (doughy cookie))`
是一个 one-to-one一对一函数吗?

> 是的，你现在就该来一份!

去点一份((chocolate chip) (doughy cookie))吧!
或者你自己做一份，这样不是更好吗

```
(define cookies
  (lambda ()
    (bake
     (quote (350 degrees))
     (quote (12 minutes))
     (mix
      (quote (walnuts 1 cup))
      (quote (chocolate-chips 16 ounces))
```

```
      (mix
       (mix
        (quote (flour 2 cups))
        (quote (oatmeal 2 cups))
        (quote (salt 5 teaspoon))
        (quote (baking-powder 1 teaspoon))
        (quote (baking-soda 1 teaspoon)))
       (mix
        (quote (eggs 2 large))
        (quote (vanilla 1 teaspoon))
        (cream
         (quote (butter 1 cup))
         (quote (sugar 2 cups)))))))))
```

还记得我们第五章末的rember和insertL吗

> 我们用equal?替换了eq?

你能用你eq?或者equal?写一个函数rember-f吗

> 还不能，因为我们还没告诉你怎么弄

你如何用 rember 删除 (b c a) 中的第一个a元素

> 把参数 a 和 (b c a) 传给rember

你如何用rember删除 (b c a) 中的第一个c元素

> 把参数 c 和 (b c a) 传给rember

你如何能够把 rember-f 的 eq? 用 equal? 代替

> 把equal? 作为参数传给 rember
> 笔记：这和C语言的函数指针作用一样啊。

`(rember-f test? a l)`
是什么，其中
test? 是 `=`
a 是 `5`
l 是 `(6 2 5 3)`

> `(6 2 3)`
>
> 注：
> Lisp:`(rember-f (function =) 5 '(6 2 5 3))`

`(rember-f test? a l)`
是什么，其中
test? 是`eq?`,
a 是`jelly`,
l 是`(jelly beans are good)`

> `(beans are good)`

`(rember-f test? a l)`
是什么，其中
test? 是`equal?`，
a 是`(pop corn)`，
l 是`(lemonade (pop corn) and (cake))`

> `(lemonade and (cake))`

试着写出函数 rember-f

```
(define rember-f
(lambda (test? a l)
(cond
((null? l) (quote()))
(else (cond
       ((test? (car l) a) (cdr l))
       (else (cons (car l)
                   (rember-f test? a
                             (cdr l)))))))))
```

> 注：
> Lisp：`(funcall test? (car l) a)`
> 当调用一个函数参数或者未定义函数时使用funcall。

那简短版呢

```
(define rember-f
(lambda (test? a l)
(cond
((null? l) (quote ()))
((test? (car l) a) (cdr l))
(else (cons (car l)
            (rember-f test? a
                      (cdr l)))))))
```

若 test? 是`eq?`，
`(rember-f test? a l)`
如何工作的

> 若 test? 是 eq?，`(rember-f test? a l)`如同 rember。

若 test? 是`equql?`，
`(rember-f test? a l)`
如何工作的

> 这是用 equal? 替换了 eq? 的 rember

现在我们有四个函数做差不多的事情

> 是的:
> 用 = 的 rember
> 用 equal? 的 rember
> 用 eq? 的 rember
> 还有
> rember-f

而 rember-f 的行为可以向其它的那样

> Let's generate all versions with rember-f

函数可以返回什么类型的值

> 表和原子

函数本身呢

> 可以的，只是你还不知道

你能说说 (lambda (a l) ...) 是什么吗

> 一个函数，有a和l两个参数。

这是什么

```
(lambda (a)
  (lambda (x)
    (eq? x a)))
```

> 这是一个函数，当传入参数 a 时返回函数`(lambda (x) (eq? x a))`，a 就是那个参数。

这就是所谓的"Curry-ing?"

> 感谢您，Moses Schonfinkel (1889-1942)

这不叫"Schonfinkel-ing"

> 感谢您，Haskell B.Curry (1900-1982)

用(define ...)给函数定义一个函数名

```
(define eq?-c
(lambda (a)
(lambda (x)
 (eq? x a))))
```

> 注.在Lisp中:

```
(defun eq?-c (a)
(function
(lambda (x)
 (eq x a))))
```

`(eq?-c k)`是什么，其中 k 是 `salad`

> 它的值是一个函数，把x作为函数参数测试其是否与salad是eq?相等的

所以让我们用`(define ...)`为它定义一个函数名
`(define eq?-salad (eq?-c k))`
其中 k 是 salad

注：
Lisp：
`(setq eq?-salad (eq?-c 'salad))`
使用setq定义的函数能够用funcall调用。

> 好。

`(eq?-salad y)`
其中 y 是 salad

> #t

> 注：
> Lisp：
> `(funcall eq?-salad y)`
> 因为 eq?-salad 还没有定义

`(eq?-salad y)`其中 y 是 `tuna`

> #f

我们需要给 eq?-salad 定义吗

> 不需要，我们可以这样`((eq?-c x) y)`，其中 x 是 salad，y 是 tuna
>
> 注：
> Lisp：
> `(funcall (eq?-c x) y)`
> 因为(eq?-c x)是未定义的函数

现在重写 rember-f 为一个参数test?的函数，返回一个如同用eq?替换test?的rember。

```
(define rember-f
(lambda (test?)
(lambda (a l)
 (cond
   ((null? l) (quote ()))
   ((test? (car l) a) (cdr l))
   (else (cons (car l) ...))))))
```

一个好开端

请用你自己的话描述`(rember-f test?)`的结果，其中 test? 是 eq?

> 结果是一个带两个参数 a，l 的函数。它比较表和 a，并且删除第一个出现的 a。

给`(rember-f test?)`的返回值起个名字，其中 test? 是 eq?

> `(define rember-eq? (rember-f test?))`，其中 test? 是 eq?

`(rember-eq? a l)`
是什么，其中
a 是`tuna`，
l 是`(tuna salad is good)`

> `(salad is good)`

若 test? 是 eq?，我们需要把`(rember-f test?)`起名为 rember-eq? 吗

> 不需要，我们可以直接`((rember-f test?) a l)`，其中 test? 是 eq?，a 是 tuna，l 是 (tuna salad is good)。

现在完成 rember-f 中的`(cons (car l) ...)`行让 rember-f 能够运行

```
(define rember-f
(lambda (test?)
(lambda (a l)
 (cond
   ((null? l) (quote ()))
   ((test? (car l) a) (cdr l))
   (else (cons (car l)
           ((rember-f test?) a (cdr l))))))))
```

`((rember-f eq?) a l)`
是什么，其中
a 是`tuna`，
l 是`(shrimp salad and tuna salad)`

> `(shrimp salad and salad)`

`((rember-f eq?) a l)`
是什么，其中
a 是`eq?`，
l 是`(equal? eq? eqan? eqlist? eqpair?)`

> `(equal? eqan? eqlist? eqpair?)`
> 前一个eq?是函数，而后一个eq?是原子

现在，把insertL改为insertL-f，方法可把rember改为rember-f一样。

```
(define insertL-f
(lambda (test?)
(lambda (new old l)
 (cond
  ((null?  l) (quote ()))
  ((test? (car l) old)
   (cons new (cons old (cdr l))))
  (else (cons (car l)
              ((insertL-f test?) new old (cdr l)))))))))
```

作为练习，再写出insertR

```
(define insertR-f
(lambda (test?)
(lambda (new old l)
 (cond
  ((null? l) (quote ()))
  ((test? (car l) old)
   (cons old (cons new (cdr l))))
  (else
   (cons (car l)
         ((insertR-f test?) new old (cdr l)))))))))
```

---

insertR-f 和 insertL-f 相似吗

> 只有中间的一点点不一样

你能写出既能在左边插入又能在右边插入的函数insert-g吗

> 如果你能，给自己来点咖啡蛋糕放松放松吧!不能也无所谓，别放弃，等下你就能看到了。

哪一片段不一样

> 第二行稍有不同。在insertL中是
>
> ((eq? (car l) old)
> (cons new (cons old (cdr l))))
>
> 但是在insertR中是
>
> ((eq? (car l) old)
> (cons old (cons new (cdr l))))

请描述出来!

> 我们这样说:
> “两个函数 cons old 和 new 到 cdr l 上的顺序不一样。

所以我们怎样消除差别

> 你大概猜出来了:传入一个函数来描述适当的cons方式。

定义一个 seqL 函数
用来:获取三个参数,第二个参数cons到第三个参数上,第一个参数cons到前面的结果上。

```
(define seqL
(lambda (new old l)
(cons new (cons old l))))
```

那下边这个呢

```
(define seqR
  (lambda (new old l)
    (cons old (cons new l))))
```

> 定义一个 seqL 函数用来:获取三个参数,第一个参数cons到第三个参数上,第二个参数cons到前面的结果上。

你知道为什么我们写这两个函数吗

> 因为他们表达出了insertL和insertR的不同之处

试试用一个参数seq写出函数 insert-g,其中在seq为seqL时返回insertL,在seq是seqR时,返回insertR

```
(define insert-g
(lambda (seq)
(lambda (new old l)
 (cond
  ((null? l) (quote ()))
  ((eq? (car l) old)
   (seq new old (cdr l)))
  (else (cons (car l)
              ((insert-g seq) new old (cdr l))))))))
```

现在用 insert-g 定义 insertL

> `(define insertL (insert-g seqL))`

还有 insertR

> `(define insertR (insert-g seqR))`

两个函数有什么异样的

> 之前我们可能会这样写
>
> `(define insertL (insert-g seq))`
>
> 其中 seq 是 seqL，以及
>
> `(define insertR (insert-g seq))`
>
> 其中 seq 是 seqR。
>
> 但是对于对函数传递参数，“其中...”是不需要的。

需要给seqL和seqR命名吗

> 不必，我们直接在定义里传递它们。

再次定义 insertL，这次不要用seqL

```
(define insertL
(insert-g
(lambda (new old l)
 (cons new (cons old l)))))
```

> 笔记：就是把seqL的定义直接替换`(define insertL (insert-g seqL))`中的 seqL。

这个更好点吗

> 是的，因为你不需要记住许多名字。你可以`(rember func-name "your-mind")`，其中 func-name 是 seqL

你还记得 subst 的定义吗

```
(define subst
(lambda (new old l)
(cond
((null? l) (quote ()))
((eq? (car l) old)
 (cons new (cdr l)))
(else
 (cons (car l)
       (subst new old (cdr l)))))))
```

这个看起来类似吗

> 是的，就像 insertL 和 insertR。仅仅cond的第二行不一样。

为 subst 定义一个类似 seqL 或者 seqR 的函数

```
(define seqS
(lambda (new old l)
(cons new l)))
```

现在用 insert-g 定义subst

> `(define subst (insert-g seqS))`

那你认为yyy是什么

```
(define yyy
   (lambda (a l)
      ((insert-g seqrem) #f a l)))
```

其中

```
(define seqrem
  (lambda (new old l) l))
```

> Surprise！这是我们的老朋友 rember。
> 提示：一步步推导
> `(yyy a l)`
> 其中 a 是 sausage
> l 是`(pizza with sausage and bacon)`
> #f是其什么作用的?
>
> 笔记：首先 (insert-gseqrem)返回的是一个函数。该函数输入new，old，l三个参数，输出时当发现l中的某个元素与old相同就跳过，恰好与rember的作用类似，只是new参数多余了。于是当定义yyy函数时，对(insert-g seqrem)函数带入的三个参数中，第一个是与函数作用无关的，可以是任意的，取个#f也无妨。
>
> 各位看官就自己判断，个人认为这样没错。

领略到了抽象的力量了吧

第九戒

从通用模式抽象出新的函数。

我们之前见到过一些相像的函数吗

> 有的

还记得第六章的 value 吗

```
(define value
(lambda (nexp)
(cond
((atom? nexp) nexp)
((eq? (operator nexp) (quote +))
 (+ (value (1st-sub-exp nexp))
    (value (2nd-sub-exp nexp))))
((eq? (operator nexp) (quote x))
 (x (value (1st-sub-exp nexp))
    (value (2nd-sub-exp nexp))))
(else
 (^ (value (1st-sub-exp nexp))
    (value (2nd-sub-exp nexp)))))))
```

看出相似性了没

> 最后三分支除了 +，×，^ 之外都一样。

你能写出一个 atom-to-function 吗，要满足下面的条件：

1. 一个参数x
2. 如果(eq? x (eqote +)) 返回函数+
   如果(eq? x (eqote ×)) 返回函数×
   否则返回函数^

```
(define atom-to-function
(lambda (x)
(cond
 ((eq? x (quote +)) +)
 ((eq? x (quote +)) x)
 (else ^))))
```

`(atom-to-function (operator nexp))`
是什么，其中
nexp 是 `(+ 5 3)`

> 函数 +，而不是原子 +。

你能用 atom-to-function 重写只有两个cond分支的value吗

> 当然

```
(define value
(lambda (nexp)
(cond
 ((atom? nexp) nexp)
 (else
  ((atom-to-function (operator nexp))
   (value (1st-sub-exp nexp))
   (value (2nd-sub-exp nexp)))))))
```

这个版本是不是比第一个版本简短多了?

> 是的，但是都 OK。我们没改变含义。

来一个苹果怎么样

> One a day keeps the doctor away。每天坚持吃个苹果，永远无需大夫看我。

下面是 multirember

```
(define multirember
  (lambda (a lat)
    (cond
     ((null? lat) (quote ()))
     ((eq? (car lat) a)
      (multirember a (cdr lat)))
     (else
      (cons (car lat)
            (multirember a (cdr lat)))))))
```

写一个函数 multirember-f

> 没问题。

```
(define multirember-f
(lambda (test?)
(lambda (a lat)
 (cond
  ((null? lat) (quote ()))
  ((test? a (car lat))
   ((multirember-f test?) a (cdr lat)))
  (else
   (cons (car lat)
         ((multirember-f test?) a (cdr lat)))))))))
```

`((multirember-f test?) a lat)` 是什么，其中 test 是 `eq?`， a 是 `tuna`， lat 是 `(shrimp salad tuna salad and tuna)`

> `(shrimp salad salad and)`

难道不是easy的很吗

> 是啊

用 multirember-f 定义 multirember-eq?

```
(define multirember-eq?
(multirember-f test?)
```

我们真的需要给 multirember-f 传递tuna吗

> 当 multirember-f 访问lat中所有元素时，它总是寻找tuna

当 multirember-f 访问lat时test?改变吗

> 不会，test? 一直是 eq?，如同 a 总是 tuna。

我们能把a和test?合并吗

> 其实，test?可以是一个参数，与tuna比较

怎么做

> 新参数test?取得一个参数并把它同tuna比较

下面是一种写法

```
(define eq?-tuna
  (eq?-c k))
```

其中k是tuna，你能想出一个其他方法来表示吗

> 下面是另一种方法

```
(define eq?-tuna
(eq?-c (quote tuna)))
```

你见过包含原子的定义吗

> 见过，0，(quote ×), (quote +),还有更多

或许我们现在该写一个与 multirember-f 类似的 multiremberT 函数。 multiremberT用 eq?-tuna 函数和lat来计算，而不是输入参数test?返回一个函数。

> 这个没什么难度

```
(define multiremberT
(lambda (test? lat)
(cond
((null? lat) (quote ()))
((test? (car lat))
 (multiremberT test? (cdr lat)))
(else (cons (car lat)
            (multiremberT test? (cdr lat)))))))

```

`multirememberT test? lat)`

其中

test? 是`eq?-tuna`,

lat 是`(shrimp salad tuna salad and tuna)`

> `(shrimp salad salad and)`

简单吗

> 不算难

那这个呢

```
(define multirember&co
  (lambda (a lat col)
    (cond
     ((null? lat)
      (col (quote ()) (quote ())))
     ((eq? (car lat) a)
      (multirember&co a (cdr lat)
                      (lambda (newlat seen)
                        (col newlat
                             (cons (car lat) seen)))))
     (else
      (multirember&co a (cdr lat)
                      (lambda (newlat seen)
                        (col (cons (car lat) newlat) seen)))))))
```

> 看起来好复杂!

这个看起来简单点:

```
(define a-friend
  (lambda (x y)
    (null? y)))
```

> 是的，这个很简单。这个函数输入两个参数并查询第二个参数是否为空表，而忽视第一个参数。

`(multirember&co a lat col)`是什么,

其中 a 是`tuna`,

lat 是`(strawberries tuna and swordifish)`,

col 是`a-friend`

> 这个可不简单。

所以让我们试试简单点的例子吧。

`(multirember&co a lat col)`,

其中 a 是`tuna`,

lat是`()`,

col 是 `a-friend`

> #t真，因为给的参数得到第一个分支查询， a-friend保证第二个参数为空。

`(multirember&co a lat col)`是什么，
其中 a 是`tuna`，
lat 是`(tuna)`，
col 是 `a-friend`

> multirember&co 查询`(eq? (car lat) (quote tuna))`，其中lat是`(tuna)`。然后递归`()`。

---

multirember&co自然递归的其它参数是什么

> 第一个参数时tuna。第三个参数是一个新的函数

第三个参数名是什么

> col

你知道现在col表示什么吗

> col是“collector”的简称，有时称为“continuation”连续。

这是新的collector

```
(define new-friend
  (lambda (newlat seen)
    (col newlat
      (cons (car lat) seen))))
```

其中(car lat)是tuna，col是 a-friend，你能把这个定义稍稍改写一下吗

> ```
> (define new-friend
> (lambda (newlat seen)
> (col newlat
>  (cons (quote tuna) seen))))
> ```

我们能在这个定义里把col替换为 a-friend 吗

> 可以：
>
> ```
> (define new-friend
> (lambda (newlat seen)
> (a-friend newlat
>  (cons (quote tuna) seen))))
> ```
>
> 笔记：就是cond第二个分支的那个lambda匿名函数

现在呢?

> multirember&co 查找出(null? lat)是真，也就是它把collector作用在了两个空表上。

这是哪个collector?

> new-friend

---

a-friend 与 new-friend有何不同

> new-friend使用 a-friend 查询空表，还有值`(cons (quote tuna) (quote ()))`

原来那个 collector 对这个参数做什么了

> 返回#f，因为第二个参数是(tuna),不是空表。

`(multirember&co a lat a-friend)`
是什么值，其中
a 是`tuna`,
lat 是`(and tuna)`

> 这次 multirember&co 对另一个friend递归

```
(define latest-friend
(lambda (newlat seen)
(a-friend (cons (quote and) newlat) seen)))
```

这次这个 multirember&co 的递归的值是什么

> #f，因为`(a-friend ls1 ls2)`是 #f，其中 ls1 是 (and)，ls2 是 (tuna)。

`(multirember&co a lat f)`做些什么

> 查询lat中的每一个atom原子看和a是否eq？。不同的放在 ls1，相同的放在 ls2，最后，它确定(f ls1 ls2)的值。
>
> 笔记：ls1就是那个newlat,seen就是ls2。准确的的说最后ls1是当那个(car lat)不是a时不断(cons (car lat) newlat)得到的一个表；ls2是当(car lat)是a时，不断(cons (car lat))的一个表。(f ls1 ls2)为真值表示没有把a找到并放入ls2，最后所有的原子都到了newlat中，seen一直为空；假表示ls2中找到了所有的a，它是非空了。函数 multirember&co在递归的三个分支中使用了不同的col函数去处理，最后递归的结果是把lat分成两个lat，以是否是a为条件。其中每深一次的递归 multirember&co 代入的col都是比上一层的递归多一层col。

最后的问题:
`(multirember&co (quote tuna) ls col)`
的值是什么，其中
ls 是`(strawberries tuna and swordfish)`,
col 是

```
(define last-friend
  (lambda (x y)
    (length x)))
```

> 3，因为 ls 包含三个不是tuna的原子，所以 last-friend作用于(strawberries and swordifish)和(tuna)

是的!

> 这可真是奇怪的食物，不过我们已经见识过外国菜了。

第十戒

```
<p>用函数，一次collect不止一个值</p>
```

下面这个是老朋友

```
(define multiinsertL
  (lambda (new old lat)
    (cond
     ((null? lat) (quote ()))
     ((eq? (car lat) old)
      (cons new (cons old
                      (multiinsertL new old
                                    (cdr lat)))))
     (else (cons (car lat)
                 (multiinsertL new old
                               (cdr lat)))))))
```

还记得 multiinsertR 吗

> 当然了
>
> ```
> (define multiinsertR
> (lambda (new old lat)
> (cond
> ((null? lat) (quote ()))
> ((eq? (car lat) old)
>  (cons old
>        (cons new
>              (multiinsertR new old
>                            (cdr lat)))))
> (else (cons (car lat)
>             (multiinsertR new old
>                           (cdr lat)))))))
> ```

现在试试写个 multiinsertLR
提示： 如果oldL和oldR是不同的，multiinsertLR 把 new 插入到 oldL 的左边和 oldR的右边

> 下面这个把两个函数和在一起了

```
(define multiinsertLR
(lambda (new oldL oldR lat)
(cond
((null? lat) (quote ()))
((eq? (car lat) oldL)
 (cons new
       (cons oldL
             (multiinsertLR new oldL oldR
                            (cdr lat)))))
((eq? (car lat) oldR)
 (cons oldR
       (cons new
             (multiinsertLR new oldL oldR
                            (cdr lat)))))
(else
 (cons (car lat)
       (multiinsertLR new oldL oldR
                      (cdr lat)))))))
```

---

multiinsertLR&co 之与 multiinsert 如同multirember&co 之与 multirember

> 这意味着 multiinsertLR&co 需要比 multiinsertLR 更多的参数的参数?

是的。参数是什么呢?

> 是一个collector函数

当 multiinsertLR&co 完成，它会对new lat，对左插入的数，右插入的数使用col。你能写出 multiinsertLR&co 的要略吗

```
(define multiinsertLR&co
(lambda (new oldL oldR lat col)
(cond
((null? lat)
 (col (quote ()) 0 0))
((eq? (car lat) oldL)
 (multiinsertLR&co new oldL oldR
                   (cdr lat)
                   (lambda (newlat L R)
                     ...)))
((eq? (car lat) oldR)
 (multiinsertLR&co new oldL oldR
                   (cdr lat)
                   (lambda (newlat L R)
                     ...)))
(else
 (multiinsertLR&co new oldL oldR
                   (cdr lat)
                   (lambda (newlat L R)
                     ...))))))
```

为什么当`(nul? lat)`为真时，col作用于`(quote () 0 0)`

> 空lat既没有oldL也也没有oldR。这表示oldL和oldR发现了0次， multiinsertLR 当lat为空时返回的是 ()。

那么

```
(multiinsertLR&co
  (quote  cranberries)
  (quote fish)
  (quote chips)
  (quote ())
  col)
```

的值是什么

> 是`(col (quote ()) 0 0)`，但是因为我们不知道col是什么，还无法确定它。

当(car lat) 既不等于 oldL也不等于oldR时, multiinsertLR&co 对三个参数作用新的collctor对吗

> 是的，第一个参数是lat， 原本是 multiinsertL 作用(cdr lat), oldL， 和oldR。第二个和第三个是对应插入oldL左边和oldR右边的次数。

然后，因仅当出现oldL或者oldR才复制，所以 multiinsertLR&co 会对(cons (car lat) newlat)使用col对吗

> 是的，没错。所以我们知道了最后一个分支情况的collector了

```
(lambda (newlat L R)
(col (cons (car lat) newlat) L R))
```

为什么col的第二个和第三个参数是 L 和 R

> 如果(car lat) 即不是oldL也不是oldR，我们不需要插入任何新的元素。所以L和R对(cdr lat)和lat都是对的。

下面这个是我们现在得到的，新添了一个collector

```
(define multiinsertLR&co
  (lambda (new oldL oldR lat col)
    (cond
     ((null? lat)
      (col (quote ()) 0 0))
     ((eq? (car lat) oldL)
      (multiinsertLR&co new oldL oldR
                        (cdr lat)
                        (lambda (newlat L R)
                          (col (cons new
                                     (cons oldL newlat))
                               (add1 L) R))))
     ((eq? (car lat) oldR)
      (multiinsertLR&co new oldL oldR
                        (cdr lat)
                        (lambda (newlat L R)
```

```
                          ....)))
    (else
     (multiinsertLR&co new oldL oldR
                       (cdr lat)
                       (lambda (newlat L R)
                         (col (cons (car lat) newlat) L R)))))))
```

> 未完成的那个collector和新添加的collector很类似。只是对 R 加 1，而不是对 L 加 1；另外，是cons old到cons new newlat的结果上。

那你能填写省略号吗

> 当然了， 最后这个 collector 是
>
> ```
> (lambda (newlat L R)
> (col (cons oldR (cons new newlat)) L (add1 R)))
> ```

`(multiinsertLR&co new oldL oldR lat col)`
的值是什么，其中
new 是 salty，
oldL 是 fish，
oldR 是 chips，
lat 是 (chips and fish or fish and chips)

> 是`(col newlat 2 2)`的值，其中 newlat 是
> `(chips salty and salty fish or salty fish and chips salty)`

这个对身体健康吗(译注：指的是chips salty and salty fish or salty fish and chips salty)

> 盐太多了。或许点心甜一些。

还记得什么是 *-函数吗

> 当然了，所有的*-函数都能对各种list进行操作
>
> > 空表
> > cons 原子到list表上的lsit表
> > cons list表到list表上的lsit表

现在写一个函数 evens-only*。它删除嵌套list中所有的奇数。下面这是even?

```
(define even?
  (lambda (n)
    (= (x (÷ n 2) 2) n)))
```

> 我们已经讲过怎么写这种函数， evens-only*只是个练习：

```
(define evens-only*
(lambda (l)
(cond
((null? l) (quote ()))
((atom? (car l))
 (cond
  ((even? (car l))
   (cons (car l)
         (evens-only* (cdr l))))
  (else (evens-only* (cdr l)))))
(else (cons (evens-only* (car l))
            (evens-only* (cdr l)))))))
```

`(evens-only* l)`的值是什么，其中 l 是 `((9 1 2 8) 3 10 ((9 9) 7 6) 2)`

> `((2 8) 10 (() 6) 2)`

如果 l 是`((9 1 2 8) 3 10 ((9 9) 7 6) 2)`，
那么 l 中奇数的总和是多少

> 9 + 1 + 3 + 9 + 9 + 7 = 38

如果 l 是`((9 1 2 8) 3 10 ((9 9) 7 6) 2)`，
那么 l 中偶数的乘积是多少

> 2 × 8 × 1 0 × 6 × 2 = 1920

你能写出一个 evens-only*&co 函数吗，它把奇数从参数中删除来构建嵌套列表，同时把所有偶数相乘，奇数相加

> 这可是全明星哦!

下面这里是个要略。你能解释(evens-only*&co (car l) ...) 是干什么的吗

```
(define evens-only*&co
  (lambda (l col)
    (cond
     ((null? l)
      (col (quote ()) 1 0))
     ((atom? (car l))
      (cond
       ((even? (car l))
        (evens-only*&co (cdr l)
                        (lambda (newl p s)
                          (col (cons (car l) newl)
                               (x (car l) p) s))))
       (else (evens-only*&co (cdr l)
                             (lambda (newl p s)
                               (col newl
                                 p (+ (car l) s))))))
      (else (evens-only*&co (car l)
            ...))))))
```

> 它访问(car l)中的每一个数，然后收集表中的偶数，得到偶数的乘积，奇数的和。

函数 evens-only*&co 在访问了所有的 (car l)中的数后做什么了

> 如同前边的例子一样，使用collector，不过我们还没有定义它。

这个collector做了什么

> 用evens-only*&co访问(cdr l)中的所有偶数，然后计算了偶数的乘积，奇数的和。

这就是说那个collector大体上应该是这样的对么

```
(lambda (al ap as)
  (evens-only*&co (cdr l)
    ...))
```

> 是的。

当`(evens-only*&co (cdr l) ...)`执行完毕之后呢

> 那个还没定义的collector就开始执行了

`(evens-only*&co (cdr l) ...)`的collector做了什么

> 它把list中car和cdr的结果cons到一起，然后做相应的乘积和求和。最后把值传给最开始输入的那个老collector
>
> ```
> (lambda (al ap as)    ;collector of (evens-only*&co (cer l) ...)
> (evens-only*&co (cdr l)
> (lambda (dl dp ds)
>  (col (cons al dl)
>     (x ap dp)
>     (+ as ds)))))
> ```
>
> 笔记：这个情形下，car l不是原子而是表，稍稍麻烦点，(car l)和(cdr l)需要分别递归得到两个部分的偶数乘积和奇数，最后对应相乘和相加。

理解这些了吗

> perfect!

`(evens-only*&co l the-last-friend)` 的值是什么，其中 l 是`((9 1 2 8) 3 10 ((9 9) 7 6) 2)`， the-last-friend的定义如下：

```
(define the-last-friend
  (lambda (newl product sum)
    (cons sum
      (cons product newl))))
```

> `(38 1920 (2 8) 10 (() 6) 2)`

哇哦！你的脑子搅起来了吧？

> 去吃块椒盐饼干吧，别忘了芥末哦。

你想来点鱼子酱吗?

> 那就去找它吧。

`(looking a lat)` 是什么，其中 a 是 `caviar`， lat 是 `(6 2 4 caviar 5 7 3)`

> #t 真，caviar 当然是 lat 了

`(looking a lat)`, 其中 a 是 `caviar`， lat 是 `(6 2 grits caviar 5 7 3)`

> #f

你察觉到什么不同吗

> 是啊，caviar 不是一直在 lat 中吗

没错，但是 lat 中第一个数是多少

> 6

lat 的第六个元素是什么

> 7

lat 的第七个元素是什么

> 3

所以 looking 是找不到 caviar 的

> 是的，因为第三个元素是 grits，这个不是 caviar

下面这是 looking

```
(define looking
  (lambda (a lat)
    (keep-looking a (pick 1 lat) lat)))
```

写个函数 keep-looking

> 我们不期待你能懂这个。

`(looking a lat)`, 其中 a 是 `caviar`, lat 是 `(6 2 4 caviar 5 7 3)`

> #t,因为`(keep-looking a 6 lat)`和`(keep-looking a (pick 1 lat) lat)`答案相同。

`(pick 6 lat)`是什么，其中 lat 是`(6 2 4 caviar 5 7 3)`

> 7

所以我们该干什么

> `(keep-looking a 7 lat)`, 其中 a 是`caviar`， lat 是`(6 2 4 caviar 5 7 3)`

`(pick 7 lat)`是什么，其中 lat 是`(6 2 4 caviar 5 7 3)`

> 3

`(keep-looking a 3 lat)`是什么，其中 a 是`caviar`， lat 是`(6 2 4 caviar 5 7 3)`

> 也就是 (keep-looking a 4 lat)。

也就是?

> #t。

写出 keep-looking

```
(define keep-looking
  (lambda (a sorn lat)
    (cond
      ((number? sorn) (keep-looking a (pick sorn lat) lat))
      (else (eq? sorn a)))))
```

你能猜出 sorn 表示什么吗

> 符号或者数(Symbol or number)

---

keep-looking 有什么特殊的地方吗

> 它没有递归 lat 的部分

我们把这个称为非自然递归"unnatural" recursion。

> 的确不自然。

---

keep-looking 慢慢的接近它的目标了吗

> 是的，各方面证据确凿。

它总是接近自己的目标吗

> 有时候表中可能既没有 caviar 也没有 grits。

一个表可能是一个 tup。

> 是的，如果我们 looking (7 2 4 7 5 6 3),我们将永远不会停止 looking。

`(looking a lat)` 是什么，其中 a 是 `caviar`， lat 是 `(7 1 2 caviar 5 6 3)`

> 这个太奇怪了！

是很奇怪。发生了什么？

> 我们不停的 looking...

像 looking 这样的函数称为 partial function。那么你认为我们之前见过的函数叫什么呢？

> total。

你能定义一个函数，对于一些参数，它永远达不到他的目标吗？

```
(define eternity
  (lambda (x)
    (eternity x)))
```

多少个参数可以让函数 eternity 到达它的目标？

> 没有。这怕是最不自然的递归了。

eternity 是 partial 的吗？

> 它是最 partial 的 function 了。

`(shift x)` 是什么，其中 x 是 `((a b) c)`

> `(a (b c))`

`(shift x)` 是什么，其中 x 是 `((a b) (c d))`

> `(a (b (c d)))`

定义 shift 函数

> 小事一桩；这连递归都不是！

```
(define shift
(lambda (pair)
(build (first (first pair))
       (build (second (first pair))
              (second pair)))))
```

描述 shift 做了什么

> 下面是我们的说法：
> “函数 shift 输入参数为一个 pair。输入参数的第一个元件是一个 pair。函数把第一个元件的 fisrt 部分移到了第二个元件中，形成一个新的 pair。”

现在看看下面这个函数：

```
(define align
  (lambda (pora)
    (cond
      ((atom? pora) pora)
      ((a-pair? (first pora)) (align (shift pora)))
      (else (build (first pora) (align (second pora)))))))
```

这与函数 keep-looking 有什么共同点

> 两个函数都在递归时改变参数但是两种情况的改变都不能保证接近目标。

为什么我们不能保证 align 的进展?

> 在 cond 第二行 shift 为 align 产生的参数不是原来参数的一部分。

那样违反了哪个戒律?

> 第七戒

新的参数至少比原来的小?

> align里边可没看出来。

为什么不是?

> 函数 shift 仅仅把 pair 中的顺序调整了一下。

所以?

> 得到的结果和 shift 的参数都有同样多的原子。

你能写出一个函数来计align的参数的原子数目吗?

> 没问题：

```
(define length*
  (lambda (pora)
    (cond
      ((atom? pora) 1)
      (else
       (+ (length* (first pora))
          (length* (second pora)))))))
```

---

align是一个 partial function 吗?

> 我们还不知道。或许有参数可以让它 align。

输入给 align 的参数在递归时，有别的什么变化吗?

> 有的。pair 的第一个元件变得更加简单，而第二个元件变得更加复杂。

第一个元件是怎样变得简单的?

> 它仅仅是原来 pair 的第一个元件的一部分。

这不就是意味着 length* 是错误的检测参数长度的函数？你能找到更好的函数吗?

> 更好的函数应该更加注意第一个元件。

那我们需要对第一个元件的多少注意力呢?

> 至少是多一倍。

意思是像 weight* 这样的吗

```
(define weight*
  (lambda (pora)
    (cond
      ((atom? pora) 1)
      (else
       (+ (x (weight* (first pora)) 2)
          (weight* (second pora)))))))
```

> 这才对。
> 笔记：真不知道作者把这个length*和这个weight*放在这是什么意思。

`(weight* x)` 是什么，其中 x 是 `((a b) c)`

> 7

这意味着参数变得简单了?

> 是的， align的参数的weight*依次变得更小了。

---

align是一个 partial function?

> 不是，它对任何参数都得到一个值。

下面这个类似align的函数 shuffle ， 使用了第七章的 revpair 函数替代了 shift:

```
(define shuffle
  (lambda (pora)
    (cond
      ((atom? pora) pora)
      ((a-pair? (first pora)) (shuffle (revpair pora)))
      (else (build (first pora) (shuffle (second pora)))))))
```

这表示 shuffle 是 total function 吗?

> 我们不知道。

让我们试试。`(shuffle x)` 的值是什么，其中 x 是 `(a (b c))`

> `(a (b c))`

`(shuffle x)`, 其中 x 是 `(a b)`

> (a b)

让我们试试有趣的东西。`(shuffle x)` 的值是什么，其中 x 是 `((a b) (c d))`。

> 为了确定这个值，我们必须找出 `(shuffle (revpair pora))` 的值，其中 pora 是 `((a b) (c d))`

我们现在打算怎么做?

> 我们打算确定 `(shuffle pora)` 的值，其中 pora 是 `((c d) (a b))`。

那不是意味着我们需要知道 `(shuffle (revpair pora))` 的值，其中 `(revpair pora)` 是 `((a b) (c d))`

> 是的。

然后呢?

> shuffle 函数不是 total function, 因为它再一次交换了 pair 的元件，这意味着我们又重头开始了。

这个是 total function吗

```
(define C
  (lambda (n)
    (cond
      ((one? n) 1)
      (else
       (cond
         ((even? n) (C (÷ n 2)))
         (else (C (add1 (× 3 n)))))))))
```

> 对于0，它没有值，其它的就天知道了。感谢您，Lothar Collatz (1910-1990)。

`(A 1 0)` 的值是多少

> 2

`(A 1 1)`

> 3

`(A 2 2)`

> 7

下面是A的定义

```
(define A
  (lambda (n m)
    (cond
      ((zero? n) (add1 m))
      ((zero? m) (A (sub1 n) 1))
      (else (A (sub1 n) (A n (sub1 m)))))))
```

> 感谢您，Wilhelm Ackermann (1853-1946)。

A与 shuffle 和 looking 有什么共同点?

> A的参数像 shuffle和 looking的一样在递归的时候减小都不是必要的。

举个例子?

> 很简单：`(A 1 2)` 需要求 `(A 0 (A 1 1))`。这又需要我们求 `(A 0 3)`。

A总是给出答案吗?

> 是的，它是 total function。

那(A 4 3)是多少?

> 对于实际情况，这是没有答案的。

什么意思?

> 在你读完这一页之后很久很久(A 4 3)的值都难以计算完成。

但是答案没有出来——真奇怪，因为它们吃掉了所有的答案。

The Walrus and The Carpenter

——Lewis Carroll

如果我们能写一个函数来告诉我们哪些函数对任意输入都有返回值不是很好吗?

> 当然可以。既然我们已经知道有一些函数永远没有返回值或者返回值得到的太慢了，我们应该搞一些这样的工具。

Okey，我们开始写吧。

> 听起来好复杂。一个函数可以对输入许多不同的参数都运行正常。

那我们把它做简单点。作为热身练习，让我们关注一种函数，它检查某些函数是否在输入空表时会停止。

> 这样简化了很多。

下面这是函数的开头：

```
(define will-stop?
  (lambda (f)
      ...))
```

你能填写省略的部分吗?

> 它做什么啊?

will-stop? 对所有参数都有返回值吗?

> 这个简单：我们说它要么返回 #t 要么返回 #f，这依赖于当输入参数(某个函数)对空表()作用时是否停止。

will-stop?是 total function吗?

> 是的。它总是返回 #t 或者 #f。

那我们来看一些例子。`(will-stop? f)`，其中 f 是`length`

> 我们知道`(length l)`是 0，其中 l 是`()`。

所以?

> `(will-stop? length)`的值是 #t 。

当然。其它例子呢? `(will-stop? eternity)`的值是什么?

> 先前我们已经知道`(eternity (quote ()))`不会返回一个值。

就是说`(will-stop? eternity)`的值是 #f 吗?

> 对。

我们需要更多的例子吗?

> 还需更多例子。
> 笔记：也可以参见维基百科的停机问题。

Okey，下面的这个对于 will-stop 是一个有意思的参数。

```
(define last-try
  (lambda (x)
    (and (will-stop? last-try) (eternity x))))
```

(will-stop? last-try) 是什么?

> 它是做什么的?

我们需要用()做测试。

> 如果我们想要得到`(last-try (quote ()))`，的值我们必须求得`(and (will-stop? last-try) (eternity (quote ())))`

`(and (will-stop? last-try) (eternity (quote ())))`
的值是什么?

> 这依赖于`(will-stop? last-try)`的值。

这有两种可能。我们假设`(will-stop? last-try)`是 #f

> Okey，那么`(and #f (eternity (quote ())))`是 #f，因为 (and #f ...) 总是 #f。

所以`(last-try (quote ()))`停止了，对吗?

> 是的。
> 笔记：停止的意思是，它得到了一个具体的值 #f，然后运行结束了。这里先不管(eternity '())的值，因为(and #f ...)总是得到#f，只要遇到一个值为false的表达式，后面的表达式不再求值，直接返回 #f.所以程序正常地结束了。于是悖论产生了：我们在函数内部假定(will-stop? last-try)是不会结束的？？？

那 will-stop?不就会预言反了吗?

> 是的。我们假设的是当`(will-stop? last-try)`是 #f 时,last-try 不停止。

所以我们假设`(will-stop? last-try)`错了?

> 是的。它必须返回 #t，因为 will-stop? 总是给出一个答案。我们称之为 total。

好的。如果`(will-stop? last-try)`是 #t ，那么`(last-try (quote ()))`的值是什么?

> 现在我们只需要判别`(and #t (eternity (quote ())))`，这和`(eternity (quote ()))`的值是相同的。

`(eternity (quote ()))` 的值是什么?

> 它没有值。我们知道它不会停的。

但是这表示我们又错了!

> 是的，因为这次我们假设的是 `(will-stop? last-try)` 是 #t 。
> 笔记：这一次假定(will-stop? last-try)返回 #t，也就是说传递空列表'()给last-try函数，它是能正常结束，并返回一个值。但是我们又错了，程序会在这里无限递归下去：(eternity '())，永远也不会结束。那么我们刚刚的假设不是又错了吗。也就是说，will-stop?函数结论上无法写出来。图灵已经证明过这个问题是无解的了。。。
>
> 一个相似的悖论：
> 理发师悖论：村子里有个理发师，这个理发师有条原则是，对于村里所有人，当且仅当这个人不自己理发，理发师就给这个人理发。如果这个人自己理发，理发师就不给这个人理发。无法回答的问题是，理发师给自己理发么?

你认为这说明了什么?

> 这是我们的表述：
> “我们认真的推导了两种可能的情况。如果我们定义 will-stop 那么 (will-stop? last-try) 必须是要么 #t 要么 #f。但是它不能——因为 will-stop? 没有作到定义要求。这就是说 will-stop?不能定义出来。”

这是唯一的吗?

> 是的。它是我们能够确切描述但是无法用我们的语言定义的函数。

这个问题有解决方法吗?

> 没有。感谢您，Alan M. Turing (1912-1954) 和Kurt Godel (1906-1978)。

`(define ...)` 是什么?

> 这是个有趣的问题。我们刚才看到(define ...)对 will-stop?不起作用。

所以递归的定义是什么?

> 抱紧，深吸一口气，等你准备好了再投身前进。

> 笔记：书里边下面的内容Y算子，但是讲的效果不好。建议直接参见维基百科的Y算子。或者参考其他书。

这是函数 length 吗

```
(define length
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (length (cdr l)))))))
```

> 是啊。

假如我们没有(define ...)呢？我们依旧可以定义 length 吗?

> 没有(define ...), length与什么都没有联系，和 length 的函数体更没有关系。

下面这个函数做了什么

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (eternity (cdr l))))))
```

> 他确定了空表的长度，其它的什么也没做。

当我们把它作用在非空表上呢

> 没有答案。如果我们给 eternity 一个参数，它不会给出答案。

这个看起来像是 length 函数的定义有什么用?

> 它对非空表都没有答案。

假设我们可以对这个定义一个函数名，那么可以怎么起?

> length0。因为它只能确定空表的长度。

你如何写一个函数来确定含有1个或者更少项的表的长度呢?

> 我们可以这样子:

```
(lambda (l)
(cond
((null? l) 0)
(else (add1 (length0 (cdr l))))))
```

差不多，但是 length0 没有使用 (define ...) 去定义。

> 所以，length0 被它的定义代替。

```
(lambda (l)
(cond
((null? l) 0)
(else (add1 ((lambda (l)
               (cond
                 ((null? l) 0)
                 (else (add1 (eternity (cdr l))))))
             (cdr l))))))
```

给这个函数其个什么名字好呢?

> 简单： length≤1。

下面这个函数是不是可以确定含有两个或者更少的项的表的长度呢?

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 ((lambda(l)
                   (cond
                     ((null? l) 0)
                     (else (add1 ((lambda(l)
                                    (cond
                                      ((null? l) 0)
                                      (else (add1 (eternity (cdr l))))))
                                  (cdr l))))))
                 (cdr l))))))
```

> 是的，这是 length≤2 。我们把 eternity替换为下一个版本的 length。

现在你认为n递归是什么?

> 什么意思?

现在，我们已经知道如何确定没有项的表，有一项的表，有两项的表的长度。我们怎样使函数 length 返回?

> 如果我们能写出一个 length0,length≤1,length≤2,...的无穷函数，那么我们就能写出函数 length∞来。这样我们就可以处理所有长度的表的情况了。

我们能搞出多长的表?

> 空表，或者1个元素，或者2个元素，或者3个元素，...，1001个，...

但是我们写不出来无穷函数。

> 是写不出来啊。

我们在这些个函数里边重复了许多模式。

> 是的。

这些模式看起来是什么样子的?

> 所有的这些个程序都包含像 length 的部分。或许我们应该把这个函数抽象出来：见第九戒。

我们开始吧!

> 我们需要一个看起来像是 length 函数，它以 (lambda (length) ...) 开头。

是说像这样子吗?

```
((lambda (length)
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (length (cdr l)))))))) eternity)
```

> 没错，就是这样。它创建了 length0。
> 笔记：使用 beta 规约 (就是 eternity 替换 length) 就可以得到 length0

同样的方式重写 length≤1。

```
((lambda (f)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (f (cdr l)))))))
 ((lambda (g)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1 (g (cdr l)))))))
  eternity))
```

> 笔记：使用 beta 规约 (eternity替换g)，再次使用 beta 规约(第二大块代码替换 f)就可以得到 length≤1。

我们必须把参数命名为 length 吗?

> 不需要，我们直接用f和g。只要我们是一贯的，一切都还好。

---

length≤2呢?

```
((lambda (length)
(lambda (l)
(cond
  ((null? l) 0)
  (else (add1 (length (cdr l)))))))
((lambda (length)
(lambda (l)
 (cond
   ((null? l) 0)
   (else (add1 (length (cdr l)))))))
((lambda (length)
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (length (cdr l)))))))
eternity)))
```

> 笔记：使用 beta 规约3次得到 length≤2。从后往前。

更进一步了，但是依然有重复。

> 是的，让我们消除重复。

我们应该从哪里开始?

> 命名一个函数，这个函数以 length 为参数，并且返回一个类似 length 的函数。

起个什么名字好呢?

> “make length”如何?

好的。就这样子，写个 length0。

> 没问题。

```
((lambda (mk-length)
(mk-length eternity))
(lambda (length)
(lambda (l)
(cond
  ((null? l) 0)
  (else (add1 (length (cdr l))))))))
```

下面这是 length≤1

```
((lambda (mk-length)
   (mk-length
    (mk-length eternity)))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l)))))))))
```

> 没错。下面这是length≤2。

```
((lambda (mk-length)
(mk-length
(mk-length
(mk-length eternity))))
(lambda (length)
(lambda (l)
(cond
  ((null? l) 0)
  (else (add1 (length (cdr l))))))))
```

你能写出类似的 length≤3 吗?

> 当然。就是这样。

```
((lambda (mk-length)
(mk-length
(mk-length
(mk-length
 (mk-length eternity)))))
(lambda (length)
(lambda (l)
(cond
  ((null? l) 0)
  (else (add1 (length (cdr l)))))))))
```

递归像是什么样?

> mk-length 就像无穷层的塔一样，应用一个 arbitrary function。

我们必须要一个无穷层的塔吗?

> 当然没必要。每次使用 length, 我们都是对一个有限的个数来使用，但是我们永远不知道该是多少。

我们能猜需要多少吗?

> 当然，但是我们可能猜的数不够大。

我们什么时候知道我们猜的数不够大?

> 当我们应用到了 mk-length 中最内层的 eternity时。

要是当执行 eternity时，我们能够创建 mk-length 来执行呢?

> 那样只能把问题向后推一次，即使那样我们又能做什么呢?

好，那么既然给 mk-length 输入的是什么参数无所谓，那我们一开始把 mk-length 作为参数输入到 mk-length中去。

> 好主意。我们对 eternity 调用 mk-length以及对cdr调用其结果，这样我们就能使塔有更多层。

下面这个还是 length0 吗?

```
((lambda (mk-length)
  (mk-length mk-length))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
               (length (cdr l)))))))))
```

> 还是length0啊。我们甚至可以用 mk-length 替代 length。

> 笔记：就是alpha规约，除了自变量名替换了什么都没变。

```
((lambda (mk-length)
(mk-length mk-length))
(lambda (mk-length)
(lambda (l)
 (cond
   ((null? l) 0)
   (else (add1
           (mk-length (cdr l))))))))
```

我们为什么要那样做?

> 所有的名字都是等效的，但是一些名字比其它的名字更加等效。All names are equal, but some names are more equal than others.

> 注：George Orwell(1903-1950)

的确：只要我们名字持续，一切都好。

> mk-length 是一个比 length 更加通用的名字。如果我们使用 mk-length，那么它总是提示我们 mk-length 的第一个参数是 mk-length。

既然把 mk-length 作为参数传递给了 mk-length，那么我们能用参数创造递归来使用吗?

> 能。当我们执行了一次 mk-length，我们得到 length≤1

```
((lambda (mk-length)
(mk-length mk-length))
(lambda (mk-length)
(lambda (l)
 (cond
   ((null? l) 0)
   (else (add1
           ((mk-length eternity) (cdr l))))))))
```

假设 l 是(apple)问下面执行的结果是什么?

```
(define l '(apple))
(((lambda (mk-length)
    (mk-length mk-length))
  (lambda (mk-length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1
                ((mk-length eternity) (cdr l))))))))
 l)
```

> 这是一个好例子。用纸和笔算算吧。
> 笔记：(mk-length (cdr l))即(mk-length ())即0，于是加1,得到1。

我们能不止一次这样做吗?

> 能，只要不断向 mk-length 传递参数 mk-length就行。我们可以经常这样做。

你把下面这个函数称作什么?

```
((lambda (mk-length)
  (mk-length mk-length))
 (lambda (mk-length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1
                ((mk-length mk-length) (cdr l))))))))
```

> 当然是length啰。

它是怎么工作的?

> 当它要expire(挂掉？)时它通过把 mk-length 传给自己不断添加递归。

还剩下一个问题：它再也不含像函数 length 的部分了。

```
((lambda (mk-length)
  (mk-length mk-length))
 (lambda (mk-length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1
                ((mk-length mk-length) (cdr l))))))))
```

你能解决这个问题吗?

> 我们可以把这个新的应用从 mk-length 中抽取出来，把它称为 length。

为什么?

> 因为它的确创建了函数 length。

那下面这个如何?

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (mk-length mk-length))))
```

> 这个看起来好些。

让我们看看他是否执行无误。

> Okey。

下面这个代码值是多少?

```
(define l '(apple))

(((lambda (mk-length)
    (mk-length mk-length))
  (lambda (mk-length)
    ((lambda (length)
       (lambda (l)
         (cond
           ((null? l) 0)
           (else (add1 (length (cdr l)))))))
     (mk-length mk-length))))
 l)
```

> 应该是 1。

首先我们需要计算

```
((lambda (mk-length)
   (mk-length mk-length))   ;;第一个 mk-length展开，得到后面的一问的回答。
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (mk-length mk-length))))
```

> 没错，因为这个表达式是我们需要对l执行的函数。

所以我们需要求得

```
((lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (mk-length mk-length)))
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
```

```
    (mk-length mk-length))))
```

> 是的。

但是我们就必须求得

```
((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
 ((lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (mk-length mk-length)))
  (lambda (mk-length)
    ((lambda (length)
       (lambda (l)
         (cond
           ((null? l) 0)
           (else (add1 (length (cdr l)))))))
     (mk-length mk-length)))))
```

> 是的。什么时候才是头呢？我们难道不需要求得下面这个表达式的值吗？
>
> ```
> ((lambda (length)
> (lambda (l)
> (cond
>   (( null? l) 0)
>   (else ( add1 ( length ( cdr l)))))))
> ((lambda (length)
> (lambda (l)
>  (cond
>    ((null? l) 0)
>    (else (add1 ( length ( cdr l)))))))
> ((lambda (mk-length)
> ((lambda (length)
>    (lambda (l)
>      (cond
>        ((null? l) 0)
>        (else (add1 (length (cdr l)))))))
>  ( mk-length mk-length)))
> (lambda (mk-length)
> ((lambda (length)
>    (lambda (l)
>      (cond
>        ((null? l) 0)
>        (else (add1 (length (cdr l)))))))
>  (mk-length mk-length))))))
> ```

是的，没有尽头了啊。怎么回事？

> 因为我们持续不断的把 mk-length参数传递给它自己，不停的，不停的...

奇怪吗?

> 因为当我们给 mk-length 传递一个参数，他返回的是一个函数。实际上，它并不介意我们传递它的是什么参数。

但是现在既然我们已经从能够创建 length函数的函数中提取出(mk-length mk-length)来，它就不会返回任何函数了。

> 它不返回函数，我们该怎么办?

把那个对 mk-length 的应用的最后一个正确的版本转成一个函数。

```
((lambda (mk-length)
  (mk-length mk-length))
 (lambda (mk-length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1
                ((mk-length mk-length) (cdr l))))))))
```

> 怎么弄?

下面这里是一个不一样的做法。如果 f 是一个函数的参数，那么(lambda (x) (f x)) 是一个函数的参数吗?

> 是的。

如果`(mk-length mk-length)`返回一个函数的参数，那么

```
(lambda (x)
  ((mk-length mk-length) x))
```

返回一个函数的参数吗?

> 没错。
>
> ```
> (lambda (x)
> ((mk-length mk-length) x))
> ```
>
> 是一个函数。

好的。现在让我们把这个用在 mk-length对自己的应用上。

```
((lambda (mk-length)
(mk-length mk-length))
(lambda (mk-length)
(lambda (l)
 (cond
   ((null? l) 0)
   (else (add1
          ((lambda (x)
           ((mk-length mk-length) x)) (cdr l)))))))))
```

把新的函数移出来，这样我们就可以把 length 移回去。

```
((lambda (mk-length)
(mk-length mk-length))
(lambda (mk-length)
((lambda (length)
 (lambda (l)
   (cond
     ((null? l) 0)
     (else (add1
            (length (cdr l)))))))
(lambda (x)
 ((mk-length mk-length) x)))))
```

这样移动函数有问题吗?

> 没有问题，我们只是反着来做了一个名字替换值罢了。这里我们提取值然后给它一个名字。

我们可以把盒子里看起来像是 length 的函数提取出来然后给它个名字吗?

> 可以啊，它并不依赖于 mk-length。

这是正确的函数吗?

```
((lambda (le)
   ((lambda (mk-length)
      (mk-length mk-length))
    (lambda (mk-length)
      (le (lambda (x)
            ((mk-length mk-length) x))))))
 (lambda (length)
   (lambda (l)
     (cond
      ((null? l) 0)
      (else (add1 (length (cdr l)))))))))
```

> 是的。

我们得到了了什么?

> 我们把原来的函数 mk-length 抽取出来。

让我们把函数中看起来像 length 的函数分离出来。

> 简单：

```
(lambda (le)
((lambda (mk-length)
 (mk-length mk-length))
(lambda (mk-length)
 (le (lambda (x)
       ((mk-length mk-length) x))))))
```

这个函数有名字吗？

> 有，它被称为Y 算子(Y combinator)。

```
(define Y
(lambda (le)
((lambda (f) (f f))
(lambda (f)
  (le (lambda (x) ((f f) x)))))))
```

(define ...)又能用了？

> 能用了，现在我们搞清楚什么是递归了。

你知道为什么 Y 算子正常执行吗？

> 把本章再读一遍你就懂了。

(Y Y) 是什么？

> 谁知道呢，管用就行。

笔记：这里可以参见维基百科的Y算子。讲解很简明很清楚，不像这一章讲的明显效果不好。

你的帽子还合脑袋吗？

> 头大成了这样子，恐怕不合适了吧。

entry(条目) 是由 list 组成的 pair， pair 的第一个 list 是 set 。另外，两个 list 的长度必须是相同的。举出几个 entry(条目) 的例子。

复习，set 就是不含有重复原子的列表.

> 例如
> `((appetizer entree beverage) (pate boeuf vin)`
> 和
> `((appetizer entree beverage) (beer beer beer))`
> 还有
> `((beverage dessert) ((food is) (number on with us)))`

我们如何用一个集合的名字和一个列表的值构建一个 entry(条目)

> `(define new-entry build)`
> 试试用这个函数来构建前边的 entry(条目) 例子。

`(lookup-in-entry name entry)` 的值是多少，其中

name 是 `entree`

entry 是:

```
((appetizer entree beverage)
 (food tastes good))
```

> tastes

那假设 name 是 dessert 呢?

> 这种情况下， lookup-in-entry 什么也不做。

我们如何把这个完成?

> 当在 entry(条目) 中没有找到 name， lookup-in-entry 调用另一个参数

你认为这个额外的函数需要多少个参数?

> 我们认为需要一个，name。为什么?

这里是我们对 `lookup-in-entry` 的定义。

```
(define lookup-in-entry
  (lambda (name entry entry-f)
    (lookup-in-entry-help name
                          (first entry)
                          (second entry)
                          entry-f)))
```

补全函数 `lookup-in-entry-help`

```
(define lookup-in-entry-help
  (lambda (name names values entry-f)
    (cond
      (_____ _____)
      (_____ _____)
      (_____ _____))))
```

```
(define lookup-in-entry-help
(lambda (name names values entry-f)
(cond
((null? names) (entry-f name))
((eq? (car names) name)
 (car values))
(else (lookup-in-entry-help name
                            (cdr names)
                            (cdr values)
                            entry-f)))))
```

> 笔记：entry-f 就是那个额外的函数来专门处理(second entry)为空表的情况。

---

table表格(也称作 environment环境)是entry(条目)为成员的表。空表为()。举出其它的例子。

```
(((appetizer entree beverage) (pate boeuf vin))
((beverage dessert) ((food is) (number one with us))))
```

定义函数 `extend-table`, 该函数输入一个 entry 和一个 table 作为参数，然后把新的 entry 放在旧的 table 表格前建立一个新的 table 表格。

> `(define extend-table cons)`

`(lookup-in-table name table table-f)` 是什么，其中 name 是 `entree`，table 是

```
(((entree dessert) (spaghetti spumoni))
 ((appetizer entree beverage) (food tastes good)))
```

table-f 是 `(lambda (name) ...)`

> 可能是 spaghetti 或者 tastes, 但是 lookup-in-table 是按照表中的 entry 的顺序来查找的。所以是 spaghetti。

写出函数 lookup-in-table
提示：别忘了用一些帮助。

```
(define lookup-in-table
(lambda (name table table-f)
(cond
 ((null? table) (table-f name))
 (else (lookup-in-entry name
                        (car table)
                        (lambda (name)
                          (lookup-in-table name (cdr table) table-
f)))))))
```

你能描述一下，下面这个函数的作用吗?

```
(lambda (name)
  (lookup-in-table name (cdr table) table-f))
```

> 当 name 在第一个 entry 中没有找到时的执行动作。

前面的preface中我们提到 sans serif 字体表示原子。这一点并不要紧。不用注意原子是否是 sans serif 字体了。(p177)

我们为 expression表达式选择了好的表示吗?

> 是的。它们都是 S-expression，所以它们都可以作为函数的数据。

什么样的函数?

> 比如 value。

还记得第六章的 value 吗?

> 对表达式调用 value 返回的是它的自然值。

`(car (quote (a b c)))` 的值是多少?

> 不知道。

```
(cons rep-a
  (cons rep-b
    (cons rep-c
      (quote ()))))
```

的值是多少? 其中 rep-a 是a，rep-b 是b，rep-c 是c

> `(a b c)`

非常棒。那这个值呢?

```
(cons rep-car
      (cons (cons rep-quote
                  (cons
                   (cons rep-a
                         (cons rep-b
                               (cons rep-c
                                     (quote ()))))))
                   (quote ()))
            (quote ())))
```

其中
rep-car 是 `car`
rep-quote 是 `quote`
rep-a 是 `a`，rep-b 是 `b`，rep-c 是 `c`

> 它是表达式 `(car (quote (a b c)))` 的表示。

`(car (quote (a b c)))` 的值是多少?

> a

`(value e)` 的值是多少? 其中e是 `(car (quote (a b c)))`

> a

`(value e)` 的值是多少? 其中e是 `(quote (car (quote (a b c))))`

> `(car (quote (a b c)))`

`(value e)` 的值是多少? 其中e是 `(add1 6)`

> 7

`(value e)` 的值是多少? 其中e是 `(quote nothing)`

> nothing

`(value e)` 的值是多少? 其中e是 `nothing`

> nothing没有value

`(value e)` 是什么，其中e是

```
((lambda (nothing)
   (cons nothing (quote ())))
 (quote (from nothing comes something)))
```

> `((from nothing comes something))`

`(value e)` 是什么，其中e是

```
((lambda (nothing)
   (cond
     (nothing (quote something))
     (else (quote nothing))))
 #t)
```

> `something`。

---

e的类型是什么，其中e是6

> *const

---

e的类型是什么，其中e是#f

> *const

(value e)的值是什么，其中e是#f

> #f

---

e的类型是什么，其中e是cons

> *const

`(value e)` 的值是什么，其中e是 `car`

> `(primitive car)`

---

e的类型是什么，其中e是 `(quote nothing)`

> *quote

---

e的类型是什么，其中e是 `nothing`

> *identifier

---

e的类型是什么，其中e是 `(lambda (x y) (cons x y))`

> *lambda

---

e的类型是什么，其中e是

```
((lambda (nothing)
   (cond
     (nothing (quote something))
     (else (quote nothing))))
 #t)
```

> *application

---

e的类型是什么，其中e是

```
(cond
  (nothing (quote something))
  (else (quote nothing)))
```

> *cond

你认为有多少种类型?

> 我们找到六种:
> *const
> *quote
> *identifier
> *lambda
> *cond
> *application

你认为我们如何表示类型?

> 我们选择函数，它被称为“action”执行动作。

如果action都是那些应用到特定类型的表达式时会做该做的事情的函数，那么 value 应该是什么?

> 你猜到了。它必须找出传递给它的类型然后使用相应的执行动作。

记得第八章的 atom-to-function 吗?

> 当我们重写 value 时，我们需要 atom-to-function。

下面这个函数能够对每一个 S-expression 执行正确的过程。

```
(define expression-to-action
  (lambda (e)
    (cond
      ((atom? e) (atom-to-action e))
      (else (list-to-action e)))))
```

写出函数 atom-to-action。
注：这里不考虑病态的(ill-formed) S-expression 表达式，例如(quote a b), (), (lambda (#t) #t), (lambda (5) 5), (lambda (car) car), (lambda a a), (cond (3 c) (else b) (6 a)), (1 2)等。它们可以在传递给value之前用一个合适的函数来侦测。

```
(define atom-to-action
  (lambda (e)
    (cond
      ((number? e) *const)
      ((eq? e #t) *const)
```

```
      ((eq? e #f) *const)
      ((eq? e (quote cons)) *const)
      ((eq? e (quote car)) *const)
      ((eq? e (quote cdr)) *const)
      ((eq? e (quote null?)) *const)
      ((eq? e (quote eq?)) *const)
      ((eq? e (quote atom?)) *const)
      ((eq? e (quote zero?)) *const)
      ((eq? e (quote add1)) *const)
      ((eq? e (quote sub1)) *const)
      ((eq? e (quote number?)) *const)
      (else *identifier))))
```

现在定义函数 `list-to-action`

```
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
       (cond
         ((eq? (car e) (quote quote)) *quote)
         ((eq? (car e) (quote lambda)) *lambda)
         ((eq? (car e) (quote cond)) *cond)
         (else *application)))
      (else *application))))
```

假设 expression-to-action 执行正常，我们就可以定义 value 和 meaning

```
(define value
  (lambda (e)
    (meaning e (quote ()))))

(define meaning
  (lambda (e table)
    ((expression-to-action e) e table)))
```

value 中的`(quote ())`是什么?

> 它是空表格。函数 value 以及它用的函数叫做解释器。
>
> 注：数 value 函数是个 Scheme(或者Lisp)中eval的近似。

Actions do speak louder than words.

---

action需要多少个参数?

> 两个，表达式 e和一个 table 表格

下面这个是constant的action执行动作。

```
(define *const
  (lambda (e table)
    (cond
      ((number? e) e)
      ((eq? e #t) #t)
      ((eq? e #f) #f)
      (else (build (quote primitive) e)))))
```

对吗?

> 对，对于数，它直接返回表达式，对于#t，它返回true，对于#f它返回false
> 其它的constant类型的原子则代表 primitive。

下面是 *quote 的 action 执行动作

```
(define *quote
  (lambda (e table)
    (text-of e)))
```

定义辅助函数 text-of

> `(define text-of second)`

我们用过table吗?

> 还没呢，不过一会就用到 table 了。

我们为甚么要用到 table?

> 存储 identifier 的 value。

假设 table 存储着 identifier 的 value，写出 *identifier 的执行动作。

```
(define *identifier
  (lambda (e table)
    (lookup-in-table e table initial-table)))
```

下面是 initial-table

```
(define initial-table
  (lambda (name)
    (car (quote ()))))
```

什么时候用它?

> 我们希望永远都不要用到。为什么?

> 笔记：lookup-in-table 的第三个参数是用来处理某个name对应是空表的情况，这时table是有问题的，table本来是给出identifier标识符的对应值的。没有给出identifier的值，于是直接抛出一个错误了，(car (quote))是违反primitive 的car操作定义的。

`(lambda (x) x)` 这样的表达式的 value 是什么?

> 我们不知道，但是我们知道它是一个 non-primitive 函数。

`non-primitive` 函数与 `primitive` 有什么不同?

> 我们知道 primitive 做了什么； non-primitive 是由它们的参数和函数体构成。

所以当我们用一个 non-primitive, 我们需要存储它的参数和函数体。

> 至少，所幸的是，这只是一个 lambda 表达式的 cdr。

我们还需要存储什么?

> 我们还需要把 table 表格存储了，方便后边使用。

我们如何表示它?

> 当然是存储在 list 表中。

下面是 *lambda 的 action 执行动作。

```
(define *lambda
  (lambda (e table)
    (build (quote non-primitive)
      (cons table (cdr e)))))
```

`(meaning e table)` 是什么，
其中e是`(lambda (x) (cons x y))`，
table 是`(((y z) ((8) 9)))`。

```
(non-primitive ( (((y z) ((8) 9)))   (x)   (cons x y) ))
                      ~~~~~~~~~~~    ````   ^^^^^^^^^^
                        table        formals    body
```

定义辅助函数来提取三个元素的list(即，table，formals，body)。写出 table-of formals-of 和 body-of

> `(define table-of first)`

> `(define formals-of second)`

> `(define body-of third)`

请您用自己的话描述(cond ...)

> cond行是特殊的格式。它一行行往下查找。如果问题部分为假，它往下查找。否则运行为真的部分。如果遇到了else行，cond认为它为真，直接执行else部分。

下面这个 evcon 就是做我们刚才说的那些事情。

```
(define evcon
  (lambda (lines table)
    (cond
     ((else? (question-of (car lines)))
      (meaning (answer-of (car lines))
               table))
     ((meaning (question-of (car lines))
               table)
      (meaning (answer-of (car lines))
               table))
     (else (evcon (cdr lines) table)))))
```

定义else? question-off answer-of

```
(define else?
  (lambda (x)
    (cond
      ((atom? x) (eq? x (quote else)))
      (else #f))))

(define question-of first)

(define answer-of second)
```

我们违反了第一戒律了吗?

> 是的，我们么有查询(null? lines), 所以 cond 中的问题得至少有一个为真。

现在用函数 evcon 来写一个 *cond 执行动作。

```
(define *cond
  (lambda (e table)
    (evcon (cond-lines-of e) table)))

(define cond-lines-of cdr)
```

辅助函数有用吗?

> 有用，它们使可读性提高了。你已经知道了。

你现在理解了 *cond 吗?

> 可能还没有。

你如何熟悉*cond?

> 最好的方式是来个例子。一个好例子：
> `(*cond e table)`, 其中e是`(cond (coffee klastsch) (else party))`, table 是
> `(((coffee) (#t)) ((klastsch party) (5 (6))))`

见过 table 是怎么用的吗?

> 见过啊， *lambda 和 *identifier 都用过。

但是 identifier 如何查找 table?

> 使用为一个我们还没有定义的 action： *application。

---

application 是如何表示的?

> 一个 application 就是一个 list，它的 car 是一个值是函数的 expression 表达式。

---

application 与(and ...) (or ...) (cond...) 等有什么不同之处。

> 一个 application 必须确定它的所有参数。

在我们执行一个函数之前，我们必须知道所有参数吗?

> 必须。

写一个函数 evlis ，它的参数是由一个用来表示参数的list和一个table构成，返回每一个参数的含义。

```
(define evlis
  (lambda (args table)
    (cond
      ((null? args) (quote ()))
      (else
       (cons (meaning (car args) table)
             (evlis (cdr args) table))))))
```

在确定 application 的含义之前我们还需要什么?

> 我们需要知道它的 function-of 的含义。

下面是 *application

```
(define *application
  (lambda (e table)
  (apply
    (meaning (function-of e) table)
    (evlis (arguments-of e) table))))
```

对吗?

> 当然对了。我们只需要正确定义 apply, function-of, arguments-of。

写出 function-of 和 arguments-of

```
(define function-of car)

(define arguments-of cdr)
```

有多少个类型的函数?

> 两种：primitive 和 non-primitive。

这两种函数表示什么?

> `(primitive primitive-name)` 和 `(non-primitive (table formals body))`, 表`(table formals body)`称为 closure record。

写出 primitive? 和 non-primitive?

```
(define primitive?
  (lambda (l)
    (eq? (first l) (quote primitive))))

(define non-primitive?
  (lambda (l)
    (eq? (first l) (quote non-primitive))))
```

现在我们可以写出函数 apply 了。

```
(define apply
  (lambda (fun vals)
    (cond
     ((primitive? fun)
      (apply-primitive (second fun) vals))
     ((non-primitive? fun)
      (apply-closure (second fun) vals)))))
```

> 注：如果 fun 既不计算 primitive 也不计算 non-primitive， 例如表达式((lambda (x) (x 5)) 3), 那就没有答案。函数 apply 是Scheme (Lisp) 中apply的近似。

填空

```
(define apply-primitive
  (lambda (name vals)
    (cond
      ((eq? name _____)
       (cons (first vals) (second vals)))
      ((eq? name (quote car))
       (car (first vals)))
      ((eq? name (quote cdr))
       (_____ (first vals)))
      ((eq? name (quote null?))
       (null? (first vals)))
      ((eq? name (quote eq?))
```

```
        (_____ (first vals) _____)))
      ((eq? name (quote atom?))
        (_____ (first vals)))
      ((eq? name (quote zero?))
        (zero? (first vals)))
      ((eq? name (quote add1))
        (add1 (first vals)))
      ((eq? name (quote sub1))
        (sub1 (first vals)))
      ((eq? name (quote number?))
        (number? (first vals))))))
```

> 1.(quote cons)
> 2.cdr
> 3.eq?
> 4.(second vals)
> 5.:atom?

```
(define :atom?
  (lambda (x)
    (cond
      ((atom? x) #t)
      ((null? x) #t)
      ((eq? (car x) (quote primitive)) #t)
      (else #f))))
```

> 注：函数 apply-primitive 能够查找 application 的 cdr 。The function apply·primitive could check for applications of cdr to the empty list or sub1 to 0, etc.

---

apply-closure 是唯一没有定义的函数了?

> 只剩它了， apply-closure 必须展开 table。

我们如何得到`(f a b)`的值，其中f是`(lambda (x y) (cons x y))`, a是1，b是(2)

> 这个很是麻烦。不过我们知道怎样做才能得到(cons x y)的含义。

我们为什么做这个?

> 这个不需要 apply-closure。

你能概括最后两个步骤吗?

> 对一个 value 的 list 应用一个 non-primitive(一个closure)同下面这个做法是一致的：对closure的body和它的展开的table查找结果。其中table展开条目entry格式为(formals values)。在条目 entry 中，formals 是 closure 的formals，values是 evlis的的结果。

所有这些你都遵守了吗?

> 如果没有，下面是 apply-closure 的定义。

```
(define apply-closure
  (lambda (closure vals)
    (meaning (body-of closure)
             (extend-table
              (new-entry (formals-of closure) vals)
              (table-of closure)))))
```

这是一个复杂的函数，得要一个例子。

> 以下内容假设：
> closure是

```
((((u v w)
   (1 2 3))
  ((x y z)
   (4 5 6)))
  (x y)
   (cons z x))
```

> vals 是 `((a b c) (d e f))`

---

meaning 的新参数是什么?

> meaning 新的参数e将是(cons z x)，table将是
> `(((x y) ((a b c) (d e f))) ((u v w) (1 2 3) (4 5 6)))`

`(cons z x)` 的含义是什么,其中z是6，x是(a b c)

> 就是(meaning e table), 其中e是(cons z x),table是
> `(((x y) ((a b c) (d e f))) ((u v w) (1 2 3)) ((x y z) (4 5 6)))`

让我们找出所有的参数的含义， (evlis args table), 其中args是
`(((x y) ((a b c) (d e f))) ((u v w) (1 2 3)) ((x y z) (4 5 6)))`

> 我们必须先确定(meaning e table), 其中e是z;(meaning e table)其中e是x。

`(meaning e table)` 是什么，其中e是 `z`

> 6， 用的 *identifier。

`(meaning e table)` 是什么，其中e是x

> `(a b c)`, 用的 *identifier。

所以 evlis 的结果是什么?

> `(6 (a b c))`， 因为evlis返回一个list的含义。

`(meaning e table)` 是什么，e是 `cons`

> `(primitive cons)`, 用的 *const。

我们现在可以 `(apply fun vals)` 了，其中fun是 `(primitive cons)`, vals是 `(6 (a b c))`, 我们应该用哪个path路径?

> apply-primitive path路径。

`(apply-primitive name vals)` 选择cond的哪个分支? 其中name是 `cons`， vals是 `(6 (a b c))`

> 第三个:

```
1 ((eq? name (quote cons))
2  (cons (first vals) (second vals)))。
```

我们完成了吗?

> 完成了。

但是(define ...)呢?

> 没有用到define，因为递归可以用Y combinator算子获得。

不需要定义(define ...)吗?

> 需要，不过需要看本书的后续 Seasoned Schemer。

意思就是如果我们用Y combinator 转换一下，就可以在解释器上运行解释器?

> 是的，but don't bother。

---

value有什么不同之处?

> 它查找 expression 表达式的表示。

---

will-stop?能查找 expression 的表示吗?

> 这个很有帮助。

有用吗?

> No, don't bother——we can play the same game again. 我们可以定义一个像 last-try? 函数那样可以展现不能定义will-stop?函数的函数。

---

else

> 好了，该是宴会的时候了。

结语

你已经到达了中场休息。你的选择呢？你可以马上接下册 The Seasoned Schemer， 或者你可以读一些我们下面提到的书。所有这些书都是经典，有一些比较老；然而它们都能经受的住时间的检验，值得你们去选择。有一些与数学，逻辑等等都无关，有些必须有数学基础，但仅仅是讲些有趣的故事，另外一些值得去发现。无需怀疑：这些书不是用来当作续集来读，就是给你娱乐休闲的。在 The Seasoned Schemer 的结尾你可以找着一些 Scheme 和 Common Lisp 的书。不要感到到必须读续集，而是该花点时间下面的这些书。当你放松点了，消化了强加给你的卡路里，你再继续读下册。Enjoy!

Abbott, Edwin A. Flatland . Dover Publications, Inc., New York, 1952. (Original publication: Seeley and Co. , Ltd . , London, 1884.)

Carroll, Lewis. The Annotated Alice: Alice 's Adventures in Wonderland and Through the Looking Glass. Clarkson M. Potter, Inc., New York, 1960. Introduction and notes by Martin Gardner. Original publications under different titles: Alice 's Adventures Under Ground and Through the Looking Glass and What Alice Found There, Macmillan and Company, London 1865 and 1872, respectively.

Halmos, Paul R. Naive Set Theory. Litton Educational Publishers, New York, 1960.

Hein, Piet. Grooks. The MIT Press, Cambridge, Massachusetts, 1960.

Hofstadter, Douglas R. Godel, Escher, Bach: an Eternal Golden Bmid. Basic Books, Inc . , New York, 1979.

Nagel, Ernest and James R. Newman . Godel 's Proof. New York University Press, New York, 1958.

P6lya, Gyorgy. How to Solve It. Doubleday and Co. , New York, 1957.

Smullyan, Raymond. To Mock a Mockingbird And Other Logic Puzzles Including an Amazing Adventure in Combinatory Logic. Alfred A. Knopf, Inc., New York, 1985.

Suppes, Patrick. Introduction to Logic. Van Nostrand Co. , Princeton , New Jersey, 1957.