欢迎来到Lisp的世界

本章的目标是尽快让你编程. 在本章结束的时候,你会掌握足够的Common Lisp的知识,可以开始写程序了.

0.1 范式(form)

你可以通过使用Lisp而学习它,这是千真万确的,因为Lisp是交互式语言. 任何Lisp系统都包含一个叫做顶层(toplevel)的交互式前端. 你在顶层中输入Lisp表达式,系统打印它们的值.

Lisp通常打印一个提示符表示它正在等待你的输入. 许多Common Lisp的实现用> 作为顶层提示符. 我们在这儿也用此符号.

最简单的Lisp表达式之一是一个整数. 如果我们在提示符后面输入1,

```
> 1
1
>
```

系统会打印它的值,跟着另一个提示符,表示它在等待更多的输入.

在这种情况下,打印出来的值和我们输入的一样. 象1这样的数叫做自身求值的. 当我们输入一个需要做些求值工作的表达式时,事情变得有趣起来. 例如,如果想把两个数加起来,我们输入:

```
> (+ 2 3)
5
```

在表达式(+ 2 3)中,+叫做操作符,数2和3叫做变元.

在日常生活中我们会把此表达式写为2 + 3,但在Lisp中我们把+写在最前面,后面跟着变元,整个表达式被一对括号围住:(+ 2 3). 因为操作符在前,这叫做前缀表示法. 一开始这样写表达式有点怪,但事实上这种表示法是Lisp最好的东西之一.

比如,我们想把三个数加起来,用通常的表示法我们要写+两次:

```
2 + 3 + 4
```

而在Lisp中我们仅需增加一个变元:

```
> (+ 2 3 4)
```

通常我们用+,它必须有两个变元:一个在左边,一个在右边. 前缀表示法的弹性意味着,在Lisp中,+可以接受任意数目的变元,包括零个:

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 4)
9
> (+ 2 3 4 5)
14
```

因为操作符可以接受不同数目的变元,我们需要用括号指示表达式的开始和结束.

表达式可以嵌套. 即表达式中的变元本身可能是个复杂的表达式:

```
> (/ (- 7 1) (- 4 2))
3
```

用自然语言来说,七减一的结果被四减二的结果除.

另一个Lisp表示法的漂亮之处是:它无所不包. 所有的Lisp表达式要么是象1这样原子(atom),要么是放在括号中由零个或多个表达式组成的表(list). 这些是合法的Lisp表达式:

```
2   (+ 2 3)   (+ 2 3 4)   (/ (- 7 1) (- 4 2))
```

正如我们将要看到的,所有的Lisp代码都采取这种形式. 象C这样的语言有着更复杂的语法:算术表达式用中缀表示法;函数调用类似前缀表示法,自变量用逗号隔开;表达式用分号隔开;而代码块用花括号分隔. 在Lisp中我们用单一的记号表达所有这些概念.

0.2 求值

在上一节中,我们在顶层里输入表达式,Lisp显示它们的值. 在这节里我们仔细观察一下表达式是如何求值的.

在Lisp中,+是一个函数,形如(+ 2 3)的表达式是函数调用. 当Lisp对函数调用求值时,它做这样两步:

1. 首先变元从左至右被求值. 在此例中,每个变元求值到自身,所以变元的值分别是2和3.

2. 变元的值传给以操作符命名的函数. 在此例中,即+函数,它返回5.

如果任何变元本身是函数调用,它们按上述规则求值. 这是对(/ (- 7 1) (- 4 2))求值时发生的情况:

1. Lisp计算(- 7 1): 7求值为7,1求值为1. 它们被传给函数-,它返回6.

2. Lisp计算(- 4 2): 4求值为4,2求值为2. 它们被传给函数-,它返回2.

3. 6和2的值传给函数/,它返回3.

并不是所有的Lisp操作符都是函数,但大多数都是. 而函数总是按这种方式求值的. 变元从左至右被求值,它们的值被传给函数,函数返回整个表达式的值.这叫做Common Lisp的求值规则.

一个不遵守上述规则的操作符是quote. quote是一个特殊操作符,这意味着它有自己独特的求值规则. 这个规则是:什么也不做.quote接受一个变元,并且一字不差地返回它:

```
> (quote (+ 3 5))
(+ 3 5)
```

为了方便,Common Lisp定义'作为quote的简记法. 通过在任何表达式前面加上' 你能获得与调用quote同样的效果:

```
> '(+ 3 5)
(+ 3 5)
```

用简记法比用quote普遍得多.

Lisp提供quote作为一种保护表达式以防被求值的手段. 下一节会解释这种保护是多么有用.

从麻烦中解脱出来

如果你输入了一些Lisp不能理解的东西,它会打印一条出错信息并把你带到一个叫中断循环(break loop)的顶层中去.中断循环给了有经验的程序员弄清出错原因的机会,不过一开始你唯一需要知道的事是如何从中断循环中出来. 如何返回顶层取决于你用的Lisp环境. 在这个假设的环境里,用:abort出来:

```
> (/ 1 0)
Error: Division by zero.
       Options: :abort, :backtrace
>> :abort
>
```

附录A展示了如何调试Lisp程序,以及一些最常见错误的例子.

0.3 数据

Lisp提供所有我们能在其它语言中找得到的数据类型,和一些我们找不到的.一种我们早已使用的数据类型是整数,它写为一列数字:256. 另一种和其它语言一样有的是字符串,它表示为一列用双引号括起来的字符:"ora et labora".整数和字符串都求值到自身.

另两种我们不常在其它语言中发现的是*符号*和*表*. 符号是单词. 通常它们被转换成大写,不管你如何输入:

```
> 'Artichoke
ARTICHOKE
```

符号(通常)不求值为自身,因此如果你想引用一个符号,请象上面那样用'引用它.

表表示为被括号包围的零个或多个元素. 元素可以是任何类型,包括表. 你必须引用表,否则Lisp会以为它是函数调用:

```
> '(my 3 "Sons")
(MY 3 "Sons")
> '(the list (a b c) has 3 elements)
(THE LIST (A B C) HAS 3 ELEMENTS)
```

请注意一个引号保护整个表达式,包括里面的表达式.

你可以调用list来构造表. 因为list是一个函数,它的变元被求值. 这是+调用在list调用里的例子:

```
> (list 'my (+ 2 1) "Sons")
(MY 3 "Sons")
```

现在我们处于欣赏Lisp最非同寻常特征之一的位置上. Lisp程序表达为表. 如果变元的机动性和优雅性没能说服你Lisp记号是一种有价值的工具,这点应该能使你信服. 这意味着Lisp程序可以生成Lisp代码. Lisp程序员能(而且经常)为自己编写能写程序的程序.

我们到第10章才考虑这种程序,但即使在现阶段理解表达式和表的关系也是很重要的,而不是被它们弄糊涂. 这就是为何我们使用quote. 如果一个表被引用了, 求值返回它本身; 如果没有被引用,它被认为是代码,求值返回它的值:

```
> (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)
```

此处第一个变元被引用了,所以生成了一个表. 第二个变元没有被引用,视之为函数调用,得出一个数字.

在Common Lisp中有两种方法表示空表. 你可用一对不包含任何东西的括号来表示空表,或用符号nil来表示它. 你用哪种方法表示空表都没有关系,不过它会被显示成nil:

```
> ()
NIL
> nil
NIL
```

你不必引用nil(虽然这也没什么害处)因为nil求值到自身.

0.4 表的操作

函数cons构造表. 如果它的第二个变元是表,它返回一个新表,新表的第一个元素就是第一个变元:

```
> (cons 'a '(b c d))
(A B C D)
```

我们可以通过把新元素cons到空表来构造新表. 我们在上一节见到的list函数只不过是一个把几样东西cons到nil上去的方便办法:

```
> (cons 'a (cons 'b nil))
(A B)
> (list 'a 'b)
(A B)
```

基本的提取表中元素的函数是car和cdr.[1] 表的car就是它的第一个元素,而cdr是第一个元素后面的所有东西:

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

你能用car和cdr的组合来取得表中任何元素. 如果你想取第三个元素,可以这样:

```
> (car (cdr (cdr '(a b c d))))
C
```

但是,你可以用third更容易地做同样的事:

```
> (third '(a b c d))
C
```

0.5 真值

符号t是Common Lisp中表示真的缺省值. 就象nil,t求值到自身. 函数listp返回真如果它的变元是一个表:

```
> (listp '(a b c))
T
```

一个函数叫做断言如果它的返回值被解释成真或假. Common Lisp的断言的名字通常以p结尾.

假在Common Lisp中用nil(空表)来表示. 如果我们传给listp的变元不是表,它返回nil:

```
> (listp 27)
NIL
```

因为nil扮演两个角色,函数null返回真如果它的变元是空表:

```
> (null nil)
T
```

[1] car和cdr的名字来源于表在第一个Lisp实现中的内部表示. car表示“contents of the address part of the registe”而cdr表示“contents of the decrement part of the register.”

而函数not返回真如果它的变元是假:

```
> (not nil)
T
```

它们完全做的是同样的事情.

要if是Common Lisp中最简单的条件语句. 它一般接受三个变元:一个测试表达式, 一个then表达式和一个else表达式. 测试表达式被求值. 如果它返回真,则then 表达式被求值并返回结果. 如果它返回假,则else表达式被求值并返回它的结果:

```
> (if (listp '(a b c))
      (+ 1 2)
      (+ 5 6))
3
> (if (listp 27)
      (+ 1 2)
      (+ 5 6))
11
```

就象quote,if是特殊操作符. 它不能用函数来实现,因为函数调用的变元总是要求值的,而if的特点是只有最后两个变元中的一个被求值.

if的最后一个变元是可选的. 如果你省略它,它缺省为nil:

```
> (if (listp 27)
      (+ 2 3))
NIL
```

虽然t是真的缺省表示,任何不是nil的东西在逻辑上下文中被认为是真:

```
> (if 27 1 2)
1
```

逻辑操作符and和or就象条件语句. 两者都接受任意数目的变元,但只求值能够确定返回值的数目的变元. 如果所有的变元都是真(不是nil),那么and返回最后变元的值:

```
> (and t (+ 1 2))
3
```

但如果其中一个变元是假,所有它后面的变元都不求值了. or也类似,只要它碰到一个是真的变元就继续求值了.

这两个操作符是宏. 就象特殊操作符,宏可以规避通常的求值规则. 第10章解释如何编写你自己的宏.

0.6 函数

你可以用defun来定义新的函数. 它通常接受三个以上变元:一个名字,一列参数, 和组成函数体的一个或多个表达式. 这是我们定义third的一种可能:

```
> (defun our-third (x)
    (car (cdr (cdr x))))
OUR-THIRD
```

第一个变元表示函数名将是our-third. 第二个变元,表(x),说明函数将接受一个变元:x. 象这样用作占位符的符号叫做*变量*. 当变量代表传给函数的变元, 就象x所做的,它又叫做参数.

定义的其余部分,(car (cdr (cdr x))),即通常所说的函数体. 它告诉Lisp,为了计算函数的返回值,它该做些什么. 所以,对我们给出的作为变元的任何x,调用our-third会返回(car (cdr (cdr x))):

```
> (our-third '(a b c d))
C
```

既然我们看到了变量,就更容易理解什么是符号了. 他们是变量名,是一种有自己名称的对象. 这就是为什么符号要象表一样必须被引用. 表必须被引用是因为不如此的话,它就会被当作代码;符号必须被引用是因为不如此的话,它就会被当作变量.

你可以把函数定义想象成某个Lisp表达式的一般形式. 下面的表达式测试1和4之和是否大于3:

```
> (> (+ 1 4) 3)
T
```

通过把这些特殊数字换成变量,我们可以写一个函数测试任何两个数之和是否大于第三个:

```
> (defun sum-greater (x y z)
    (> (+ x y) z))
SUM-GREATER
> (sum-greater 1 4 3)
T
```

Lisp对程序,过程或函数不加区别. 函数做了所有的事情(事实上构成了语言本身的大部分). 你可以认为你的函数中的一个是主函数,但通常你能在顶层里调用任何一个函数. 这意味着,当你写程序的时候,你能一小段一小段地测试它们.

0.7 递归

我们在上一节中定义的函数还调用了其它函数为自己服务. 比如sum-greater调用了+和>. 函数可以调用任何函数,包括它本身.

自己调用自己的函数是递归的. Common Lisp函数member测试某样东西是否是一个表的元素. 这是定义成递归函数的简化版本:

```
(defun our-member (obj lst)
  (if (null lst)
      nil
```

```
    (if (eql (car lst) obj)
        lst
        (our-member obj (cdr lst)))))
```

断言eql测试它的两个变元是否相同;除此之外,定义中所有东西我们以前都见过. 这是它的运行情况:

```
> (our-member 'b '(a b c))
(B C)
> (our-member 'z '(a b c))
NIL
```

our-member的定义符合下面的自然语言描述. 为了测试一个对象obj是否是表lst 的成员,我们

1. 首先检查lst是否为空. 如果是空的,显然obj不是它的成员,我们做完了.

2. 否则如果obj是lst的第一个元素,它就是成员

3. 否则只有当obj是lst其余部分的成员时,它才是lst的成员.

当你设法理解一个递归函数是如何工作的时候,把它翻译成这样的描述会有帮助的.

许多人一开始觉得递归函数很不好理解. 很多困难来自对函数的一个错误的比喻. 有一种趋势把函数想象成某种形式的机器. 原料就象参数一样到来;一些工作被转包给其它函数;最后完成的产品被组装好运出去,就象返回一个值. 如果我们用这种比喻,递归就自相矛盾了. 机器怎么能把活儿转包给自己? 它已经忙着干活了.

一个更好的比喻是把函数想象成经历的过程. 在过程中递归是很自然的事情. 我们经常在日常生活中看到递归过程. 比如,假设一个历史学家对欧洲历史上的人口变化感兴趣. 检查文献的过程可能是这样的:

1. 取得一份文献

2. 查找有关人口变化的信息

3. 如果该文献提到其它可能有用的文献,检查它们

这个例子简单得足以理解,但它是递归的,因为第三步可以伴有一个或多个同样的过程.

因此不要把our-member想象为测试某样东西是否在表中的机器,而是把它理解为决定某样东西是否在表中的规则. 如果我们以这种见解看待函数,递归的悖论就不存在了.[2]

[2]理解递归有困难的读者可以参考以下文献中的任何一种:

Touretzky, David S. Common Lisp: A Gentle Introduction to Symbolic Computation. Benjamin/Cummings, Redwood City (CA), 1990, Chapter 8.

Friedman, Daniel P., and Matthias Felleisen. The Little Lisper. MIT Press, Cambridge, 1987.

0.8 阅读Lisp

上一节我们定义的our-member以五个括号结尾. 更加复杂的函数定义可能以七八个括号结尾. 初学Lisp的人看到这么多括号会感到气馁. 这叫人如何去读这样的代码? 更不用说写了. 我怎么分辨括号之间的匹配?

回答是,你不需要做这些. Lisp程序员通过缩进,而不是括号来读写程序. 当他们写代码的时候,他们让文本编辑器显示哪个括号匹配哪个. 任何一个优秀的编辑器,特别是Lisp系统附带的,应该能做到括号匹配. 在这样的编辑器中,当你输入了一个括号,它会指示和它匹配的那个括号. 如果你的编辑器不做括号匹配,现在就停下来,研究一个如何使它做这件事,因为没有这个功能,事实上不可能写Lisp 程序的.

在vi中,你可以用:set sm来打开括号匹配. 在emacs中,M-x lisp-mode是获得该功能的好办法.

有了好的编辑器,当你写程序的时候,括号匹配不再是个问题. 而且因为Lisp缩进有通用的惯例,你阅读Lisp代码也不是个问题了. 因为人人都用相同的习惯,你可以通过缩进阅读代码,忽略括号.

不管多么有经验的Lisp黑客,会发现our-member的定义很难读懂,如果它写成这个样子:

```
(defun our-member (obj lst) (if (null lst) nil (if
(eql (car lst) obj) lst (our-member obj (cdr lst)))))
```

如果代码适当地缩进,他就没有困难了. 你可以忽略大部分的括号而读懂它:

```
defun our-member (obj lst)
  if null lst
     nil
     if eql (car lst) obj
        lst
        our-member obj (cdr lst)
```

事实上,当你在纸上写Lisp代码的时候,这就是一个可行的办法. 以后你输入的时候,可以充分利用编辑器的匹配括号的功能.

0.9 输入和输出

到目前为止,我们一直在利用顶层暗中使用i/o. 对实际的交互式的程序,这可能还不够. 在这一节中,我们看一些输入输出函数.

Common Lisp中最一般的输出函数是format. 它接受两个以上变元:第一个表示输出到哪儿,第二个是字符串模板,剩下的变元通常是对象,它们的打印表示(printed representation)将被插入到模板中去. 这是个典型的例子:

```
> (format t "~A plus ~A equals ~A.~%" 2 3 (+ 2 3))
2 plus 3 equals 5.
NIL
```

注意两样东西打印在这儿. 第一行是format打印的. 第二行是format调用的返回值,就象通常一样由顶层打印. 通常象format这样的函数不会直接在

顶层,而是在程序内部被调用,因此返回值就不会被看见. format的第一个变元t表示输出将被送到缺省的地方去. 通常这会是顶层. 第二个变元是充当输出模板的字符串. 在它里面,每个 A*表示一个将被填充的位置, 而 %表示新行符. 这些位置依次被后面的变元的值填充.

标准的输入函数是read. 当没有变元时,它从缺省的地方---通常是顶层---读入. 下面这个函数提示用户输入,然后返回任何输入的东西:

```
(defun askem (string)
  (format t "~A" string)
  (read))
```

它运行如下:

```
> (askem "How old are you? ")
How old are you? 29
29
```

请记住read会永远等在那儿直到你输入什么东西并(通常要)敲入回车. 因此调用read而不打印明确的提示信息是不明智的,否则你的程序会给人以已经死掉的印象,但实际上它在等待输入.

第二个要了解read的是它非常强大:它是一个完整的Lisp语法分析器. 它并不是读入字符再把它们当作字符串返回. 它分析所读到的东西,并返回所产生的Lisp 对象. 在上例中, 它返回一个数.

askem虽然很短,但它展示了一些我们以前在函数定义中没有看到的内容. 它的函数体包含多个表达式. 函数体可以包含任意多个表达式,当函数被调用时,它们依次被求值,函数会返回最后一个表达式的值.

在以前的章节中,我们坚持所谓的‘‘纯粹’’的Lisp---即没有副作用的Lisp. 副作用是指作为表达式求值的后果改变了外部世界的状态. 当我们对一个纯粹的Lisp 表达式,例如(+ 1 2)求值,没有出现副作用;它仅返回一个值. 但当我们调用format,它不仅返回值,还打印了一些东西. 这是一种副作用.

如果我们要写没有副作用的代码,那么定义有多个表达式的函数体就没有什么意义. 最后一个表达式的值作为函数的返回值被返回了,但前面的表达式的值都被扔掉了. 如果这些表达式没有副作用,你就不知道为什么Lisp要费劲去计算它们.

0.10 变量

let是Common Lisp里最常用的操作符之一,它让你引入新的局部变量:

```
> (let ((x 1) (y 2))
    (+ x y))
3
```

一个let表达式有两部分. 第一部分是一列创造新变量的指令,每个形如*(变量表达式)*. 每个变量会被赋予相应的表达式的值. 在上例中,我们创造了两个变量x 和y,它们分别被赋予初值1和2. 这些变量只在let的体内有效.

变量和值的列表的后面是一组表达式,它们将被依次求值. 在此例中,只有一个表达式:对+的调用. 最后一个表达式的值作为let的值被返回. 下面是一个使用let 的更具选择性的askem的版本:

```
(defun ask-number ()
  (format t "Please enter a number. ")
  (let ((val (read)))
    (if (numberp val)
        val
        (ask-number))))
```

此函数造了变量val来存放read返回的对象. 因为它有此对象的名称,它可以在作出是否要返回对象之前察看一下你的输入值. 你可能已经猜到,numberp是测试它的自变量是否是数字的断言.

如果用户输入的不是数字,ask-number调用它自己. 结果产生了一个坚持要得到一个数的函数:

```
> (ask-number)
Please enter a number. a
Please enter a number. (ho hum)
Please enter a number. 52
52
```

象目前我们看到的变量都叫做局部变量. 它们只在特定的环境中是有效的. 另外有一类叫做全局变量的变量,它们在任何地方都是可见的.[3]

通过传给defparameter一个符号和一个值,你可以构造全局变量:

```
> (defparameter *glob* 99)
*GLOB*
```

这样的变量可以在任何地方存取,除非在一个表达式中,定义了一个相同名字的局部变量. 为了避免这种情况的出现,习惯上全局变量的名字以星号开始和结束. 我们刚才定义的变量可读作''星--glob--星''.

你还可以用defconstant定义全局常数:

```
(defconstant limit (+ *glob* 1))
```

你不需要给常数起一个与众不同的名字,因为如果使用相同的名字作为变量,就会出错. 如果你想知道某个符号是否是全局变量或常数的名字,请用boundp:

```
> (boundp '*glob*)
T
```

0.11 赋值

Common Lisp中最普通的赋值操作符是setf. 我们可以用它对全局或局部变量进行赋值:

```
> (setf *glob* 98)
98
> (let ((n 10))
```

[3] 真正的区别在于词法变量和特殊变量的不同,不过我们得到第六章才会考虑它.

```
    (setf n 2)
    n)
2
```

如果第一个自变量不是局部变量的名字,它被认为是全局变量:

```
> (setf x (list 'a 'b 'c))
(A B C)
```

即你可以通过赋值隐含地新建全局变量.不过在源文件中明确地使用defparameter是较好的风格.

你能做的远不止给变量赋值. setf的第一个自变量不但可以是变量名,还可以是表达式. 在这种情况下,第二个自变量的值被插入到第一个所涉及到的位置:

```
> (setf (car x) 'n)
N
> x
(N B C)
```

setf的第一个自变量几乎可以是任何涉及特定位置的表达式. 所有这样的操作符在附录D中都被标记为''settable''.

你可以给setf偶数个自变量. 形如

```
(setf a b
      c d
      e f)
```

的表达式相当于连续三个单独的setf调用:

```
(setf a b)
(setf c d)
(setf e f)
```

0.12 函数化编程法

函数化编程法的意思是编写通过返回值来工作的程序,而不是修改什么东西. 它是Lisp中占支配地位的范例. 大多数Lisp内置函数被调用是为了得到它们的返回值, 而不是它们的副作用.

例如函数remove,它接受一个对象和一个表,返回一个排除了那个对象的新表:

```
> (setf lst '(c a r a t))
(C A R A T)
> (remove 'a lst)
(C R T)
```

为什么不说remove从表中删除一个对象? 因为这不是它所做的事情. 原来的表没有被改变:

```
> lst
(C A R A T)
```

那么如果你真想从表中删掉一些元素怎么办? 在Lisp中,你通常这样做类似的事情:把表传给某个函数,然后用setf来处理返回值. 为了把所有的a从表x中删掉, 我们这样做:

```
(setf x (remove 'a x))
```

函数化编程法本质上意味着避免使用诸如setf的函数. 乍一看连想象这种可能性都很困难,别说试着去做了. 怎么能仅凭返回值就能构造程序?

完全不利用副作用是有困难的. 但随着学习的深入,你会惊讶地发现真正需要副作用的地方极少. 你使用副作用越少,你也就越进步.

函数化编程最重要的优点之一是它允许交互式测试. 在纯粹的函数化代码中,当你写函数的时候就可以测试它们. 如果它返回期望的值,你可以肯定它是正确的. 这些额外的信心,聚集在一起会产生巨大的影响. 当你在程序中修改了任何地方, 你会得到即时的转变. 而这种即时的转变会带来一种全新的编程风格. 就象电话与信件相比,赋予我们新的通讯方式.

0.13 迭代

当我们想做一些重复的事情时,用迭代比用递归更自然些. 典型的例子是用迭代生成某种表格. 函数

```
(defun show-squares (start end)
  (do ((i start (+ i 1)))
      ((> i end) 'done)
    (format t "~A ~A~%" i (* i i))))
```

打印从start到end之间的整数的平方:

```
> (show-squares 2 5)
2 4
3 9
4 16
5 25
DONE
```

do宏是Common Lisp中最基本的迭代操作符. 就象let,do也会产生变量,它的第一个自变量是关于变量规格的表. 表中的每个元素具有如下形式:

```
(variable initial update)
```

其中variable是符号,而initial和update是表达式. 一开始每个变量会被赋予相应的initial的值;在迭代的时候它会被赋予相应的update的值. show-squares中的do仅产生了一个变量i. 第一次迭代的时候,i被赋予start的值,在以后的迭代中它的值会每次增加1.

do的第二个自变量是包含一个或多个表达式的表. 第一个表达式用来测试迭代是否应该停止. 在上例中,测试是(> i end). 其余的表达式会在迭代停

止后依次计算,并且最后一个的值作为do的返回值. 因此show-squares总是会返回done.

余下的自变量组成了循环体. 它们在每次迭代的时候依次被求值. 在每次迭代的时候变量先被更新,然后终止测试被计算,再是(如果测试失败)循环体被计算.

作为对比,这是递归版本的show-squares:

```
(defun show-squares (i end)
  (if (> i end)
      'done
      (progn
        (format t "~A ~A~%" i (* i i))
        (show-squares (+ i 1) end))))
```

此函数中的唯一新面孔是progn. 它接受任意数量的表达式,对它们依次求值,然后返回最后一个的值.

对一些特殊情况Common Lisp有更简单的迭代操作符. 比如,为了遍历表的所有元素,你更可能用dolist. 这个函数返回表的长度:

```
(defun our-length (lst)
  (let ((len 0))
    (dolist (obj lst)
      (setf len (+ len 1)))
    len))
```

此处dolist接受形如(variable expression)的自变量,然后是表达式块. variable相继与expression返回的表中元素绑定,表达式块被计算. 因此上面的循环在意思是,对lst中的每个obj,len增加1.

此函数的一个显然的递归版本是:

```
(defun our-length (lst)
  (if (null lst)
      0
      (+ (our-length (cdr lst)) 1)))
```

即,如果表为空,它的长度就是0;否则它的长度是它的cdr的长度加上1. 此版本清楚一些,但因为它不是尾递归的(见13.2节),它的效率不那么高.

0.14 函数作为对象

函数在Lisp中就象符号,字符串和表一样,是常规的对象. 如果我们给function一个函数的名字,它会返回相关的对象. 就象quote,function是特殊操作符,因此我们不必引用自变量:

```
> (function +)
#<Compiled-Function + 17BA4E>
```

这个模样很奇怪的返回值是函数在典型Common Lisp实现中可能的显示方式.

到目前为止我们涉及到的对象具有这样的特点:Lisp显示它们与我们输入的模样是一致的. 此惯例不适合函数. 一个象+这样的内置函数在内部可能是一段机器码. Common Lisp的实现可以选择任何它喜欢的外部表示方式.

就象我们用'作为quote的简记法,我们可以用#'作为function的简写:

```
> #'+
#<Compiled-Function + 17BA4E>
```

此简记法叫做sharp-quote.

就象其它的对象,函数可以作为自变量传递. 一个接受函数作为自变量的是apply. 它接受一个函数和一列自变量,并返回那个函数应用这些自变量后的结果:

```
> (apply #'+ '(1 2 3))
6
> (+ 1 2 3)
6
```

它能接受任意数目的自变量,只要最后一个是表:

```
> (apply #'+ 1 2 '(3 4 5))
15
```

函数funcall能做同样的事情,不过它不需要把自变量放在表中:

```
> (funcall #'+ 1 2 3)
6
```

宏defun创造一个函数并给它一个名字. 但函数并不是必须需要名字,因此我们也不需要用defun来定义它们. 就象其它Lisp对象一样,我们可以直接引用函数.

为了直接引用一个整数,我们用一列数字;为了直接引用函数,我们用所谓的lambda表达式. 一个lambda表达式是包含以下元素的表:符号lambda,一列参数, 然后是零个或多个表达式组成的函数体.

这个lambda表达式代表接受两个数并返回它们之和的函数:

```
(lambda (x y)
  (+ x y))
```

(x y)是参数表,跟在它后面的是函数体.

可以认为lambda表达式是函数的名称. 就象普通的函数名,lambda表达式可以是函数调用的第一个元素:

```
> ((lambda (x) (+ x 100)) 1)
101
```

而通过在lamda表达式之前附加#',我们得到了相应的函数:

```
> (funcall #'(lambda (x) (+ x 100))
           1)
101
```

此种表达法让我们使用匿名函数.

`lambda`是什么?

`lambda`表达式中的`lambda`不是操作符. 它仅是个符号.[4] 它在早期的Lisp方言里有一种作用:函数的内部形态是表,因此区别函数和普通表的唯一办法是查看第一个元素是否是符号`lambda`.

在Common Lisp中你能把函数表示为表,但它们在内部被表示成独特的函数对象. 因此`lambda`不再是必需的. 如果要求把函数

```
(lambda (x) (+ x 100))
```

表示成

```
((x) (+ x 100))
```

也没有什么矛盾,但Lisp程序员已经习惯了函数以符号`lambda`开始,因此Common Lisp保留了此传统.

0.15 类型

Lisp用非同寻常的灵活手段来处理类型. 在许多语言中,变量是有类型的,你得指定变量的类型才能使用它. 在Common Lisp中,值具有类型,而不是变量. 你可以假想每个对象都贴了指明它的类型的标签. 这种方法叫做显示类型. 你不需要去声明变量的类型,因为变量可以装任何类型的对象.

虽然类型声明不是必需的,为了效率的缘故你可能会用到它们. 类型声明在13.3 节中讨论.

Common Lisp的内置类型构成了一个类型的层次结构. 一个对象通常具有多种类型. 比如,数27是类型`fixnum`,`integer`,`rational`,`real`,`number`,`atom`,和`t`,以一般性的增长为序. (Numeric类型在第9章中讨论)类型`t`是所有类型的超集,因此任何对象都是类型`t`.

函数`typep`接受一个对象和一个类型说明符,如果对象是那种类型就返回真:

```
> (typep 27 'integer)
T
```

当我们碰到各种内置类型时,我们会介绍它们.

0.16 展望

本章仅蜻蜓点水般地介绍了一下Lisp. 然而一种非同寻常的语言的形象已经浮现出来了. 首先,该语言有单一的语法来表达所有的程序结构. 此语法基于一种Lisp 的对象---表. 函数,作为独特的Lisp对象,可以表示为表. 而且Lisp本身就是Lisp程序,几乎全是由与你自己定义的函数没有任何区别的函数组成的.

[4]在Ansi Common Lisp中还有一个lambda宏,它能让你把#'(lambda (x) x)写成(lambda (x) x). 由于使用这个宏模糊了lambda表达式和符号化的函数名(其中你得作用#') 的对称性,它最多不过具有美观的外表.

如果你还不完全清楚所有这些概念之间的关系,请不必担心. Lisp引入了这么多新颖的概念,你得花时间去熟悉它们. 不过至少得说明一件事: 令人吃惊的优雅思想蕴藏其中.

Richard Gabriel曾半开玩笑地说C是适合写Unix的语言.[5] 我们也可以说Lisp是编写Lisp的语言. 但这是两种不同的陈述. 一种可以用自己来编写的语言是和一种擅长编写某些特定类型的应用的语言完全不同的. 它开启了新的编程方法:你不但在语言中编程,你还可以改进语言以适合你程序的需要. 如果你想理解Lisp编程的本质,这个思想是个很好的起点.

总结

1. Lisp是交互式语言. 如果你在顶层输入表达式,Lisp会打印它的值.
2. Lisp程序由表达式组成. 表达式可以是一个原子,或是一个表, 表的第一个元素是操作符,后面跟着零个或多个自变量. 前缀表达式意味着操作符可接受任意多个自变量.
3. Common Lisp函数调用的求值规则:从左至右对自变量求值,然后把这些值传给由操作符表示的函数. quote有它自己的求值规则:它原封不动地返回自变量.
4. 除了通常的数据类型,Lisp还有符号和表. 由于Lisp程序由表组成,很容易编写能写程序的程序.
5. 三个基本的表处理函数是cons:它创造一个表;car:它返回表的头一个元素; cdr:它返回第一个元素之后的所有东西.
6. 在Common Lisp里, t表示真,nil表示伪. 在逻辑上下文中,除了nil之外的任何东西都算作真. 基本的条件语句是if. and和or操作符就象条件语句.
7. Lisp主要是由函数构成的. 你可用defun来定义新的函数.
8. 调用自己的函数是递归的. 递归函数应该被认为是一个过程而不是机器.
9. 括号不是个问题,因为程序员利用缩进来读写Lisp.
10. 基本的i/o函数是read:它包含了完整的Lisp语法分析器,和format:它基于模板产生输出.
11. 你可以用let创造新的局部变量,用defparameter创造新的全局变量.
12. 赋值操作符是setf. 它的第一个自变量可以是表达式.
13. 函数化编程法---它意味着避免副作用---是Lisp中占支配地位的范例.
14. 基本的循环操作符是do.

[5] Gabriel, Richard P. Lisp: Good News, Bad News, How to Win Big. AI Expert, June 1991, p. 34.

15. 函数是常规的Lisp对象. 它们可以作为自变量被传递,可以表示成lambda表达式.

16. 值有类型,而变量没有类型

练习

1. 解释以下表达式求值后的结果:

   a. (+ (- 5 1) (+ 3 7))

   b. (list 1 (+ 2 3))

2. 给出3种不同的能返回(a b c)的cons表达式

3. 用car和cdr定义一个函数,它返回表的第四个元素.

4. 定义一个函数,它接受两个自变量,返回两个中较大的一个.

5. 这些函数做了什么?

```
a. (defun enigma (x)
     (and (not (null x))
          (or (null (car x))
              (enigma (cdr x)))))

b. (defun mystery (x y)
     (if (null y)
         nil
         (if (eql (car y) x)
             0
             (let ((z (mystery x (cdr y))))
               (and z (+ z 1))))))
```

6. 在下面的表达式中,x处应该是什么可得出结果?

```
a. > (car (x (cdr '(a (b c) d))))
   B

b. > (x 13 (/ 1 0))
   13

c. > (x #'list 1 nil)
   (1)
```

7. 只用本章介绍的操作符,定义一个函数,它接受一个表作为自变量,并返回t 如果表的元素中至少有一个类型是表.

8. 给出函数的迭代和递归版本:它

   a. 接受一个正整数,并打印这么多数目的点.

   b. 接受一个表,返回符号a在表中出现的次数.

9. 一位朋友想写一个函数,它返回表中所有非nil元素之和. 他写了此函数的两个版本, 但没有一个能正确工作. 请指出错误在哪里,并给出正确的版本:

   a.
   ```
   (defun summit (lst)
     (remove nil lst)
     (apply #'+ lst))
   ```

   b.
   ```
   (defun summit (lst)
     (let ((x (car lst)))
       (if (null x)
           (summit (cdr lst))
           (+ x (summit (cdr lst))))))
   ```