第5章

函　数

5

有了语法和语义规则以后，所有Lisp程序的三个最基本组成部分就是函数、变量和宏。在第3章里构建数据库时，这三个组件已经全部用到了，但是我没有详细提及它们是如何工作的，如何更好使用它们。接下来的几章将专门讲解这三个主题，先从函数开始。就跟其他语言里一样，函数提供了用于抽象和功能化的基本方法。

Lisp本身是由大量函数组成的。其语言标准中有超过四分之三的名字用于定义函数。所有内置的数据类型纯粹是用操作它们的函数来定义的。甚至连Lisp强大的对象系统也是构建在函数的概念性扩展——广义函数（generic function）之上的，第16章将会介绍它们。

而且，尽管宏对于Lisp风格有着重要的作用，但最终所有实际的功能还是由函数来提供的。宏运行在编译期，因此它们生成的代码，即当所有宏被展开后将实际构成程序的那些代码，将完全由对函数和特殊操作符的调用所构成。更不用说，宏本身也是函数了——尽管这种函数是用来生成代码，而不是用来完成实际的程序操作的。[1]

5.1 定义新函数

函数一般使用DEFUN宏来定义。DEFUN的基本结构看起来像这样：

```
(defun name (parameter*)
  "Optional documentation string."
  body-form*)
```

任何符号都可用作函数名。[2]通常函数名仅包含字典字符和连字符，但是在特定的命名约定里，其他字符也允许使用。例如，将值的一种类型转换成另一种的函数有时会在名字中使用→，一个将字符串转换成微件（widget）的函数可能叫做`string->widget`。最重要的一个命名约定

① 尽管函数对于Common Lisp很重要，但将其说成是函数型语言却并不是非常合适。尽管一些Common Lisp特性（例如其列表管理函数）被设计用于函数型编程风格，并且Lisp在函数型编程史上也有其突出的地位——McCarthy引进了许多被认为对函数型编程非常重要的思想，但Common Lisp其本意是被用于支持多种不同的编程风格的。在Lisp家族里，Scheme最接近“纯”函数型语言，而就算它也有一些特性，但相比于诸如Haskell和ML这类语言，其在纯粹程度上也不够格。

② 严格来讲是几乎任何符号。如果你使用了由语言标准所定义的名字作为你自己的函数的名字，其后果尚未可知。尽管如此，你在第21章会看到，Lisp的包系统允许你在不同的命名空间里创建名字，因此这实际上不是问题。

是在第2章里提到的那个，即要用连字符而不是下划线或内部大写来构造复合名称。因此，frob-widget比frob_widget或frobWidget更具有Lisp风格。一个函数的形参列表定义了一些变量，将用来保存函数在调用时所传递的实参。[1]如果函数不带有实参，则该列表就是空的，写成()。不同种类的形参分别负责处理必要的、可选的、多重的以及关键字实参。我将在下一节里讨论相关细节。

如果一个字符串紧跟在形参列表之后，那么它应该是一个用来描述函数用途的文档字符串。当定义函数时，该文档字符串将被关联到函数名上，并且以后可以通过**DOCUMENTATION**函数来获取。[2]

最后，一个**DEFUN**的主体可由任意数量的Lisp表达式所构成。它们将在函数被调用时依次求值，而最后一个表达式的值将被作为整个函数的值返回。另外**RETURN-FROM**特殊操作符可用于从函数的任何位置立即返回，我很快就会谈到它。

第2章里所写的hello-world函数，形式如下：

```
(defun hello-world () (format t "hello, world"))
```

现在可以分析一下该程序的各个部分了。它的名字是hello-world，形参列表为空，因此不接受任何参数，它没有文档字符串，并且它的函数体由一个表达式所构成：

```
(format t "hello, world")
```

下面是一个更复杂一些的函数：

```
(defun verbose-sum (x y)
  "Sum any two numbers after printing a message."
  (format t "Summing ~d and ~d.~%" x y)
  (+ x y))
```

这个函数称为verbose-sum，它接受的两个实参分别与形参x和y一一对应并且带有一个文档字符串，以及一个由两个表达式所组成的主体。由“+”调用所返回的值将成为verbose-sum的返回值。

5.2 函数形参列表

关于函数名或文档字符串就没有更多可说的了，而本书其余部分将用很多篇幅来描述所有可在一个函数体里做的事情，因此就只需讨论形参列表了。

很明显，一个形参列表的基本用途是为了声明一些变量，用来接收传递给函数的实参。当形参列表是一个由变量名所组成的简单列表时，如同在verbose-sum里那样，这些形参被称为必

① 因为Lisp的函数表示法与lambda演算之间的历史关系，形参列表有时也称为lambda列表。

② 例如：

```
(documentation 'foo 'function)
```

将返回函数foo的文档字符串。尽管如此，请注意文档字符串是用来给人看的，而没有任何程序意义上的用途。一个Lisp实现不要求保存它们，实际上可在任何时候丢弃它们，因此可移植的程序不应该依赖于它们的存在。在某些实现里，一个由实现所定义的全局变量需要在使用文档字符串之前被设置成指定的值。

要形参。当函数被调用时，必须为它的每一个必要形参都提供一个实参。每一个形参被绑定到对应的实参上。如果一个函数以过少或过多的实参来调用的话，Lisp就会报错。

但是，Common Lisp的形参列表也给了你更灵活的方式将函数调用实参映射到函数形参。除了必要形参以外，一个函数还可以有可选形参，或者也可以用单一形参绑定到含有任意多个额外参数的列表上。最后，参数还可以通过关键字而不是位置来映射到形参上。这样，Common Lisp的形参列表对于几种常见的编码问题提供了一种便利的解决方案。

5.3　可选形参

虽然许多像verbose-sum这样的函数只有必要形参，但并非所有函数都如此简单。有时一个函数将带有一个只有特定调用者才会关心的形参，这可能是因为它有一个合理的默认值。例如一个可以创建按需增长的数据结构的函数。由于数据结构可以增长，那么从正确性角度来说，它的初始尺寸就无关紧要了。那些清楚知道自己打算在数据结构中放置多少个元素的调用者们，可以通过设置特定的初始尺寸来改进其程序的性能，而多数调用者只需让实现数据结构的代码自行选择一个好的通用值就可以了。在Common Lisp中，你可以使用可选形参，从而使两类调用者都满意。不在意的调用者们将得到一个合理的默认值，而其他调用者们有机会提供一个指定的值。[1]

为了定义一个带有可选形参的函数，在必要形参的名字之后放置符号&optional，后接可选形参的名字。下面就是一个简单的例子：

```
(defun foo (a b &optional c d) (list a b c d))
```

当该函数被调用时，实参被首先绑定到必要形参上。在所有必要形参都被赋值以后，如果还有任何实参剩余，它们的值将被赋给可选形参。如果实参在所有可选形参被赋值之前用完了，那么其余的可选形参将自动绑定到值NIL上。这样，前面定义的函数会给出下面的结果：

```
(foo 1 2)      → (1 2 NIL NIL)
(foo 1 2 3)    → (1 2 3 NIL)
(foo 1 2 3 4)  → (1 2 3 4)
```

Lisp仍然可以确保适当数量的实参被传递给函数——在本例中是2到4个。而如果函数用太少或太多的参数来调用的话，将会报错。

当然，你会经常想要一个不同于NIL的默认值。这时可以通过将形参名替换成一个含有名字跟一个表达式的列表来指定该默认值。只有在调用者没有传递足够的实参来为可选形参提供值的时候，这个表达式才会被求值。通常情况只是简单地提供一个值作为表达式：

---

① 在那些不直接支持可选形参的语言里，程序员们通常可以找到模拟它们的方式。一种技术是使用可区分的“no-value”值供调用者传递，以说明它们想要一个给定形参的默认值。例如在C语言中，通常使用NULL作为这样的可区分值。尽管如此，这种在函数和其调用者之间的协议完全是自组织的——在某些函数或某些实参中，NULL可能是一个可区分值，而在另一些函数或实参中，这样的特殊值可能是－1或一些由#define所定义的常量。在像Java这种支持用多个定义重载单个方法的语言里，可选形参也可以通过提供多个具有相同名称，但不同实参个数的方法（method）来模拟，这时当一个方法使用较少的实参来调用时，会以默认值代替缺少的实参去调用“真实”的那个方法。

```
(defun foo (a &optional (b 10)) (list a b))
```

上述函数要求将一个实参绑定到形参a上。当存在第二个实参时，第二个形参b将使用其值，否则使用10。

```
(foo 1 2) → (1 2)
(foo 1)   → (1 10)
```

不过有时可能需要更灵活地选择默认值。比如可能想要基于其他形参来计算默认值。默认值表达式可以引用早先出现在形参列表中的形参。如果要编写一个返回矩形的某种表示的函数，并且想要使它可以特别方便地产生正方形，那么可以使用一个像这样的形参列表：

```
(defun make-rectangle (width &optional (height width)) ...)
```

除非明确指定否则这将导致height形参带有和width形参相同的值。

有时，有必要去了解一个可选形参的值究竟是被调用者明确指定还是使用了默认值。除了通过代码来检查形参的值是否为默认值（假如调用者碰巧显式传递了默认值，那么这样做终归是无效的）以外，你还可以通过在形参标识符的默认值表达式之后添加另一个变量名来做到这点。该变量将在调用者实际为该形参提供了一个实参时被绑定到真值，否则为NIL。通常约定，这种变量的名字与对应的真实形参相同，但是带有一个-supplied-p后缀。例如：

```
(defun foo (a b &optional (c 3 c-supplied-p))
  (list a b c c-supplied-p))
```

这将给出类似下面的结果：

```
(foo 1 2)   → (1 2 3 NIL)
(foo 1 2 3) → (1 2 3 T)
(foo 1 2 4) → (1 2 4 T)
```

5.4 剩余形参

可选形参仅适用于一些较为分散并且不能确定调用者是否会提供值的形参。但某些函数需要接收可变数量的实参，比如说前文已然出现过的一些内置函数。FORMAT有两个必要实参，即流和控制串。但在这两个之后，它还需要一组可变数量的实参，这取决于控制串需要插入多少个值。+函数也接受可变数量的实参——没有特别的理由限制它只能在两个数之间相加，它可以对任意数量的值做加法运算（它甚至可以没有实参，此时返回0——加法的底数）。下面这些都是这两个函数的合法调用：

```
(format t "hello, world")
(format t "hello, ~a" name)
(format t "x: ~d y: ~d" x y)
(+)
(+ 1)
(+ 1 2)
(+ 1 2 3)
```

很明显，也可以通过简单地给它一些可选形参来写出接受可变数量实参的函数，但这样将会

非常麻烦，光是写形参列表就已经足够麻烦了，何况还要在函数体中处理所有这些形参。为了做好这件事，还不得不使用一个合法函数调用所能够传递的那么多的可选形参。这一具体数量与具体实现相关，但可以保证至少有50个。在当前所有实现中，它的最大值范围从4096到536 870 911。[①] 汗，这种绞尽脑汁的无聊事情绝对不是Lisp风格。

相反，Lisp允许在符号**&rest**之后包括一揽子形参。如果函数带有**&rest**形参，那么任何满足了必要和可选形参之后的其余所有实参就将被收集到一个列表里成为该**&rest**形参的值。这样，**FORMAT**和+的形参列表可能看起来会是这样：

```
(defun format (stream string &rest values) ...)
(defun + (&rest numbers) ...)
```

5.5　关键字形参

尽管可选形参和剩余形参带来了很大的灵活性，但两者都不能帮助应对下面的情形。假设有一个接受四个可选形参的函数，如果在多数的函数调用中，调用者只想为四个参数中的一个提供值，并且更进一步，不同的调用者甚至有可能将分别选择使用其中一个参数。

想为第一个形参提供值的调用者将会很方便——只需传递一个可选实参，然后忽略其他就好了。但是所有其他的调用者将不得不为所不关心的一到三个形参传递一些值。这不正是可选形参想来解决的问题吗？

当然是。问题在于可选形参仍然是位置相关的——如果调用者想要给第四个可选形参传递一个显式的值，就会导致前三个可选形参对于该调用者来说变成了必要形参。幸好我们有另一种形参类型，关键字形参，它可以允许调用者指定具体形参相应所使用的值。

为了使函数带有关键字形参，在任何必要的**&optional**和**&rest**形参之后，可以加上符号**&key**以及任意数量的关键字形参标识符，后者的格式类似于可选形参标识符。下面就是一个只有关键字形参的函数：

```
(defun foo (&key a b c) (list a b c))
```

当调用这个函数时，每一个关键字形参将被绑定到紧跟在同名键字后面的那个值上。如第4章所述，关键字是以冒号开始的名字，并且它们被自动定义为自求值常量。

如果一个给定的关键字没有出现在实参列表中，那么对应的形参将被赋予其默认值，如同可选形参那样。因为关键字实参带有标签，所以它们在必要实参之后可按任意顺序进行传递。例如foo可以用下列形式调用：

```
(foo)                → (NIL NIL NIL)
(foo :a 1)           → (1 NIL NIL)
(foo :b 1)           → (NIL 1 NIL)
(foo :c 1)           → (NIL NIL 1)
(foo :a 1 :c 3)      → (1 NIL 3)
(foo :a 1 :b 2 :c 3) → (1 2 3)
```

① 常量**CALL-ARGUMENTS-LIMIT**将告诉你这个与具体实现有关的数值。

```
(foo :a 1 :c 3 :b 2) → (1 2 3)
```

如同可选形参那样，关键字形参也可以提供一个默认值形式以及一个supplied-p变量名。在关键字形参和可选形参中，这个默认值形式都可以引用那些早先出现在形参列表中的形参。

```
(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ a b)))
  (list a b c b-supplied-p))
(foo :a 1)           → (1 0 1 NIL)
(foo :b 1)           → (0 1 1 T)
(foo :b 1 :c 4)      → (0 1 4 T)
(foo :a 2 :b 1 :c 4) → (2 1 4 T)
```

同样，如果出于某种原因想让调用者用来指定形参的关键字不同于实际形参名，那么可以将形参名替换成一个列表，令其含有调用函数时使用的关键字以及用作形参的名字。比如说下面这个foo的定义：

```
(defun foo (&key ((:apple a)) ((:box b) 0) ((:charlie c) 0 c-supplied-p))
  (list a b c c-supplied-p))
```

可以让调用者这样调用它：

```
(foo :apple 10 :box 20 :charlie 30) → (10 20 30 T)
```

这种风格在想要完全将函数的公共API与其内部细节相隔离时特别有用，通常是因为想要在内部使用短变量名，而不是API中的描述性关键字。不过该特性不常被用到。

5.6 混合不同的形参类型

在单一函数里使用所有四种类型形参的情况虽然罕见，但也是可能的。无论何时，当用到多种类型的形参时，它们必须以这样的顺序声明：首先是必要形参，其次是可选形参，再次是剩余形参，最后才是关键字形参。但在使用多种类型形参的函数中，一般情况是将必要形参和另外一种类型的形参组合使用，或者可能是组合**&optional**形参和**&rest**形参。其他两种组合方式，无论是**&optional**形参还是**&rest**形参，当与**&key**形参组合使用时，都可能导致某种奇怪的行为。

将**&optional**形参和**&key**形参组合使用时将产生非常奇怪的结果，因此也许应该避免将它们一起使用。问题出在如果调用者没有为所有可选形参提供值时，那么没有得到值的可选形参将吃掉原本用于关键字形参的关键字和值。例如，下面这个函数很不明智地混合了**&optional**形参和**&key**形参：

```
(defun foo (x &optional y &key z) (list x y z))
```

如果像这样调用的话，就没问题：

```
(foo 1 2 :z 3) → (1 2 3)
```

这样也可以：

```
(foo 1)  → (1 nil nil)
```

但是这样的话将报错：

```
(foo 1 :z 3) → ERROR
```

这是因为关键字:z被作为一个值填入到可选的y形参中了，只留下了参数3被处理。在这里，Lisp期待一个成对的关键字/值，或者什么也没有，否则就会报错。也许更坏的是，如果该函数带有两个&optional形参，上面最后一个调用将导致值:z和3分别被绑定到两个&optional形参上，而&key形参z将得到默认值NIL，而不声明缺失了东西。

一般而言，如果正在编写一个同时使用&optional形参和&key形参的函数，可能就应该将它变成全部使用&key形参的形式——它们更灵活，并且总会可以在不破坏该函数的已有调用的情况下添加新的关键字形参。也可以移除关键字形参，只要没人在使用它们。[①]一般而言，使用关键字形参将会使代码相对易于维护和拓展——如果需要为函数添加一些需要用到新参数的新行为，就可以直接添加关键字形参，而无需修改甚至重新编译任何调用该函数的已有代码。

虽然可以安全地组合使用&rest形参和&key形参，但其行为初看起来可能会有一点奇怪。正常地来讲，无论是&rest还是&key出现在形参列表中，都将导致所有出现在必要形参和&optional形参之后的那些值被特别处理——要么作为&rest形参被收集到一个形参列表中，要么基于关键字被分配到适当的&key形参中。如果&rest和&key同时出现在形参列表中，那么两件事都会发生——所有剩余的值，包括关键字本身，都将被收集到一个列表里，然后被绑定到&rest形参上；而适当的值，也会同时被绑定到&key形参上。因此，给定下列函数：

```
(defun foo (&rest rest &key a b c) (list rest a b c))
```

将得到如下结果：

```
(foo :a 1 :b 2 :c 3)  → ((:A 1 :B 2 :C 3) 1 2 3)
```

5.7　函数返回值

目前写出的所有函数都使用了默认的返回值行为，即最后一个表达式的值被作为整个函数的返回值。这是从函数中返回值的最常见方式。

但某些时候，尤其是想要从嵌套的控制结构中脱身时，如果有办法从函数中间返回，那将是非常便利的。在这种情况下，你可以使用RETURN-FROM特殊操作符，它能够立即以任何值从函数中间返回。

在第20章将会看到，RETURN-FROM事实上不只用于函数，它还可以用来从一个由BLOCK特殊操作符所定义的代码块中返回。不过DEFUN会自动将其整个函数体包装在一个与其函数同名的代码块中。因此，对一个带有当前函数名和想要返回的值的RETURN-FROM进行求值将导致函数立即以该值退出。RETURN-FROM是一个特殊操作符，其第一个“参数”是它想要返回的代码块名。该名字不被求值，因此无需引用。

---

① 有四个标准函数同时接受&optiona参数和&key参数——READ-FROM-STRING、PARSE-NAMESTRING、WRITE-LINE和WRITE-STRING。在标准化的过程中，出于跟早期Lisp方言向后兼容的目的它们被原样保留了下来。READ-FROM-STRING应该是新的Lisp程序员最常用到的函数之一，一个诸如(read-from-string s :start 10)这样的调用看起来会忽略:start关键字形参，直接从索引位置0而非10处开始读取。这是因为READ-FROM-STRING还有两个&optional形参将参数:start和10覆盖了。

下面这个函数使用了嵌套循环来发现第一个数对——每个都小于10，并且其乘积大于函数的参数，它使用RETURN-FROM在发现之后立即返回该数对：

```
(defun foo (n)
  (dotimes (i 10)
    (dotimes (j 10)
      (when (> (* i j) n)
        (return-from foo (list i j))))))
```

必须承认的是，不得不指定正在返回的函数名多少会有些不便——比如改变了函数的名字，就需要同时改变RETURN-FROM中所使用的名字。[1]但在事实上，显式的RETURN-FROM调用在Lisp中出现的频率远小于return语句在源自C的语言里所出现的频率，因为所有的Lisp表达式，包括诸如循环和条件语句这样的控制结构，都会求值得到一个值。因此在实践中这不是什么问题。

5.8 作为数据的函数——高阶函数

使用函数的主要方式是通过名字来调用它们，但有时将函数作为数据看待也是很有用的。例如，可以将一个函数作为参数传给另一个函数，从而能写出一个通用的排序函数，允许调用者提供一个比较任意两元素的函数，这样同样的底层算法就可以跟许多不同的比较函数配合使用了。类似地，回调函数（callback）和钩子（hook）也需要能够保存代码引用便于以后运行。由于函数已经是一种对代码比特进行抽象的标准方式，因此允许把函数视为数据也是合理的。[2]

在Lisp中，函数只是另一种类型的对象。在用DEFUN定义一个函数时，实际上做了两件事：创建一个新的函数对象以及赋予其一个名字。在第3章里我们看到，也可以使用LAMBDA表达式来创建一个函数而无需为其指定一个名字。一个函数对象的实际表示，无论是有名的还是匿名的，都只是一些二进制数据——以原生编译的Lisp形式存在，可能大部分是由机器码构成。只需要知道如何保持它们以及需要时如何调用它们。

特殊操作符FUNCTION提供了用来获取一个函数对象的方法。它接受单一实参并返回与该参数同名的函数。这个名字是不被引用的。因此如果一个函数foo的定义如下。

```
CL-USER> (defun foo (x) (* 2 x))
FOO
```

就可以得到如下的函数对象。[3]

```
CL-USER> (function foo)
#<Interpreted Function FOO>
```

事实上，你已经用过FUNCTION了，但它是以伪装的形式出现的。第3章里用到的#'语法就是

① 另一个宏RETURN不要求使用一个名字。尽管如此，你不能用它来代替RETURN-FROM从而避免指定函数名，它是一个从称为NIL的块中返回的语法糖。我将在第20章里跟BLOCK以及RETURN-FROM一起讨论其细节。

② 当然了，Lisp并不是唯一的将函数视为数据的语言。C使用函数指针；Perl使用子例程引用（subroutine reference）；Python使用与Lisp相似的模式；C#使用了代理（delegate），其本质上是带有类型的函数指针；而Java则使用了相当笨重的反射（reflection）和匿名类机制。

③ 一个函数对象的确切打印格式将随着不同的实现而有所不同。

FUNCTION的语法糖，正如“'”是**QUOTE**的语法糖一样。[1]因此也可以像这样得到foo的函数对象。

```
CL-USER> #'foo
#<Interpreted Function FOO>
```

一旦得到了函数对象，就只剩下一件事可做了——调用它。Common Lisp提供了两个函数用来通过函数对象调用函数：**FUNCALL**和**APPLY**。[2]它们的区别仅在于如何获取传递给函数的实参。

FUNCALL用于在编写代码时确切知道传递给函数多少实参时。**FUNCALL**的第一个实参是被调用的函数对象，其余的实参被传递到该函数中。因此，下面两个表达式是等价的：

```
(foo 1 2 3) ≡ (funcall #'foo 1 2 3)
```

不过，用**FUNCALL**来调用一个写代码时名字已知的函数毫无意义。事实上，前面的两个表达式将很可能被编译成相同的机器指令。

下面这个函数演示了**FUNCALL**的另一个更有建设性的用法。它接受一个函数对象作为实参，并使用实参函数在min和max之间以step为步长的返回值来绘制一个简单的ASCII式柱状图：

```
(defun plot (fn min max step)
  (loop for i from min to max by step do
        (loop repeat (funcall fn i) do (format t "*"))
        (format t "~%")))
```

FUNCALL表达式在每个i值上计算函数的值。内层**LOOP**循环使用计算得到的值来决定向标准输出打印多少星号。

请注意，不需要使用**FUNCTION**或#'来得到fn的函数值。因为它是作为函数对象的变量的值，所以你需要它被解释成一个变量。可以用任何接受单一数值实参的函数来调用plot，例如内置的函数**EXP**，它返回以e为底以其实参为指数的值。

```
CL-USER> (plot #'exp 0 4 1/2)
*
*
**
****
*******
************
********************
**********************************
**************************************************************
NIL
```

然而，当实参列表只在运行期已知时，**FUNCALL**的表现不佳。例如，为了再次调用plot函数，假设你已有一个列表，其包括一个函数对象、一个最小值和一个最大值以及一个步长。换句

---

① 思考**FUNCTION**的最佳方式是将其视为一个特殊类型的引用。**QUOTE**一个符号可以避免其被求值，从而得到该符号本身而不是由该符号所命名的变量的值。**FUNCTION**同样规避了正常的求值规则，但并非是避免其符号被求值，而是使其作为一个函数的名字来求值，就好像它作为函数调用表达式中的函数名那样。

② 实际上还有第三个，特殊操作符**MULTIPLE-VALUE-CALL**，但是我将保留到第20章中当我讨论到返回多值的表达式时再讨论它。

话说，这个列表包含了你想要作为实参传给plot的所有的值。假设这个列表保存在变量plot-data中，可以像这样用列表中的值来调用plot：

```
(plot (first plot-data) (second plot-data) (third plot-data) (fourth plot-data))
```

这样固然可以，但仅仅为了将实参传给plot而显式地将其解开，看起来相当讨厌。

这就是需要**APPLY**的原因。和**FUNCALL**一样，**APPLY**的第一个参数是一个函数对象。但在这个函数对象之后，它期待一个列表而非单独的实参。它将函数应用在列表中的值上，这就使你可以写出下面的替代版本：

```
(apply #'plot plot-data)
```

更方便的是，**APPLY**还接受“孤立”（loose）的实参，只要最后一个参数是个列表。因此，假如plot-data只含有最小、最大和步长值，那么你仍然可以像这样来使用**APPLY**在该范围上绘制**EXP**函数：

```
(apply #'plot #'exp plot-data)
```

APPLY并不关心所用的函数是否接受**&optional**、**&rest**或**&key**实参——由任何孤立实参和最后的列表所组合而成的实参列表必定是一个合法的实参列表，其对于该函数来说带有足够的实参用于所有必要形参和适当的关键字形参。

5.9 匿名函数

一旦开始编写或只是使用那些可以接受其他函数作为实参的函数，你就必然发现，有时不得不去定义和命名一个仅使用一次的函数，尤其是你可能从不用名字来调用它时，这会让人相当恼火。

觉得没必要用**DEFUN**来定义一个新函数时，可以使用一个**LAMBDA**表达式创建匿名的函数。第3章里讨论过，一个**LAMBDA**表达式形式如下：

```
(lambda (parameters) body)
```

可以将**LAMBDA**表达式视为一种特殊类型的函数名，其名字本身直接描述函数的用途。这就解释了为什么可以使用一个带有#'的**LAMBDA**表达式来代替一个函数名。

```
(funcall #'(lambda (x y) (+ x y)) 2 3) → 5
```

甚至还可以在一个函数调用表达式中将**LAMBDA**表达式用作函数名。由此一来，我们可以在需要时以更简洁方式来书写前面的**FUNCALL**表达式如下：

```
((lambda (x y) (+ x y)) 2 3) → 5
```

但几乎没人这样做。它唯一的用途是来强调将**LAMBDA**表达式用在任何一个正常函数名可以出现

的场合都是合法的。[1]

在需要传递一个作为参数的函数给另一个函数，并且需要传递的这个函数简单到可以内联表达时，匿名函数特别有用。例如，假设想要绘制函数2x，你可以定义下面的函数：

```
(defun double (x) (* 2 x))
```

并随后将其传给plot：

```
CL-USER> (plot #'double 0 10 1)
**
****
******
********
**********
************
**************
****************
******************
********************
NIL
```

但如果写成这样将会更简单和清晰：

```
CL-USER> (plot #'(lambda (x) (* 2 x)) 0 10 1)
**
****
******
********
**********
************
**************
****************
******************
********************
NIL
```

LAMBDA表达式的另一项重要用途是制作闭包（closure），即捕捉了其创建时环境信息的函数。你在第3章里使用了一点儿闭包，但要深入了解闭包的工作原理及其用途，更多的还是要从变量而非函数的角度去考察，因此我将在下一章里讨论它们。

---

① 在Common Lisp里也可以不带前缀#'来使用一个**LAMBDA**表达式作为**FUNCALL**的参数（或是其他一些接受函数参数的函数，诸如**SORT**和**MAPCAR**），像这样：

```
(funcall (lambda (x y) (+ x y)) 2 3)
```

这是合法的，并且出于一个诡异的理由，它跟带有#'的版本等价。在历史上，**LAMBDA**表达式本身并非是可以求值的表达式。这就是说，**LAMBDA**并非是一个函数、宏或特殊操作符的名字。并且一个以符号**LAMBDA**开始的列表会被Lisp作为特殊的语法构造来识别成一种函数名。

如果还是这样，那么`(funcall (lambda (...) ...))`就不合法，因为**FUNCALL**是一个函数，而一个函数调用的正常求值规则将要求该**LAMBDA**表达式被求值。不过，在ANSI标准化过程晚期，为了有可能实现ISLISP（另一种同时被标准化的Lisp方言，将其作为用户层面的兼容层构建在Common Lisp之上），人们定义了一个**LAMBDA**宏，其展开成一个包装在**LAMBDA**表达式外围的**FUNCTION**调用。换句话说，下面的**LAMBDA**表达式：

```
(lambda () 42)
```

当其出现在一个会被求值的上下文中时，将展开成下面这样：

```
(function (lambda () 42)) ; or #'(lambda () 42)
```

这使其可以合法地用在需要求值的位置上，例如作为**FUNCALL**的参数。换句话说，它纯粹是一个语法糖。多数人要么总是在值位置上的**LAMBDA**表达式前使用#'，要么总是不用。在本书中，我将总是使用#'。