第一章　Lisp 简介

1.1　符号计算

到目前为止，我们所做的只是以简单的袖珍计算器的方式操纵数字。Lisp 比计算器更有用有两个主要原因。首先，它允许我们操作数字以外的对象，其次，它允许定义在后续计算中可能有用的新对象。我们将依次研究这两个重要属性。

除了数字，Lisp 还可以表示字符（字母）、字符串和任意符号，我们可以自由地将这些符号解释为数学世界之外的事物。Lisp 还可以通过将多个对象组合成一个列表来构建非原子对象。这种能力是基本的，在语言中得到了很好的支持；事实上，Lisp 这个名字是 LISt Processing 的缩写。

下面是一个列表计算的实例：

```
> (append '(Pat Kim) '(Robin Sandy)) => (PAT KIM ROBIN
  SANDY)
```

这个表达式将两个名字的列表连接在一起。计算此表达式的规则与数值计算的规则相同：将函数（在本例中为 append）应用于参数的值。

不寻常的部分是引号（'），它用于阻止对以下表达式的求值，并按字面意思返回它。如果我们只有表达式（Pat Kim），则可以通过将 Pat 视为函数并将其应用于表达式 Kim 的值来对其进行评估。这不是我们想要的。引号指示 Lisp 将列表视为一段数据，而不是函数调用：

```
> '(Pat Kim) => (PAT KIM)
```

在其他计算机语言（和英语）中，引号通常成对出现：一个标记开始，一个标记结束。在 Lisp 中，单引号用于标记表达式的开头。由于我们总是知道一个表达式有多长——要么到原子的末尾，要么到列表的匹配括号——我们不需要明确的标点符号来告诉我们表达式在哪里结束。引号可以用于列表，如'(Pat

Kim)，也可以用于符号，如'Robin，事实上还可以用于其他任何东西。以下是一些示例：

```
> 'John => JOHN

> '(John Q Public) => (JOHN Q PUBLIC)

> '2 => 2

> 2 => 2

> '(+ 2 2) => (+ 2 2)

> (+ 2 2) 4

> John => *Error: JOHN is not a bound variable*

> (John Q Public) => *Error: JOHN is not a function*
```

请注意，'2 的计算结果为 2，因为它是一个引号表达式，而 2 的计算值为 2，是因为数字本身的计算结果。同样的结果，不同的原因。相比之下，'John 求值为 John 是因为它是一个引号表达式，但求值 John 会导致错误，因为求值一个符号意味着得到符号的值，而没有给 John 赋值。

符号计算可以嵌套，甚至可以与数值计算混合。以下表达式使用内置函数 list 以与我们之前看到的略有不同的方式构建名称列表。然后，我们将看到如何使用内置函数 length 查找列表中的元素数量：

```
> (append '(Pat Kim) (list '(John Q Public) 'Sandy))
(PAT KIM (JOHN Q PUBLIC) SANDY)

> (length (append '(Pat Kim) (list '(John Q Public) '
    Sandy)))
4
```

关于符号，有四点要强调：

- 首先，重要的是要记住，Lisp 不会给它所操纵的对象附加任何外部意义。例如，我们自然会认为（Robin Sandy）是两个名字的列表，而（John Q

Public）是一个人的名字、中间的首字母和姓氏的列表。Lisp 没有这样的先入为主的观念。对于 Lisp 来说，Robin 和 xyzzy 都是非常好的符号。

- 其次，为了进行上述计算，我们必须知道 append、length 和 + 是 Common Lisp 中定义的函数。学习一门语言涉及记忆词汇（或知道在哪里查找），以及学习形成表达式和确定其含义的基本规则。Common Lisp 提供了 700 多个内置函数。在某些时候，读者应该翻阅参考文本，看看里面有什么，但大多数重要功能都在本书的第一部分中介绍。

- 第三，请注意 Common Lisp 中的符号不区分大小写。我的意思是，输入 John, john, 和 jOhN 都引用了相同的符号，通常打印为 JOHN。

- 第四，请注意，符号中允许使用各种各样的字符：数字、字母和其他标点符号，如 “+” 或 “!” 构成符号的确切规则有点复杂，但通常的惯例是使用主要由字母组成的符号，单词之间用破折号（-）分隔，最后可能还有一个数字。一些程序员在命名变量时更为自由，并包含 “?! $/<=>” 等字符。例如，在 Lisp 中，一个将美元转换为日元的函数可能会用符号 $-to-yen 或 $->yen 命名，而在 Pascal 或 C 中，可能会使用 DollarsToYen、dollars_to_yen 或 dol2yen 这样的符号。这些命名约定有一些例外，将在出现时处理。

1.2 变量

我们已经了解了符号计算的一些基础知识。现在，我们继续讨论编程语言最重要的特征：能够根据其他对象定义新对象，并为这些对象命名以供将来使用。在这里，符号再次发挥着重要作用，它们用于命名变量。变量可以取一个值，该值可以是任何 Lisp 对象。给变量赋值的一种方法是使用 setf:

```
> (setf p '(John Q Public)) => (JOHN Q PUBLIC)
> p => (JOHN Q PUBLIC)
> (setf x 10) => 10
> (+ x x) => 20
> (+ x (length p)) => 13
```

在将值（John Q Public）赋值给名为 p 的变量后，我们可以引用名为 p 的值。同样，在为名为 x 的变量赋值后，我们也可以引用 x 和 p。

符号也用于在 Common Lisp 中命名函数。每个符号都可以用作变量或函数的名称，或两者兼而有之，尽管很少（也可能令人困惑）同时使用符号名称。例如，append 和 length 是命名函数但没有值作为变量的符号，pi 不命名函数，而是一个值为 3.1415926535897936（或其附近）的变量。

1.3 特殊形式

细心的读者会注意到 setf 违反了求值规则。我们之前说过，像 +、-和 append 这样的函数通过首先评估它们的所有参数，然后将函数应用于结果来工作。但是 setf 并不遵循这个规则，因为 setf 根本不是一个函数。相反，它是 Lisp 基本语法的一部分。除了原子和函数调用的语法，Lisp 还有少量的语法表达式。它们被称为特殊形式。它们与其他编程语言中的语句具有相同的目的，并且确实具有一些相同的语法标记，例如 if 和 loop。Lisp 的语法和其他语言有两个主要区别。首先，Lisp 的语法形式总是列表，其中第一个元素是少数特权符号之一。setf 是这些符号之一，因此 (setf x 10) 是一种特殊形式。其次，特殊形式是返回值的表达式。这与大多数语言中的语句形成鲜明对比，后者具有效果但不返回值。

在计算像 (setf x(+ 1 2)) 这样的表达式时，我们将符号 x 命名的变量设置为值 (+ 1 2)，即 3。如果 setf 是一个普通函数，我们将计算符号 x 和表达式 (+ 1 2)，并对这两个值进行处理，这根本不是我们想要的。setf 之所以被称为特殊形式，是因为它做了一些特殊的事情：如果它不存在，就不可能编写一个为变量赋值的函数。Lisp 的哲学是提供少量的特殊形式来完成原本无法完成的事情，然后期望用户将其他所有内容都作为函数编写。

特殊形式一词被混淆地用来指代像 setf 这样的符号和以它们开头的表达式，比如 (setf x 3)。在《Common LISPcraft》一书中，Wilensky 通过调用 setf 一个特殊函数并为 (setf x 3) 保留特殊形式来解决歧义。这个术语意味着 setf 只是另一个函数，但它是一个特殊的函数，因为它的第一个参数没有被求值。在 Lisp 主要是一种解释性语言的时代，这种观点是有道理的。现代观点认为，setf 不应被视为某种异常函数，而应被视为由编译器专门处理的特殊语法的标记。因此，特殊形式 (setf x (+ 2 1)) 应被视为 x = 2+1 的等价物。当存在混淆的风险时，我们将 setf 称为特殊形式运算符，将 (setf x 3) 称为特殊格式表达式。

原来引号只是另一种特殊形式的缩写。表达式'x 等价于（quote x），一个计算结果为 x 的特殊形式表达式。本章中使用的特殊形式运算符是：

- defun：define function
- defparameter：define special variable
- setf：set variable or field to new value
- let：bind local variable(s)

- case : choose one of several alternatives
- if : do one thing or another, depending on a test
- function(#') : refer to a function
- quote(') : introduce constant data

1.4 列表

目前为止我们看到了两个操作 list 的函数：append 和 length。下面是更多关于 list 的处理函数：

```
> p => (JOHN Q PUBLIC)

> (first p) => JOHN

> (rest p) => (Q PUBLIC)

> (second p) => Q

> (third p) => PUBLIC

> (fourth p) => NIL

> (length p) => 3
```

函数 first、second、third 和 fourth 的命名很恰当：first 返回列表的第一个元素，second 给出第二个元素，以此类推。函数 rest 并不那么明显；它的名字代表 “列表中第一个元素之后的其余部分”。符号 nil 和形式 () 完全同义；它们都是空列表的表示。nil 也用于表示 Lisp 中的 “false” 值。因此，(fourth p) 是 nil，因为 p 中没有第四个元素。请注意，列表不一定只由原子组成，也可以包含子列表作为元素：

```
> (setf x '((1st element) 2 (element 3) ((4)) 5))
((1ST ELEMENT) 2 (ELEMENT 3) ((4)) 5)

> (length x) => 5

> (first x) => (1ST ELEMENT)

> (second x) => 2

> (third x) => (ELEMENT 3)
```

```
> (fourth x) => ((4))

> (first (fourth x)) => (4)

> (first (first (fourth x))) => 4

> (fifth x) => 5

> (first x) => (1ST ELEMENT)

> (second (first x)) => ELEMENT
```

目前为止，我们知道了如何访问 list 的部分。下面是如何建立新的 list：

```
> p => (JOHN Q PUBLIC)

> (cons 'Mr p) => (MR JOHN Q PUBLIC)

> (cons (first p) (rest p)) => (JOHN Q PUBLIC)

> (setf town (list 'Anytown 'USA)) => (ANYTOWN USA)

> (list p 'of town 'may 'have 'already 'won!) =>
((JOHN Q PUBLIC) OF (ANYTOWN USA) MAY HAVE ALREADY WON!)

> (append p '(of) town '(may have already won!)) =>
(JOHN Q PUBLIC OF ANYTOWN USA MAY HAVE ALREADY WON!)

> p => (JOHN Q PUBLIC)
```

函数 cons 代表 “构造”。它接受一个元素和一个列表作为参数，并构造一个新的列表，其第一个是元素，其余的是原始列表。list 接受任意数量的元素作为参数，并返回一个按顺序包含这些元素的新列表。我们已经看到 append，它类似于 list；它接受任意数量的列表作为参数，并将它们全部附加在一起，形成一个大列表。因此，要附加的参数必须是列表，而要列出的参数可以是列表或原子。值得注意的是，这些函数会创建新的列表；他们不会修改旧的。当

我们说（append p q）时，效果是创建一个全新的列表，该列表以 p 中的相同元素开始，p 本身保持不变。

现在，让我们远离列表上的抽象函数，考虑一个简单的问题：给定一个人的名字以列表的形式，我们如何提取姓氏？对于（JOHN Q PUBLIC），我们可以只使用函数 third，但对于没有中间名的人来说，这是行不通的。Common Lisp 中有一个名为 last 的函数；也许这会奏效。我们可以尝试：

```
> (last p) => (PUBLIC)

> (first (last p)) => PUBLIC
```

事实证明，last 反常地返回最后一个元素的列表，而不是最后一个元件本身。因此，我们需要将 first 和 last 组合起来，以找出实际的最后一个要素。我们希望能够保存我们所做的工作，并给它一个适当的描述，比如姓氏。我们可以使用 setf 保存 p 的姓氏，但这无助于确定任何其他姓氏。相反，我们想定义一个新函数，该函数计算表示为列表的任何名称的姓氏。下一节正是这样做的。

1.5 定义新的函数

特殊形式 defun 代表 “define function”。在这里用于定义一个名为 last-name 的新函数:

```
(defun last-name (name)
"Select the last name from a name represented as a list
    ."
(first (last name)))
```

我们为新函数命名为 last-name。它有一个由单个参数组成的参数列表: (name)。这意味着函数接受一个参数，我们称之为 name。它还有一个文档字符串，说明函数的作用。这在任何计算中都没有使用，但文档字符串是调试和理解大型系统的关键工具。定义的主体是 (first (last name)))，这是我们之前用来挑选 p 的姓氏的。不同的是，在这里我们想挑选任何名字的姓氏，而不仅仅是特定名字 p 的姓氏。

一般来说，函数定义采用以下形式（其中文档字符串是可选的，所有其他部分都是必需的):

(defun function-name (parameter...) ”documentation string” function-body...)

函数名必须是一个符号，参数通常是符号（稍后会解释一些复杂情况），函数体由一个或多个表达式组成，在调用函数时对其进行求值。最后一个表达式作为函数调用的值返回。

一旦我们定义了 last-name，我们就可以像使用任何其他 Lisp 函数一样使用它:

```
> (last-name p) => PUBLIC

> (last-name '(Rear Admiral Grace Murray Hopper)) =>
  HOPPER

> (last-name '(Rex Morgan MD)) => MD

> (last-name '(Spot)) => SPOT

> (last-name '(Aristotle)) => ARISTOTLE
```

```
(defun first-name (name)
"Select the first name from a name represented as a list
    ."
(first name))

> p => (JOHN Q PUBLIC)

> (first-name p) => JOHN

> (first-name '(Wilma Flintstone)) => WILMA

> (setf names '((John Q Public) (Malcolm X)
(Admiral Grace Murray Hopper) (Spot)
(Aristotle) (A A Milne) (Z Z Top)
(Sir Larry Olivier) (Miss Scarlet))) =>

((JOHN Q PUBLIC) (MALCOLM X) (ADMIRAL GRACE MURRAY
    HOPPER)
(SPOT) (ARISTOTLE) (A A MILNE) (Z Z TOP) (SIR LARRY
    OLIVIER)
(MISS SCARLET))

> (first-name (first names)) => JOHN
```

1.6 使用函数

正如我们上面所做的那样，定义一个名称列表的一个好处是，它使测试我们的函数变得更加容易。考虑以下表达式，它可用于测试 last-name 函数：

```
> (mapcar #'last-name names)
(PUBLIC X HOPPER SPOT ARISTOTLE MILNE TOP OLIVIER
   SCARLET)
```

#' 符号从函数的名称映射到函数本身。这类似于'x 符号。内置函数 mapcar 传递了两个参数，一个函数和一个列表。它返回一个列表，该列表是通过在输入列表的每个元素上调用函数构建的。换句话说，上面的 mapcar 调用相当于：

```
(list (last-name (first names))
      (last-name (second names))
      (last-name (third names))
      ...)
```

mapcar 的名字来源于它将函数 “映射” 到每个参数上。car 部分的名字是指 Lisp 函数 car，first 的旧名字。cdr 是 rest 的旧名称。这些名称代表 “地址寄存器的内容” 和 “递减寄存器的内容 “，这是在 IBM 704 上首次实现 Lisp 时使用的指令。我相信你会同意 first 和 rest 是更好的名字，每当我们谈论列表时，它们都会被用来代替 car 和 cdr。然而，当我们考虑一对不被视为列表的值时，我们偶尔会继续使用 car 和 cdr。请注意，一些程序员仍然使用 car 和 cdr 作为列表。

下面是一些 mapcar 的例子：

```
> (mapcar #'- '(1 2 3 4)) => (-1 -2 -3 -4)

> (mapcar #'+ '(1 2 3 4) '(10 20 30 40)) => (11 22 33
   44)
```

最后一个例子表明，mapcar 可以传递三个参数，在这种情况下，第一个参数应该是一个二进制函数，它将应用于其他两个列表的相应元素。一般来说，mapcar 期望一个 n 元函数作为其第一个参数，然后是 n 个列表。它首先将函数应用于通过收集每个列表的第一个元素而获得的参数列表。然后，它将该函数应用于每个列表的第二个元素，依此类推，直到其中一个列表耗尽。它返回已计算的所有函数值的列表。

现在我们已经了解了 mapcar，让我们用它来测试 last-name 函数：

```
> (mapcar #'first-name names)
(JOHN MALCOLM ADMIRAL SPOT ARISTOTLE A Z SIR MISS)
```

我们可能会对这些结果感到失望。假设我们想要一个名字的版本，忽略了 Admiral 和 Miss 等头衔，得到了“真实”的名字。我们可以按以下方式进行：

```
(defparameter *titles*
  '(Mr Mrs Miss Ms Sir Madam Dr Admiral Major
    General)
  "A list of titles that can appear at the start
   of a name.")
```

我们引入了另一种新的特殊形式 defparameter，它定义了一个参数——一个在计算过程中不会改变的变量，但当我们想到要添加的新东西时，它可能会改变。defparameter 形式既给变量一个值，又可以在后续的函数定义中使用该变量。在这个例子中，我们练习了提供描述变量的文档字符串的选项。Lisp 程序员中广泛使用的一种惯例是，通过在两端用星号拼写特殊变量的名称来标记它们。这只是一个惯例；在 Lisp 中，星号只是另一个没有特定含义的字符。

接下来，我们为 first-name 给出了一个新的定义，它取代了之前的定义。这个定义说，如果名字的第一个单词是标题列表中的成员，那么我们想忽略这个单词，并返回名字中其余单词的名字。否则，我们使用第一个单词，就像以前一样。另一个内置函数 member 测试其第一个参数是否是作为第二个参数传递的列表中的元素。

特殊形式 if 具有形式 (if test then-part else-part)。在 Lisp 中执行条件测试有许多特殊形式；if 对于这个例子来说是最合适的。通过首先计算测试表达式来计算 if 表单。如果为真，则对 then-part 进行求值，并将其作为 If 表单的值返回；否则，将评估并返回 else-part。虽然一些语言坚持条件测试的值必须为真或假，但 Lisp 更宽容。该测试可以合法地评估任何值。只有值 nil 被认为是 false；所有其他值都被视为真。在下面的 first-name 定义中，如果名字的第一个元素在标题列表中，函数 member 将返回一个非 nil (因此为 true) 的值，如果不是，则将返回 nil (因而为 false)。虽然所有非空值都被认为是真的，但按照惯例，常数 t 通常用于表示真值。

```
(defun first-name (name)
  "Select the first name from a name represented
   as a list."
```

```
        (if (member (first name) *titles*)
        (first-name (rest name))
        (first name)))
```

当我们将新的 first-name 映射到名字列表上时，结果更令人鼓舞。此外，该函数通过跳过头衔来获得“正确”的结果。

```
> (mapcar #'first-name names)
(JOHN MALCOLM GRACE SPOT ARISTOTLE A Z LARRY SCARLET)

> (first-name '(Madam Major General Paula Jones))
PAULA
```

通过 tracing the execution of first-name，并查看传递给函数和从函数返回的值，我们可以看到这是如何工作的。特殊形式的跟踪和未跟踪用于此目的。

```
> (trace first-name)
(FIRST-NAME)

> (first-name '(John Q Public))
(1 ENTER FIRST-NAME: (JOHN Q PUBLIC))
(1 EXIT FIRST-NAME: JOHN)
JOHN

> (first-name '(Madam Major General Paula Jones)) =>
        (1 ENTER FIRST-NAME: (MADAM MAJOR GENERAL PAULA
            JONES))
                (2 ENTER FIRST-NAME: (MAJOR GENERAL
                    PAULA JONES))
                        (3 ENTER FIRST-NAME: (GENERAL
                            PAULA JONES))
                                (4 ENTER FIRST-NAME: (
                                    PAULA JONES))
                                (4 EXIT FIRST-NAME:
                                    PAULA)
                        (3 EXIT FIRST-NAME: PAULA)
```

```
                    (2 EXIT FIRST-NAME: PAULA)
          (1 EXIT FIRST-NAME: PAULA)
PAULA

> (untrace first-name) => (FIRST-NAME)

> (first-name '(Mr Blue Jeans)) => BLUE
```

函数 first-name 被称为递归的，因为它的定义包括对自身的调用。不熟悉递归概念的程序员有时会觉得它很神秘。但递归函数实际上与非递归函数没有什么不同。任何函数都需要为给定的输入返回正确的值。另一种看待这一要求的方法是将其分为两部分：函数必须返回一个值，并且不能返回任何不正确的值。这个由两部分组成的要求与第一个要求等效，但它使思考和设计功能定义变得更加容易。

1.7 高阶函数

Lisp 中的函数不仅可以被 “调用” 或作为参数，还可以像任何其他类型的对象一样被操纵。将另一个函数作为参数的函数称为高阶函数。mapcar 就是一个例子。为了演示高阶函数式编程，我们将定义一个名为 mappend 的新函数。它有两个参数，一个函数和一个列表。mappend 将函数映射到列表的每个元素上，并将所有结果附加在一起。

```
(defun mappend (fn the-list)
        "Apply fn to each element of list and append the
            results."
        (apply #'append (mapcar fn the-list)))
```

现在我们做一些实验，看看 apply 和 mappend 是如何工作的。第一个例子将加法函数应用于四个数字的列表。

```
> (apply #'+ '(1 2 3 4)) => 10
```

下一个示例将 append 应用于一个包含两个参数的列表，其中每个参数都是一个列表。如果参数不是列表，那将是一个错误。

```
> (apply #'append '((1 2 3) (a b c))) => (1 2 3 A B C)
```

现在我们定义一个新函数 self-and-double，并将其应用于各种参数。

```
> (defun self-and-double (x) (list x (+ x x)))

> (self-and-double 3) => (3 6)

> (apply #'self-and-double '(3)) => (3 6)

> (mapcar #'self-and-double '(1 10 300)) => ((1 2) (10
    20) (300 600))

> (mappend #'self-and-double '(1 10 300)) => (1 2 10 20
    300 600)
```

当 mapcar 被传递一个函数和一个包含三个参数的列表时，它总是返回一个包含 3 个值的列表。每个值都是对相应参数调用函数的结果。相比之下，当调用

mappend 时，它返回一个大列表，该列表等于 mapcar 将生成的所有附加值。使用不返回列表的函数调用 mappend 将是一个错误，因为 append 希望将列表作为其参数。

现在考虑以下问题：给定一个元素列表，返回一个由原始列表中的所有数字和这些数字的负数组成的列表。例如，给定列表 (1 2 3)，返回 (1 -1 2 -2 3 -3)。使用 mappend 作为组件可以很容易地解决这个问题：

```
(defun numbers-and-negations (input)
        "Given a list, return only the numbers and their
            negations."
        (mappend #'number-and-negation input))

(defun number-and-negation (x)
        "If x is a number, return a list of x and -x."
        (if (numberp x)
        (list x (- x))
        nil))

> (numbers-and-negations '(testing 1 2 3 test)) => (1 -1
    2 -2 3 -3)
```

下面显示的 mappend 的替代定义没有使用 mapcar；相反，它一次构建一个元素的列表：

```
(defun mappend (fn the-list)
        "Apply fn to each element of list and append the
            results."
        (if (null the-list)
                nil
                (append (funcall fn (first the-list))
                        (mappend fn (rest the-list)))))
```

funcall 类似于 apply；它也将函数作为其第一个参数，并将该函数应用于参数列表，但在 funcall 的情况下，参数是单独列出的：

```
> (funcall #'+ 2 3) => 5
```

```
> (apply #'+ '(2 3)) => 5

> (funcall #'+ '(2 3)) => *Error: (2 3) is not a number
    .*
```

分别等价于 (+ 2 3)、(+ 2 3) 和 (+ '(2 3))。

到目前为止，我们使用的每个函数都是在 Common Lisp 中预定义的，或者引入了一个 defun，它将函数与名称配对。也可以使用特殊语法 lambda 在不给函数命名的情况下引入函数（匿名函数）。

一般来说，lambda 表达式的形式是：

(lambda (parameters...) body...)

lambda 表达式只是函数的非原子名称，就像 append 是内置函数的原子名称一样。因此，它适合在函数调用的第一个位置使用，但如果我们想得到实际的函数，而不是它的名称，我们仍然必须使用 #' 符号。例如：

```
> ((lambda (x) (+ x 2)) 4) => 6

> (funcall #'(lambda (x) (+ x 2)) 4) => 6
```

为了理解这种区别，我们必须清楚如何在 Lisp 中计算表达式。正常的计算规则规定，通过查找符号所引用的变量的值来计算符号。因此，(+ x 2) 中的 x 是通过查找名为 x 的变量值来计算的。列表的计算方式有两种。如果列表的第一个元素是特殊形式运算符，则根据该特殊形式的语法规则对列表进行求值。否则，列表表示函数调用。第一个元素作为一个函数以独特的方式进行评估。这意味着它可以是符号或 lambda 表达式。在任何一种情况下，由第一个元素命名的函数都会应用于列表中其余元素的值。这些值由正常的评估规则确定。如果我们想在函数调用的第一个元素之外的位置引用一个函数，我们必须使用 #' 符号。否则，表达式将按正常求值规则求值，不会被视为函数。例如：

```
> append => *Error: APPEND is not a bound variable*

> (lambda (x) (+ x 2)) => *Error: LAMBDA is not a
    function*

> (mapcar #'(lambda (x) (+ x x))
'(1 2 3 4 5)) =>
```

```
(2 4 6 8 10)

> (mappend #'(lambda (l) (list l (reverse l)))
'((1 2 3) (a b c))) =>
((1 2 3) (3 2 1) (A B C) (C B A))
```

习惯于其他语言的程序员有时看不到 lambda 表达式的意义。lambda 表达式非常有用有两个原因。

首先，用多余的名称弄乱程序可能会很混乱。正如写 (a+b)*(c+d) 比发明像 temp1 和 temp2 这样的变量名来保存 a+b 和 c+d 更清晰一样，将函数定义为 lambda 表达式也更清晰，而不是为它发明一个名称。

其次，更重要的是，lambda 表达式使在运行时创建新函数成为可能。这是一种强大的技术，在大多数编程语言中是不可能的。这些运行时函数，称为闭包。

1.8 其他数据类型

到目前为止，我们只看到了四种 Lisp 对象：数字、符号、列表和函数。Lisp 实际上定义了大约 25 种不同类型的对象：向量、数组、结构、字符、流、哈希表等。在这一点上，我们将再介绍一个，字符串。正如您在下面看到的，字符串和数字一样，会自行计算。字符串主要用于打印消息，而符号用于它们与其他对象的关系以及命名变量。字符串的打印表示在每端都有一个双引号 (")。

```
> "a string" => "a string"

> (length "a string") => 8

> (length "") => 0
```

1.9 Lisp 求值规则

我们现在可以总结 Lisp 的求值规则。

- 每个表达式要么是 list，要么是 atom。
- 每个要计算的列表都是一个特殊的形式表达式 (special form expression) 或函数应用 (function application)。
- 特殊形式表达式被定义为第一个元素是特殊形式运算符的列表。表达式根据运算符的特殊求值规则进行求值。例如，setf 的评估规则是根据正常评估规则评估第二个参数，将第一个参数设置为该值，并将该值作为结果返回。defun 的规则是定义一个新函数，并返回函数的名称。引用的规则是返回第一个未求值的参数。符号'x 实际上是特殊形式表达式 (quote x) 的缩写。同样，符号 #'f 是特殊形式表达式 (function f) 的缩写。

```
'John    (quote John) => JOHN

(setf p 'John) => JOHN

(defun twice (x) (+ x x)) => TWICE

(if (= 2 3) (error) (+ 5 6)) => 11
```

- 函数应用 (function application) 的评估方法是首先评估参数（列表的其余部分），然后找到列表中第一个元素命名的函数并将其应用于评估的参数列表。

```
(+ 2 3) => 5
(- (+ 90 9) (+ 50 5 (length '(Pat Kim)))) => 42
```

- 每个原子 (atom) 要么是符号 (symbol)，要么是非符号 (nonsymbol)。
- 符号的计算结果为分配给该符号命名的变量的最新值。符号由字母组成，可能还有数字，很少还有标点符号。为了避免混淆，我们将使用主要由字母 a-z 和 “-” 字符组成的符号，只有少数例外。
- 非符号原子会自行计算。目前，数字和字符串是我们所知的唯一非符号原子。数字由数字组成，可能还有小数点和符号。也有科学记数法、有理

数和复数以及不同基数的数字的规定，但我们不会在这里详细描述。字符串两边用双引号分隔。

1.10 What Makes Lisp Different?

原文链接：What Makes Lisp Different?。

1.11 Exercises

- **Exercise 1.1** 定义一个 last-name 版本，以处理”Rex Morgan MD”、”Morton Downey，Jr.” 以及你能想到的任何其他例子。
- **Exercise 1.2** 编写一个函数进行指数运算，或将一个数字提升到整数幂。例如：(power 3 2) = 9。
- **Exercise 1.3** 编写一个函数,对表达式中的原子数进行计数。例如:(count-atoms ’(a (b) c)) = 3。请注意，这里有点模糊：(a nil c) 应该算作三个原子还是两个原子，因为它等价于 (a () c)?
- **Exercise 1.4** 编写一个函数，对一个表达式在另一个表达式中的任何位置出现的次数进行计数。示例：(count-anywhere ’a ’(a ((a) b) a))=>3。
- **Exercise 1.5** 编写一个函数来计算两个数字序列的点积，表示为列表。点积是通过将相应的元素相乘，然后将得到的乘积相加来计算的。例子：(dot-product ’(10 20) ’(3 4)) = 10 x 3 + 20 x 4 = 110

1.12 Answers

- **Answer 1.2**

```
(defun power (x n)
        "Power raises x to the nth power. N must be
            an integer >= 0.
        This executes in log n time, because of the
           check for even n."
        (cond ((= n 0) 1)
        ((evenp n) (expt (power x (/ n 2)) 2))
        (t (* x (power x (- n 1))))))
```

- **Answer 1.3**

```
(defun count-atoms (exp)
        "Return the total number of non-nil atoms in
            the expression."
        (cond ((null exp) 0)
        ((atom exp) 1)
        (t (+ (count-atoms (first exp))
        (count-atoms (rest exp))))))

(defun count-all-atoms (exp &optional (if-null 1))
        "Return the total number of atoms in the
           expression,
        counting nil as an atom only in non-tail
           position."
        (cond ((null exp) if-null)
        ((atom exp) 1)
        (t (+ (count-all-atoms (first exp) 1)
        (count-all-atoms (rest exp) 0)))))
```

- **Answer 1.4**

```
(defun count-anywhere (item tree)
```

```
        "Count the times item appears anywhere
           within tree."
        (cond ((eql item tree) 1)
        ((atom tree) 0)
        (t (+ (count-anywhere item (first tree))
        (count-anywhere item (rest tree))))))
```

- **Answer 1.5**

```
(defun dot-product (a b)
        "Compute the mathematical dot product of two
            vectors."
        (if (or (null a) (null b))
        0
        (+ (* (first a) (first b))
        (dot-product (rest a) (rest b)))))

(defun dot-product (a b)
        "Compute the mathematical dot product of two
            vectors."
        (let ((sum 0))
        (dotimes (i (length a))
        (incf sum (* (elt a i) (elt b i))))
        sum))

(defun dot-product (a b)
        "Compute the mathematical dot product of two
            vectors."
        (apply #'+ (mapcar #'* a b)))
```

第二章　一个简单的 Lisp 程序

2.1　A Grammar for a Subset of English

本章我们将开发的程序生成随机的英语句子。以下是一小部分英语的简单
语法：Sentence => Noun-Phrase + Verb-Phrase Noun-Phrase => Article + Noun Verb-Phr