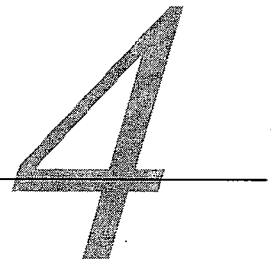
第4章 语法和语义



在一阵旋风似的介绍之后，我们将坐下来用几章的篇幅对已经用到的那些特性来一次更加系统化的介绍。我将从对Lisp的基本语法和语义元素的概述开始，这意味着我必须先回答下面这个问题……

4.1 括号里都可以有什么

Lisp的语法和源自Algol的语言在语法上有很多不同。两者特征最明显的区别在于前者大量使用了括号和前缀表示法。说来也怪，大量的追随者都喜欢这样的语法。Lisp的反对者们总是将这种语法描述成“奇怪的”和“讨厌的”，他们说Lisp就是“大量不合理的多余括号”（Lots of Irritating Superfluous Parentheses）的简称；而Lisp的追随者则认为，Lisp的语法是它的最大优势。为什么两个团体之间会有如此对立的见解呢？

本章不可能完整地描述Lisp的语法，因为我还没有彻底地解释Lisp的宏，但我可以从一些宝贵的历史经验来说明保持开放的思想是值得的：John McCarthy首次发明Lisp时，曾想过实现一种类Algol的语法，他称之为M-表达式。尽管如此，他却从未实现这一点。他在自己的文章*History of Lisp*[①]中对此加以解释。

> 这个精确定义M-表达式以及将其编译或至少转译成S-表达式的工程，既没有完成也没有明确放弃，它只不过是被无限期地推迟了。（相比S-表达式）更加偏爱类FORTRAN或类Algol表示法的新一代程序员也许会最终实现它。

换句话说，过去45年以来，实际使用Lisp的人们已经喜欢上了这种语法，并且发现它能使该语言变得更为强大。接下来的几章将告诉你为什么会这样。

4.2 打开黑箱

在介绍Lisp的语法和语义之前，值得花一点时间来看看它们的定义及其与许多其他语言的语法所存在的差异。

---

① http://www-formal.standford.edu/jmc/history/lisp/node3.html。

在大多数编程语言里，语言的处理器（无论是解释器还是编译器）的操作方式都类似于黑箱作业：一系列表示程序文本的字符被送进黑箱，然后它（取决于是解释器还是编译器）要么执行预想行为，要么产生一个编译版本的程序并在运行时执行这些行为。

当然，在黑箱的内部，语言的处理器通常划分成子系统，各自负责一部分将程序文本转换成具体行为或目标代码的任务。一个典型的任务划分思路是将处理器分成三个阶段，每个阶段为下一个阶段提供内容：一个词法分析器将字符流分拆成语元并将其送进一个解析器，解析器再根据该语言的语法在程序中构建一个表达式的树形表示。这棵树被称为抽象语法树，它随即被送进一个求值器，求值器要么直接解释它，要么将其编译成某种其他语言（比如机器码）。由于语言处理器是一种黑箱，所以处理器所使用的包括语元和抽象语法树在内的数据结构，只对语言的实现者有用。

而在Common Lisp中，分工则有点不同，无论从实现者的角度还是从语言定义方式的角度上来说都是这样。与一个从文本到程序行为一步到位的单一黑箱有所不同的是，Common Lisp定义了两个黑箱，一个将文本转化成Lisp对象，而另一个则用这些对象来实现语言的语义。前一个箱子称为读取器，后一个称为求值器。[①]

每个黑箱都定义了一个语法层面。读取器定义了字符串如何被转换为我们称之为S-表达式[②]的Lisp对象。由于S-表达式语法可适用于由任意对象及其他列表所组成的列表，因此S-表达式可用来表达任意树形表达式，这跟由非Lisp语言的语法解析器所生成的抽象语法树非常相似。

求值器随后定义了一种构建在S-表达式之上的Lisp形式（form）的语法。并非所有的S-表达式都是合法的Lisp形式，更不用说所有字符序列都是合法的S-表达式了。举个例子，`(foo 1 2)`和`("foo" 1 2)`都是S-表达式，但只有前者才是一个Lisp形式，因为一个以字符串开始的列表对于Lisp形式来说是没有意义的。

这样的黑箱划分方法带来了一系列后果。其中之一是可将S-表达式（正如第3章那样）用作一种可暴露的数据格式来表达源代码之外的数据，用`READ`来读取它再用`PRINT`来打印它。[③]另一个后果则在于，由于语言的语义是用对象树而非字符串定义而成的，因此很容易使用语言本身而非文本形式来生成代码。完全从手工生成代码的好处很有限——构造列表和构造字符串的工作量大致相同。尽管如此，真正的优势在于可通过处理现有数据来生成代码。这就是Lisp宏的本意，我将在后续章节详加论述。目前我将集中在Common Lisp所定义的两个层面上：读取器所理解的S-表达式语法以及求值器所理解的Lisp表达式语法。

---

① 和任何语言的实现者一样，Lisp实现者们有许多方式可以实现一个求值器，从一个解释那些直接送进求值器的对象的“纯”解释器，到一个将对象转化成机器码并执行的编译器。在这两者之间，还有些实现将输入编译成类似虚拟机字节码这样的中间形式，然后解释执行字节码。近年来，多数Common Lisp实现都使用某种形式的编译，甚至在运行期求值代码的时候也是这样。

② 术语“S-表达式”有时代表文本表示，而有时则代表从文本表示中读取到的对象。通常从上下文中可以清楚地判断它的含义，或者怎样理解都无所谓。

③ 并非所有的Lisp对象都可以被写成一种可以被读回来的形式，但任何你可以用**READ**读取的东西都可以被**PRINT**打印成可读的形式。

4.3 S-表达式

S-表达式的基本元素是列表（list）和原子（atom）。列表由括号所包围，并可包含任何数量的由空格所分隔的元素。原子是所有其他内容。[1]列表元素本身也可以是S-表达式（换句话说，也就是原子或嵌套的列表）。注释从技术角度来讲不是S-表达式，它们以分号开始，直到一行的结尾，本质上将被当作空白来处理。

这就差不多了。列表在句法上十分简单，你需要知道的句法规则只有那些用来形成不同类型原子的规则了。在本节里我将描述几种常用原子类型的规则，这些原子包括：数字、字符串和名字。之后我将说明由这些元素所组成的S-表达式是如何作为Lisp形式求值的。

数字的表示方法很简单。任何数位的序列将被读取为一个数字，它们可能有一个前缀标识（+或-），还可能会有一个十进制点（.）或者斜杠（/），或是以一个指数标记结尾。例如：

```
123       ; 整数一百二十三
3/7       ; 比值七分之三
1.0       ; 默认精度的浮点数一
1.0e0     ; 同一个浮点数的另一种写法
1.0d0     ; 双精度的浮点数一
1.0e-4    ; 等价于万分之一的浮点数
+42       ; 整数四十二
-42       ; 整数负四十二
-1/4      ; 比值负四分之一
-2/8      ; 负四分之一的另一种写法
246/2     ; 整数一百二十三的另一种写法
```

这些不同的形式代表着不同类型的数字：整数、比值和浮点数。Lisp也支持复数，但它们有其自己的表示法，第10章将予以介绍。

从这些示例可见，你可以用多种方式来表示同一个数字。但无论怎样书写，所有的有理数（整数和比值）在内部都被表示成“简化”形式。换句话说，表示－2/8或246/2的对象跟表示－1/4或123的对象并没有什么不同。与此相似，1.0和1.0e0也只是同一个数字的不同写法而已。但另一方面，1.0、1.0d0和1则可能会代表不同的对象，因为不同精度的浮点数和整数都是不同类型的对象。我将在第10章详细讨论不同类型数字的特征。

正如在前面章节所看到的那样，字符串是由双引号所包围着的。在字符串中，一个反斜杠会转义接下来的任意字符，使其被包含在字符串里。两个在字符串中必须被转义的字符是双引号和反斜杠本身。所有其他的字符无需转义即可被包含在一个字符串里，无论它们在字符串之外有何含义。下面是一些字符串的示例：

```
"foo"     ; 含有f、o和o的字符串
"fo\o"    ; 同一个字符串
"fo\\o"   ; 含有f、o、\和o的字符串
"fo\"o"   ; 含有f、o、"和o的字符串
```

Lisp中所使用的名字，诸如**FORMAT**、hello-world和*db*均由称为符号的对象所表示。读

① 空列表即()，也可写成**NIL**，既是原子也是列表。

取器对于一个给定名的用途毫不知情——无论其究竟用作变量名、函数名还是其他什么东西。读取器只是读取字符序列并构造出此名所代表的对象。[1]几乎任何字符都可以出现在一个名字里，不过空白字符不可以，因为列表的元素是用空格来分隔的。数位也可以出现在名字里，只要整个名字不被解释成一个数字。类似地，名字可以包含句点，但读取器无法读取一个只由句点组成的名字。有十个字符被用于其他句法目的而不能出现在名字里，它们是：开括号和闭括号、双引号和单引号、反引号、逗号、冒号、分号、反斜杠以及竖线。而就算是这些字符，如果你愿意的话，它们也可以成为名字的一部分，只需将它们用反斜杠进行转义，或是将含有需要转义的字符名字用竖线包起来。

这种读取器将名字转化成符号对象的方式有两个重要特征，一个是它如何处理名字中的字母大小写，另一个是它如何确保相同的名字总被读取成相同的符号。当读取名字时，读取器将所有名字中未转义的字符都转化成它们等价的大写形式。这样，读取器将把foo、Foo和FOO都读成同一个符号FOO。但\f\o\o和|foo|将都被读成foo，这是和符号FOO不同的另一个对象。这就是为什么在REPL中定义一个函数时，它的名字会被打印成大写形式的原因。近年来，标准的编码风格是将代码全部写成小写形式，然后让读取器将名字转化成大写。[2]

为了确保同一个文本名字总是被读取成相同的符号，读取器保留这个符号——在已读取名字并将其全部转化成大写形式以后，读取器在一个称为包（package）的表里查询带有相同名字的已有符号。如果无法找到，则将创建一个新符号并添加到表里。否则就将返回已在表中的那个符号。这样，无论在任何地方，同样的名字出现在任何S-表达式里，都会用同一个对象去表示它。[3]

因为在Lisp中名字可以包含比源自Algol的语言更多的字符，故而命名约定在Lisp中也相应地有所不同，诸如可以使用像hello-world这类带有连字符的名字。另一个重要约定是全局变量名字在开始和结尾处带有“*”。类似地，常量名都以“+”开始和结尾。而某些程序员则将特别底层的函数名前加%甚至%%。语言标准所定义的名字只使用字母表字符（A-Z）外加*、+、-、/、1、2、<、=、>以及&。

只用列表、数字、字符串和符号就可以描述很大一部分的Lisp程序了。其他规则描述了字面向量、单个字符和数组的标识，我将在第10章和第11章里谈及相关的数据类型时再讲解它们。目前的关键是要理解怎样用数字、字符串和由符号借助括号所组成的列表来构建S-表达式，从而表示任意的树状对象。下面是一些简单的例子。

```
x               ; 符号X
()              ; 空列表
(1 2 3)         ; 三个数字所组成的列表
("foo" "bar")   ; 两个字符串所组成的列表
(x y z)         ; 三个符号所组成的列表
```

① 事实上，正如你后面将要看到的，名字从本质上是一种独立的概念。根据不同的上下文，你可以使用相同的名字来同时引用一个变量或者函数，更不用说其他几种可能性了。

② 事实上，读取器的大小写转化行为是可以定制的，但是要想理解何时以及怎样改变它，则需要在相关的名字、符号和其他程序元素中，相对我已经涉及的内容做更加深入的讨论。

③ 我将在第21章里讨论符号和包之间关系的更多细节。

```
(x 1 "foo")    ; 由一个符号、一个数字和一个字符串所组成的列表
(+ (* 2 3) 4) ; 由一个符号、一个列表和一个数字所组成的列表
```

下面这种四元素列表就稍显复杂了，它含有两个符号、空列表以及另一个列表——其本身又含有两个符号和一个字符串：

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

4.4　作为 Lisp 形式的 S-表达式

在读取器把大量文本转化为S-表达式以后，这些S-表达式随后可以作为Lisp形式被求值。或者只有它们中的一些可以——并不是每个读取器可读的S-表达式都有必要作为Lisp形式来求值的，Common Lisp的求值规则定义了第二层的语法来检测哪种S-表达式可看作Lisp形式[①]。这一层面的句法规则相当简单。任何原子（非列表或空列表）都是一个合法的Lisp形式，正如任何以符号为首元素的列表那样。[②]

当然，Lisp形式的有趣之处不在于其语法，而在于它们被求值的方式。为了便于讨论，你可以将求值器想象成一个函数，它接受一个句法良好定义的Lisp形式作为参数并返回一个值，我们称之为这个形式的值。当然，当求值器是一个编译器时，情况会更加简化一些——在那种情况下，求值器被给定一个表达式，然后生成在其运行时可以计算出相应值的代码。但是这种简化可以让我们从不同类型的Lisp形式如何被这个假想的函数求值的角度来描述Common Lisp的语义。

作为最简单的Lisp形式，原子可以被分成两个类别：符号和所有其他内容。符号在作为Lisp形式被求值时会被视为一个变量名，并且会被求值为该变量的当前值。[③]第6章将讨论变量是如何得到这个值的。你也需要注意，某些特定的“变量”其实是编程领域的早期产物“常值变量”(constant variable)。例如，符号`PI`命名了一个常值变量，其值是最接近数学常量π的浮点数。

所有其他的原子，包括你已经见过的数字和字符串，都是自求值对象。这意味着当这样的表达式被传递给假想函数时，它会简单地直接返回自身。第2章在REPL里键入的`10`和`"hello, world"`实际上就是自求值对象的例子了。

把符号变成自求值对象也是可能的——它们所命名的变量可以被赋值成符号本身的值。两个以这种方式定义的常量是`T`和`NIL`，即所谓的真值和假值。我将在4.8节里讨论它们作为布尔值的角色。

另一类自求值符号是*关键字符号*——以名字冒号开始的符号。当读取器保留这样一个名字时，它会自动定义一个以此命名的常值变量并以该符号作为其值。

---

① 当然如同其他语言那样，其他层面的纠错也存在于Lisp中。例如，从读取`(foo 1 2)`得到的S-表达式在句法上是良好定义的，但是只有当`foo`是一个函数或宏的名字时，它才可以被求值。

② 另一种很少用到的Lisp形式的类型是那种第一个元素是lambda表达式的列表。我将在第5章里讨论这类形式。

③ 存在另一种可能性——可以定义出符号宏（symbol macro），它可被稍有不同地求值。我们无需理会它们。

当我们开始考虑列表的求值方式时，事情变得更加有趣了。所有合法的列表形式均以一个符号开始，但是有三种类型的列表形式，它们会以三种相当不同的方式进行求值。为了确定一个给定的列表是哪种形式，求值器必须检测列表开始处的那个符号是一个函数、宏还是特殊操作符的名字。如果该符号尚未定义，比如说当你正在编译一段含有对尚未定义函数的引用的代码时，它会被假设成一个函数的名字。[1]我将把这三种类型的形式称为*函数调用形式*（function call form）、*宏形式*（macro form）和*特殊形式*（special form）。

4.5 函数调用

函数调用形式的求值规则很简单，对以Lisp形式存在的列表其余元素进行求值并将结果传递到命名函数中。这一规则明显的有着一些附加的句法限制在函数调用形式上：除第一个以外，所有的列表元素它们自身必须是一个形态良好的Lisp形式。换句话说，函数调用形式的基本语法应如下所示，其中每个参数本身也是一个Lisp形式：

```
(function-name argument*)
```

这样下面这个表达式在求值时将首先求值1，再求值2，然后将得到的值传给+函数，再返回3：

```
(+ 1 2)
```

像下面这样更复杂的表达式也采用相似的求值方法，不过在求值参数`(+ 1 2)`和`(- 3 4)`时需要先对它们的参数求值，然后再对它们应用相应的函数：

```
(* (+ 1 2) (- 3 4))
```

最后，值3和-1被传递到*函数里，从而得到-3。

正如这些例子所显示的这样，许多在其他语言中需用特殊语法来处理的事务在Lisp中都可用函数来处理。Lisp的这种设计对于保持其语法正则化很有帮助。

4.6 特殊操作符

然而，并非所有的操作都可定义成函数。由于一个函数的所有参数在函数被调用之前都将被求值，因此无法写出一个类似第3章里用到的`IF`操作符那样的函数。为了说明这点，可以假设有下面这种形式：

```
(if x (format t "yes") (format t "no"))
```

如果`IF`是一个函数，那么求值器将从左到右依次对其参数表达式求值。符号`x`将被作为产生某个值的变量来求值，然后`(format t "yes")`将被当成一个函数调用来求值，在向标准输出打

① 在Common Lisp中一个符号可以同时为操作符（函数、宏或特殊操作符）和变量命名。这是Common Lisp和Scheme的主要区别之一。这一区别有时被描述成Common Lisp是一种Lisp-2而Scheme则是Lisp-1——一个Lisp-2有两个命名空间，一个用于操作符而另一个用于变量，但一个Lisp-1仅使用单一的命名空间。两种选择都有其各自的优点，而拥护者们总是在无休止地争论哪种更好。

印“yes”以后得到NIL。接下来(format t "no")将被求值，打印出“no”同时也得到NIL。只有当所有三个表达式都被求值以后，它们的结果值才被传递给IF，而这时已经无法控制两个FORMAT表达式中的哪一个会被求值了。

为了解决这个问题，Common Lisp定义了一些特殊操作符，IF就是其中之一，它们可以做到函数无法做到的事情。它们总共有25个，但只有很少一部分直接用于日常编程。[①]

当列表的第一个元素是一个由特殊操作符所命名的符号时，表达式的其余部分将按照该操作符的规则进行求值。

IF的规则相当简单：求值第一个表达式。如果得到非NIL，那么求值下一个表达式并返回它的值。否则，返回第三个表达式的求值，或者如果第三个表达式被省略的话，返回NIL。换句话说，一个IF表达式的基本形式是像下面这样：

```
(if test-form then-form [ else-form ])
```

其中*test-form*将总是被求值，然后要么是*then-form*要么是*else-form*。

一个更简单的特殊操作符是QUOTE，它接受一个单一表达式作为其“参数”并简单地返回它，不经求值。例如，下面的表达式求值得到列表(+ 1 2)，而不是值3:

```
(quote (+ 1 2))
```

这个列表没有什么特别的，你可以像用LIST函数所创建的任何列表那样处理它。[②]

QUOTE被用得相当普遍，以至于读取器中内置了一个它的特别语法形式。除了能像下面这样写之外：

```
(quote (+ 1 2))
```

也可以这样写：

```
'(+ 1 2)
```

该语法是读取器所理解的S-表达式语法的小扩展。从求值器的观点来看，这两个表达式看起来是一样的：一个首元素为符号QUOTE并且次元素是列表(+ 1 2)的列表。[③]

一般来说，特殊操作符所实现的语言特性需要求值器作出某些特殊处理。例如，有些的操作符修改了其他形式的求值环境。其中之一是LET，也是我将在第6章详细讨论的特殊操作符，它用来创建新的变量绑定。下面的形式求值得到10，因为在第二个x的求值环境中，它是由LET赋值为10的变量名：

```
(let ((x 10)) x)
```

---

① 其他操作符提供了有用但有时晦涩难懂的特性。我将在它们所支持的特性里讨论它们。

② 确实有一点区别——像引用列表这样的字面对象，也包括双引号里的字符串、字面数组和向量（以后你将看到它的语法），一定不能被修改。一般而言，任何你打算修改的列表都应该用LIST来创建。

③ 该语法是读取宏（reader macro）的一个例子。读取宏可以修改读取器用来将文本转化成Lisp对象的语法。事实上，定义你自己的读取宏也是有可能的，但这是该语言的一种很少被用到的机制。多数Lisp程序员提到该语言的语法扩展时，他们指的是正规的宏，我将在稍后讨论它们。

4.7 宏

虽然特殊操作符以超越了函数调用所能表达的方式扩展了Common Lisp语法，但特殊操作符的数量在语言标准中是固定的。然而宏却能提供给语言用户一种语法扩展方式。如同在第3章里看到的那样，宏是一个以S-表达式为其参数的函数，并返回一个Lisp形式，然后对其求值并用该值取代宏形式。宏形式的求值过程包括两个阶段：首先，宏形式的元素不经求值即被传递到宏函数里；其次，由宏函数所返回的形式（称其为*展开式*（expansion））按照正常的求值规则进行求值。

重点在于要清醒地认识到一个宏形式求值的两个阶段。当你在REPL中输入表达式时很容易忘记这一点，因为两个阶段相继发生并且后一阶段的值被立即返回了。但是当Lisp代码被编译时，这两个阶段所发生时间却是完全不同的，因此关键在于对于何时发生什么要保持清醒。例如，当使用函数**COMPILE-FILE**来编译整个源代码文件时，文件中所有宏形式将被递归展开，直到代码中只含有函数调用形式和特殊形式。这些无宏的代码随后被编译成一个FASL文件——**LOAD**函数知道如何去加载它。但编译后的代码直到文件被加载时才会被执行。因为宏在编译期会生成其展开式，它们可以用相对大量的工作来生成其展开式，而无需在文件被加载时或是当文件中定义的函数被调用时再付出额外的代价。

由于求值器在将宏形式传递给宏函数之前并不对它们求值，因此它们不需要是格式良好的Lisp形式。每个宏都为其宏形式中的符号表达式指定了一种含义，用以指明宏将如何使用它们生成展开式。换句话说，每个宏都定义了它们自己的局部语法。例如，第3章的backwards宏就定义了一种语法，合法的backwards形式列表必须与合法的Lisp形式列表反序。

我会在本书中经常地提到宏。眼下最为重要的是认识到宏，虽然跟函数调用在句法上相似，但却有着相当不同的用途，并提供了一种嵌入编译器的钩子。[①]

---

① 对于缺少或没有Lisp宏使用经验的人们，以及那些甚至被C的预处理器所荼毒过的人们来说，当他们意识到宏调用跟正常函数调用一样时可能会紧张。但这在实践中却并不是一个问题，原因如下。一方面是因为宏形式通常在格式上与函数调用不同。例如，你会写成这样：

```
(dolist (x foo)
  (print x))
```

而不是这样：

```
(dolist (x foo) (print x))
```

也不会是这样：

```
(dolist (x foo)
        (print x))
```

后面两种形式使**DOLIST**显得像是一个函数。一个好的Lisp开发环境会自动正确地格式化宏调用，甚至对于用户定义的宏也是如此。

就算一个**DOLIST**形式被写在了单行里，也有几条线索可以说明它是一个宏。其一，表达式(x foo)只有在x是一个函数或宏的名字时本身才有意义。将这一现象和随后作为变量出现的x联系起来，就会很容易发现**DOLIST**是一个正在创建名为x的变量绑定的宏。命名约定也有帮助——通常作为宏的循环结构都带有一个以do开始的名字。

4.8 真、假和等价

最后两个需要了解的基本知识是Common Lisp对于真和假的表示法以及两个Lisp对象“等价”的含义。真和假的含义在这里是直截了当的：符号NIL是唯一的假值，其他所有的都是真值。符号T是标准的真值，可用于需要返回一个非NIL值却又没有其他值可用的情况。关于NIL，唯一麻烦的一点是，它是唯一一个既是原子又是列表的对象：除了用来表示假以外，它还用来表示空列表。[1]这种NIL和空列表的等价性被内置在读取器之中：如果读取器看到了()，它将作为符号NIL读取它。它们是完全可以互换的。并且如同我前面提到的那样，因为NIL是一个以符号NIL作为其值的常值变量名，所以表达式nil、()、'nil以及'()求值结果是相同的——未引用形式将被看成是对值为符号NIL的常值变量的引用来进行求值，而在引用形式中，QUOTE特殊操作符将会直接求解出符号NIL。基于同样的理由，t和't的求值结果也完全相同：符号T。

使用诸如“完全相同”这样的术语理所当然会引申出两个值“等价”的这个问题上。在后面的章节里将会看到，Common Lisp提供了许多特定于类型的等价谓词：=用来比较数字，CHAR=用来比较字符，依此类推。本节将讨论四个“通用”等价谓词——这些函数可以被传入任何两个Lisp对象，然后当它们等价时返回真，否则返回假。按照介绍的顺序，它们是EQ、EQL、EQUAL和EQUALP。

EQ用来测试“对象标识”，只有当两个对象相同时才是EQ等价的。不幸的是，数字和字符的对象标识取决于这些数据类型在特定Lisp平台上实现的方式。带有相同值的两个数字或字符可能会被EQ认为是等价的也可能会是不等价的。语言实现者有足够的空间将表达式(eq x x)合法地求值为真或假，这种情况更多会发生在x恰好为数字或字符时。

因此不该将EQ用于比较可能是数字或字符的值上。在个别实现的特定值上，它可能会以一种可预测的方式工作，但如果切换了语言实现，则它将不保证以相同的方式工作。切换实现可能意味着只是简单地把实现升级到一个新版本——而如果Lisp实现者改变了表示数字或字符的方式，那么EQ的行为也将很有可能发生改变。

因此，Common Lisp定义了EQL来获得与EQ相似的行为，除此之外，它也可以保证当相同类型的两个对象表示相同的数字或字符值时，它们是等价的。因此，(eql 1 1)能被确保是真。而(eql 1 1.0)则被确保是假，因为整数值1和浮点数1.0是不同类型的对象。

关于何时使用EQ以及何时使用EQL，这里有两种观点：“凡有可能就用EQ”阵营认为，当知道不存在比较数字或字符时就应该使用EQ，他们认为(a)这是一种说明你不在比较数字或字符的方式，以及(b)因为EQ不需要检查它的参数是否为数字或字符，所以它将会稍微更有效率。

但“总是使用EQL”阵营则认为永远不该使用EQ，因为(a)每次有人（包括你在内）阅读你的代码时，看到了一个EQ，就得停下来检查它是否被正确使用了（换句话说，它永远不该被用来比

① 使用空列表作为假是Lisp作为列表处理语言所留下的遗产，就好比是C语言使用整数0作为假是其作为字节处理语言所留下的遗产。但并非所有的Lisp都以相同的方式处理布尔值。Common Lisp和Scheme的许多细微差异中的另一个就是Scheme使用一个单独的假值#f，#f无论跟符号nil还是空列表都是不同的值，并且即便是nil和空列表，两者也是不同的。

较数字或字符），这样就会丢失潜在获得的代码清晰性，以及(b)EQ和EQL之间的效率差异相比于实际的性能瓶颈来说微不足道。

本书中的代码是以“总是使用EQL”风格写成的。[1]

另外两个等价谓词EQUAL和EQUALP更为通用，因为它们可以操作在所有类型的对象上，但又不像EQ或EQL那样基础。它们每个都定义了相比EQL稍微宽松一些的等价性，允许认为不同的对象是等价的。除了它们曾经被过去的Lisp程序员认为是有用的之外，由这些函数所实现的特殊含义的等价性没有什么特别的。如果这些谓词不能满足需要，也可以自己定义谓词函数，以自己所需方式来比较不同类型对象。

EQUAL相比EQL的宽松之处在于，它将在递归上具有相同结构和内容的列表视为等价。EQUAL也认为含有相同字符的字符串是等价的，它对于位向量和路径名也定义了比EQL更加宽松的等价性，我将在未来的章节里讨论这两个数据类型。对于所有其他类型，它回退到EQL的水平上。

EQUALP甚至更加宽松之外，它和EQUAL是相似的。它在考察两个含有相同字符的字符串的等价性时忽略了大小写的区别。它还认为如果两个字符只在大小写上有区别，那么它们就是等价的。只要数字表示相同数学意义上的值，它们在EQUALP下面就是等价的。因此，(equalp 1 1.0)是真的。由EQUALP等价的元素所组成的列表也是EQUALP等价的。同样地，带有EQUALP元素的数组也是EQUALP等价的。和EQUAL一样，还有一些我尚未涉及的其他数据类型，EQUALP可认为两个对象是等价的，但EQL或EQUAL则不会。对于所有的其他数据类型，EQUALP回退到EQL的水平上。

4.9 格式化 Lisp 代码

严格说起来，代码格式化既不是句法层面也不是语法层面上的事情，好的格式化对于阅读和编写流利而又地道的代码而言非常重要。格式化Lisp代码的关键在于正确缩进它。这一缩进应当反映出代码结构，这样就不需要通过数括号来查看代码究竟写到哪儿了。一般而言，每一个新的嵌套层次都需要多缩进一点儿，并且如果折行是必需的，位于同一个嵌套层次的项应当按行对齐。这样，一个需要跨越多行的函数调用可能会被写成这样：

```
(some-function arg-with-a-long-name
               another-arg-with-an-even-longer-name)
```

那些实现控制结构的宏和特殊形式在缩进上稍有不同：“主体”元素相对于整个形式的开括号缩进两个空格。就像这样：

```
(defun print-list (list)
  (dolist (i list)
    (format t "item: ~a~%" i)))
```

尽管如此，但也不需要太担心这些规则，因为一个像SLIME这样的优秀Lisp环境将会帮你做

① 甚至是语言标准，在关于EQ或EQL哪一个应当被使用方面也有一点歧义，对象标识（object identity）是由EQ定义的，但是标准在谈论对象时定义了术语“相同”来表达EQL，除非明确提到了另外的谓词。因此，如果你想要在技术上100%正确的话，你可以说(- 3 2)和(- 4 3)求值到“相同”（same）的对象而不是“同样”（identical）的对象。不得不承认，这个问题有些无聊。

到这点。事实上，Lisp正则语法的优势之一就在于，它可以让诸如编辑器这样的软件相对容易地知道应当如何缩进。由于缩进的本意是反映代码的结构，而结构是由括号来标记的，因此很容易让编辑器帮你缩进代码。

在SLIME中，在每行开始处按下Tab键将导致该行被适当地缩进，或者也可以通过将光标放置在一个开括号上并键入C-M-q来重新缩进整个表达式。或者还可以在函数内部的任何位置通过键入C-c M-q来重新缩进整个函数体。

的确，有经验的Lisp程序员们倾向于依赖编辑器来自动处理缩进，这样不但可以确保代码美观，还可以检测笔误：一旦熟悉了代码该如何缩进，那么一个错误放置的括号就将立即由于编辑器所给出的奇怪缩进而被发现。例如，假设要编写一个如下所示的函数：

```
(defun foo ()
  (if (test)
    (do-one-thing)
    (do-another-thing)))
```

而你不小心忘记了test后面的闭括号。如果不数括号，那么很可能就会在DEFUN形式的结尾处添加一个额外的括号，从而得到下面的代码：

```
(defun foo ()
  (if (test
    (do-one-thing)
    (do-another-thing))))
```

尽管如此，如果一直都在每行的开始处按Tab来缩进的话，就不会得到这样的代码。相反，将得到如下的代码：

```
(defun foo ()
  (if (test
       (do-one-thing)
       (do-another-thing))))
```

看到then和else子句被缩进到了条件语句的位置，而不是仅仅相对于IF稍微缩进了一点，你立即就能看出有错误发生。

另一个重要的格式化规则是，闭括号总是位于与它们所闭合的列表最后一个元素相同的行。也就是说，不要写成这样：

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)
  )
)
```

而一定要写成这样：

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)))
```

结尾处的)))可能看起来令人生畏，但是一旦代码缩进正确，那么括号的意义就不存在了，没有

必要通过将它们分散在多行来加以突出。

最后，注释应该根据其适用范围被前置一到四个分号，如同下面所说明的：

```
;;;; 四个分号用于文件头注释。

;;;带有三个分号的注释将通常作为段落注释应用到接下来的一大段代码上。

(defun foo (x)
  (dotimes (i x)
    ;; 两个分号说明该注释应用于接下来的代码上。
    ;; 注意，该注释与其所应用的代码具有相同的缩进。
    (some-function-call)
    (another i)                    ; 本注释仅用于此行
    (and-another)                  ; 这个也是一样
    (baz)))
```

现在可以开始了解Lisp的主要程序构造的更多细节了：函数、变量和宏。下一章先来看看函数。