第10章 数字、字符和字符串

尽管函数、变量、宏和25个特殊操作符组成了语言本身的基本构造单元，但程序的构造单元则是你所使用的数据结构。正如Fred Brooks在《人月神话》里提到的，“数据的表现形式是编程的根本。”[1]

Common Lisp为现代语言中常见的大多数数据类型都提供了内置支持：数字（整数、浮点数和复数）、字符 、字符串、数组（包括多维数组）、列表、哈希表、输入和输出流以及一种可移植地表示文件名的抽象。函数在Lisp中也是第一类（fist-class）数据类型。它们可以被保存在变量中，可以作为实参传递，也可以作为返回值返回以及在运行期创建。

而这些内置类型仅仅是开始。它们被定义在语言标准中，因此程序员们可以依赖于它们的存在，并且也因为它们可以跟语言的其余部分紧密集成，从而使其可以更容易地高效实现。但正如你将在后续章节里看到的那样，Common Lisp另外还提供了几种定义新的数据类型的方式，能定义对其的操作，并能将它们与内置数据类型集成起来。

但目前将先从内置数据类型开始讲起。因为Lisp是一种高阶语言，不同的数据类型的具体实现细节在很大程度上是隐藏的。从语言用户的角度来看，内置数据类型是由操作它们的函数所定义的。因此为了学习一个数据类型，你只需学会那些与之一起使用的函数就行了。另外，多数内置数据类型都具有Lisp读取器所理解并且Lisp打印器可使用的特殊语法。所以你才能将字符串写成`"foo"`，将数字写成123、1/23和1.23，以及把列表写成`(a b c)`。我将在描述操作它们的函数时具体描述不同对象的语法。

本章将介绍内置的“标量”数据类型：数字、字符和字符串。从技术上来讲，字符串并不是真正的标量。字符串是字符的序列，你可以访问单独的字符并使用一个操作在序列上的函数来处理该字符串。但我在这里讨论字符串则是因为多数字符串的相关函数会将它们作为单一值来处理，同时也是因为某些字符串函数与它们的字符组成部分之间有着紧密的关系。

10.1 数字

正如Barbie所说，数学很难。[2]虽说Common Lisp并不能使其数学部分变得简单一些，但它确

① Fred Brooks，《人月神话》（Boston:Addison-Wesley,1995），p.103。

② Mattel's Teen Talk Barbie。

实可以比其他编程语言在这方面简单不少，考虑到它的数学传统这并不奇怪。Lisp最初是由一位数学家设计而成的，用作研究数学函数的工具。并且MIT的MAC项目的主要项目之一——Macsyma符号代数系统也是由Maclisp（一种Common Lisp的前身）写成的。此外，Lisp还一直在MIT这类院校用作教学语言，它能很好地支持精确比值，省得计算机科学教授们要给学生们解释为什么10/4=2。Lisp还曾经多次在高性能数值计算领域与FORTRAN竞争。

Lisp是一门用于数学的良好语言，其原因之一是它的数字更加接近于真正的数学数字，而不是易于在有穷计算机硬件上实现的近似数字。例如，Common Lisp中的整数几乎可以是任意大而不是限制在一个机器字的大小上。[1]而两个整数相除将得到一个确切的比值而非截断的值。并且比值是由成对的任意大小的整数表示的，所以比值可以表示任意精度的分数。[2]

另一方面，对于高性能数值编程，你可能想要用有理数的精度来换取使用硬件的底层浮点操作所得到的速度。因此Common Lisp也提供了几种浮点数，它们可以映射到适当的硬件支持浮点表达的实现上。[3]浮点数也被用于表示其真正数学值为无理数的计算结果。

最后，Common Lisp支持复数——通过在负数上获取平方根和对数所得到的结果。Common Lisp标准甚至还指定了复域上无理和超越函数的主值和分支切断。

10.2　字面数值

可以用多种方式来书写字面数值，第4章已经介绍了一些例子。但你需要牢记Lisp读取器和Lisp求值器之间的分工——读取器负责将文本转化成Lisp对象，而Lisp求值器只处理这些对象。对于一个给定类型的数字来说，它可以有多种不同的字面表示方式，所有这些都将被Lisp读取器转化成相同的对象表示。例如，你可以将整数10写成10、20/2、#xA或是其他形式的任何数字，但读取器将把所有这些转化成同一个对象。当数字被打印回来时，比如在REPL中，它们将以一种可能与输入该数字时不同的规范化文本语法被打印出来。例如：

```
CL-USER> 10
10
CL-USER> 20/2
10
```

① 很明显，一个具有有限内存的计算机可以表示的数字大小在事实上仍然是有限的。更进一步，特定Commom Lisp实现中使用的大数的实际表示在其所能表达的数字大小上可能有另外的限制，但这些限制通常会超出“天文数字”般大的数字。例如，整个宇宙中原子的数量预计少于2^{269}，而当前的Common Lisp实现可以轻易处理最大或超出2^{262144}的数字。

② 那些对于使用Common Lisp密集的数值计算感兴趣的人们应该注意到，如果单纯比较数值代码的性能，那么和诸如C或FORTRAN这样的语言比起来，Common Lisp可能会更慢。这是因为在Common Lisp中即便是(+ a b)这样简单的表达式也比其他语言中看起来等价的a+b做更多的事。由于Lisp中动态类型的机制以及对诸如任意精度有理数和复数的支持，一个看来简单的加法操作比两个已知表达为机器字的数字相加做更多的事。尽管如此，你可以使用声明来告诉Comom Lisp关于你所使用的数字类型的信息，从而使其生成与C或FORTRAN编译器做相同工作的代码。为这类优化调节数字代码超出了本书的范围，但这确实是可能的。

③ 尽管标准并未要求，但许多Common Lisp实现都支持IEEE浮点算术标准，*IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985*（Institute of Electrical and Electronics Engineers，1985年）。

```
CL-USER> #xa
10
```

整数的语法是可选的符号（+或-）后接一个或多个数字。比值的写法则依次组合了一个可选符号、一个代表分子的数位序列、一个斜杠（/）以及另一个代表分母的数位序列。所有的有理数在读取后都被“规范化”，这就是10和20/2都被读成同一个数字的原因，3/4和6/8也是这样，有理数以“简化”形式打印，整数值以整数语法来打印，而比值被打印成分值和分母约分到最简的形式。

用十进制以外的进制来书写有理数也是可能的。如果前缀是#B或#b，一个字面有理数将以二进制来读取，其中0和1是唯一的合法数字。前缀是#O或#o代表一个八进制数（合法数字0-7），而#X或#x则代表十六进制数（合法数字0-F或o-f）。你可以使用#nR以2到36的其他进制书写有理数，其中n代表进制数（一定要以十进制书写）。超过9的附加“数字”从字母A-Z或a-z中获取。注意，这些进制指示符将应用到整个有理数上——不可能以一种进制来书写比值的分值，而用另一种进制来书写分母。另外，你可以将整数而非比值写成以十进制小数点结尾的十进制数。[①]下面是一些有理数的例子，带有它们对应的规范化十进制表示：

```
123                                → 123
+123                               → 123
-123                               → -123
123.                               → 123
2/3                                → 2/3
-2/3                               → -2/3
4/6                                → 2/3
6/3                                → 2
#b10101                            → 21
#b1010/1011                        → 10/11
#o777                              → 511
#xDADA                             → 56026
#36rABCDEFGHIJKLMNOPQRSTUVWXYZ → 8337503854730415241050377135811259267835
```

也可以用多种方式来书写浮点数。和有理数不同，表示浮点数的语法可以影响数字被读取的实际类型。Common Lisp定义了四种浮点数子类型：短型、单精度、双精度和长型。每一个子类型在其表示中可以使用不同数量的比特，这意味着每个子类型可以表示跨越不同范围和精度的值。更多的比特可以获得更宽的范围和更高的精度。[②]

浮点数的基本格式是一个可选符号后跟一个非空的十进制数字序列，同时可能带有一个嵌入的小数点。这个序列可能后接一个代表“计算机科学计数法”[③]的指数标记。指数标记由单个字

① 通过改变全局变量*READ-BASE*，也有可能实现无需使用特别的进制标记即可改变读取器在数字上使用的默认基数。不过这样可能会导致严重的混乱。

② 由于浮点数的目的是为了有效使用浮点硬件，因此每个Lisp实现都允许将这四种子类型映射到适当的原生浮点类型上。如果硬件支持少于四种相区别的表示方法，这些类型中的一种或几种可能是等价的。

③ “计算机科学计数法”加引号是因为，尽管其自从FORTRAN时就被广范用在计算机语言里，但它实际上和真正的科学计数法很不相同，确切地说，像1.0e4这样的数字代表10000.0，而在真正的科学计数法中它表示为1.0×10^4。而进一步产生混淆的是，在真正的科学计数法中字母e代表自然对数的底。因此，$1.0 \times e^4$从表面上看类似于1.0e4，但其却是一个完全不同的值，约等于54.6。

母后跟一个可选符号和一个数字序列组成，其代表10的指数用来跟指数标记前的数字相乘。该字母有两重作用：它标记了指数的开始并且指出该数字应当使用的浮点表示方式。指数标记s、f、d、l（以及它们等价的大写形式）分别代表短型、单精度、双精度以及长型浮点数。字母e代表默认表示方式（单浮点数）。

没有指数标记的数字以默认表示来读取，并且必须含有一个小数点后面还至少有一个数字，以区别于整数。浮点数中的数字总是以十进制数字来表示——#B、#X、#O和#R语法只用在有理数上。下面是一些浮点数的例子，带有它们的规范表示形式：

```
1.0       → 1.0
1e0       → 1.0
1d0       → 1.0d0
123.0     → 123.0
123e0     → 123.0
0.123     → 0.123
.123      → 0.123
123e-3    → 0.123
123E-3    → 0.123
0.123e20 → 1.23e+19
123d23    → 1.23d+25
```

最后，复数有它们自己的语法，也就是#C或#c跟上一个由两个实数所组成的列表，分别代表复数的实部和虚部。事实上因为实部和虚部必须同为有理数或相同类型的浮点数，所以共有五种类型的复数。

不过你可以随意书写它们。如果复数被写成由有理数和浮点数组成，该有理数将被转化成一个适当表示的浮点数。类似地，如果实部和虚部是不同表示法的浮点数，使用较小表示法的那一个将被提高。

尽管如此，没有复数可以具有一个有理的实部和一个零的虚部，因为这样的值从数学上讲是有理的，所以它们将用对应的有理数值来表示。同样的数学论据对于由浮点数所组成的复数也是成立的，但其中那些带有零虚部的复数总是一个与代表实部的浮点数不同的对象。下面是一些以复数语法写成的数字的例子：

```
#c(2      1)    → #c(2 1)
#c(2/3  3/4)    → #c(2/3 3/4)
#c(2    1.0)    → #c(2.0 1.0)
#c(2.0  1.0d0) → #c(2.0d0 1.0d0)
#c(1/2  1.0)    → #c(0.5 1.0)
#c(3      0)    → 3
#c(3.0  0.0)    → #c(3.0 0.0)
#c(1/2    0)    → 1/2
#c(-6/3   0)    → -2
```

10.3　初等数学

基本的算术操作即加法、减法、乘法和除法，通过函数+、-、*、/支持所有不同类型的Lisp数字。使用超过两个参数来调用这其中的任何一个函数，这种作法将等价于在前两个参数上调用

相同的函数而后再在所得结果和其余参数上再次调用。例如，(+ 1 2 3)等价于(+ (+ 1 2) 3)。当只有一个参数时，+和*直接返回其值，-返回其相反值，而/返回其倒数。[1]

```
(+ 1 2)              → 3
(+ 1 2 3)            → 6
(+ 10.0 3.0)         → 13.0
(+ #c(1 2) #c(3 4)) → #c(4 6)
(- 5 4)              → 1
(- 2)                → -2
(- 10 3 5)           → 2
(* 2 3)              → 6
(* 2 3 4)            → 24
(/ 10 5)             → 2
(/ 10 5 2)           → 1
(/ 2 3)              → 2/3
(/ 4)                → 1/4
```

如果所有实参都是相同类型的数（有理数、浮点数或复数），则结果也将是同类型的，除非带有有理部分的复数操作的结果产生了一个零虚部的数，此时结果将是一个有理数。尽管如此，浮点数和复数是有传播性的。如果所有实参都是实数但其中有一个或更多是浮点数，那么其他实参将被转化成以实际浮点实参的“最大”浮点表示而成的最接近浮点值。那些“较小”表示的浮点数也将被转化成更大的表示。同样，如果实参中的任何一个是复数，则任何实参数会被转化成等价的复数。

```
(+ 1 2.0)            → 3.0
(/ 2 3.0)            → 0.6666667
(+ #c(1 2) 3)        → #c(4 2)
(+ #c(1 2) 3/2)      → #c(5/2 2)
(+ #c(1 1) #c(2 -1)) → 3
```

因为/不作截断处理，所以Common Lisp提供了4种类型的截断和舍入用于将一个实数（有理数或浮点数）转化成整数：**FLOOR**向负无穷方向截断，返回小于或等于实参的最大整数；**CEILING**向正无穷方向截断，返回大于或等于参数的最小整数；**TRUNCATE**向零截断，对于正实参而言，它等价于**FLOOR**，而对于负实参则等价于**CEILING**；而**ROUND**舍入到最接近的整数上，如果参数刚好位于两个整数之间，它舍入到最接近的偶数上。

两个相关的函数是**MOD**和**REM**，它返回两个实数截断相除得到的模和余数。这两个函数与**FLOOR**和**TRUNCATE**函数之间的关系如下所示：

```
(+ (* (floor    (/ x y)) y) (mod x y)) ≡ x
(+ (* (truncate (/ x y)) y) (rem x y)) ≡ x
```

因此，对于正的商它们是等价的，而对于负的商它们产生不同的结果。[2]

① 出于数学一致性的考虑，+和*也可以不带参数被调用。这种情况下，它们将返回适当的值：+返回0，而*返回1。

② 严格来讲，**MOD**等价于Perl和Python中的%操作符，而**REM**等价于C和Java中的%。（从技术上来讲，%在C中的行为直到C99标准时才明确指定。）

函数1+和1-提供了表示从一个数字增加或减少一个的简化方式。注意它们与宏INCF和DECF有所不同。1+和1-只是返回一个新值的函数，而INCF和DECF会修改一个位置。下面的恒等式显示了INCF/DECF、1+/1-和+/-之间的关系：

```
(incf x)    ≡ (setf x (1+ x)) ≡ (setf x (+ x 1))
(decf x)    ≡ (setf x (1- x)) ≡ (setf x (- x 1))
(incf x 10) ≡ (setf x (+ x 10))
(decf x 10) ≡ (setf x (- x 10))
```

10.4　数值比较

函数=是数值等价谓词。它用数学意义上的值来比较数字，而忽略类型上的区别。这样，=将把不同类型在数学意义上等价的值视为等价，而通用等价谓词EQL将由于其类型差异而视其不等价。（但通用等价谓词EQUALP使用=来比较数字。）如果它以超过两个参数被调用，它将只有当所有参数具有相同值时才返回真。如下所示：

```
(= 1 1)                          → T
(= 10 20/2)                      → T
(= 1 1.0 #c(1.0 0.0) #c(1 0))    → T
```

相反，只有当函数/=的全部实参都是不同值时才返回真。

```
(/= 1 1)        → NIL
(/= 1 2)        → T
(/= 1 2 3)      → T
(/= 1 2 3 1)    → NIL
(/= 1 2 3 1.0)  → NIL
```

函数<、>、<=和>=检查有理数和浮点数（也就是实数）的次序。跟=和/=相似，这些函数也可以用超过两个参数来调用，这时每个参数都跟其右边的那个参数相比较。

```
(< 2 3)         → T
(> 2 3)         → NIL
(> 3 2)         → T
(< 2 3 4)       → T
(< 2 3 3)       → NIL
(<= 2 3 3)      → T
(<= 2 3 3 4)    → T
(<= 2 3 4 3)    → NIL
```

要想选出几个数字中最小或最大的那个，你可以使用函数MIN或MAX，其接受任意数量的实数参数并返回最小或最大值。

```
(max 10 11)     → 11
(min -12 -10)   → -12
(max -1 2 -3)   → 2
```

其他一些常用函数包括ZEROP、MINUSP和PLUSP，用来测试单一实数是否等于、小于或大于零。另外两个谓词EVENP和ODDP，测试单一整数参数是否是偶数或奇数。这些函数名称中的P后缀是一种谓词函数的标准命名约定，这些函数能够测试某些条件并返回一个布尔值。

10.5 高等数学

目前为止，你所看到的函数只是初级的内置数学函数。Lisp也支持对数函数**LOG**，指数函数**EXP**和**EXPT**，基本三角函数**SIN**、**COS**和**TAN**及其逆函数**ASIN**、**ACOS**和**ATAN**，双曲函数**SINH**、**COSH**和**TANH**及其逆函数**ASINH**、**ACOSH**和**ATANH**。它还提供了函数用来获取一个整数中单独的位，以及取出一个比值或一个复数中的部分。完整的函数列表参见任何Common Lisp参考。

10.6 字符

Common Lisp字符和数字是不同类型的对象。本该如此——字符不是数字，而将其同等对待的语言当字符编码改变时（比如说从8位ASCII到21位Unicode[①]）可能会出现问题。由于Common Lisp标志并未规定字符的内部表示方法，当今几种Lisp实现都使用Unicode作为其“原生”字符编码，尽管从标准化组织的观点来看，Unicode在Common Lisp自身的标准化成型时期只是昙花一现。

字符的读取语法很简单：`#\`后跟想要的字符。这样，`#\x`就是字符x。任何字符都可以用在`#\`之后，包括“"”、“(”和空格这样的特殊字符。但以这种方式来写空格字符对我们来说可续性不高，特定字符的替代语法是`#\`后跟该字符的名字。具体支持的名字取决于字符集和所在的Lisp实现，但所有实现都支持名字*Space*和*Newline*。这样就应该写成用`#\Space`来代替“`#\ `”，尽管后者在技术上是合法的。其他半标准化的名字（如果字符集包含相应的字符实现就必须采用的名字）是Tab、Page、Rubout、Linefeed、Return和Backspace。

10.7 字符比较

可以对字符做的主要操作，除了将它们放进字符串（我将在本章后面讨论这点）之外，还可将它们与其他字符相比较。由于字符不是数字，所以不能使用诸如<和>这样的数值比较函数。作为替代，有两类函数提供了数值比较符的特定于字符的相似物：一类是大小写相关的，而另一类是大小写无关的。

数值=的大小写相关相似物是函数**CHAR=**。像=那样，**CHAR=**可以接受任意数量的实参并只在它们全是相同字符时才返回真。大小写无关版本是**CHAR-EQUAL**。

其余的字符比较符遵循了相同的命名模式：大小写相关的比较符通过在其对应的数值比较符前面加上CHAR来命名；大小写无关的版本拼出比较符的名字，前面加上CHAR和一个连字符。不过，<=和>=被拼写成了其逻辑等价形式**NOT-GREATERP**和**NOT-LESSP**，而不是更确切的LESSP-OR-EQUALP和GREATERP-OR-EQUALP。和它们的数值等价物一样，所有这些函数都接受一个或更多参数。表10-1总结了数值和字符比较函数之间的关系。

① 甚至像Java也会产生问题，它基于Unicode注定将成为未来主流字符编码这一理论，而从一开始就被设计使用Unicode字符。产生问题是因为Java字符被定义为16位值，而Unicode3.1标准将Unicode字符集范围扩展到了需要21位表示。太惨了。

表10-1　字符比较函数

数值相似物	大小写相关	大小写无关
=	CHAR=	CHAR-EQUAL
/=	CHAR/=	CHAR-NOT-EQUAL
<	CHAR<	CHAR-LESSP
>	CHAR>	CHAR-GREATERP
<=	CHAR<=	CHAR-NOT-GREATERP
>=	CHAR>=	CHAR-NOT-LESSP

其他处理字符的函数包括测试一个给定字符是否是字母或者数字字符，测试一个字符的大小写，获取不同大小写的对应字符，以及在代表字符编码的数值和实际字符对象之间转化。对于完整的细节，请参见你喜爱的Common Lisp参考。

10.8　字符串

如前所述，Common Lisp中的字符串其实是一种复合数据类型，即一维字符数组。因此，我将在下一章讨论用来处理序列的许多函数时谈及许多字符串应用，因为字符串只不过是一种序列。但字符串也有其自己的字面语法和一个用来进行字符串特定操作的函数库。本章将讨论字符串的这些方面，其余部分留到第11章再作介绍。

正如你所看到的那样，字面字符串写在闭合的双引号里。你可以在一个字面字符串中包括任何字符集支持的字符，除了双引号（"）和反斜杠（\）。而如果你将它们用一个反斜杠转义的话也可以包括这两个字符。事实上，反斜杠总是转义其下一个字符，无论它是什么，尽管这对于除了“"”和“\”本身之外的其他字符并不必要。表10-2显示了不同的字面字符串是如何被Lisp读取器读取的。

表10-2　字面字符串

字面字符串	内　容	说　明
"foobar"	foobar	纯字符串
"foo\"bar"	foo" bar	反斜杠转义引号
"foo\\bar"	foo\bar	第一个反斜杠转义第二个反斜杠
"\"foobar\""	"foobar"	反斜杠转义引号
"foo\bar"	foobar	反斜杠“转义” *b*

注意，REPL将以可读的形式原样打印字符串，并带有外围的引号和任何必要的转义反斜杠。因此，如果想要看到一个字符串的实际内容，就要使用诸如**FORMAT**这种原本就是设计用于打印出可读性良好输出的函数。例如，下面是在REPL中输入一个含有内嵌引号的字符串时所看到的输出。

```
CL-USER> "foo\"bar"
"foo\"bar"
```

另一方面，**FORMAT**将显示出实际的字符串内容：[①]

```
CL-USER> (format t "foo\"bar")
foo"bar
NIL
```

10.9 字符串比较

你可以使用一组遵循了和字符比较函数相同的命名约定的函数来比较字符串，只不过要将前缀的CHAR换成STRING（参见表10-3）。

表10-3 字符串比较函数

数值相似物	大小写相关	大小写无关
=	STRING=	STRING-EQUAL
/=	STRING/=	STRING-NOT-EQUAL
<	STRING<	STRING-LESSP
>	STRING>	STRING-GREATERP
<=	STRING<=	STRING-NOT-GREATERP
>=	STRING>=	STRING-NOT-LESSP

但跟字符和数字比较符不同的是，字符串比较符只能比较两个字符串。这是因为它们还带有关键字参数，从而允许你将比较限制在每个或两个字符串的子字符串上。这些参数:start1、:end1、:start2和:end2指定了起始和结束参数字符串中子字符串的起始和终止位置（左闭右开区间）。因此请看下面这个表达式。

```
(string= "foobarbaz" "quuxbarfoo" :start1 3 :end1 6 :start2 4 :end2 7)
```

在两个参数中比较子字符串"bar"并返回真。参数:end1和:end2可以为**NIL**（或者整个关键字参数被省略）指示相应的子字符串扩展到字符串的结尾。

当参数不同时返回真的比较符，也即**STRING=**和**STRING-EQUAL**之外的所有操作符，将返回第一个字符串中首次检测到不匹配的索引。

```
(string/= "lisp" "lissome") → 3
```

如果第一个字符串是第二个字符串的前缀，返回值是第一个字符串的长度，也就是一个大于字符串中最大有效索引的值。

```
(string< "lisp" "lisper") → 4
```

当比较子字符串时，返回值仍然是该字符串作为整体的索引，例如，下面的调用比较子字符串"bar"和"baz"；但返回了5，因为它是“r”在第一个字符串中的索引：

```
(string< "foobar" "abaz" :start1 3 :start2 1) → 5   ; N.B. not 2
```

① 注意：尽管如此，并非所有的字面字符串都可以通过将其作为**FORMAT**的第二个参数传递来打印，因为特定的字符序列对于**FORMAT**有特殊的含义。为了安全地通过**FORMAT**打印一个任意字符串，比如说，一个变量s的值，你应当写为(format t "~a" s)。

其他字符串函数允许你转化字符串的大小写以及从一个字符串的一端或两端修剪字符。而且如同我前面提到的，由于字符串实际上是一种序列，我将在下一章讨论的所有序列函数都可用于字符串。例如，你可以用**LENGTH**函数来检查字符串的长度并获取和设定字符串中的个别字符，使用通用序列元素访问函数**ELT**或者使用通用数组元素访问函数**AREF**。你还可以使用特定于字符串的访问函数**CHAR**。这些以及其他一些函数都是下一章的主题，让我们继续吧。