

ELICA 5.0

不仅仅“只是另一个 Logo”

简明手册

著书并插图：Pavel Boytchev, Elica Team
电子邮件地址：pavel@elica.net
www.elica.net

修订版本：3.2
2001年9月

目录

1.致谢	3
2.前言	4
3.LOGO 初步	6
3.1 认识字	6
3.2 实体类型	6
3.3 如何声明字和表?	7
3.4 如何声明过程?	7
3.5 如何声明变量?	8
3.6 如何声明注释?	10
3.7 内建的字(关键字)	11
3.7.1 打印文本(print)	11
3.7.2 创建和更改变量(local, make, ob)	11
3.7.3 执行指令(run)	12
3.7.4 控制程序的走向(if, repeat, while)	12
3.7.5 定义过程(to...end, output)	14
4.LOGO 海龟	17
4.1 跳舞的海龟(FD, BK)	17
4.2 左转, 右转(LT, RT)	17
4.3 做更多的事(HOME, PU, PD, PD?)	18
4.4 时隐时现(HIDE, SHOW, SHOWN?)	18
5.LOGO 和语言	21
5.1 这是什么?(number?, word?, list?)	21
5.2 我要构造(word, list, se, set, fput, lput)	21
5.3 我不想要它的全部, 我只需要一点点。(first, last, bf, bl, item)	22
5.4 其它过程(count, ascii, char, wait)	23
6.Logo 和数学	25
6.1 数学运算符	25
6.1.1 算术运算符(+, -, *, /, ^)	25
6.1.2 整数运算符(idiv, imod)	25
6.2 数学函数	25
6.2.1 符号函数(abs, neg, sign)	25
6.2.2 指数函数(sqrt, exp, logn)	26
6.2.3 舍去函数(trunc, round)	26
6.2.4 三角函数(sin, cos, tan, cotan)	26
6.2.5 随机函数	26
7.Logo 和逻辑	29
7.1 比较运算符(=, <, >, <=, >=, <>)	29
7.2 逻辑运算符(and, or, not)	29

1. 致谢

非常感谢 Bob Gorman, 尽管适逢周年庆典, 他仍抽出足够的时间逐字阅读并完善这本手册。更要感谢 Andreas Micheler, 他不怕初次校订的麻烦, 给出许多可以更加细致改进这本手册的有价值的建议。

特别感谢 Bojidar Sendov 和 Jenny Sendova, 是他们指引我穿越编程语言的迷宫, 帮助我看到出口标识着“Logo”。

2. 前言

科学家们非常努力地工作以使计算机能够明白你的意图，但直到这项工作完成，你仍需通过一种计算机能明白的语言与之交流。这样的语言称为编程语言，因为它的字是计算机能够明白的指令。按照前面所说，程序员就是编写程序的人。

Logo 是最容易学习的编程语言之一。正是因为它是基于这样的设计思想，你可以付出比学习其他编程语言少得多的努力来学习它。

Elica 是基于 Logo 编程语言的软件开发系统。Elica Logo 是 Logo 语言家族的一员，并且是 Elica 创建的应用程序的核心。“Elica”是由 Educational Logo Interface for Creative Actives 的字头组成。

Elica 程序是包含指令的普通文本文件。就象书本上的文字一样，这些指令包含字和标点。Logo 内建了识别机制，当遇到一些特定的字时，它知道该做些什么。为了让它能够明白得更多，你需要教会它新的字，并且让它知道当你在指令中运用这些字时，它该如何做出反应。你应当有正确的思想准备，教会它是困难的。当然，这意味着你不仅要充当老师，你还要定义。拥有了定义的能力，你几乎可以用 Logo 做任何事。

这本手册可以在你学习如何用 Elica 编制程序的时候，向你提供一些指导性的教程。它的描述非常容易理解，适合于大多数的读者阅读。不管怎么说，这本手册不能涵盖 Elica Logo 的全部细节。当你读完之后，你会发觉我们讨论的大部分内容对于所有的 Logo 软件都或多或少是通用的。

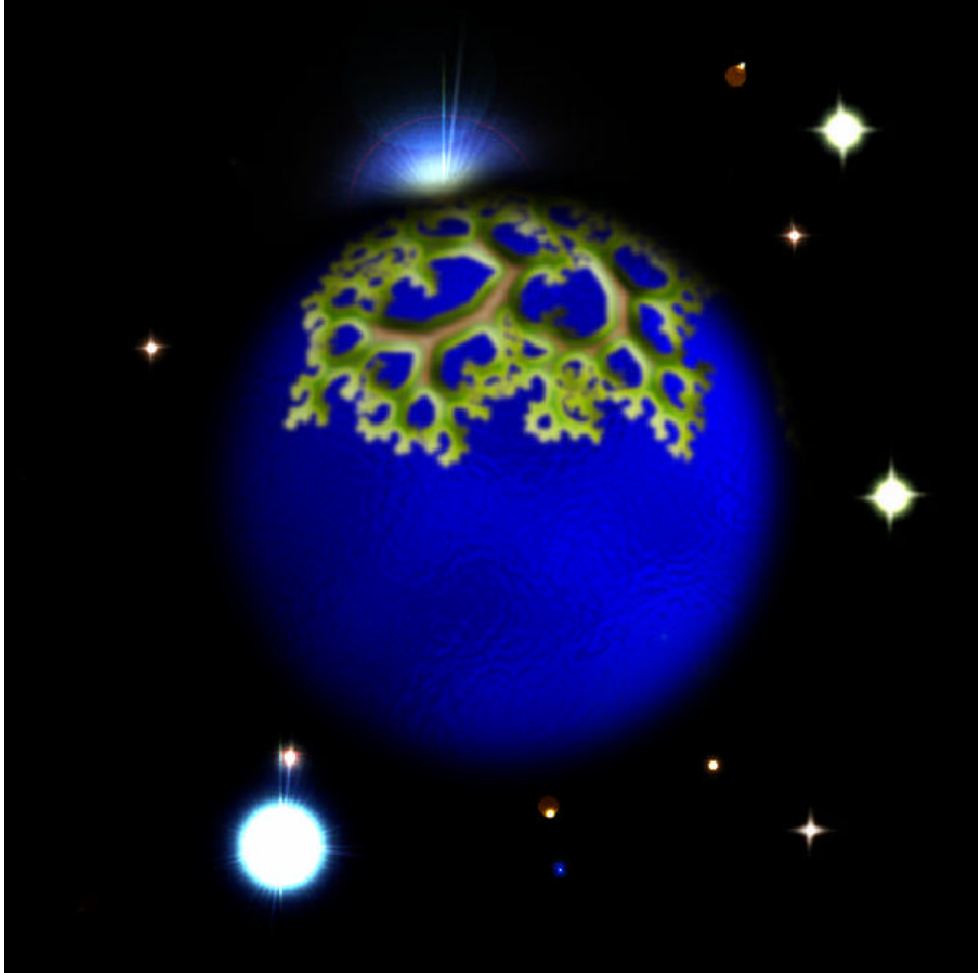
对于 Logo 的高级用户，最好是快速浏览一下《简明手册》，然后继续阅读《完全手册》，在那里 Elica Logo 将被详细说明。

对于那些喜好阅读以科技词汇写成的手册，并且在听到一些不可思议的词如多态性，多重继承和高次函数时感觉很好，那么找其它的手册去看会更好一些。

好了，让我们开始吧。

三维海龟

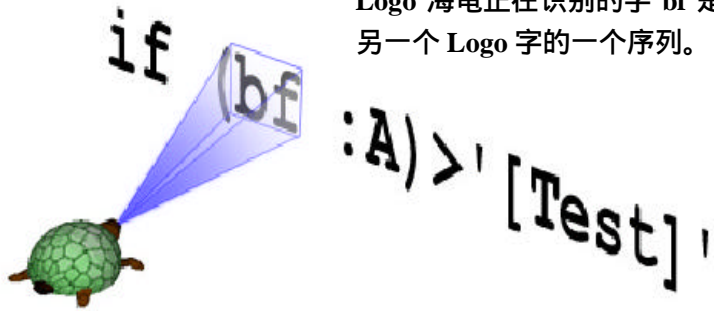
不规则的碎片凸出于行星表面



3. Logo 初步

3.1 认识字

当你在阅读或者编写 Logo 程序的时候,过了一段时间,你会直觉地明白单个的字是程序的组成部分。但通常编程语言是与自然语言不同的。这就是为什么你需要懂得如何让 Logo 识别你的语句。



Logo 海龟正在识别的字 bf 是另一个 Logo 字的一个序列。

以下是 Logo 所用规则的简短列表：

- 一组位于同一行上的,在一对单引号'...'之间的任何字符,称为一个字。
- 另外,一个字也可由连续的一组由空格或符号如方括号[], 小括号(), 大括号{}, 分号;, 冒号:, 双引号“, 加号+和逗号, 构成的字符, 它们被用来构成 Logo 程序, 通常是单个的字。
- 如果减号-位于一个空格和一个数字之间, 那么它是字的一部分, 并包含这数字。
- 只包含>, =或<的连续的一组字符通常也是一个字。

3.2 实体类型

数据类型是 Logo 能够处理的实体类型。数据类型是与数据相关的类型的简称。按惯例 Logo 识别两种常规的实体类型：字和表。尽管 Elica 能够识别这些实体, 但它没有关于如何处理它们的知识。这就好象你能够读出外语, 但你并不明白其中的含意。

字是最基本的实体类型。当你学会怎样在程序中识别 Elica 语句时, 你已经知道什么是字了。当 Elica 在分析你写下的语句时, 它运用了相似的识别机制。一些字包含字母, 另一些包含了数字并构成了有效的数值。

表是由若干字组成的。起始的字必须是左方括号[, 结尾的字必须是右方括号]。它们之间的字被称为表的元素。

表[A AB ABC]包含三个字。不包含任何元素的表被称作空表, 记作[]。表中还可以嵌套表, 这有一个包含三个元素的表[ELI [A AB ABC] CA], 其中第二个元素是[A AB ABC], 它本身也是一个表。表对于 Logo 来说十分重要, 因为你发给 Logo 的所有指令都是以表的形式写成的。

3.3 如何声明字和表？

假设你想命令 Logo 打印房子。计算机并没有你想象的那么智能化, 因此它无法确定你的真正意图。是打印房子的图像还是打印文字“房子”？这就是为什么你在编写程序时要给 Logo 一些提示以告诉 Logo 如何正确处理那些容易混淆的语句, 这不仅显得文雅而且是

必须的。

如果你想把字当作一般自然语言中的字来使用（例如你想打印文本的时候），那么你可以在字的前面加上一个双引号”。”cat,”X,”Case23 被视为字’cat’,’X’和’Case23’。如果你声明 cat 来代替”cat，这将要求 Logo 把它用于一个过程（过程是指令的同义词）。过程我们稍后再叙述。

Logo 只认识字和表



Logo 中的数值实际上也是字。这是为什么你可以在它们前面加上”来声明。当 Logo 读取到数值时，它显然不会把它当成一个过程，所以我们并不真正需要引用标记。因此，你可以直接声明数值：56, 31.8,-300。

正如数值一样，表也不容易混淆。当你想声明表的时候，你只需按照它本身的形式来声明，但别忘了把它用方括号[...]框起来，否则 Logo 不会明白它是一个表。

3.4 如何声明过程

正如你日常说话一样，声明字和表只用名词。只有动词才能表述如何对这些实体进行处理。按惯例 Logo 喜欢看到过程名被直接使用（没有专门的标记）。如果你想打印 5 和 6，你只需用如下命令 print 5 6。许多过程需要它所处理的实体能够很容易地找到。这就是这些实体（称作实参，形参或输入）通常写在过程名之前或后的原因。

当你定义了一个新的过程之后，你还要定义它是否在过程名之前或之后寻找参数，或者前后都要寻找。例如，操作符+的前后各需要一个参数。使用已经定义好的过程，你需要知道它们的名称以及参数的放置位置。如果你没有把参数的位置摆正确，一些过程也会出错。

这里正是个合适的地方来区分一下实参和形参。你可以认为实参是真实提供给过程使用的参数（也称作实际的参数），而形参是过程在形式上需要的参数（形式上的参数）。在大多数情况下，实参和形参具有相同的意义。



一些过程能够与几个参数一起发挥作用。如果你不确定它到底需要几个参数，或者你提供的参数多于或少于它需要的，又或者，最后的但并非不重要的，你想更好地控制 Logo 解释程序的方式，那么，你必须将过程和它的所有参数都用圆括号 (...) 括起来。当 Logo 看到被圆括号括起来的过程，它会试着强制该过程使用所有的参数。

在接下来的例子中，你可以看到被称作 first 的过程首先按通常的方式提取出一个字符串的头一个字母并打印输出 c, 接着它被强制使用两个参数 (第二个参数指示要从字符串的头部开始提取的字母的个数), -瞧- 它打印输出 ca。第三条指令将打印输出 c 2, 因为数值将被 print 处理, Logo 并没有得到它必须被 first 使用的线索。

```
Print first "cat          ;c
print (first "cat 2)      ;ca
print first "cat 2       ;c 2
```

因此，声明一个过程的一般规则如下所写：

参数 参数 ... 过程名 参数 参数...

如果你要明确指定那几个参数要提供给过程，那么要这样使用圆括号 (...)：

(参数 参数 ... 过程名 参数 参数...)

你已经看到操作过程具有强大的活力。Logo 所作的每一件事都是通过某种形式的过程。这就是为什么我们要用其它更专门的术语来描述不同类型的过程。那些不需要返回结果的过程被称作指令或命令。那些通过计算并返回结果的过程被称作函数。需要在两边或左边提供参数的函数被称作操作符。而那些在参数中仍包含过程的过程经常被归为表达式。

总之，不用太用心地去学习这些术语，因为关于过程的分类只是相对的，并且并不正式。你所要了解的就是，指令，命令，函数，操作符，表达式（还有一串其它的字）都是过程的不同风味。

3.5 如何声明变量？

如果 Logo 是你学习的第一个编程语言，那么你可能从没接触过变量。什么是变量？为什么编程谣言需要它们？设想一下你负责编写火灾避险的指令。当你在编写的时候，你可以用“值班的人”来指代不同的人。你这样写是因为你不知道发生火灾的当时是谁值班。当然，你可以把名字加到指令中去，但这样做之后，你不得不为每一个可能值班的人提供单独的指令。

变量就象是柜子。你可以把一把钥匙或一串钥匙——变量值——放在里面。要打开这个柜子，你就需要另一把钥匙——变量名。现在，海龟正背着标着 LOGO 的钥匙去打开装着钥匙串[A AB ABC]的柜子。



在 Logo 中，你可以把字作为值的名称。重要的一点在于，你可以改变这个值并且仍使用同样的名称。由此，我们把这个名称和与之相联系的值称作变量。你可以改变变量的值并保持程序不用修改。这就是计算机程序的最重要的特性之一——你只需编写一次程序然后设定不同的值来运行它。

Elica Logo 中的变量以它的名称，值和位置来定义。你已经知道变量名是字，并且它们的值可以是任何类型——字，数值，表和过程。

当你想声明变量名时，暂时不必考虑它的值，按照字的规则声明，并在字的前面加上

双引号“，创建变量时经常要这么做。如果你要达此目的地，你只需以很自然的方式要求 Logo 来建立它。

```
Make " a 5
make " fellow " mike
make " LOGO [A AB ABC]
```

第一条指令创建一个值为 5 的变量 a，第二条指令创建了变量 fellow，并把字 mike 赋予它，第三条指令创建了变量 LOGO，并赋予它一个表。Make 是 Logo 中预先建立的过程之一，它已经知道如何处理这个过程。稍后你将看到关于过程 make 的集中描述。

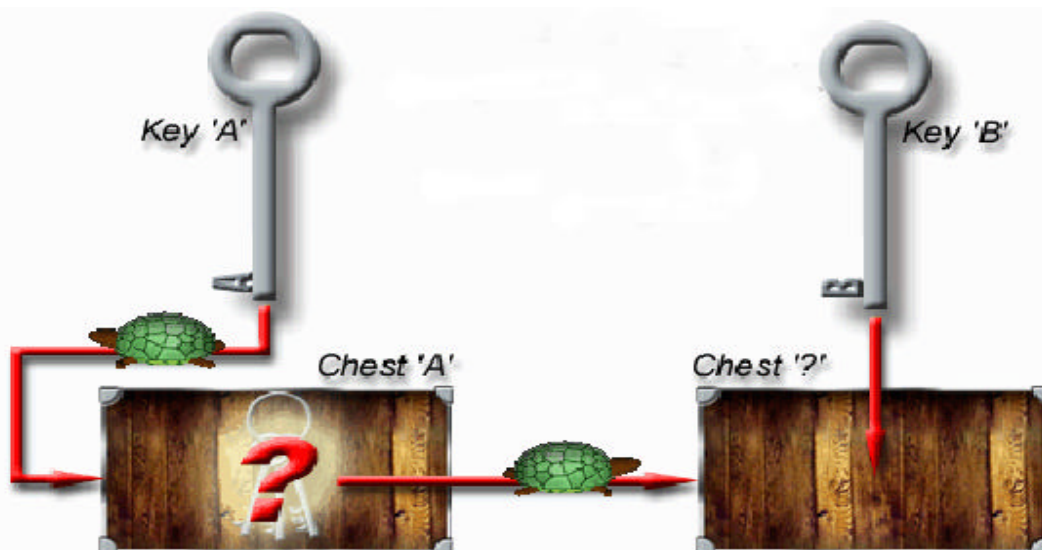
什么是值？怎样把值赋予变量？在 Logo 中有一个专的过程，你可以用它来达此目的地。如果你在一个字的前面加上冒号：，那么 Logo 将会试图读取与该字同名的变量的内容。如果我们要让 Logo 求得存储在变量 a 和 b 中的两个数值的总和，我们需要以:a+b 这样的形式向 Logo 下达指令。

Logo 是一种强大的语言。为什么？有多种说法，最酷的一种是 Logo 能够处理那些无法预先知道变量名的变量。看下面的例子。

```
Make :a " b
```

Logo 会怎样解释它呢？它首先会找出与变量 a 相联系的值，这个值（不是 a）将被用作声明变量名。

在钥匙和柜子的世界里，命令 make :a "b 有着非常不可思议的解释。海龟用钥匙'a'打开柜子（也就是这把钥匙唯一能打开的柜子）。然后它往里面看，希望能找到另一把钥匙。当海龟找到了钥匙，它就用这把钥匙打开一个新柜子，然后海龟把钥匙'b'放进这个新柜子并进入休眠状态。



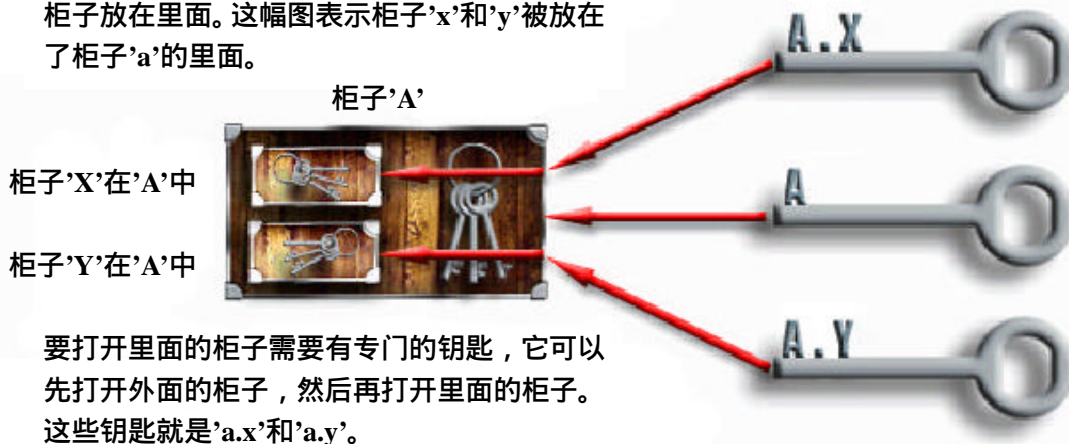
除了名称和值，变量还具有位置的属性。所谓位置是有关给定变量存在于何处的描述。哪里是变量可以存在的位置呢？在 Logo 中，每个变量都可以被放置在另一个变量中。换句话说，每个变量不不仅可以包含一个值，还可以包含一个变量，甚至可以包含多个变量。一些人把这些变量称作子变量，另一些人称作局部变量，还有第三组人群——严格的程序员们——称作域，成员，或方法。这种形式是如此多见，无所谓你怎么来称呼它——Logo 不使用这些词语。它们只被人类使用。当你与你的朋友或同学谈论 Logo 时，你们只有使用你们都知道的（或喜欢的）术语，交谈和讨论才会顺利。

接下来让我们回到关于位置的话题。如果你打算定义一个变量 *a*，并使它位于变量 *b* 之中，那么，你可以使用这样的变量名 *b.a*，这将指示 Logo 在寻找变量 *a* 的值时，可以定位到变量 *b* 的内部。当然，你可以变量隐藏得更深一些，例如 *d.c.b.a*，意为：*a* 是 *b* 的子变量，*b* 在 *c* 中，*c* 又是 *d* 的局部变量。

```
Make " a.x 10
make " a.y 20
make " b :a
print :b.x :b.y ; 10 20
```

仔细看上面的例子。它展示了局部变量的一个基本的特性。当 Logo 获取一个变量的值时，包含了它的所有子变量。头两行定义了一个变量 *a*，它包含了两个子变量 *x*, *y*。当 Logo

这些柜子的最大的奥秘在于你可以把其它的柜子放在里面。这幅图表示柜子'*x*'和'*y*'被放在了柜子'*a*'的里面。



在创建变量 *b* 时，同时把变量 *a* 的子变量 *x*, *y* 一同复制给 *b*。因而从此刻起，有两个名为 *x* 的变量。当你谈及两个局部变量，例如 *cat.x* 和 *dog.y*，读作 *cat* 的 *x*，*dog* 的 *y* 会很方便。为变量和过程取个好名字真的非常重要。发挥你的想象力和创造力，不要犹豫，把命名的时间延长一些。因为一个好的名字通常会使得做事条理清楚。

3.6 如何声明注释？

注释，或者备注，是你在与 Logo 交谈时也与自己交谈的一种方式。通常注释可以当作你的备忘录。然而，Logo 对它完全不理不睬。在注释里，举例来说，您可以写下或多或少的文字来描述你的程序，程序的片断，或者一行复杂的程序是做什么的。

你也可以在你认为可能出现错误，或你有些主意但还不知如何实现，或者有其它问题的地方加上注释。这对你来说，不论在现在还是将来都很重要。如果你与朋友分享或交换程序时，它也许对你的朋友很重要。

不要低估了注释的价值。

设想一下，一位朋友把他编写的一个程序给了你。如果里面有一些提示或笔记，你读起来不是更容易些吗？如果你的朋友在不知如何进行下去的地方有些标记，你完善这个程序不会更容易些吗？如果你的朋友在怀疑有错误的地方加上注释，你帮助他修正错误不会更容易些吗？

但你必须去除那些对程序来说已经过了时的注释。如果你不这样做，那么还不如没有注释。

你可以在注释中随心所欲地加入内容，没有固定的模式可以遵循，只有一点——Logo 必须知道注释从何而始，从何而终。

Logo 中有两种类型的注释——短的和长的。短注释以分号 (;) 开始，直到这一行结束。

```
Make "a 105 ;current check number
make "b ":a+1 ;next check number
```

长注释以花括号 { (左大括号) 开始，结束于花括号 } (右大括号)。被称作长注释，是因为可以开始于一行而结束于其它行。长注释可以同时具有短注释和长注释的功能。无论何时使用，不要忘记使用了 { 之后在结束时一用要加上}。

```
{This is code calculates the
  average of three numbers}
make "m (:a+:b+:c)/3
```

3.7 内建的词 (关键字)

通过使用一些内建的词，你可以教会 Logo 做事。Elica 已经懂得如何对这些词做出反应。Elica 在最初诞生时就已经被灌输了这些知识。



在许多编程语言中，内建的词被称作关键字。在 Logo 中只有以下几个关键字：`print`, `local`, `make`, `ob`, `run`, `if`, `while`, `repeat`, `output`, `to` 和 `end`。

3.7.1 打印文本 (print)

任何 Logo 都可以执行的最普通的过程是将一个或多个值打印输出到屏幕上。指示 Logo 这样做的指令就是 `print`。如果你想让系统打印输出一些值，你只需在一行中使用 `print` 指令，后面紧跟需要打印输出的值。下个例子展示了如何打印输出数字 5，字 `five`，变量 `pos` 的值以及包含两个数值型参数的过程 `+` 的结果。

```
Print 5 "five :pos 1+2
```

如果你没有明确指定 Logo 需要打印输出的值，系统将打印输出一个空行。这个技巧被用来调整程序的输出格式。

3.7.2 创建和改变变量 (local, make, ob)

另一个 Logo 能够执行的常用的过程是创建变量并且改变它的值。Logo 这么做的时候，它需要执行两个指令。当 Logo 执行过程 `local` 时，它将在当前活动的过程内部创建一个新的变量。因而这个变量将是它的子变量。如果变量已经存在，它将什么也不做。在大多数情况下，当一个过程已经完成，它所有的局部变量会被清除。`Local` 与 `print` 有些类似，它使用可以在它的后面找到的所有的值。但与

使用 `local` 这个命令就相当于制作了一个带有钥匙孔的空柜子。



print 不同之处在于，local 只寻找字并把它用作变量名。

Local “x “y “done

另一个创建变量的指令是 make。你已经见过它了。它除了可以创建变量，还可以改变变量的值。Logo 需要二个参数来达到此目地——变量名以及它的新值。因此这个指令的使用方式如下：

```
make "a 1
make "b :a+10
```

还有一个创建变量的指令——它就是 ob，它的名称来自 Geomland——一个非常好的并且是强大的面向几何的 Logo 实现。这个指令与 make 的工作方式一样，除了它同时也创建变量之间的链接。来看下面的例子：

```
ob "a 1
ob "b :a*:a
ob "c :b*:b
print :a :b :c           ;prints 1 1 1
```

```
ob "a 2
print :a :b :c           ;prints 2 4 16
```

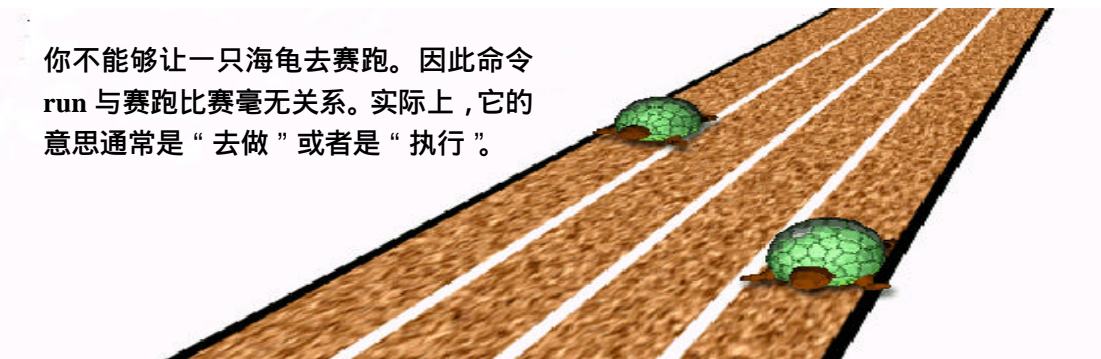
```
ob "a 3
print :a :b :c           ;prints 3 9 81
```

通过使用 ob 你可以很礼貌地要求 Logo 记住各个变量之间的关系。当你改变了它们中的一个，与之相关的变量都会被自动重新计算！

3.7.3 执行指令(run)

当 Logo 读取你的程序时，它知道按照什么样的顺序来执行什么样的指令。如果有这样的情况，当你想要 Logo 运行你在编写程序时还不知道的指令时，这些指令可能由你的程序生成或者是包含在外部的文本文件中。在这种情况下，这就要用 run 这个指令来告诉 Logo 来执行这些指令。

```
Run :mycommands
```



当然，你可以用表来替代变量或者表达式。但在这种方式下，以下两行将具有同样的效果：

```
run [print :x :y]
print :x :y
```

指令 `run` 也可以用来达到另一个目的。如果它没有找到一系列的指令，只找到了一个字，那么它将把这个字当作文件名，并在其中寻找指令。

Run “vectors

上面的例子要求 Logo 寻找文件 `vectors.eli` 并载入它，尽可能多地学习。

3.7.4 控制程序的走向 (if,repeat,while)

包括 Logo 在内的编程语言的强大在于它们具有控制程序走向的能力。如果只允许你用指令编写那种从头到尾逐行执行的程序，你就不会那么容易地教会 Logo 去做那些真正很酷的事了。

这三个内建的指令可以用来控制 Logo 如何执行程序。

一个是 `if`。它被用来告诉 Logo 只有当给定的条件都满足时，才能执行下面的一系列指令。

```
If :a>0 [print “positive]
```

在这里，你告诉 Logo 只有当 `a` 的值大于 0 时才能打印输出字 `positive`。很显然，Logo 使你不仅能够定义当条件满足时做什么，而且能够定义当条件不满足时做什么：

```
if :a>0  
  [print “positive]  
  [print “negative “or “zero]
```

为了使程序行能够短一些，Logo 默认不仅在 `if` 所在的行寻找指令列表，而且在下一行寻找指令列表。总之，条件语句必须总是与 `if` 在同一行——就在 `if` 之后。说到条件，Logo 只是通过条件的值来判断是否满足条件。如果值为字 `true`，那么就满足条件，反之，则不满足条件。

另一个控制程序走向的指令是 `repeat`。你可以通过使用这个指令来要求 Logo 重复执行一系列的指令若干次。看下面的指令：

使用 `repeat` 来重复一件同样的事若干次。



```
repeat 5[print “Hello]
```

等同于下面的指令：

```
print “Hello  
print “Hello  
print “Hello  
print “Hello  
print “Hello
```

当然，如果你想把一个过程重复成千上

万次，使用 `repeat` 将会很有

效率。我们推荐不仅在重复次数多的时候使用 `repeat`，而且在不知道重复的次数时也用 `repeat`。在许多情况下，你并不知道需要重复的次数，但你可以通过计算得到。如下面所示，`chars` 的值决定了重复执行表中指令的次数。

```
Make “I 65
```

`while` 循环可以因条件的设定而持续一天以上的时间。



指令 `if` 被用来定义要走哪一条路去执行下面的指令。



```

repeat :chars
[
  print char :i
  make "I :i+1
]

```

使用 `repeat` 重复执行指令称作 `repeat` 循环。还有一种循环类型叫做 `while` 循环。正如你所猜想的那样，它使用指令 `while`。

```

While :a<10
[
  print :a
  make "a :a+1
]

```

`while` 循环与 `repeat` 循环类似，它们都是要求 Logo 多次指行一系列的指令。不同之处在，如果你使用 `while`，那么只有当条件不满足时才停止重复执行。上面的例子命令 Logo 在 `a` 的值小于 10 的时候打印输出 `a` 中的值。

3.7.5 定义过程 (to...end, output)

你已经知道要让 Logo 做事需要指定专门的过程。要教会 Logo 去做一些新的工作，必须有一种描述新过程的方式。一个新过程的定义开始于字 `to`，结束于字 `end`。紧跟 `to` 的后面你必须声明过程名（前面不加”）以及过程的参数。如果过程的前面需要参数，则在它的前面声明，如果过程的后面需要参数，则在它的后面声明。在这些声明和字 `end` 之间，你必须把构成过程的指令包括进去。

```

To greater :x :y
  if :x>:y [print "yes] [print "no]
end

```

上面定义了一个称为 `greater` 的过程，它需要两个参数。当你把过程定义给 Logo 之后，例如：

```
greater 5 6
```

将会打印输出 `no`。Logo 会给过程的每一个参数创建一个局部变量。变量名从第一行中的声明中获得。在本例中它们是 `x` 和 `y`。值得注意的是，你必须在每个参数的名字前面加上 `:`。为什么呢？首先，这会让 Logo 记住这些参数需要被赋值。其次，这是 Logo 能够区分过程名和参数名的唯一途径。

参数的顺序是由你和使用你定义的过程的人决定的，即使是自己使用这个过程，你也要用同样的顺序。例如，你可以用多种不同的方式定义过程 `greater`，但无论你如何选择，都要类似下面这样：

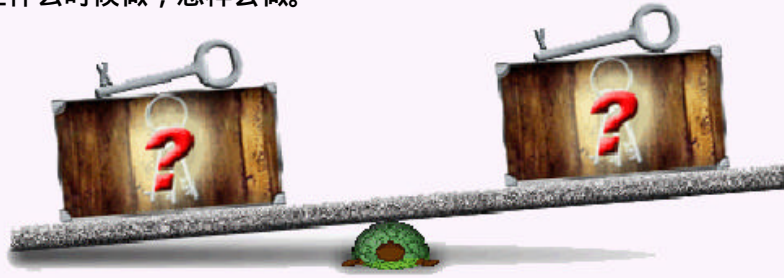
```

to :x greater :y
  if :x>:y [print "yes] [print "no]
end
5 greater 6

```

在 Logo 中能够做到的事，你都可以在过程中完成。例如，你可以定义变量和过程。但是，过程有一个特别的地方。有时你可能要定义一个过程，并且要返回一个值以便在以后的计算中使用。

在很多时候,把参数分开放置是会很便利的。在比较两个值大小的过程中,你可能更愿意把过程名放在参数之间.....这取决于你决定在什么时候做,怎样去做。



这样的值可以通过指令 `output` 来返回。如果你希望修改上面的例子以使它并不打印输出比较的结果,而是返回这个结果,那么你需要进行如下的声明:

```
to :x greater :y
  if :x>:y [output "yes] [output "no]
end
```

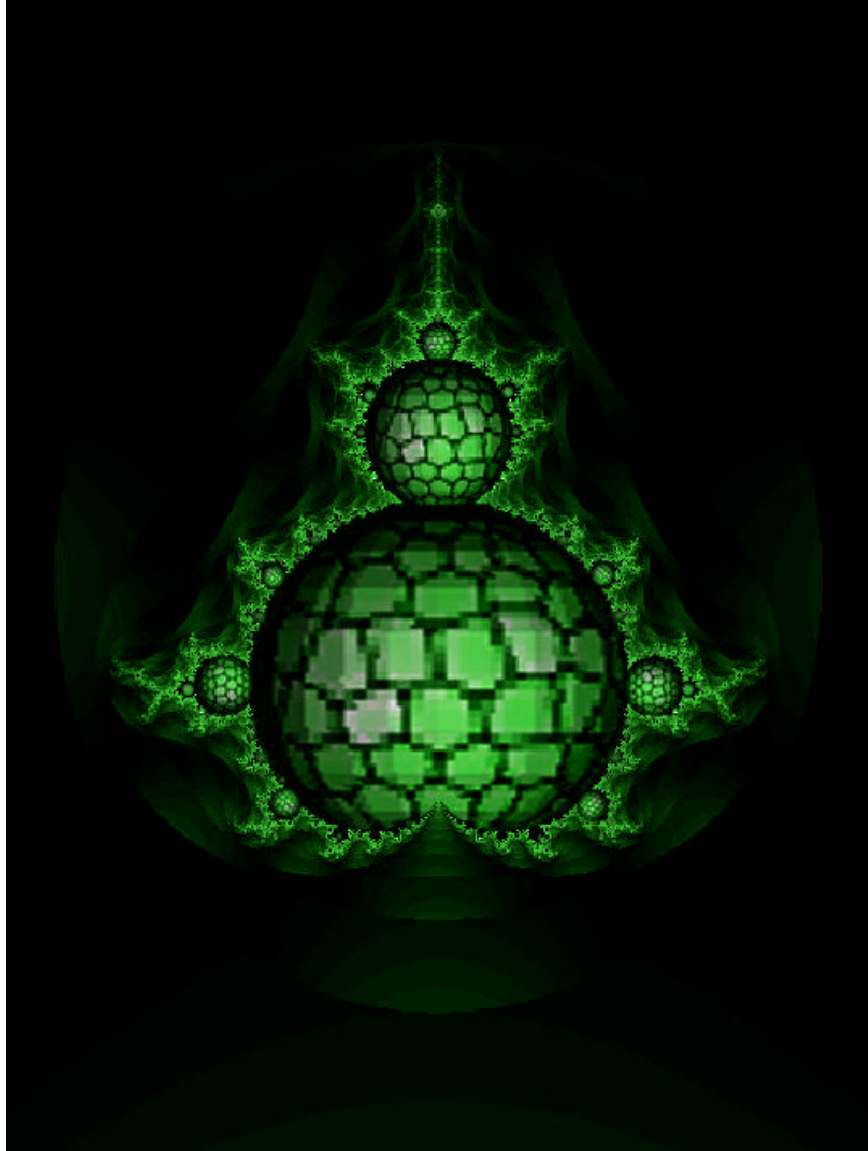
```
print 5 greater 6 ;no
if (5 greater 1)="yes [print "really?] ;yes
```

就如同上面所示,返回值可以被用在 `if` 语句之中。

当 Logo 执行了指令 `output`,它会立即停止执行这个过程。这个特性可被用于当你想提前中止过程的时候。这时,你可以只使用指令 `output` 而并不用指定返回何值。

Mandelbrot 分形

用海龟的身体建成



4. Logo 海龟

海龟是 Logo 的脸面。从很早的时候开始, Logo 软件就能控制一只海龟画出美丽的图形。甚至现在许多听说过 Logo 的人也听说过海龟作图。

海龟作图有广泛的含义——从海龟的程序控制一直到复杂的分形图形。但是所有这些都有一個中心角色——那就是海龟。

好了, 传统 Logo 的海龟是一个在你和你的程序控制之下的小三角形。你告诉它你想做什么, 它就会很好地完成你给的任务。

Elica 海龟本身具有预建立的一些最基本的过程, 如果你想让海龟画房子和树, 那么你需要先教会它。但现在 Elica 在启动时, 这些预建立的知识已经可以利用了。因此, 如果你喜欢使用海龟, 你的第一条命令必须是:

```
run "turtle
```

这将载入一个指令文件, 它将教会你的海龟基本的过程。

4.1 跳舞的海龟 (fd, bk)

理解海龟怎样移动的最好的办法就是把你自己想象成海龟。如果你听到向前 10 步的命令, 最大的可能是你将向前走 10 步。这与海龟要做的事一样。不同之处只是它的步子要比你的小得多, 并且过程 'forward' 的名字实际上被缩写为 'fd'。按照同样的方式, 后退由过程 'bk' 来完成。

下面的指令将要求海龟前进十步, 然后后退五步。

```
Fd 10
```

```
bk 5
```

在屏幕上你将看到海龟的踪迹显示为一条线段, 海龟则停在线段的中点。

这多么象在跳舞——你就是设计舞蹈动作的人, 而海龟则是跳舞的人。

4.2 左转, 右转 (lt, rt)

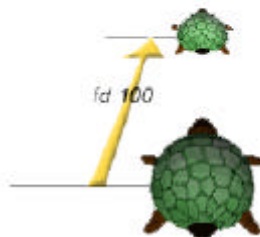
前进和后退不久将变成令人厌烦的舞步, 除非你领会到怎样使海龟左转和右转。因为这非常重要, 海龟已经知道如何做, 它只是在等待你的命令。让它转向左面只要说 lt (它来自于 left 或 left turn) 或 rt (right 或 right turn)。当然, 海龟不仅希望你告诉它朝哪个方向转, 而且希望知道转的程度。这个程度将被作为 lt 和 rt 的参数提供, 并且必须以度的形式来表示。

什么是度? 度一直被古代的数学家用来测量角度。因此, 如今我们仍用它来表示角的大小。一度有多大? 这个问题回答起来很复杂。一个数学家会告诉你一个圆周有 360 度, 但是很难想象一个圆周的 360 分之一是多大。

你上次是什么时候吃的一份比萨饼? 你还记得整个比萨饼被分成了几份吗? 如果它被分成了六份, 那么每一份是 60 度。如果分成了八份——每一份将是 45 度。请注意每一份角度的大小与比萨饼的大小无关。如果你把每个十美分的比萨饼和每个 20 美分的比萨饼都平均分成六份, 那么总共十二份每份都将是同样的 60 度。

另一种描述一度有多大的方式是看有指针的钟表。秒针每秒转 6 度, 因此, 一度是秒

前进和后退是海龟仅有的两种移动方式, 但要作出美丽的图形已经足够了。



转向左面并不是总意味着转向屏幕的左面。

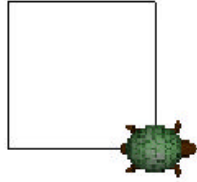


如果海龟的头是朝下的, 那么它的左面则相当于你的右面。

针一秒所转度数的六分之一。如果没有秒针也没关系。分针每分钟转 6 度，因而，一分钟相当于 6 度大小。一小时相当于 30 度。3 小时相当于 90 度。记住这个数值——90 度。当我们人类互相说向左转，我们的意思是向左转 90 度。Logo 海龟是计算机生灵，它不会象我们这样行事。这就是为什么你必须总是得告诉它转身的角度，尽管这对你来说是显而易见的。

现在，让我们来尝试一个小例子：

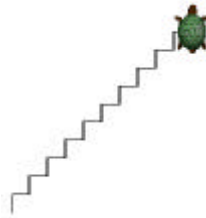
```
fd 50
lt 90
fd 50
lt 90
fd 50
lt 90
fd 50
```



如果你是一只海龟并且严格遵循这些指令，你将会画出一个正方形。你向前十步，左转，再向前十步，左转，向前十步，左转，向前走最后十步。

现在，让我们画出右面的梯子：

```
repeat 10
[ fd 10
  rt 90
  fd 10
  lt 90
]
```



4.3 做更多的事 (home, pu, pd, pd?)

除了这四个过程——fd, bk, lt 和 rt，海龟也知道如何直接跳到指定的位置，这被称作回家。（通常）家指的是屏幕的中央。过程的名称是 home。当你念了这个咒语，海龟直接跳回家里，并把它的头朝向上方（也就是朝向屏幕的上方）。

你已经注意到了当海龟在屏幕上移动时留下了踪迹，因此你可以跟踪它的路径。有时，当你画一些十分复杂的图形时，不可能用一条路径画出来。因此，你会要海龟停止在它的后面留下踪迹。那很容易，因为海龟能够留下踪迹不是因为它的脚脏，而是因为它拿着一只笔。在一开始，海龟把笔放下来，因此，当它移动时，笔就画出了图形。如果要想海龟把笔抬起来，可以用指令 pu (pen up)。从这时开始，海龟移动时将不会留下踪迹。

在你再次开始画图时，你要用相反的指令——pd (pen down)。无论何时，你都可以检测笔是抬起的还是放下的。如果你问海龟 pd?，它将以字“true 或”false 作出反应。

If pd?

```
[print “Pen is down’]
[print “Pen is up’]
```

4.4 时隐时现 (hide, show, shown?)

当画家正在作画时，他离画很近。但当他作画完毕时，他会从他的画旁撤回以使别人能够欣赏他的作品。海龟也是一样。你控制它画了一个美丽的图形，当海龟仍然待在那里，就在它最后停下的位置。当然，你可以让它走远一点躲起来，这很容易——抬起画笔，一直走到屏幕边缘以外。

但是，这有一个更好的方法。对海龟说 hide，它将戴上隐身帽。它仍然在那里，一点

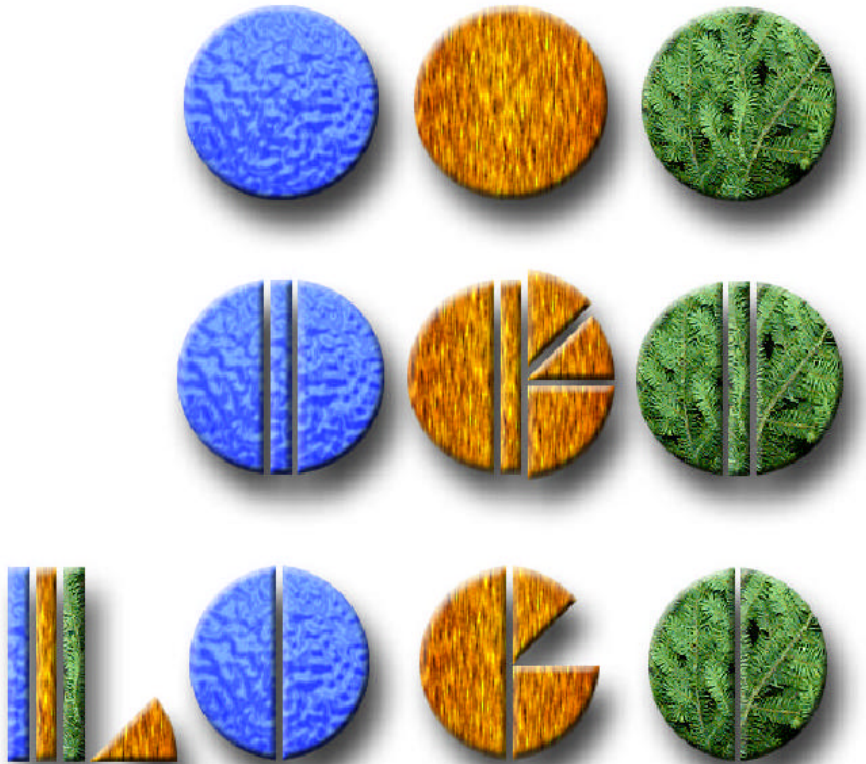
也没有改变位置，只是无法看见而已。

现在你仍然可以让海龟跳舞，但你将无法知道它在哪里，如果画笔被放下，那么将会画出笔迹。作为一个额外的作用，隐藏画笔可以加快画图的速度。

当你想重新显示海龟，你可以用指令 `show`。正如 `pu`, `pd` 和 `pd?` 的情形一样，你可以用指令 `shown?` 来获知海龟当前是隐藏的还是显示的。

Logo 命令

可以通过这种方式构造



5. Logo 和语言

如果你已经使用过其它的 Logo 软件，你就会想知道字和表的处理函数在哪儿。如果你让 Elica 处理表，它的核心并不知道如何去作。幸运的是，你可以教会它如何做。

Elica 诞生时已经有了学习 Logo 的课程。这里面就有关于如何教会 Elica 所有 Logo 过程的课程。为了让 Elica 成为一个完整的 Logo，你要用下面的指令：

```
run "logo
```

这会告诉 Elica 载入文件 logo.eli 并从里面学习。事实上，并不需要你来做这些事，因为 Elica 会自动为你做好这些事。

Logo 最重要的特性之一就是你可以让它处理字和表。显然，自然语言的句子经常是以字的列表来呈现的。因而 Logo 对语言的处理就称为字表处理。

5.1 这是什么？（number? Word? List?）

编写处理语句的程序时，你经常要检测隐藏在不同变量中的变量值的类型。你可以让 Logo 来做这些事，使用以下指令：number?，word?，以及 list?。这些指令将返回 true（如果它们的参数类型与它们的名称符合）或 false。

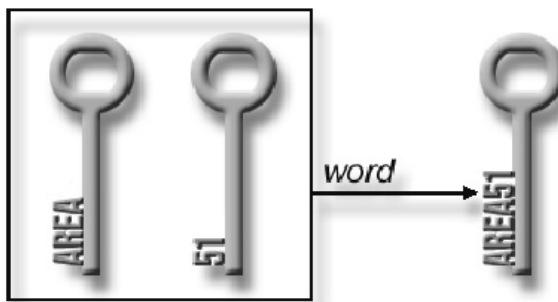
```
Print number? 5      ;true
print word? "K12     ;true
print list? "mice    ;false
print list? [m I c e] ;true
```

当你创建自己的数据类型查询函数时，这会是一个很好的实践。按照它们所要检测的数据类型来命名，并且，当然，在后面加上一个问号。

5.2 我要构造(word, list, se, set, fput, lput)

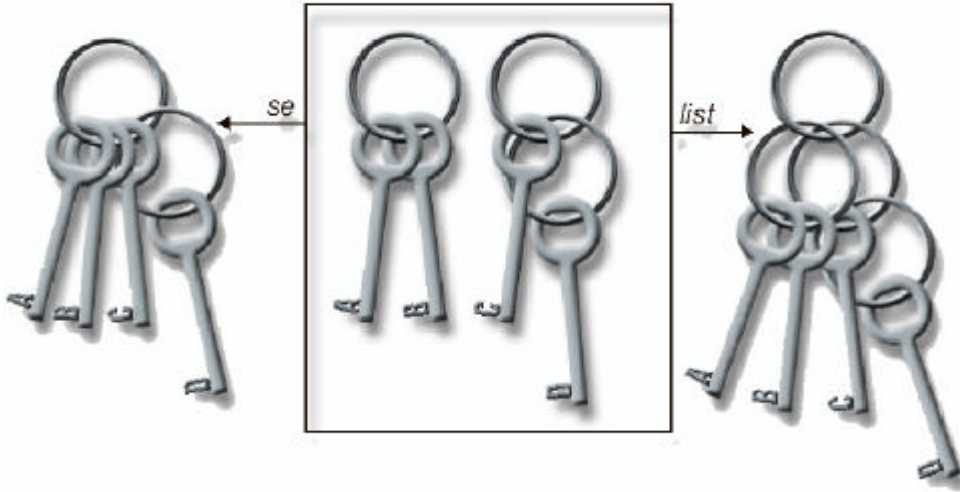
语言处理的核心是创建新的字和表。对于每种类型，Logo 都有相应的一个或多个函数来创建值。这些函数都可以接受多个参数。函数 word 被用来连接几个字成为一个字。List 被用来利用参数中提供的值作为元素来创建表。与之类似的函数是 se。它的名称来自于 sentence。就象 list，se 创建一个表，但是参数中的表都先被解包，它们里面的元素都作为所得到的表中的元素。

用钥匙 word 打开一个装看创建新钥匙过程的柜子。



```
Print word "area 51      ;area51
print list [a b] [c [d]] ;[[a b] [c [d]]]
print (list [a] "b [c])  ;[[a] b [c]]
print se [a b] [c [d]]   ;[a b c [d]]
```

柜子 `se` 和 `list` 装着的过程以两种不同的方式来创建新的钥匙串。`se` 把钥匙串合并到一个新钥匙环上，`list` 把钥匙串一起套入一个新的钥匙环上。



还有两个函数——`fput` 和 `lput`。它们被用来构建一个新表，并且同时决定新元素是加在给定表的开始还是结尾。

```
Print fput "x [a b]      ;[x a b]
print fput [x] [a b]   ;[[x] a b]
print lput "x [a b]    ;[a b x]
print lput [x] [a b]  ;[[a b] [x]]
```

5.3 我不想要它的全部，我只需要一点点。（`first`, `last`, `bf`, `bl`, `item`）

构建字和表是一个单向的过程。为了拥有全部的能力，`logo.eli` 教会了 `Elica` 不仅知道如何创建值，而且知道如何分解和提取它们的指定部分。

过程 `first` 要求 `Logo` 抽取值中排在前面的元素。如果值是一个字，那么得到的就是这个字的前面的字母。如果值是一个表，那么得到的就是这个表或设置的前面的元素。

在抽取前面的元素时，`Logo` 的缺省行为只是抽取最前面的一个元素。如果你想要抽取更多的元素，你可以强制 `Logo` 这样做。你只需简单地加上一个提供抽取数目的参数。

```
Print first "mint      ;m
print (first "mint 2)  ;mi
print first [a b f]    ;a
print (first [a b f] 2) ;[a b]
make "myset first :myset
```

事物具有对称性。`Logo` 不仅知道如何抽取前面的元素，而且知道如何抽取后面的元素。可以用指令 `last`。

```
Print last "mint      ;t
print (last [a b f] 2) ;[b f]
```

当对应到钥匙串上，`first`、`last`、`bf` 和 `bl` 被用来取出前面的或后面的钥匙来用，或者用剩下的钥匙。



为了方便，当你指定了哪一部分不保留的时候，Logo 也知道如何抽取字或表的其余部分。这里有两个函数：`bf` 和 `bl`，它们来自于 `butfirst` 和 `butlast`。`Bf` 所做的就是找出所有的元素，除了前面的一个或几个。类似的，`bl` 返回除了后面几个的所有元素。

```
Print bf "mint      ;int
print (bl "mint 3)  ;m
print (bf [a b f] 2) ;[f]
print bl [a b f]   ;[a b]
```

所有的这些用于提取的函数都被描述成与元素的开始，结束的位置有关。有时你需要面对一些特殊的位置：第-4个，第十个，第*i*个。你可以通过巧妙使用 `first` 和 `bf` 来达到目的，但 Logo 提供了另一个有用的函数 `item`。它有两个参数——你要提取的元素的指示和你要提取元素的对象。

```
Print item 3 [screen] ;r
print item 4 "clock   ;c
```

5.4 其它过程 (count, ascii, char, wait)

还有一些其它类别的过程。这一节，我们来做一些简短的描述。

字（表）的大小是指它包含的字母（元素）的数量，用 `count` 来计算。这个函数不能被用于设置列表，

```
print count "book    ;4
print count [b [o o] k] ;3
```

在你处理以 ASCII 提供的字母时，字母与数字之间的转换就显得非常重要。有如下两个函数：`ascii` 返回字母的 ASCII 码，`char` 返回 ASCII 码所对应的字母。

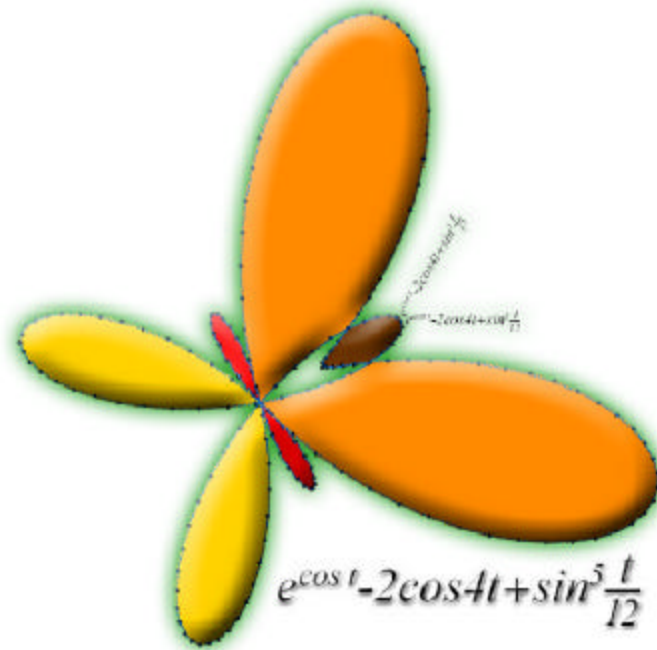
```
print ascii "B      ;66
print char 65       ;A
```

程序运行中的延迟可以用 `wait` 来实现。它只需要一个参数——等待的秒数。

```
wait 10    ;will pause for 10 seconds
wait 0.5   ;will pause for half a second
```

一只蝴蝶

由参数方程定义



6. Logo 和数学

最初制造计算机的目的是为了帮助人们进行快速和精确的计算。这种需要影响了包括 Logo 在内的大部分编程语言。因而你可以使用一些数学运算符和函数来编写一些计算的程序。

6.1 数学运算符

数学运算符在 Logo 中以过程的方式提供，它们需要两个参数——一个在左边，一个在右边。

6.1.1 算术运算符 (+, -, *, /, ^)

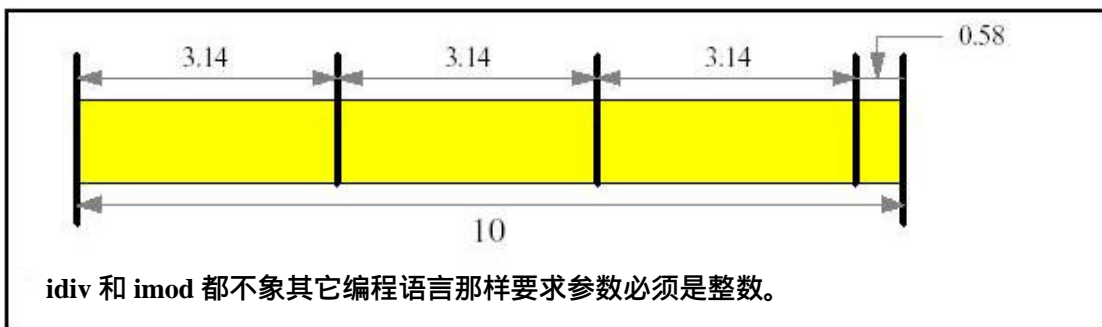
首先 Logo 要通过执行 run “logo 来学习如何使用这几个重要的运算符，它们可以用来构建数学表达式。运算符+用来把数值相加，-用来做减法，*用来做乘法，/用来做除法，而^用来做幂运算。下面的例子展示了如何使用这些运算符：

```
print 2+4      ;6
print 2-4      ;-2
print 2*4      ;8
print 2/4      ;0.5
print 2^4      ;16
print 2^3+3*4  ;20
```

6.1.2 整数运算符 (idiv, imod)

Logo 中只有两个整数运算符——idiv 和 imod。前一个被用来求得一个数是另一个数的多少整数倍。后一个被用来求得余数。简单举几个例子：

```
print 10 idiv 3.14 ;3
print 10 imod 3.14 ;0.58
print 3*3.14+0.58 ;10
```



在上面的指令中，10 除以 3.14 商 3，余 0.58。

6.2 数学函数

Logo 中的数学函数赋予你完全的能力来应付复杂的计算。所有的数学函数都以过程的方式提供，就象数学运算符一样。当然，所有的函数的参数都要放在右边。

有一些数学运算你已经在学校里学过了，另一些对你来说还很陌生。如果你发现了你不知道的函数，请查阅详细的数学资料。

6.2.1 符号函数 (abs, neg, sign)

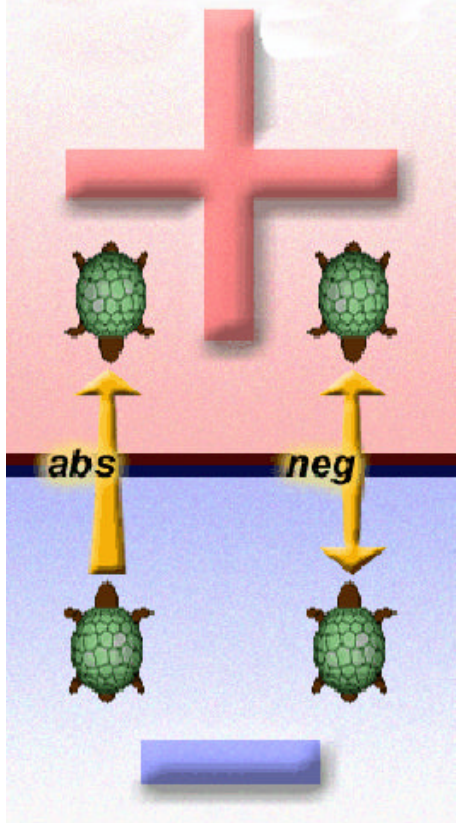
在 Logo 中，如果你想检测或改变一个数值的符号状态时，你要用到符号函数。如

果你使用函数 `abs`，那么 Logo 将移去数值的符号，将它转换为正数。

另一个函数是 `neg`，它将把一个正数变为一个负数，反之亦然。实际上，求一个数的相反数等同于用 0 去减它。

```
print abs 20 ;20
```

`neg` 是位于正值和负值之间的一个双向门，而 `abs` 只允许通向正值一边。



```
print abs -15 ;15
```

```
print neg 10 ;-10
```

```
print neg -7 ;7
```

`abs` 和 `neg` 都返回了同样的数，归根到底是因为符号的改变。为了查看数值的符号，你要用到函数 `sign`。如果是负数则返回 -1，正数则返回 +1，0 则返回 0。

```
print sign -3 ;-1
```

```
print sign 0 ;0
```

```
print sign 4 ;1
```

6.2.2 指数函数 (`sqrt`, `exp`, `logn`)

指数函数处理数值的幂运算。Logo 从文件 `logo.eli` 中学会了三个函数。`sqrt` 被用来计算数值的平方，`exp` 可以求得 e 的任意次方 (e 是一个非常重要的数学常量，约等于 2.718——更精确的值可以通过 `exp 1` 计算)。第三个函数 `logn` 是函数 `exp` 的逆运算。

```
print sqrt 3 ;9
```

```
print exp 1 ;2.71828182845905
```

```
print logn 10 ;2.30258509299405
```

6.2.3 舍去函数 (`trunc`, `round`)

我们经常希望能够得到整数的计算结果，但是计算机给的是却不是。这时你希望能让 Logo 按你定义的方式来修改数值。最简单的方式是截去

数值的小数部分。这可用函数 `trunc` 来实现。另一种方式你可以定义你所要保留的精度，这可以用 `round` 来实现。

```
print trunc 3.2 ;3
```

```
print trunc -4.8 ;-4
```

```
print round 319.425 0 ;319
```

```
print round 319.425 2 ;319.43
```

```
print round 319.425 -1 ;320
```

6.2.4 三角函数 (`sin`, `cos`, `tan`, `cotan`)

所有这四个三角函数：`sin`, `cos`, `tan`, `cotan` 都被 Logo 支持。它们只需要一个放在右边的参数，并且用度来表示（不是弧度或其它的单位）。

```
print "sin(30)= ' sin 30
```

```
print "cos(30)= ' cos 30
```

```
print "tan(30)= ' tan 30
```

```
print "cotan(30)= ' cotan 30
```

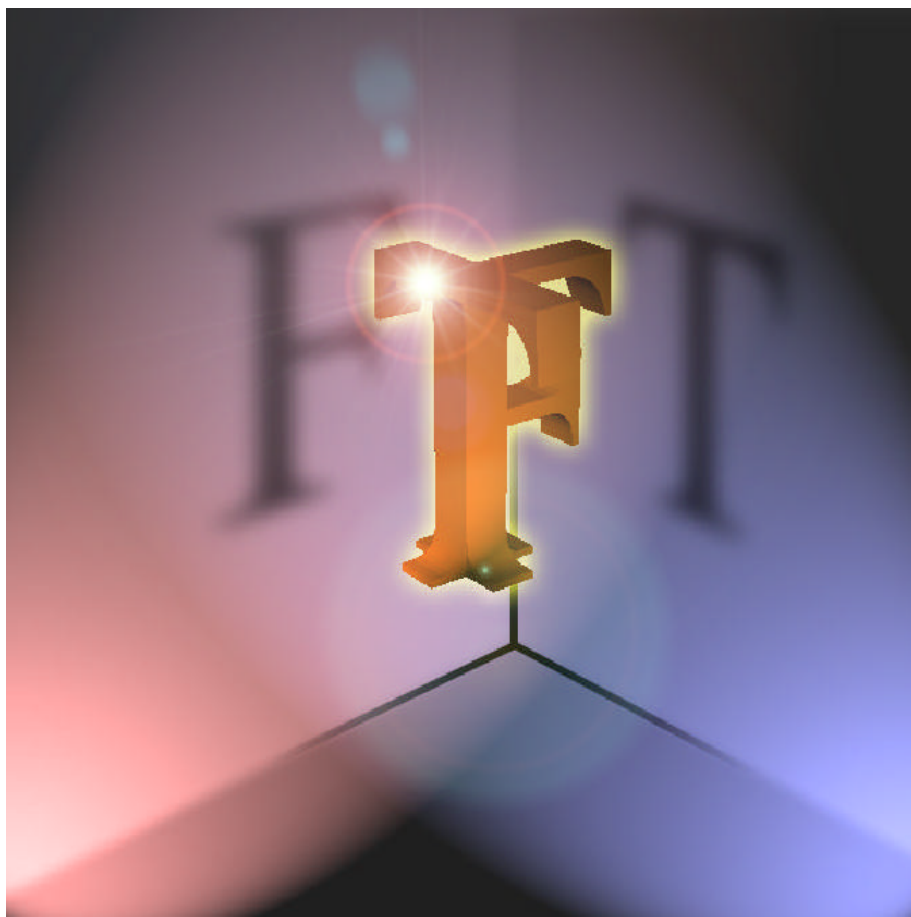
6.2.5 随机函数

除了上面提到的函数外，还有一个独特的函数。在许多模拟中，你会用到随机数。Logo 中有这样一个函数，无论你何时调用它，它都会返回一个随机数。它就是 `random`。它可以不带参数运行，也可以带着参数运行。如果没有参数，它将返回 0 与 1 之间的随机数。如果有参数，那么它将返回 0 与参数值之间的随机数。在两种情况下，随机数可以等于 0，但总是小于它的上限。

```
print random      ;3.346296521136537
print random      ;0.175248484592885
print random 100  ;98.8024020334706
print random 100  ;60.4138494469225
```

真或假

它们都是现实的映射



7.Logo 和逻辑

Logo 与逻辑的关系体现在你编写程序时可以加入逻辑表达式。这些表达式与其它的表达式不同，因为它们经过求值运算后得到 true（有效）和 false（无效）。

7.1 比较运算符（=, <, >, <=, >=, <>）

当你对两个值进行比较时要用到什么样的过程呢？有六种不同的方式，它们每个都对应着一个运算符。通过它们你可以了解两个值的关系是相等（=）还是不相等（<>），一个值比另一个值是小于（<），大于（>），小于等于（<=）还是大于等于（>=）。

如下所示，这些值可以是数值，字和表：

```
print 4=5           ;false
print 41>12         ;true
print "mouse<"mice ;false
print "[h a t s]>=[c a t s] ;true
print "[h a t s]<>[c a t s] ;true
```

数值按它们本身的大小做比较。字按照字母顺序比较（它们在字典中的排列顺序），但大写字母总是排在最前面，因而 A...Z 排在 a...z 的前面。表的比较通过一种更复杂的方式。当你要求 Logo 比较两个表时，它就开始逐个比较里面的元素。当它遇到一对不相等的元素时它们之间的关系就决定了这两个表的关系。如果一个表比另一个表短，并且它里面的元素与另一个表相同，那么这个短的表小一些。

在任何时候，如果你对两个无法比较的值进行比较，例如，你想知道 5 和 [1 2 3] 是否相等，Logo 将返回 false 作为结果。反之，它将比较这两个值并返回 true 或 false。

7.2 逻辑运算符（and, or, not）

逻辑运算符能处理逻辑值 true 和 false，而不能处理非逻辑值。运算符 and 在两个参数都为真时返回 true。运算符 or 在两个参数中至少有一个为真时返回 true。第三个运算符 not 在参数为 false 时返回 true，反之，返回 false。

如果你要在 if 和 while 结构中使用复杂的条件判断，那么使用条件运算符会很合适。

```
if (5<:a) and (:a<10)
```

```
  [print :a "'is between 5 and 10']
```

```
if (5<=:a) or (:a>=10)
```

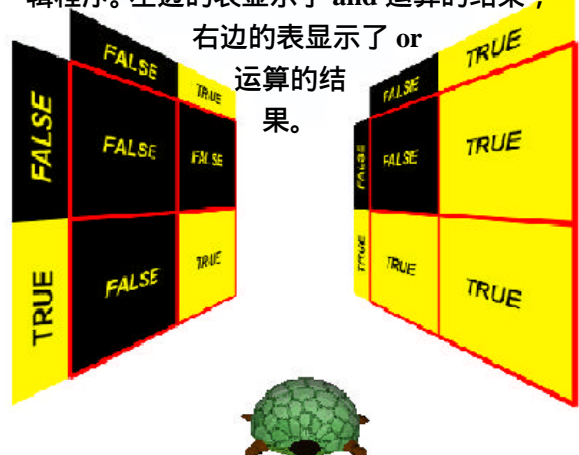
```
  [print :a "'is not between 5 and 10']
```

```
if not (5<:a)
```

```
  [print :a "'is less or equal to 5']
```

Logo 的逻辑世界帮助人们编写和体验逻辑程序。左边的表显示了 and 运算的结果，

右边的表显示了 or 运算的结果。



当你在使用逻辑运算符时，要注意如果象下面这样使用，Logo 是不会领会你的意图的：

```
print :a>b and :c
```

它能够进行这样的操作，但结果可能不是你所要求的。因此你必须明确地告诉海龟你的意图是什么：

```
print (:a>b) and (:a>c)
```

```
print (:a>b) and :c
```

```
print :a>(:a and :c)
```

译者的话

非常感谢 Elica 的作者为我们编写了这个册子。我认为它不仅是介绍了 Logo 本身，这
里面的很多地方对学习其它的编程语言也是不无裨益的。由于时间仓促，错误在所难免，
希望见谅。

刘红军
2002 年 8 月
于大连