

GNU Emacs Lisp 编程入门

Programming in Emacs Lisp:
An Introduction

(美) Robert J. Chassell 著

毛文涛 吕芳 译
洪峰 审校



机械工业出版社
China Machine Press

Free Software Foundation

GNU技术文档精粹

GNU Emacs Lisp编程入门

(美) Robert J. Chassell 著

毛文涛、吕芳 译

洪峰 审校



机械工业出版社
China Machine Press

本书的作者罗伯特·卡塞尔是自由软件基金会的合创人之一，也是理查德·斯托曼博士青年时期结交的挚友，他精通GNU Emacs Lisp的每一个方面。本书是一本GNU Emacs Lisp的编程入门，全书循序渐进地介绍了GNU Emacs Lisp编程的各种基础知识和方法，文笔流畅、讲解透彻，对GNU Emacs用户提高对它的理解和运用帮助极大。

Robert J. Chassell: Programming in Emacs Lisp: An Introduction.

Copyright © 1990, 1991, 1992, 1993, 1994, 1995, 1997, 1999, Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

图书在版编目 (CIP) 数据

GNU Emacs Lisp编程入门 / (美) 卡塞尔 (Chassell, R. J.) 著; 毛文涛等译. -北京: 机械工业出版社, 2001. 5

(GNU技术文档精粹)

书名原文: Programming in Emacs Lisp: An Introduction

ISBN 7-111-08862-X

I. G… II. ①卡… ②毛… III. LISP语言-程序设计 IV. TP312

中国版本图书馆CIP数据核字 (2001) 第19381号

机械工业出版社 (北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑: 姚蕾

北京昌平奔腾印刷厂印刷 · 新华书店北京发行所发行

2001年5月第1版第1次印刷

787mm × 1092mm 1/16 · 13.5印张

印数: 0 001-5 000册

定价: 38.00元

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换

致中国读者

Calendars, email, writing in general, programming, and debugging programs: GNU Emacs gives you tools for all these actions, and gives you even more. GNU Emacs is truly an integrated environment. Emacs Lisp is the language in which most of Emacs is written. It is a simple yet powerful language that is easily understood and learned.

FSF-CHINA led by Hong Feng is doing us all a great service by translating this document from English into Chinese. The work will bring the joy, the efficiency, and the power of GNU Emacs to many people. I hope you will gain as much from reading this book as I did from writing it.

My best wishes to you. *Bob Chassell* Robert J. Chassell

译者序

GNU Emacs 长期以来一直是自由软件基金会的旗舰产品。它是由理查德·斯托曼 (Richard Stallman) 博士为 GNU 工程开发的第一个自由软件。在所有目前已开发的 GNU 软件中, GNU Emacs 的作用和地位是非常突出的, 因为几乎所有其他的自由软件基金会的工具都是用 GNU Emacs 编写出来的。

从编程实践上看, GNU Emacs 有许多特点。其中最为突出的一个特点是斯托曼在创造 GNU Emacs 编辑器时非常巧妙地揉合了用 Lisp 语言和 C 语言编写的代码。

斯托曼利用 Lisp 语言编写 GNU Emacs 的大部分代码不是偶然的。Lisp 语言发明于 20 世纪 50 年代, 并广泛地应用于人工智能研究领域, 而斯托曼早年曾经在麻省理工学院人工智能实验室工作过很长时间, 所以他非常熟悉 Lisp 语言的优点。Lisp 是解释性的语言, 用 Lisp 开发的程序具有良好的可读性, 因此将它用于处理文本编辑这样的任务是非常合适的。当然解释性的语言与硬件直接作用时其效率与编译性的语言相比则显得不高, 这样的任务还是由 C 语言代码来完成比较合适。

斯托曼的一个天才构想就是利用 C 语言编写与硬件直接作用的 GNU Emacs 模块(如显示模块), 而绝大多数文本编辑模块则统统利用 Lisp 语言来编写。Lisp 语言是一种功能全面的编程语言, 其解释器被嵌入了 GNU Emacs 中后, 用户便可以用它自行对 GNU Emacs 进行定制。这一几乎无限的灵活性是其他编辑器很难做到的。在 GNU Emacs 中, Emacs 的 Lisp 代码模块和 C 代码模块组织良好, 它们相互取长补短, 相得益彰。

为了保持源代码的可读性与一致性, 斯托曼将 GNU Emacs 中的 C 语言代码模块的函数名写得很像 Lisp 函数名。如果不仔细研究模块底层的细节, 那么实在很难将它们两者区分开。实际上, 如果没有特殊的目的, 作者希望你不要去辨认它们之间的区别。这样, 在扩充和维护代码时, 工作就会变得容易多了。经过这么一番匠心独运的安排, GNU Emacs 成为了一种“高级的、自带文档的、可定制的和可扩充的实时显示的编辑器”。难怪如此众多的自由软件开发人员整天都可以坐在计算机旁运行 GNU Emacs 而乐此不疲。

自从 GNU Emacs 问世以来, 众多专家一致评价说: 斯托曼的这一天才的泛对称设计思想极富艺术性, 具有方法论研究的永久价值。

由于使用 GNU Emacs 的开发人员数量众多, 运行的平台又很广泛, 因此各种使用 GNU Emacs Lisp 编写的增强功能包也源源不断地产生了, 有些功能包还成为标准 GNU Emacs 发行版本的一部分。GNU Emacs 今天仍处在不断的演进和完善过程中, 它代表着一种文化。这一切都是由于斯托曼当年的天才设计思想所引发的良好局面。

当然, 要了解和运用这些新功能, 或者自己动手开发所希望的特性来增强和扩充 GNU Emacs, 使用 GNU Emacs Lisp 编程是必不可少的, 而通过本书学习编程可以说是一个良好的起点。

本书的作者，罗伯特·卡塞尔（Robert Chassell）先生是自由软件基金会的合创人之一，也是斯托曼青年时期结交的挚友。他精通 GNU Emacs Lisp 的每一个方面。在这本编程入门著作中，他循序渐进地介绍了 GNU Emacs Lisp 编程的各种基础知识和方法，文笔流畅、讲解透彻，对 GNU Emacs 用户提高对它的理解和运用帮助极大。

另外，由于各种 Lisp 解释器大同小异，因此，一旦通过这一教程理解了 GNU Emacs Lisp 的工作原理，那么你所掌握的知识和技巧对于学习其他版本的 Lisp 语言（如 AutoLisp）或者现在日趋流行的 Python 等解释性语言也会具有触类旁通的指导意义。

这部著作的中文版是自由软件基金会中国研究院组织人员翻译的第一本 GNU 自由软件文档。我们出版这一著作以及《GNU Emacs 技术手册》、《GNU Emacs Lisp 技术手册》等中文版的主要目的就在于更好地引导读者体会编写程序、分析自由软件源代码的乐趣。这对于在中国催生新的黑客具有极大的促进作用，因为我相信黑客道的真正本质就是“热爱编写程序、并享受通过编写程序而变得更加聪明这一过程”。

自由软件基金会中国研究院

洪 峰

fred@mail.rons.net.cn

前言

GNU Emacs 文本编辑器的绝大多数代码是用一种被称为 Emacs Lisp 的编程语言编写的。用这种语言编写的代码就是这个软件——指令集——用户通过它向计算机发布命令以告诉计算机如何工作。Emacs 就是为使你能用 Emacs Lisp 编写新的代码并能方便地作为编辑器的扩展部分来安装而设计的。这也是为什么 Emacs 被称作“可扩展的编辑器”的原因。

因为 Emacs 的确提供了比编辑更多的功能，它或许应当被称为“可扩展的计算环境”，但是这个词显得口气太大。同样，在 Emacs 中做的任何事情——查找玛雅年代和月相、简化多项式、调试代码、管理文件、阅读信件以及撰写图书——所有这些活动都是“编辑”这个词所包含的。

虽然人们经常将 Emacs Lisp 与文本编辑器联系在一起，但它却是一种完整的计算机编程语言。可以像使用任何其他编程语言一样使用它。

也许你希望理解编程；也许你希望扩展 Emacs；或者也许你希望成为一名程序员。这本入门教程就是为你开始 Emacs Lisp 之旅而设计的：引导你学习编程基础，更重要的是告诉你如何自学提高。

在整本书中，你将看到为数不多的几个程序例子，你可以在 Emacs 中运行它们。如果用 GNU Emacs 的 Info 阅读本书文档，可以在例子程序出现时运行它们。（这很容易做到，我们将在例子出现时作进一步解释。）同时，你也可以将这本教程作为一本印刷的图书一样，当你坐在计算机旁运行 Emacs 时阅读。（这就是我所喜欢的方式，我喜欢印刷出来的纸版图书。）如果你身边没有一个运行的 Emacs，你仍旧可以阅读这本书，但是在这种情况下，最好将其作为一本小说或者是一本你未到过的某个国家的导游手册来阅读：这样读起来会较有趣，但是你的收获会与亲身体验不同。

本教程用许多篇幅介绍 GNU Emacs 用到的代码。教程的这些设计安排有两个目的：一是使读者熟悉真实的正在运行的代码；二是使读者熟悉 Emacs 工作的方式。弄清一个编辑器如何工作是很有趣的。同样，我希望读者养成浏览源代码的习惯。读者可以从中学习并开阔思路。有了 GNU Emacs，就像拥有一个龙穴宝藏一样。

除了将 Emacs 当做一个编辑器、将 Emacs Lisp 当做一门编程语言学习之外，书中的例子和导引将使读者通晓将 Emacs 作为 Lisp 编程环境的机会。GNU Emacs 支持编程，并提供了你将乐于使用的工具，如“M-”（这是调用 find-tag 命令的键）。你还可以学习缓冲区和对象，这些都是编辑环境的组成部分。学习 Emacs 的这些功能就像熟悉家乡周围的新路一样。

最后，我希望传授一些使用 Emacs 来学习编程时读者不知道的技巧。你可以经常用 Emacs 来解决那些困扰你的问题，并用它们做一些新奇的事情。这种自力更生不仅是一种乐趣，更是一种优点。

读者对象

这本教程是作为入门读物为那些非编程人员编写的。如果你是一名程序员，可能并不满足

这本初级读物。原因在于你可能已经通过阅读参考手册成了专家。或许本书的组织方式已经使你失去兴趣。

一位评论过本书的编程专家曾这样对我说：

我更喜欢从参考手册中学习（编程）。我“潜入”每一个段落，并在段落之间跃出“水面”呼吸空气。

当到达一个段落结尾时，我假定这个主题已经结束。我知道了需要知道的所有东西（也可能存在这样的可能性，那就是下一个段落将对这个主题作更加详细的讲解）。我希望一份认真撰写的参考手册不要出现太多的冗余，并且它应指引我学习所希望的知识。

这本入门教程并不是为这类读者撰写的！

首先，我试图就每一件事情至少说上三遍：第一次介绍它；第二次在文中详细展现它的内容；第三次在不同的地方揭示它，或者复习一下。

其次，我几乎从不将一个主题的所有内容放在一个地方讲完，更不放在某一段中。以我思考的方式而言，那样做会给读者强加过重的负担。相反，我试图仅仅解释在那种情况下你需要知道的东西（有时会增加一点点附加信息，在后面读到这些附加信息的正式介绍时无需惊讶）。

阅读本书的时候，我并不指望你第一次就学会所有的东西。通常的情况是你仅需要对某些内容略微了解。我希望已经组织好本书，为你提供了足够的信息，并提醒你哪些是重要的线索，且着重讲述它们。

你应当“潜入”某些段落，除此以外没有其他方法。但是我已尽力减少这类段落。希望本书成为一座可以攀越的小山，而不是一座使人畏缩的高峰。

《GNU Emacs Lisp 编程入门》还有一个姊妹篇，那就是《GNU Emacs Lisp 技术手册》。那本手册比本书更详细。在那本手册中，关于任何一个话题的所有信息都集中在一个地方。如果你喜欢上面引用的那位程序员所欣赏的学习方法，那么应当掉头去阅读那本技术手册。当然，阅读完这本编程入门后，在编写自己的程序时，你会发现那本技术手册很有用。

Lisp 的历史

Lisp 是20世纪50年代晚期在麻省理工学院为研究人工智能而被首先发展起来的。Lisp语言的强大功能使之也能用于其他目的，比如编写编辑器命令。

GNU Emacs Lisp 在很大程度上得益于20世纪60年代在MIT编写的 MacLisp。它同时也得益于在20世纪80年代成为标准的 Common Lisp。然而，Emacs Lisp 比 Common Lisp 简单得多（标准的 Emacs 发行版本中包含一个可选的扩展文件“cl.el”，它为 Emacs Lisp 增加了许多 Common Lisp 的特性）。

初学者注意

如果你不知道 GNU Emacs，阅读本书仍旧有益。但是，如果仅仅是想学习在计算机屏幕上如何操作，建议你学习 Emacs。可以通过在线教程自学如何使用 Emacs。为使用在线教程，按下组合键 C-h t（这意味着同时按下并释放 CTRL 和 h 键，然后按下并释放 t 键）。

同样，我经常在提到 Emacs 的标准命令时列出激活该命令时应按下的键序列，然后在括号

中给出命令名，例如：`M-C-\ (indent-region)`。这意味着`indent-region`命令通常是通过输入键序列：`M-C-\`来激活的（如果你愿意的话，可以改变激活这个命令的键序列，这称作“重新绑定”。参见16.11节，“键图”）。缩写 `M-C-\` 意味着同时输入 `META` 键、`CTRL` 键和 `\` 键。有时，像这样的一个组合键也叫做一个键和弦，因为它类似于在钢琴上演奏一个和弦。如果你的键盘没有 `META` 键，可以用前缀 `ESC` 键取代它。在这种情况下，`M-C-\` 意味着按下并释放 `ESC` 键，然后同时按下并释放 `CTRL` 键和 `\` 键。

如果用 GNU Emacs 的 `Info` 阅读这份文档，只用空格键就能翻阅整本书（可以输入 `C-h i` 然后选择 `Info` 来学习）。

关于术语的说明：当仅仅提到 `Lisp` 这个词时，常常是指各变种的 `Lisp`；但是当提到 `Emacs Lisp` 时，就是特指 `GNU Emacs Lisp` 了。

致谢

感谢所有对这本书提供了帮助的人们。特别感谢 Jim Blandy、Noah Friedman、Jim Kingdon、Roland McGrath、Frank Ritter、Randy Smith、Richard M. Stallman 和 Melissa Weisshaus。同时要感谢 Philip Johnson 和 David Stampe 耐心的鼓励。书中的所有错误都由我负责。

目 录

致中国读者	
译者序	
前言	
第1章 列表处理	1
1.1 Lisp列表	1
1.1.1 Lisp原子	1
1.1.2 列表中的空格	2
1.1.3 GNU Emacs帮助你输入列表	3
1.2 运行一个程序	3
1.3 产生错误消息	4
1.4 符号名和函数定义	5
1.5 Lisp解释器	5
1.6 求值	6
1.7 变量	7
1.8 参量	8
1.8.1 参量的数据类型	9
1.8.2 作为变量和列表的值的参量	10
1.8.3 数目可变的参量	10
1.8.4 用一个错误类型的数据对象作为参量	10
1.8.5 message函数	11
1.9 给一个变量赋值	12
1.9.1 使用set函数	13
1.9.2 使用setq函数	13
1.9.3 计数	14
1.10 小结	15
1.11 练习	15
第2章 求值实践	16
2.1 缓冲区名	16
2.2 获得缓冲区	17
2.3 切换缓冲区	18
2.4 缓冲区大小和位点的定位	19
2.5 练习	20
第3章 如何编写函数定义	21
3.1 defun特殊表	21
3.2 安装函数定义	23
3.3 使函数成为交互函数	24
3.4 interactive函数的不同选项	25
3.5 永久地安装代码	26
3.6 let函数	27
3.6.1 let表达式的各个部分	27
3.6.2 let表达式例子	28
3.6.3 let语句中的未初始化变量	29
3.7 if特殊表	29
3.8 if-then-else表达式	31
3.9 Lisp中的真与假	32
3.10 save-excursion函数	33
3.11 回顾	35
3.12 练习	37
第4章 与缓冲区有关的函数	38
4.1 查找更多的信息	38
4.2 简化的beginning-of-buffer函数定义	38
4.3 make-whole-buffer函数的定义	40
4.4 append-to-buffer函数的定义	41
4.4.1 append-to-buffer函数的交互表达式	42
4.4.2 append-to-buffer函数体	42
4.4.3 append-to-buffer函数中的save-excursion	43
4.5 回顾	45
4.6 练习	46

第5章 更复杂的函数	47	8.1.5 总结zap-to-char函数	71
5.1 copy-to-buffer函数的定义	47	8.1.6 第18版中zap-to-char函数的 实现方法	72
5.2 insert-buffer函数的定义	48	8.1.7 progn表达式主体	73
5.2.1 insert-buffer函数中的交互 表达式	48	8.2 kill-region函数	74
5.2.2 insert-buffer函数体	49	8.3 delete-region函数: 接触C	75
5.2.3 用if表达式(而不是or表达式) 编写的insert-buffer函数	49	8.4 用defvar初始化变量	76
5.2.4 函数体中的or表达式	50	8.5 copy-region-as-kill函数	77
5.2.5 insert-buffer函数中的let 表达式	51	8.6 回顾	82
5.3 beginning-of-buffer函数的完 整定义	52	8.7 查找练习	83
5.3.1 可选参量	52	第9章 列表是如何实现的	85
5.3.2 带参量的beginning-of-buffer 函数	53	第10章 找回文本	88
5.3.3 完整的beginning-of-buffer 函数	55	10.1 kill环总览	88
5.4 回顾	56	10.2 kill-ring-yank-pointer变量	88
5.5 &optional参量练习	57	10.3 练习: 使用yank函数和nthcdr函数	89
第6章 变窄和增宽	58	第11章 循环和递归	90
6.1 save-restriction特殊表	58	11.1 while	90
6.2 what-line函数	59	11.1.1 while循环和列表	91
6.3 练习: 变窄	60	11.1.2 一个例子: print-elements -of-list	92
第7章 基本函数: car、cdr、cons	61	11.1.3 使用增量计数器的循环	93
7.1 car和cdr函数	61	11.1.4 使用减量计数器的循环	96
7.2 cons函数	63	11.2 递归	98
7.3 nthcdr函数	64	11.2.1 使用列表的递归函数	99
7.4 setcar函数	65	11.2.2 用递归算法代替计数器	100
7.5 setcdr函数	66	11.2.3 使用cond的递归例子	102
7.6 练习	67	11.3 有关循环表达式的练习	102
第8章 剪切和存储文本	68	第12章 正则表达式查询	104
8.1 zap-to-char函数	69	12.1 查询sentence-end的正则表达式	104
8.1.1 interactive表达式	69	12.2 re-search-forward函数	105
8.1.2 zap-to-char函数体	70	12.3 forward-sentence函数	106
8.1.3 search-forward函数	70	12.4 forward-paragraph: 函数的金矿	109
8.1.4 progn函数	71	12.5 创建自己的“TAGS”文件	115
		12.6 回顾	116
		12.7 练习: 使用re-search-forward	117
		第13章 计数: 重复和正则表达式	118
		13.1 count-words-region函数	118

13.2 用递归的方法实现单词计数	123	16.9 自动加载	158
13.3 练习:统计标点符号的数量	127	16.10 一个简单的功能扩充: line-to-top-of-window	159
第14章 统计函数定义中的单词数	128	16.11 键图	161
14.1 计数什么?	128	16.12 X11的颜色	162
14.2 单词或者符号是由什么构成的?	129	16.13 V19中的小技巧	163
14.3 count-words-in-defun函数	130	16.14 修改模式行	163
14.4 在一个文件中统计几个函数定 义的单词数	132	第17章 调试	165
14.5 查找文件	133	17.1 debug	165
14.6 lengths-list-file函数详解	134	17.2 debug-on-entry	166
14.7 在不同文件中统计几个函数定义 的单词数	135	17.3 debug-on-quit和(debug)	168
14.8 在不同文件中递归地统计单词数	137	17.4 源代码级调试器edebg	168
14.9 为图形显示准备数据	138	17.5 调试练习	170
14.9.1 对列表排序	138	第18章 结论	171
14.9.2 制作一个文件列表	139	附录A the-the函数	173
第15章 准备柱型图	144	附录B kill环的处理	175
15.1 graph-body-print函数	148	B.1 rotate-yank-pointer函数	175
15.2 recursive-graph-body-print 函数	150	B.2 yank函数	180
15.3 需要打印的坐标轴	151	B.3 yank-pop函数	182
15.4 练习	151	附录C 带坐标轴的图	184
第16章 配置你的“.emacs”文件	152	C.1 print-graph函数的变量列表	185
16.1 全站点的初始化文件	152	C.2 print-Y-axis函数	185
16.2 为一项任务设置变量	153	C.2.1 题外话:计算余数	186
16.3 开始改变“.emacs”文件	153	C.2.2 构造一个Y轴元素	188
16.4 文本和自动填充模式	154	C.2.3 创建Y坐标轴	189
16.5 邮件别名	156	C.2.4 print-Y-axis函数的最后形式	190
16.6 缩排模式	156	C.3 print-X-axis函数	190
16.7 一些绑定键	156	C.4 打印整个图形	194
16.8 加载文件	157	C.4.1 测试print-graph函数	197
		C.4.2 绘制函数中单词和符号数的图形	198
		C.4.3 打印出来的图形	202

第1章 列表处理

对那些没有学过Lisp语言的人而言，Lisp是一种奇怪的编程语言。在Lisp代码中到处都是括号。有些人把Lisp这个词当成是“Lots of Isolated Silly Parentheses”（大量分离的愚蠢的括号）的缩写。但是这种说法是没有根据的。Lisp是指“LIST Processing”（列表处理），和通过把列表放置在括号之间来处理列表（甚至是列表的列表）的编程语言。括号标记了列表的边界。有时一个列表用一个单引号或表示标记“'”开头。列表是Lisp的基础。

1.1 Lisp 列表

在Lisp中，一个列表看起来像这个样子：'(rose violet daisy buttercup)。这个列表以单引号开始。这个列表也可以写成下面这种你可能比较熟悉的形式：

```
'(rose
  violet
  daisy
  buttercup)
```

在这个列表中，元素是四种不同的花的名称，它们之间用空格分隔开，并用括号括起来，就像花在一个用石头墙围起来的花园中一样。

列表也可以有数字，如列表(+ 2 2)一样。这个列表有一个加号“+”，后接两个“2”，它们之间用空格分隔开。

在Lisp中，数据和程序都以同样的方式表示；也就是说，它们都是由空格分隔的、由括号括起来的单词、数字或者其他列表的列表。（因为如果一个程序看起来像数据，那它就很容易作为其他程序的数据；这是Lisp的一个很有用的特性。）（附带提一下，对前面这对括号而言，它不是Lisp列表，因为其中使用了标点符号“；”和“。”来分隔不同的元素。）

下面是另一个列表，这个列表中有另外一个列表：

```
'(this list has (a list inside of it))
```

这个列表的元件是单词“this”、“list”、“has”和内部列表“(a list inside of it)”。内部列表由“a”、“list”、“inside”、“of”和“it”几个词组成的。

1.1.1 Lisp 原子

在Lisp中，我们刚才说到的词被称作原子（atom）。这个术语来自原子一词的历史含义，即原子意味着“不可分”。只要提到Lisp，我们在列表中使用的词就不可以再被分成更小的部分，这在程序中也一样。数字、单个字符（如“+”）都是如此。另一方面，不像原子，一个列表可以拆分成不同的部分。（参见第7章，“基本函数：car、cdr和cons”。）

在一个列表中，原子是由空格——分隔的。原子可以紧接着括号。

从技术上说, Lisp 中的一个列表有三种可能的组成方式: 括号和括号中由空格分隔的原子; 括号和括号中的其他列表; 括号和括号中的其他列表及原子。一个列表可以仅有一个原子或者完全没有原子。一个没有任何原子的列表就像这样: `()`, 它被称作空列表。与所有的列表都不同的是, 可以把一个空列表同时看作既是一个原子, 也是一个列表。

原子和列表的书面表示都被称作符号表达式 (*symbolic expression*), 或者更简洁地被称作 *s*-表达式 (*s-expression*)。表达式这个词, 既可以指书面的表示, 也可以指一个原子或者一个列表在计算机中的内部表示。人们常常无区别地使用表达式这个词。(同样地, 在许多书中, 表格 (*form*) 这个词也被看作是表达式的同义词)。

顺便说一下, 构成我们的宇宙的原子是在它们被认为是不可分的时候命名的。但是, 人们已经发现, 物理上的原子不再是不可分的。原子的一部分可以被分出来, 或者可以裂变成大致相等的两个部分。物理上的原子在它们的更真实的本质被发现之前就被过早地命名。在 Lisp 中, 某种类型的原子, 例如一个数组, 可以被分成更小的部分, 但是分割数组的机制与分割列表的机制是不同的。只要是涉及列表操作, 列表中的原子就是不可分的。

与英语中一样, Lisp 原子的组成字母的意义与由这些字母构成的单词的含义是不同的。例如, 代表 “South American sloth” 的单词 “ai” 与 “a” 和 “i” 这两个字母是完全不同的。

自然界中有许多种原子, 但是在 Lisp 中只有几种原子: 例如, 数字(比如 “37”、“511” 或 “1729”)和符号(比如 “+”、“foo” 和 “forward-line”)。以上列出的这些单词都是符号。在 Lisp 的日常使用习惯中, “原子” 一词不太常用, 因为程序员经常试图更明确地表示他们处理的原子类型。Lisp 编程几乎都是关于列表中的符号的(且有时是关于数字的)。(附带说明一下, 上述3个单词^①是 Lisp 中一个正确的列表, 因为它包含的是原子。在这种情况下, 原子是一些由空格分隔、用括号括起来的符号, 其中没有任何对 Lisp 而言是非法的标点符号。)

另外, 双引号中的文本——不论是句子或者是段落——都是一个原子。下面是一个这样的例子:

```
'(this list includes "text between quotation marks.")
```

在 Lisp 中, 所有用双引号括起来的文本, 包括标点符号和空格, 都是单个原子。这种原子被称作串 (*string*) (代表 “字符串” 之意), 并且它是一种事物的分类, 以便让计算机能够打印出可供阅读的信息。字符串是不同于数字和符号的一种原子, 在使用上也是不同的。

1.1.2 列表中的空格

列表中空格的数目无关紧要。从 Lisp 语言的角度来说,

```
'(this list
  looks like this)
```

与下面的列表完全等价:

```
'(this list looks like this)
```

① 这里是指英文版中前面括号中的3个单词, 即 “(and sometimes numbers)”。——译者注

上面两个例子对 Lisp 而言是同一个列表。这个列表由符号“this”、“list”、“looks”、“like”、“this”按上面这种顺序组成。

多余的空格和换行符只不过是使人们易于阅读而设计的。当 Lisp 读取表达式时，它剔除了所有多余的空格（但是原子间至少需要一个空格以使原子分隔开来）。

这看起来很奇怪，我们已经看到的例子涵盖了 Lisp 列表的几乎所有情况。其他任何一个 Lisp 列表看起来都或多或少与上面的例子相似，只是列表可能更长更复杂。简要地说，列表放在括号之间，串放在引号之间，符号看起来像一个单词，而数字看起来就像一般的数字一样。（当然，对于特定的情况，方括号、大括号、句点和一些特殊的字符都是可以使用的，然而我们暂时先不去理会它们。）

1.1.3 GNU Emacs 帮助你输入列表

如果你在 GNU Emacs 中使用 Lisp 交互模式或是 Emacs Lisp 模式来输入一个 Lisp 表达式，那么你将可以使用多种命令来使 Lisp 表达式排成易于阅读的格式。例如，按 TAB 键会使光标所在的行自动缩排到适当的位置。用于在一个区域内正确缩排的常用命令是 M-C-\\。设计缩排，是为了使读者看清列表的哪些元素是属于哪个列表的——缩排时子列表的元素要比外围列表的元素更缩进一些。

另外，当输入一个右括号时，Emacs 立即使光标跳到与之配对的左括号处，因此就可以看清它到底是哪一个列表。这个功能非常有用，因为 Lisp 中输入的每一个列表必须有一对匹配的左括号和右括号（有关 Emacs 模式的详细情况可以参见《GNU Emacs 技术手册》中“主要模式”一节）。

1.2 运行一个程序

Lisp 中的一个列表——任何列表——都是一个准备运行的程序。如果你运行它（在 Lisp 的术语中，这称为求值），计算机将完成三件事情：只返回列表本身；告诉你一个出错消息；或者，将列表中的第一个符号当做一个命令，然后执行这个命令。（当然，你通常真正希望的是上述三件事情中的最后一件）。

单引号“'”，也就是在前一节例子中的列表前面的引号，被称作一个引用（*quote*）。当单引号位于一个列表之前时，它告诉 Lisp 不要对这个列表做任何操作，而仅仅是按其原样。但是，如果一个列表前面没有引号，这个列表中的第一个符号就很特别了：它是一条计算机要执行的命令（在 Lisp 中，这些命令被称作函数）。上面说到的列表（+ 2 2）就没有引号在前面，因此 Lisp 将“+”号理解为一指令，用来对这个列表的其余部分进行操作；在这种情况下，就是将其后续的数字相加。

如果在 GNU Emacs 的 Info 中阅读到这个列表，可以这样对它求值：将光标移到下面列表的右括号之后，然后按 C-x C-e：

```
(+ 2 2)
```

你将看到数字 4 显示在回显区。（用术语来说，刚才做的就叫做：“对一个列表求值”。回显区是屏幕底部的那一行，它显示或者“回显”文本。）现在，对下面带引号的列表进行同样的操

作，将光标置于下面列表之后，然后按 C-x C-e:

```
'(this is a quoted list)
```

此时，将会看到 (this is a quoted list) 显示在回显区。

在这两种情况下，你所做的是给 GNU Emacs 内一个叫做 *Lisp* 解释器的程序一个命令，即给解释器一个命令，使之求值。*Lisp* 解释器的名字来自于由一个人来完成某项任务这个词，这个人给出一个表达式的值，即他解释了它。

同样可以对一个不是列表的一部分的原子(即不被括号括起来的原子)求值。同样，*Lisp* 解释器将人能理解的表达式翻译成计算机的语言。但是在讨论这个问题之前(参见1.7节，“变量”)，我们先讨论在出错时 *Lisp* 解释器会做些什么。

1.3 产生错误消息

如果不小心出了错，也不要太担心。现在我们将给 *Lisp* 解释器一个命令，使之产生一个错误消息。这是一个无害的动作，确实，我们时常会有意识地产生错误消息。一旦理解了这种术语，错误消息是能提供有用信息的。与其说是错误消息，不如说是有助的消息。它们像是一个给在异国他乡的游客的路标，破译它们可能很艰难，但是一旦理解了，它们就成了指路明灯。

我们将要做的，就是对一个没有引号并且其第一个元素不是一个有意义的命令的列表求值。下面是一个与我们用到过的列表几乎完全相同的列表，但是它前面没有单引号。将光标移到它后面并输入 C-x C-e:

```
(this is an unquoted list)
```

这一次，将会看到下面的内容显示在回显区:

```
Symbol's function definition is void: this
```

(另外，终端可能对你发出鸣叫声——有些终端这样做，有些不这样。有些则闪烁。这仅仅是一个示警的装置。)只要键入任何键，消息都将迅速消失，哪怕是仅仅移动了光标。

根据已有的知识，我们几乎可以读懂这条错误消息。我们知道“Symbol”一词的意义。在这种情况下，它指列表中的第一个原子，就是“this”一词。上述错误消息中的“function”一词在前面已经出现过一次。它是非常重要的一个词。对我们的目的而言，可以将它定义为：一个“函数”(function)就是一组告诉计算机做什么的计算机指令(从技术上说，这个符号告诉计算机到什么地方去寻找这些指令，但是这是一个我们暂时可以忽略的复杂问题)。

现在，我们可以理解这条错误消息了：“Symbol's function definition is void: this”。其中的“Symbol”是指“this”。这个错误消息是指没有为“this”定义让计算机执行的任何指令。

这条错误消息中稍显奇怪的用词“function definition is void”，是 Emacs Lisp 实现方式的体现：即当一个符号没有一个对应的函数定义时，那个应当包含指令的位置就是“空的”(void)。

另一方面，因为可以成功地通过对表达式 (+ 2 2) 求值来执行 2 加 2 计算，那就是说，+ 号一定有一组计算机执行的指令，这些指令就是将 + 号后面的数字加起来。

1.4 符号名和函数定义

在已经讨论过的内容的基础上，我们可以结合介绍 Lisp 的另外一个特性。这个重要的特性就是，一个符号，如 + 号，它本身并不是计算机执行的指令本身。相反，符号或许是临时用于定位函数或者一组指令的。我们所看到的只不过是一个名字而已，通过这个名字可以找到相应的指令。人的名字起着同样的作用。我可以被叫做“Bob”，然而，我并不是“B”、“o”、“b”这几个字母，而是与这个特定的生命形式相联系的有意识的人。这个符号不是我，但是它可以用于指我。

在 Lisp 中，一组指令可以连到几个名字，例如，计算机的加法指令可以连接到符号“Plus”，也可以连接到符号“+”。在人类社会，我可以叫做“Robert”，也可以叫做“Bob”，或者其他什么词。

另一方面，一个符号一次只能有一个函数定义与其连接，否则，计算机就会疑惑到底使用哪个函数。如果在人群中出现这种情况，那么只有一个人可以叫做“Bob”。然而，一个名字指向的函数定义是容易改变的。（参见3.2节“安装函数定义”。）

因为 Emacs Lisp 很大，它常以一定的方式将符号命名，这种命名方法可以确定函数属于 Emacs 的哪一个部分。因而，处理 Texinfo 的所有函数的名字都以“texinfo-”开头，所有用于阅读电子邮件的函数的名字都以“rmail-”开头。

1.5 Lisp 解释器

根据前面的内容，现在可以来说明在我们命令 Lisp 解释器对一个列表求值时它做些什么了。首先，它查看一下在列表前面是否有单引号。如果有，解释器就为我们给出这个列表。如果没有引号，解释器就查看列表的第一个元素，并判断它是否是一个函数定义。如果它确实是一个函数，则解释器执行函数定义中的指令。否则解释器打印一个错误消息。

这就是 Lisp 工作的方式。简单极了。后面我们会介绍更复杂一些的内容，但是这些是基本的。当然，为了编写 Lisp 程序，需要知道如何书写函数定义并将它们连向函数名，以及如何使得这样做不使自己和计算机都搞混。

现在，我们介绍第一种复杂的情况。除了列表之外，Lisp 解释器可以对一个符号求值，只要这个符号前没有引号也没有括号包围它。在这种情况下，Lisp 解释器将试图像变量一样来确定符号的值。（参见1.7节，“变量”。）

出现第二种复杂的情况是因为一些函数异常并且以异常的方式运行。那些异常的函数被称作特殊表（*special form*）。它们用于特殊的工作，例如定义一个函数。但是这些特殊表并不多。在下面几章，你将接触几个更加重要的特殊表。

第三种也是最后一种复杂的情况是：如果 Lisp 解释器正在寻找的函数不是一个特殊表，而是一个列表的一部分，则 Lisp 解释器首先查看这个列表中是否有另外一个列表。如果有一个内部列表，Lisp 解释器首先解释将如何处理那个内部列表，然后再处理外层的这个列表。如果还有一个列表嵌入在内层列表中，则解释器将首先解释那个列表，然后逐一往外解释。它总是首先处理最内层的列表。解释器首先处理最内层的列表是为了找到它的结果。这个结果可以由