Chapter 1

第一章

1. Introduction

简介

Most of the GNU Emacs text editor is written in the programming language called Emacs Lisp. You can write new code in Emacs Lisp and install it as an extension to the editor. However, Emacs Lisp is more than a mere “extension language”; it is a full computer programming language in its own right. You can use it as you would any other programming language.

大多数的GNU Emacs 文本编辑器是用Emacs Lisp语言编写的。你可以编写Emacs Lisp新的代码，并将其作为编辑器的扩展进行安装。然而，Emacs Lisp不单纯是一门“扩展语言”，它是凭借自身能力成为一门的完整的计算机语言。你可以像使用其他编程语言一样使用它。

Because Emacs Lisp is designed for use in an editor, it has special features for scanning and parsing text as well as features for handling files, buffers, displays, subprocesses, and so on. Emacs Lisp is closely integrated with the editing facilities; thus, editing commands are functions that can also conveniently be called from Lisp programs, and parameters for customization are ordinary Lisp variables.

Emacs Lisp是为在编辑器中使用而设计的，因此它除了有扫描和分析文本的特殊功能外还可以处理文件、缓冲区、显示器、子进程等。Emacs Lisp同编辑工具紧密结合，因此，可以从lisp程序中方便的调用编辑命令，而且自定义参数也是普通的lisp变量。

This manual attempts to be a full description of Emacs Lisp. For a beginner's introduction to Emacs Lisp, see An Introduction to Emacs Lisp Programming, by Bob Chassell, also published by the Free Software Foundation. This manual presumes considerable familiarity with the use of Emacs for editing; see The GNU Emacs Manual for this basic information.

本手册尝试完整的描述Emacs Lisp。由Bob Chassell编著、自由软件基金会出版的 Emacs Lisp编程入门可以作为Emacs Lisp的初学者入门教程。

Generally speaking, the earlier chapters describe features of Emacs Lisp that have counterparts in many programming languages, and later chapters describe features that are peculiar to Emacs Lisp or relate specifically to editing.

一般来说，前面的章节中描述了Emacs Lisp和其他编程语言类似的功能，后面的章节中描述Emacs Lisp特有的或编辑相关的功能。

This is edition 3.0 of the GNU Emacs Lisp Reference Manual, corresponding to Emacs version 23.3.

这是GNU Emacs Lisp参考手册的第三版，对应Emacs 23.3。

1.1 Caveats

前言

This manual has gone through numerous drafts. It is nearly complete but not flawless. There are a few topics that are not covered, either because we consider them secondary (such as most of the individual modes) or because they are yet to be written. Because we are not able to deal with them completely, we have left out several parts intentionally.

这本手册经过无数次的修改。它几近完成，但并非无懈可击。有一些主题并未涉及，要么我们认为它们是次要的（如大部分的个人模式），要么是尚未开始编写它们。几个部分我们不能完全论述，所以我们故意忽略了它们。

The manual should be fully correct in what it does cover, and it is therefore open to criticism on anything it says—from specific examples and descriptive text, to the ordering of chapters and sections. If something is confusing, or you find that you have to look at the sources or experiment to learn something not covered in the manual, then perhaps the manual should be fixed. Please let us know.

该手册在它涉及的部分应完是全正确的，因此，对于它所说的，从具体的例子和说明性文字到章节顺序的编排，都欢迎批评。如果有困惑的地方，或者你发现有必须查看源码或者做实验才能学到本手册中未涉及的部分，那么可能该修正手册了 。请让我们知道这些。

As you use this manual, we ask that you send corrections as soon as you find them. If you think of a simple, real life example for a function or group of functions, please make an effort to write it up and send it in. Please reference any comments to the node name and function or variable name, as appropriate. Also state the number of the edition you are criticizing.

当你使用这个手册的时候，我们要求您一发现更正就发送给我们。如果你想到一个功能或功能组在真实生活中的简单例子，请努力把它写下来，并发送给我们。请指明评论所对应的章节、函数或者变量名字，还有版本编号。

Please send comments and corrections using M-x report-emacs-bug.

请使用M-x report-emacs-bug将意见和更正发送给我们。

1.2 Lisp History

Lisp (LISt Processing language) was first developed in the late 1950s at the Massachusetts Institute of Technology for research in artificial intelligence. The great power of the Lisp language makes it ideal for other purposes as well, such as writing editing commands.

Lisp（列表处理语言）是在20世纪五十年代晚期在麻省理工学院为研究人工智能而首先发展起来的。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。

Dozens of Lisp implementations have been built over the years, each with its own idiosyncrasies. Many of them were inspired by Maclisp, which was written in the 1960s at MIT's Project MAC. Eventually the implementors of the descendants of Maclisp came together and developed a standard for Lisp systems, called Common Lisp. In the meantime, Gerry Sussman and Guy Steele at MIT developed a simplified but very powerful dialect of Lisp, called Scheme.

多年来，已完成的lisp实现有几十个，各具特质。它们中的很多得益于二十世纪六十年代MIT 的MAC项目中所编写的Maclisp。最终Maclis后裔的那些实现合并起来并发展为Lisp系统的标准，即所谓的Common Lisp。与此同时，麻省理工学院的Gerry Sussman and Guy Steele开发出一种简单但很强大的的Lisp的方言，称为Scheme。

GNU Emacs Lisp is largely inspired by Maclisp, and a little by Common Lisp. If you know Common Lisp, you will notice many similarities. However, many features of Common Lisp have been omitted or simplified in order to reduce the memory requirements of GNU Emacs. Sometimes the simplifications are so drastic that a Common Lisp user might be very confused. We will occasionally point out how GNU Emacs Lisp differs from Common Lisp. If you don't know Common Lisp, don't worry about it; this manual is self-contained.

GNU Emacs Lisp在很大程度上得益于Maclisp，少部分得益于Common lisp。如果你了解Common Lisp，你会发现许多相似之处。然而，Common Lisp的许多功能已经被省略或简化，以减少GNU Emacs内存需求。有时这些简化如此剧烈，以至于Common Lisp的用户可能会非常困惑。我们偶尔会指出GNU Emacs Lisp和 Common Lisp的不同之处。如果你不知道Common Lisp，不担心;本手册是自我完备的。

A certain amount of Common Lisp emulation is available via the cl library. see Overview.

CL库中有大量可用的Common Lisp仿真。请看概述。

Emacs Lisp is not at all influenced by Scheme; but the GNU project has an implementation of Scheme, called Guile. We use Guile in all new GNU software that calls for extensibility.

Emacs Lisp根本没有受到Scheme的影响；但是GNU 项目有一个叫做Guile的Scheme的实现。在所有要求扩展性的新GNU 软件中使用Guile。

* 1. Conventions

约定

This section explains the notational conventions that are used in this manual. You may want to skip this section and refer back to it later.

本节介绍了本手册中使用的符号约定。你可能想跳过这一节，以后回来再看。

* + 1. Some Terms

一些术语

Throughout this manual, the phrases “the Lisp reader” and “the Lisp printer” refer to those routines in Lisp that convert textual representations of Lisp objects into actual Lisp objects, and vice versa. See Printed Representation, for more details. You, the person reading this manual, are thought of as “the programmer” and are addressed as “you.” “The user” is the person who uses Lisp programs, including those you write.

在本手册中，短语“Lisp读取器”和“Lisp打印机”是指在Lisp中routines，它将Lisp对像的文本表示转换成实际的Lisp对像，反之亦然。详细信息键打印表示。，读取本手册的你被认为是“编程人员”，被称呼为“你”。“用户”是指那些使用lisp程序（包括你写的）的人。

Examples of Lisp code are formatted like this: (list 1 2 3). Names that represent metasyntactic variables, or arguments to a function being described, are formatted like this:*first-number*.

Lisp代码例子格式如下：(list 1 2 3)。元语法变量，或者正在描述的函数的参数的名字的格式如下：*first-number*。

* + 1. nil and t

In Lisp, the symbol nil has three separate meanings: it is a symbol with the name ‘nil’; it is the logical truth value false; and it is the empty list—the list of zero elements. When used as a variable, nil always has the value nil.

Lisp中，符号nil有三个独立的含义：一个名叫‘nil’的符号；布尔值为假；一个元素个数为零的空列表

As far as the Lisp reader is concerned, ‘()’ and ‘nil’ are identical: they stand for the same object, the symbol nil. The different ways of writing the symbol are intended entirely for human readers. After the Lisp reader has read either ‘()’ or ‘nil’, there is no way to determine which representation was actually written by the programmer.

对于lisp读取器而言，“（）”和“nil”是相同的：它们表示同一个对像---符号nil。以不同的方式书写符号完全是为了人类读者。不管lisp读取器读到‘()’还是‘nil’，都没有办法确定程序员实际上写的是哪种表示方式。

In this manual, we write () when we wish to emphasize that it means the empty list, and we write nil when we wish to emphasize that it means the truth value false. That is a good convention to use in Lisp programs also.

在本手册中，当想强调空列表时写做()，强调布尔值为假时写做nil。这是也是lisp程序中的一个好习惯。

(cons 'foo ()) ; 强调空列表

(setq foo-flag nil) ; 强调布尔值为假

In contexts where a truth value is expected, any non-nil value is considered to be true. However, t is the preferred way to represent the truth value true. When you need to choose a value which represents true, and there is no other basis for choosing, use t. The symbol t always has the value t.

在一个期望布尔值的上下文中，任何不为nil的值被认为是真，然而，t也是代表布尔值为真的首选方式。当你需要选择一个值来代表真，没有其他作为选择基准，就使用t。符号t使用值为t。

In Emacs Lisp, nil and t are special symbols that always evaluate to themselves. This is so that you do not need to quote them to use them as constants in a program. An attempt to change their values results in a setting-constant error. See Constant Variables.

Emacs Lisp中，nil和t是特殊的符号，它们总是用来赋值给自己。这也就是说你在程序中将它们作为常量使用的时候不需要引用它们。企图改变它们的值会引起一个常量设置错误。See Constant Variables。

— Function: booleanp object

函数：booleanp object

Return non-nil if object is one of the two canonical boolean values: t or nil.

返回值非nil值如果object是两个标准布尔值（t或者nil）中的一个。

* + 1. Evaluation Notation

求值符号

A Lisp expression that you can evaluate is called a form. Evaluating a form always produces a result, which is a Lisp object. In the examples in this manual, this is indicated with ‘’:

可以求值的lisp表达称为表。对表求值总是会产生一个lisp对像作为结果。本手册中的例子中，他将用‘⇒’指示出来。

(car '(1 2))

⇒ 1

You can read this as “(car '(1 2)) evaluates to 1.”

可以将这个读作“(car '(1 2)) 求值得 1.”

When a form is a macro call, it expands into a new form for Lisp to evaluate. We show the result of the expansion with ‘==>’. We may or may not show the result of the evaluation of the expanded form.

当表是一个宏调用时，它将展开为一个新表进行求值。我们用‘==>’来指示展开结果。我们可能会显示展开表的求值结果。

(third '(a b c))

==> (car (cdr (cdr '(a b c))))

⇒ c

Sometimes to help describe one form we show another form that produces identical results. The exact equivalence of two forms is indicated with ‘==’.

又是为了说明一个表，会显示另外一个产生同样结果的表。两个表的精确等价用‘==’表示。

(make-sparse-keymap) == (list 'keymap)

* + 1. Printing Notation

打印符号

Many of the examples in this manual print text when they are evaluated. If you execute example code in a Lisp Interaction buffer (such as the buffer ‘\*scratch\*’), the printed text is inserted into the buffer. If you execute the example by other means (such as by evaluating the function eval-region), the printed text is displayed in the echo area.

本手册中的很多例子求值时会打印文本。如果在lisp交互缓冲区中执行示例代码（比如‘\*scratch\*’缓冲区），被打印的文本会插入到缓冲区中。如果通过其他方法执行例子（比如通过求值函数eval-region），被打印文本会在回显区中显示。

Examples in this manual indicate printed text with ‘-|’, irrespective of where that text goes. The value returned by evaluating the form (here bar) follows on a separate line with ‘’.

(progn (prin1 'foo) (princ "\n") (prin1 'bar))

-| foo

-| bar

⇒ bar

本手册中的例子用‘-|’指示打印文本，不论文本到哪里。对表求值返回的值处于‘⇒’之后，出现在一个单独的行中。

* + 1. Error Messages

错误消息

Some examples signal errors. This normally displays an error message in the echo area. We show the error message on a line starting with ‘error-->’. Note that ‘error-->’ itself does not appear in the echo area.

一些例子提示错误。这样通常在回显区显示一个错误消息。我们用一个以‘error-->’开头的行来显示错误消息。注意：‘error-->’本身不会出现在回显区。

(+ 23 'x)

error--> Wrong type argument: number-or-marker-p, x

* + 1. Buffer Text Notation

缓冲区文本符号

Some examples describe modifications to the contents of a buffer, by showing the “before” and “after” versions of the text. These examples show the contents of the buffer in question between two lines of dashes containing the buffer name. In addition, ‘-!-’ indicates the location of point. (The symbol for point, of course, is not part of the text in the buffer; it indicates the place between two characters where point is currently located.)

一些例子通过显示文本的“先”“后”版本描述了缓冲区内容的修改。这些例子在包含缓冲区名字的两条虚线之间显示了正在考虑的缓冲区的内容

---------- Buffer: foo ----------

This is the -!-contents of foo.

---------- Buffer: foo ----------

(insert "changed ")

⇒ nil

---------- Buffer: foo ----------

This is the changed -!-contents of foo.

---------- Buffer: foo ----------

* + 1. Format of Descriptions

描述的格式

Functions, variables, macros, commands, user options, and special forms are described in this manual in a uniform format. The first line of a description contains the name of the item followed by its arguments, if any. The category—function, variable, or whatever—appears at the beginning of the line. The description follows on succeeding lines, sometimes with examples.

本手册中以一种统一的格式描述函数，变量，宏，命令，用户选项和特殊的表。描述的第一行包含项目名称，若有参数，紧跟其后。该行开始是类别（函数，变量或者其他）。下一行是描述，有时还有例子。

* + - 1. A Sample Function Description

一个简单的函数描述

In a function description, the name of the function being described appears first. It is followed on the same line by a list of argument names. These names are also used in the body of the description, to stand for the values of the arguments.

函数描述中，首先是函数名字，其次是变量列表，它们处于同一行。描述部分也会使用这些名字来代表参数的值。

The appearance of the keyword &optional in the argument list indicates that the subsequent arguments may be omitted (omitted arguments default to nil). Do not write &optional when you call the function.

变量列表中出现的关键词&optional表明随后出现的参数可以被省略（省略参数默认为nil）。当调用该函数时不要写&optional。

The keyword &rest (which must be followed by a single argument name) indicates that any number of arguments can follow. The single argument name following &rest will receive, as its value, a list of all the remaining arguments passed to the function. Do not write &rest when you call the function.

关键字&rest（必须紧跟一个单一的参数名字）表明其后可以使任意数量的参数。&rest之后的单一的参数名字将会接收传递给该函数的剩余参数作为它的值。

Here is a description of an imaginary function foo:

这里是一个假想函数foo的描述。

— Function: foo integer1 &optional integer2 &rest integers

The function foo subtracts integer1 from integer2, then adds all the rest of the arguments to the result. If integer2 is not supplied, then the number 19 is used by default.

函数foo从integer2减去integer1，然后将剩余的参数加到结果上。如果integer2不提供，将其默认为19。

(foo 1 5 3 9)

⇒ 16

(foo 5)

⇒ 14

More generally,

更一般的

(foo w x y...)

==

(+ (- x w) y...)

Any argument whose name contains the name of a type (e.g., integer, integer1 or buffer) is expected to be of that type. A plural of a type (such as buffers) often means a list of objects of that type. Arguments named object may be of any type. (See Lisp Data Types, for a list of Emacs object types.) Arguments with other sorts of names (e.g., new-file) are discussed specifically in the description of the function. In some sections, features common to the arguments of several functions are described at the beginning.

任何名字中包含一种类型名称（e.g., integer, integer1 or buffer）的参数被期望为该类型。一个类型的复数（比如buffers）经常意味着这种类型的对像列表。名字为object的参数可能是任何类型（See Chapter 2 [Lisp Data Types], page 8, 了解Emacs对像类型列表.）。其他名字（比如new-file）的参数将会在函数的描述中谈论。在一些节中，几个函数的参数所共有的特征会一开始被描述。

See Lambda Expressions, for a more complete description of optional and rest arguments.

查阅Lambda表达式，了解optional和rest参数的完整描述。

Command, macro, and special form descriptions have the same format, but the word `Function' is replaced by `Command', `Macro', or `Special Form', respectively. Commands are simply functions that may be called interactively; macros process their arguments differently from functions (the arguments are not evaluated), but are presented the same way.

命令、宏和特殊表的描述有相同的格式，但是‘函数’分别被‘命令’，‘宏’或‘特殊表’所代替。命令是可能交互调用的简单函数。宏处理参数的方式与函数不同（参数不会被求值），但是表示方式相同。

Special form descriptions use a more complex notation to specify optional and repeated arguments because they can break the argument list down into separate arguments in more complicated ways. ‘[optional-arg]’ means that optional-arg is optional and ‘repeated-args...’ stands for zero or more arguments. Parentheses are used when several arguments are grouped into additional levels of list structure. Here is an example:

特殊表的描述方法使用了一种更加复杂的表示方法来指出可选的和可重复的参数，因为它们能够一种更复杂的方式将参数列表分解为单独的参数。‘[optional-arg]’意味着optional-arg是可选的，‘repeated-args...’代表0个或多个参数。可以用括号将几个参数按列表结构分为几个等级。这里是一个例子：

— Special Form: count-loop (var [from to [inc]]) body...

This imaginary special form implements a loop that executes the body forms and then increments the variable var on each iteration. On the first iteration, the variable has the value from; on subsequent iterations, it is incremented by one (or by inc if that is given). The loop exits before executing body if var equals to. Here is an example:

这个假想的特殊表实现了一个执行表主体的循环，每次循环能够增加变量var的值。第一循环，变量值为from；在后面的循环中，它的每次加一（若给出inc，通过inc来加一）。如果var等于to，表体停止执行，循环结束。这里是一个列子。

(count-loop (i 0 10)

(prin1 i) (princ " ")

(prin1 (aref vector i))

(terpri))

If from and to are omitted, var is bound to nil before the loop begins, and the loop exits if var is non-nil at the beginning of an iteration. Here is an example:

如果from和to被省略，var开始循环之前被绑定为nil，如果var在一个循环之前不为nil，循环存在，这里是一个例子。

(count-loop (done)

(if (pending)

(fixit)

(setq done t)))

In this special form, the arguments from and to are optional, but must both be present or both absent. If they are present, inc may optionally be specified as well. These arguments are grouped with the argument var into a list, to distinguish them from body, which includes all remaining elements of the form.

在这个特殊表中，参数from和to是可选的，但是必须同时存在或者不存在。如果它们存在，inc也是可以选择性的指出或不支持。这些参数和参数var分组到一个list，来将它们同主体区别开来，主体包含了表的其他元素。

* + - 1. A Sample Variable Description

一个简单的变量描述

A variable is a name that can hold a value. Although nearly all variables can be set by the user, certain variables exist specifically so that users can change them; these are called user options. Ordinary variables and user options are described using a format like that for functions except that there are no arguments.

变量是一个可被赋值的名字。尽管几乎所有的变量可以被用户设置，存在一些用户可以改变的特殊变量，即所谓用户选项，描述普通的变量和用户选项的格式除了没有参数以外，与描述函数的类似。

Here is a description of the imaginary electric-future-map variable.

这里有一个假想的electric-future-map 变量的描述。

— Variable: electric-future-map

The value of this variable is a full keymap used by Electric Command Future mode. The functions in this map allow you to edit commands you have not yet thought about executing.

这个变量的值是一个被Electric Command Future mode使用的完整的键映射。这个映射的函数允许你编辑还没有被想到执行的命令。

User option descriptions have the same format, but `Variable' is replaced by `User Option'.

用户选项描述有相同的格式，只是‘变量’被替换为‘用户选项’。

1.5 Acknowledgements

致谢

This manual was written by Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman and Chris Welty, the volunteers of the GNU manual group, in an effort extending over several years. Robert J. Chassell helped to review and edit the manual, with the support of the Defense Advanced Research Projects Agency, ARPA Order 6082, arranged by Warren A. Hunt, Jr. of Computational Logic, Inc. Additional sections were written by Miles Bader, Lars Brinkhoff, Chong Yidong, Kenichi Handa, Lute Kamstra, Juri Linkov, Glenn Morris, Thien-Thi Nguyen, Dan Nicolaescu, Martin Rudalics, Kim F. Storm, Luc Teirlinck, and Eli Zaretskii.

Corrections were supplied by Drew Adams, Juanma Barranquero, Karl Berry, Jim Blandy, Bard Bloom, Stephane Boucher, David Boyes, Alan Carroll, Richard Davis, Lawrence R. Dodd, Peter Doornbosch, David A. Duff, Chris Eich, Beverly Erlebacher, David Eckelkamp, Ralf Fassel, Eirik Fuller, Stephen Gildea, Bob Glickstein, Eric Hanchrow, Jesper Harder, George Hartzell, Nathan Hess, Masayuki Ida, Dan Jacobson, Jak Kirman, Bob Knighten, Frederick M. Korz, Joe Lammens, Glenn M. Lewis, K. Richard Magill, Brian Marick, Roland McGrath, Stefan Monnier, Skip Montanaro, John Gardiner Myers, Thomas A. Peterson, Francesco Potorti, Friedrich Pukelsheim, Arnold D. Robbins, Raul Rockwell, Jason Rumney, Per Starbäck, Shinichirou Sugou, Kimmo Suominen, Edward Tharp, Bill Trost, Rickard Westman, Jean White, Eduard Wiebe, Matthew Wilding, Carl Witty, Dale Worley, Rusty Wright, and David D. Zuhn.

2 Lisp Data Types

lisp数据类型

A Lisp object is a piece of data used and manipulated by Lisp programs. For our purposes, a type or data type is a set of possible objects.

Lisp对像是一块Lisp程序使用和操纵的数据。对于我们来说，类型或数据类型是一套可能的对像。

Every object belongs to at least one type. Objects of the same type have similar structures and may usually be used in the same contexts. Types can overlap, and objects can belong to two or more types. Consequently, we can ask whether an object belongs to a particular type, but not for “the” type of an object.

每个对像属于至少一个类型。同类的对像有相似的结构，通常可能在相同的上下文中使用。类型可以重叠，对像可以属于两个或者多个类型。因此，我们可以查询一个对像是否属于一个特定类型的，但不可以问一个类型是否属于一个对像。

A few fundamental object types are built into Emacs. These, from which all other types are constructed, are called primitive types. Each object belongs to one and only one primitive type. These types include integer, float, cons, symbol, string, vector, hash-table, subr, and byte-code function, plus several special types, such as buffer, that are related to editing.  (See Editing Types.)

Emacs内置了一些基本的对像类型。它们被称为原始数据类型，它们构成了其他数据类型。每个对像属于且只属于一个基本数据类型。这些类型包括integer, float, cons, symbol, string, vector, hash-table, subr, and byte-code function，加上集中特殊的类型，比如和编辑相关的buffer。

Each primitive type has a corresponding Lisp function that checks whether an object is a member of that type.

每一个基本类型都有一个相对应lisp函数来检查是否一个对像是这种类型的一个实例。

Lisp is unlike many other languages in that its objects are self-typing: the primitive type of each object is implicit in the object itself. For example, if an object is a vector, nothing can treat it as a number; Lisp knows it is a vector, not a number.

lisp不同于其他语言的地方时他的对像是自打印的：每个对像的基本类型隐藏于其对像本身。例如，如果一个对像是一个vector，不能将他当做一个number对待，lisp知道他是一个vector，而不是一个number。

In most languages, the programmer must declare the data type of each variable, and the type is known by the compiler but not represented in the data. Such type declarations do not exist in Emacs Lisp. A Lisp variable can have any type of value, and it remembers whatever value you store in it, type and all. (Actually, a small number of Emacs Lisp variables can only take on values of a certain type.  See  Variables with Restricted Values.)

大多数的语言，程序员必须声明每个变量的数据类型，类型不由数据代表，但是为编译器所知。Emacs中不存在这样的数据声明。一个lisp变量可以拥有任何数值类型，它会记录你所存储的值，类型及所有的一切。（实际上，一小部分的Emacs Lisp变量只能为某种类型的值。 See [Variables with Restricted Values](http://www.gnu.org/software/emacs/manual/html_node/elisp/Variables-with-Restricted-Values.html#Variables-with-Restricted-Values).）

This chapter describes the purpose, printed representation, and read syntax of each of the standard types in GNU Emacs Lisp. Details on how to use these types can be found in later chapters.

本章介绍目的，打印形式，并读取每个GNU Emacs Lisp的标准类型的语法。在后面的章节中可以找到关于如何使用这些类型的详细信息。

2.1 Printed Representation and Read Syntax

打印形式和读取语法

The printed representation of an object is the format of the output generated by the Lisp printer (the function prin1) for that object. Every data type has a unique printed representation. The read syntax of an object is the format of the input accepted by the Lisp reader (the function read) for that object. This is not necessarily unique; many kinds of object have more than one syntax. See Read and Print.

一个对像的打印形式是lisp打印机为其产生的输出格式。每种数据类型的打印形式是唯一的。一个对像的读取语法是lisp读取器所接受的输入形式。这不一定是唯一的；很多对像不止一个语法。

In most cases, an object's printed representation is also a read syntax for the object. However, some types have no read syntax, since it does not make sense to enter objects of these types as constants in a Lisp program. These objects are printed in hash notation, which consists of the characters ‘#<’, a descriptive string (typically the type name followed by the name of the object), and a closing ‘>’. For example:

在大多数情况下，一个对像的打印形式也是它的读取语法。然而，一些类型没有读取语法，因为在lisp程序中将这些类型的对像作为常量输入是没有意义的。这些对像以哈希形式打印，哈希形式包含字符‘#<’，一个描述字符串（通常类型名字后面跟对像名字）和一个结束的‘>’。例如

(current-buffer)

⇒ #<buffer objects.texi>

Hash notation cannot be read at all, so the Lisp reader signals the error invalid-read-syntax whenever it encounters ‘#<’. In other languages, an expression is text; it has no other form. In Lisp, an expression is primarily a Lisp object and only secondarily the text that is the object's read syntax. Often there is no need to emphasize this distinction, but you must keep it in the back of your mind, or you will occasionally be very confused.

根本不会读取哈希形式，所以lisp读取器不管任何时候遇到‘#《’都会发出invalid-read-syntax的错误。在其他语言中，表达式是文本；它没有其他形式。在lisp中，一个表达式首先是一个lisp对像，其次又有它是对像的读取是形式时才是文本。通常没有必要去强调这种区别，但你必须把它记在脑海中，否则你有时会非常困惑。

When you evaluate an expression interactively, the Lisp interpreter first reads the textual representation of it, producing a Lisp object, and then evaluates that object (see Evaluation). However, evaluation and reading are separate activities. Reading returns the Lisp object represented by the text that is read; the object may or may not be evaluated later. See Input Functions, for a description of read, the basic function for reading objects。

当交互式的计算一个表达式时，lisp解释程序首先读入它的文本形式，产生一个lisp对像，然后对这个对像求值（see Evaluation）。然而，求值和读入是单独活动。读返回读取的文本所代表的Lisp对像;对像可能会与不会稍后背后被求值。See Input Functions，理解读入对像基本功能的描述。

2.2 Comments

注释

A comment is text that is written in a program only for the sake of humans that read the program, and that has no effect on the meaning of the program. In Lisp, a semicolon (‘;’) starts a comment if it is not within a string or character constant. The comment continues to the end of line. The Lisp reader discards comments; they do not become part of the Lisp objects which represent the program within the Lisp system.

注释是程序中为了人类读取该程序所写的文本，它并不会影响程序的意义。lisp中，一个不在字符串或者常量字符中的分号（‘;’）开始一段注释。注释一直到行的结束。lisp读取器忽略注释；它们不会成为lisp中代表程序的lisp对像的部分

The ‘#@count’ construct, which skips the next count characters, is useful for program-generated comments containing binary data. The Emacs Lisp byte compiler uses this in its output files (see Byte Compilation). It isn't meant for source files, however.

‘#@count’结构，跳过接下来的count个字符，对于程序生成包含二进制数据的注释是有用的。Emacs Lisp字节编译器在它的输入文件(see Byte Compilation)中使用这种结构。然而，它对源文件可用。

See Comment Tips, for conventions for formatting comments.

See Comment Tips,了解格式化注释的约定。

2.3 Programming Types

编程类型

There are two general categories of types in Emacs Lisp: those having to do with Lisp programming, and those having to do with editing. The former exist in many Lisp implementations, in one form or another. The latter are unique to Emacs Lisp.

Emacs Lisp中有两种类型的分类：和lisp编程有关的以及和编辑有关的。前者以这种或那种形式存在于很多lisp工具中。后者特指Emacs Lisp。

2.3.1 Integer Type

整数类型

The range of values for integers in Emacs Lisp is −536870912 to 536870911 (30 bits; i.e., -2\*\*29 to 2\*\*29 - 1) on most machines. (Some machines may provide a wider range.) It is important to note that the Emacs Lisp arithmetic functions do not check for overflow. Thus (1+ 536870911) is −536870912 on most machines.

在大多数机器上Emacs Lisp的整数的值的范围是-536870912到536870911（30位，也就是说，-2 \*\* 29 到2 \*\* 29 - 1）（有些机器可能提供了更为广阔的范围。）。要注意，重要的是Emacs Lisp的算术函数不检查溢出。因此在大多数机器上（1 + 536870911）是-536870912。

The read syntax for integers is a sequence of (base ten) digits with an optional sign at the beginning and an optional period at the end. The printed representation produced by the Lisp interpreter never has a leading ‘+’ or a final ‘.’.

整数的读取文法是一个以可选的正负号开始和可选的句点结束的十进制数字序列。lisp解释器成圣的打印格式绝没有开始的‘+’和结束的‘.’。

-1 ; The integer -1.

1 ; The integer 1.

1. ; Also the integer 1.

+1 ; Also the integer 1.

1073741825 ; Also the integer 1 on a 30-bit implementation.

As a special exception, if a sequence of digits specifies an integer too large or too small to be a valid integer object, the Lisp reader reads it as a floating-point number (see Floating Point Type). For instance, on most machines 536870912 is read as the floating-point number 536870912.0.

作为一个特殊的例外，如果一个数字序列指定一个整数过大或过小而不能称为一个有效的的整数对像，Lisp的读取器读取它作为一个浮点数（浮点型）。例如，在大多数机器上是536870912 作为浮点数536870912.0读取

See Numbers, for more information.

See Numbers了解更多信息。

2.3.2 Floating Point Type

浮点类型

Floating point numbers are the computer equivalent of scientific notation; you can think of a floating point number as a fraction together with a power of ten. The precise number of significant figures and the range of possible exponents is machine-specific; Emacs uses the C data type double to store the value, and internally this records a power of 2 rather than a power of 10.

浮点数相当于计算机的科学表示法；你可以认为浮点数是一个有十的幂次的小数部分。有效数字的确切位数和可能的指数的范围是和机器相关的;Emacs使用C数据类型double来存储该值，内部是以2的幂次方来记录，而不是10的幂次方。

The printed representation for floating point numbers requires either a decimal point (with at least one digit following), an exponent, or both. For example, ‘1500.0’, ‘15e2’, ‘15.0e2’, ‘1.5e3’, and ‘.15e4’ are five ways of writing a floating point number whose value is 1500. They are all equivalent.

浮点数的打印形式要求一个小数点（至少后跟一个数字），一个指数，或者两个都有。例如，“1500 .0”，“15e2”“15 0.0 E2”，“1 0.5 E3”，和“0.15 E4”是一个值为1500的浮点数的五种写法。他们都是等价的。

See Numbers, for more information.

See Numbers,了解更多信息。

2.3.3 Character Type

字符类型

A character in Emacs Lisp is nothing more than an integer. In other words, characters are represented by their character codes. For example, the character A is represented as the integer 65.

Emacs Lisp中的字符和整数并无不同。换而言之，字符使用它们的字符编码表示的。例如字符A用整数65来表示。

Individual characters are used occasionally in programs, but it is more common to work with strings, which are sequences composed of characters. See String Type.

程序中偶尔使用单个字符，但是更多时候和字符串一块使用。字符串是字符组成的序列。

Characters in strings and buffers are currently limited to the range of 0 to 4194303—twenty two bits (see Character Codes). Codes 0 through 127 are ASCII codes; the rest are non-ASCII (see Non-ASCII Characters). Characters that represent keyboard input have a much wider range, to encode modifier keys such as Control, Meta and Shift.

目前字符串和缓冲区中的字符的范围限制范围是0到4194303—22位(see Character Codes).。0到127的代码是ASCII码，其余的都是非ASCII码。代表键盘输入的字符有一个更广的范围，用来对修饰位，比如Control，Meta和Shift来进行编码。

There are special functions for producing a human-readable textual description of a character for the sake of messages. See Describing Characters.

对于用来传递消息的字符，有特殊的函数来来生成人类可读的文本描述。

Basic Char Syntax: Syntax for regular characters.

基本字符语法：常规字符的语法。

2.3.3.2 General Escape Syntax

一般转义语法

In addition to the specific escape sequences for special important control characters, Emacs provides several types of escape syntax that you can use to specify non-ASCII text characters.

除了特别重要的控制字符的特殊转义序列外，Emacs提供了几种类型的转义语法，您可以使用指定非ASCII文本字符。

You can specify characters by their Unicode values. ?\unnnn represents a character that maps to the Unicode code point ‘U+nnnn’ (by convention, Unicode code points are given in hexadecimal). There is a slightly different syntax for specifying characters with code points higher than U+ffff: \U00nnnnnn represents the character whose code point is ‘U+nnnnnn’. The Unicode Standard only defines code points up to ‘U+10ffff’, so if you specify a code point higher than that, Emacs signals an error.

您可以通过Unicode值指定字符。？\ unnnn表示一个映射到Unicode代码点“U + nnnn”的字符。（按照惯例，Unicode代码点以16进制给出）。为指定的代码点大于U + FFFF的字符需要稍微不同的语法：\ U00nnnnnn代表代码点是“U + NNNNNN”的字符。Unicode标准之定义了最高到“U +10 FFFF”的代码点，所以，如果你指定的代码点高于它，Emacs会发出一个错误信号。

This peculiar and inconvenient syntax was adopted for compatibility with other programming languages. Unlike some other languages, Emacs Lisp supports this syntax only in character literals and strings.

选择这种奇怪且不方便的语法是为了兼容其他编程语言。不同于其他一些语言，Emacs Lisp只有在字符和字符串支持这个语法。

The most general read syntax for a character represents the character code in either octal or hex. To use octal, write a question mark followed by a backslash and the octal character code (up to three octal digits); thus, ‘?\101’ for the character A, ‘?\001’ for the character C-a, and ?\002 for the character C-b. Although this syntax can represent any ASCII character, it is preferred only when the precise octal value is more important than the ASCII representation.

一个字符最一般的读取语法是用八进制或十六进制表示。要使用八进制，用问号“？”加反斜杠和八进制字符代码（最多三位八进制数）;因而，“? \ 101”表示字符A,“?\ 001”表示字符C-a，“?\ 002”表示字符C-b。虽然这个语法可以表示任何ASCII字符，只有在精确的八进制值比ASCII表示更重要时它才是首选，。

?\012 ⇒ 10 ?\n ⇒ 10 ?\C-j ⇒ 10

?\101 ⇒ 65 ?A ⇒ 65

To use hex, write a question mark followed by a backslash, ‘x’, and the hexadecimal character code. You can use any number of hex digits, so you can represent any character code in this way. Thus, ‘?\x41’ for the character A, ‘?\x1’ for the character C-a, and ?\x8e0 for the Latin-1 character ‘a’ with grave accent。

要使用是十六进制，用问号“？”加反斜杠、‘x’还有十六进制字符代码。您可以使用任意数量的十六进制数字，因此，可以以这种方式表示任何字符。因而，“?\x41”表示字符A,“?\x1”表示字符C-a，“?\x8e0”表示重音的Latin-1字符‘a’。

2.3.3.3 Control-Character Syntax

控制字符语法

Control characters can be represented using yet another read syntax. This consists of a question mark followed by a backslash, caret, and the corresponding non-control character, in either upper or lower case. For example, both ‘?\^I’ and ‘?\^i’ are valid read syntax for the character C-i, the character whose value is 9.

控制字符还可以用另一种读取语法来表示。这包括一个问号，后跟一个反斜杠，插入符，以及相应的非控制字符（大写或小写）。例如， ‘?\^I’ 和‘?\^i’都是值是9的字符C-i的有效的读取语法。

Instead of the ‘^’, you can use ‘C-’; thus, ‘?\C-i’ is equivalent to ‘?\^I’ and to ‘?\^i’:

可以用‘C-’替代‘^’因而，‘?\C-i’和 ‘?\^I’ 以及‘?\^i’等价。

?\^I ⇒ 9 ?\C-I ⇒ 9

In strings and buffers, the only control characters allowed are those that exist in ASCII; but for keyboard input purposes, you can turn any character into a control character with ‘C-’. The character codes for these non-ASCII control characters include the 2\*\*26 bit as well as the code for the corresponding non-control character. Ordinary terminals have no way of generating non-ASCII control characters, but you can generate them straightforwardly using X and other window systems.

在字符串和缓冲区，只有在ASCII中存在的存在的字符才能作为控制字符；但为了键盘输入的目的，你可以用‘C-’将任何字符变成一个控制字符。这些非ASCII 控制字符的字符编码共有2 \*\* 26位对应于非控制字符的编码。普通终端有没有办法产生非ASCII控制字符，但你可以直截了当地使用X和其他视窗系统生成它们。

For historical reasons, Emacs treats the <DEL> character as the control equivalent of ?:

由于历史的原因，Emacs将<DEL>等价为？的控制字符：

?\^? ⇒ 127 ?\C-? ⇒ 127

As a result, it is currently not possible to represent the character Control-?, which is a meaningful input character under X, using ‘\C-’. It is not easy to change this, as various Lisp files refer to <DEL> in this way.

因此，目前用“\ C-”代表字符Control-？是不可能的，这是一个在X下有意义的输入字符。这是不容易改变，因为很多Lisp文件以这种方式代表<DEL>。

For representing control characters to be found in files or strings, we recommend the ‘^’ syntax; for control characters in keyboard input, we prefer the ‘C-’ syntax. Which one you use does not affect the meaning of the program, but may guide the understanding of people who read it.

代表在文件或字符串的控制字符，我们建议'^'语法，键盘输入的控制字符，我们更倾向于“C-”语法。使用哪一个都不影响程序的意义，但可能会指导那些阅读它的人理解它。

A meta character is a character typed with the <META> modifier key. The integer that represents such a character has the 2\*\*27 bit set. We use high bits for this and other modifiers to make possible a wide range of basic character codes.

一个元字符是与<META>修饰键一起输入的字符。表示这样的字符的整数，有2 \*\* 27位。我们使用高位，其他修饰符用来表示尽可能广泛的基本字符代码。

In a string, the 2\*\*7 bit attached to an ASCII character indicates a meta character; thus, the meta characters that can fit in a string have codes in the range from 128 to 255, and are the meta versions of the ordinary ASCII characters. See Strings of Events, for details about <META>-handling in strings.

在一个字符串中，属于ASCII字符的2 \*\* 7位表示一个元字符，因此，一个字符串中可以和元字符匹配的代码范围从128到255，这些是普通的ASCII字符的元版本。

The read syntax for meta characters uses ‘\M-’. For example, ‘?\M-A’ stands for M-A. You can use ‘\M-’ together with octal character codes (see below), with ‘\C-’, or with any other syntax for a character. Thus, you can write M-A as ‘?\M-A’, or as ‘?\M-\101’. Likewise, you can write C-M-b as ‘?\M-\C-b’, ‘?\C-\M-b’, or ‘?\M-\002’。

元字符阅读语法使用‘M -'\’。例如，“\ M - A”代表M-A。 ‘\M-’可以和八进制字符代码、'\ C-“、或其他字符任何的语法一起使用（见下文）。M-A 可以写作 ‘?\M-A’,或者‘?\M-\101’。同样地，C-M-b 可以写为‘?\M-\C-b’, ‘?\C-\M-b’, 或 ‘?\M-\002’。

2.3.3.5 Other Character Modifier Bits

其他字符修改位

The case of a graphic character is indicated by its character code; for example, ASCII distinguishes between the characters ‘a’ and ‘A’. But ASCII has no way to represent whether a control character is upper case or lower case. Emacs uses the 2\*\*25 bit to indicate that the shift key was used in typing a control character. This distinction is possible only when you use X terminals or other special terminals; ordinary terminals do not report the distinction to the computer in any way. The Lisp syntax for the shift bit is ‘\S-’; thus, ‘?\C-\S-o’ or ‘?\C-\S-O’ represents the shifted-control-o character.

图形字符用其字符代码表示，例如，ASCII能够区分字符'A'和'a’。但是，ASCII没有办法表示表示控制字符是大写还是小写。Emacs使用第2 \*\* 25位表明Shift键用于输入控制字符。只有当使用X终端或其他特殊终端时才可能区别出来，普通终端不以任何方式向计算机报告这种区别。

The X Window System defines three other modifier bits that can be set in a character: hyper, super and alt. The syntaxes for these bits are ‘\H-’, ‘\s-’ and ‘\A-’. (Case is significant in these prefixes.) Thus, ‘?\H-\M-\A-x’ represents Alt-Hyper-Meta-x. (Note that ‘\s’ with no following ‘-’ represents the space character.) Numerically, the bit values are 2\*\*22 for alt, 2\*\*23 for super and 2\*\*24 for hyper.

X Window系统定义了字符中可设置的三个其他修饰位：hyper, super and alt。这些位的语法分别是‘\H-’, ‘\s-’ 和‘\A-’.（这些前缀中的大小写是重要的）。因而，‘?\H-\M-\A-x’表示Alt-Hyper-Meta-x。（注意‘\s’没有‘-’跟着表示空格字符）。数字上，alt的比特值是2\*\*22，super是 2\*\*23 ，hyper是2\*\*24 。

2.3.4 Symbol Type

符号类型

A symbol in GNU Emacs Lisp is an object with a name. The symbol name serves as the printed representation of the symbol. In ordinary Lisp use, with one single obarray (see Creating Symbols), a symbol's name is unique—no two symbols have the same name.

GNU Emacs Lisp中的符号是一个有名字的对像。符号名用做符号的打印形式。在普通Lisp的使用，用单一的obarray（请参阅创建符号），一个符号的名称是唯一，没有两个符号具有相同的名称。

A symbol can serve as a variable, as a function name, or to hold a property list. Or it may serve only to be distinct from all other Lisp objects, so that its presence in a data structure may be recognized reliably. In a given context, usually only one of these uses is intended. But you can use one symbol in all of these ways, independently.

符号可以作为变量，作为函数名称，或持有一个属性列表。或者，它可能只是用来有别于所有其他的Lisp对像，这样其在数据结构的存在就能可靠地识别。在给定上下文中，通常这些用途只用其一。但你可以将一个符号独立的用于所有的这些用途。

A symbol whose name starts with a colon (‘:’) is called a keyword symbol. These symbols automatically act as constants, and are normally used only by comparing an unknown symbol with a few specific alternatives.

一个名称以冒号(':')开始的符号被称为关键字符号。这些符号自动作为常数，通常只有将一个未知的符号同一些特定的替代物作比较是才会用到他们。

A symbol name can contain any characters whatever. Most symbol names are written with letters, digits, and the punctuation characters ‘-+=\*/’. Such names require no special punctuation; the characters of the name suffice as long as the name does not look like a number. (If it does, write a ‘\’ at the beginning of the name to force interpretation as a symbol.) The characters ‘\_~!@$%^&:<>{}?’ are less often used but also require no special punctuation. Any other characters may be included in a symbol's name by escaping them with a backslash. In contrast to its use in strings, however, a backslash in the name of a symbol simply quotes the single character that follows the backslash. For example, in a string, ‘\t’ represents a tab character; in the name of a symbol, however, ‘\t’ merely quotes the letter ‘t’. To have a symbol with a tab character in its name, you must actually use a tab (preceded with a backslash). But it's rare to do such a thing.

符号名字能够包含任何字符。大多数符号名称用字母，数字和标点字符'-+=\*/'写成。只要一个字符的名字足够看起来不像一个数字，它就不需要标点符号。（如果像，就在名字的前面写一个‘\’来强制将它解释为一个字符。）‘\_~!@$%^&:<>{}?’很少用，但是也没有特殊的标点符号。其他一些通过反斜杠来转义的字符可能被包含在名字中。然而，相比于反斜杠在字符串中的使用，符号名字中的反斜杠只是简单引用紧跟其后的单个字符。例如，字符串中，‘\t’ 代表制表符；然而，在符号名中，“\ t”只是引用字母“t”。要在符号名称中有一个制表符，你必须实际使用制表符（前面加一个反斜杠）。但也很少这么做。

Common Lisp note: In Common Lisp, lower case letters are always “folded” to upper case, unless they are explicitly escaped. In Emacs Lisp, upper case and lower case letters are distinct.

公用列表处理语言注意：在公用列表处理语言中，除非明确说明，小写字母总是合并为大写。在Emacs Lisp，大写和小写字母是不同的。

Here are several examples of symbol names. Note that the ‘+’ in the fifth example is escaped to prevent it from being read as a number. This is not necessary in the fourth example because the rest of the name makes it invalid as a number.

这里有几个符号名称的例子。请注意，在第五个例子“+”号被转义，以防止它被作为一个数字读取。在第四个例子中就不必这样做，因为名字剩余的部分使它不可能成为一个数字。

foo ; A symbol named ‘foo’.

FOO ; A symbol named ‘FOO’, different from ‘foo’.

1+ ; A symbol named ‘1+’

; (not ‘+1’, which is an integer).

\+1 ; A symbol named ‘+1’

; (not a very readable name).

\(\*\ 1\ 2\) ; A symbol named ‘(\* 1 2)’ (a worse name).

+-\*/\_~!@$%^&=:<>{} ; A symbol named ‘+-\*/\_~!@$%^&=:<>{}’.

; These characters need not be escaped.

Normally the Lisp reader interns all symbols (see Creating Symbols). To prevent interning, you can write ‘#:’ before the name of the symbol.

正常情况下lisp读取器intern所有的符号。在符号名字之前加上‘#’可以阻止intern。

2.3.5 Sequence Types

序列类型

A sequence is a Lisp object that represents an ordered set of elements. There are two kinds of sequence in Emacs Lisp, lists and arrays. Thus, an object of type list or of type array is also considered a sequence.

序列是一个代表有序元素集的对像。在Emacs Lisp中，有两类序列，即表和数组。因此，表对像或者数组对像都可以认为是序列。

Arrays are further subdivided into strings, vectors, char-tables and bool-vectors. Vectors can hold elements of any type, but string elements must be characters, and bool-vector elements must be t or nil. Char-tables are like vectors except that they are indexed by any valid character code. The characters in a string can have text properties like characters in a buffer (see Text Properties), but vectors do not support text properties, even when their elements happen to be characters.

数组可以继续被细分为字符串、容器、字符表以及布尔容器。容器可以存储任何类型的元素，但字符串必须存储字符，而布尔容器的元素则只能是t 或nil。字符表除了它被任意有效字符索引的特性外，很像容器。字符串里的字符们可以像缓冲区里的字符一样，具有文本属性。（请参考see Text Properties）， 但容器即使其元素碰巧是字符，也并不支持文本属性。

Lists, strings and the other array types are different, but they have important similarities. For example, all have a length l, and all have elements which can be indexed from zero to l minus one. Several functions, called sequence functions, accept any kind of sequence. For example, the function elt can be used to extract an element of a sequence, given its index. See Sequences Arrays Vectors.

表、字符串以及其他的数组类型是不同的，但它们却具有一些重要的相似性。例如，都有一个长度l，并且其元素可以以0到l减一的的数值去索引。某些被称为 序列函数，可以接受任何序列。例如，函数elt可以通过指定一个索引获取序列中的一个元素。（请参考See Sequences Arrays Vectors.）

It is generally impossible to read the same sequence twice, since sequences are always created anew upon reading. If you read the read syntax for a sequence twice, you get two sequences with equal contents. There is one exception: the empty list () always stands for the same object, nil.

因为序列总是在读取时被重新创建，因此通常不可能将同一个序列读取两次，如果你对序列的读取语法读取了两次，你将得到两个具有相同内容的序列（译注：这句话强调的应当是 将产生两个相同内容但不同的序列对像），但有一个例外，空表()总是代表相同的对像，即nil。

2.3.6 Cons Cell and List Types

Cons Cell和表类型

A cons cell is an object that consists of two slots, called the car slot and the cdr slot. Each slot can hold or refer to any Lisp object. We also say that “the car of this cons cell is” whatever object its car slot currently holds, and likewise for the cdr.

cons cell是一个具有两个槽的对像，分别称为car槽和cdr槽。每个槽都可以持有或指向任意Lisp对像。我们也用 “这个cons cell的car”来指代该对像car槽持有的任意对像，对于cdr槽持有的对像相似，可做相似称呼。

A note to C programmers: in Lisp, we do not distinguish between “holding” a value and “pointing to” the value, because pointers in Lisp are implicit.

C程序员注意：在Lisp中，我们并不区分“持有”一个值和“引用”该值，因为在Lisp中，指针是隐式的。

A list is a series of cons cells, linked together so that the cdr slot of each cons cell holds either the next cons cell or the empty list. The empty list is actually the symbol nil. See Lists, for functions that work on lists. Because most cons cells are used as part of lists, the phrase list structure has come to refer to any structure made out of cons cells.

表是一系列的cons cell，依次通过cdr链接在一起，每个cons cell的cdr槽或引用下一个cons cell，或引用空表。空表实际上是符号nil（请参考See Lists.了解对表操作的函数）。大部分cons cell都作为表的一部分，因此短语--表结构---被用来指代任何产生cons cell的结构

Because cons cells are so central to Lisp, we also have a word for “an object which is not a cons cell.” These objects are called atoms.

由于cons cell们对于表如此重要，我们也有这样一个说法，即“不是cons cell的对像”，这些对像被称为原子。

The read syntax and printed representation for lists are identical, and consist of a left parenthesis, an arbitrary number of elements, and a right parenthesis. Here are examples of lists:

表的读取语法和打印表示是相等的，并且包含一个左括号，任意数量的元素以及一个右括号。以下是表的例子：

(A 2 "A") ; A list of three elements.

() ; A list of no elements (the empty list).

nil ; A list of no elements (the empty list).

("A ()") ; A list of one element: the string "A ()".

(A ()) ; A list of two elements: A and the empty list.

(A nil) ; Equivalent to the previous.

((A B C)) ; A list of one element

; (which is a list of three elements).

(A 2 "A") ; 具有三个元素的表。

() ; 没有元素的表（空表）。

nil ; 没有元素的表（空表）。

("A ()") ; 具有一个元素的表，元素是："A ()"。

(A ()) ; 具有两个元系的空表，元素是：A和空表。

(A nil) ; 同上。

((A B C)) ; 具有一个元素的表，表的元素是 一个具有三个元素的表。

Upon reading, each object inside the parentheses becomes an element of the list. That is, a cons cell is made for each element. The car slot of the cons cell holds the element, and its cdr slot refers to the next cons cell of the list, which holds the next element in the list. The cdr slot of the last cons cell is set to hold nil.

在读取时，括号内的每个对像都变成表的一个元素。也就是说，cons cell由各个元素组成。car槽持有一个元素，其cdr引用 表中的下一个cons cell，该cons cell持有表的下一个元素。表中最后个cons cell的cdr被设置为nil。

The names car and cdr derive from the history of Lisp. The original Lisp implementation ran on an IBM 704 computer which divided words into two parts, called the “address” part and the “decrement”; car was an instruction to extract the contents of the address part of a register, and cdr an instruction to extract the contents of the decrement. By contrast, “cons cells” are named for the function cons that creates them, which in turn was named for its purpose, the construction of cells.

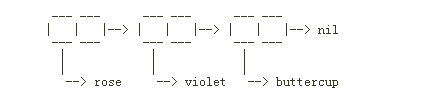
car和cdr从Lisp的历史演变而来。原始的Lisp实现运行在一个IBM 704计算机上，该计算机被分为两部分，分别叫作“地址”部分和 “剩余”部分；car是用于取得寄存器地址部分对应内容的指令，而cdr则是用于取得寄存器剩余部分对应内容的指令。相比之下，“cons cells”用于命名创建它的函数cons，即如其名所识，construction of cells。（译注：这句不翻译，否则不易理解cons cells的意义）

2.3.6.1 Drawing Lists as Box Diagrams

将表画成盒子图表

A list can be illustrated by a diagram in which the cons cells are shown as pairs of boxes, like dominoes. (The Lisp reader cannot read such an illustration; unlike the textual notation, which can be understood by both humans and computers, the box illustrations can be understood only by humans.) This picture represents the three-element list (rose violet buttercup):

可以将cons cells画成盒子对，像多米诺一样的形式对表进行图解。（Lisp读取器不能读取这种描绘方式，不像文本记法，可以被人和计算机读取，盒式描绘方式只能被人读取）此图表示了一个具有三元素的表(rose violet buttercup)：



In this diagram, each box represents a slot that can hold or refer to any Lisp object. Each pair of boxes represents a cons cell. Each arrow represents a reference to a Lisp object, either an atom or another cons cell.

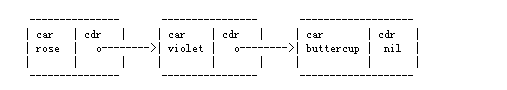
在此图解中，每个盒子代表一个可以持有或者引用任意Lisp对象的槽。每对盒子对代表一个cons cell，每个箭头代表对一个Lisp对象的引用，该对象可以是一个原子， 也可以是另一个cons cell。

In this example, the first box, which holds the car of the first cons cell, refers to or “holds” rose (a symbol). The second box, holding the cdr of the first cons cell, refers to the next pair of boxes, the second cons cell. The car of the second cons cell is violet, and its cdr is the third cons cell. The cdr of the third (and last) cons cell is nil.

在这个例子中，第一个盒子，持有着第一个cons cell的car部分，引用rose（一个符号）。第二个盒子,持有第一个cons cell的cdr，引用下 一个cons cell。第二个cons cell的car是violet，它的cdr是第三个cons cell。第三个即最后一个cons cell的cdr是nil。

Here is another diagram of the same list, (rose violet buttercup), sketched in a different manner:

下面是该表的另一个图解，(rose violet buttercup)，用另外一个方式进行描绘。

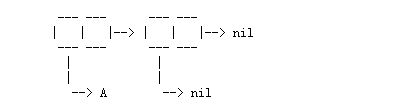


A list with no elements in it is the empty list; it is identical to the symbol nil. In other words, nil is both a symbol and a list.

没有元素的表称为空表；它等同于符号nil。换句话说，nil既是一个符号，又是一个表。

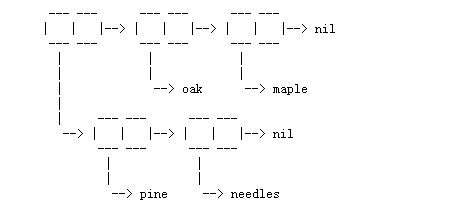
Here is the list (A ()), or equivalently (A nil), depicted with boxes and arrows:

下面是表(A ())，或者等价的(A nil)，由盒子和箭头描绘：



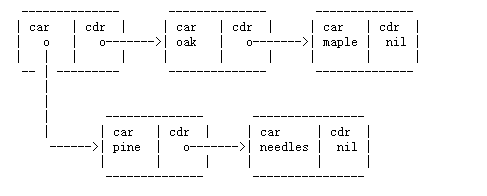
Here is a more complex illustration, showing the three-element list, ((pine needles) oak maple), the first element of which is a two-element list:

下面是一个更为复杂的刻画，显示了一个三元素表((pine needles) oak maple)，第一个元素是一个具有两个元素的表：



The same list represented in the second box notation looks like this:

上述表用第二种盒子记法如下：



2.3.6.2 Dotted Pair Notation

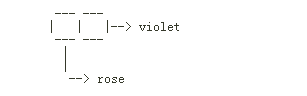
点对记法

Dotted pair notation is a general syntax for cons cells that represents the car and cdr explicitly. In this syntax, (a . b) stands for a cons cell whose car is the object a and whose cdr is the object b. Dotted pair notation is more general than list syntax because the cdr does not have to be a list. However, it is more cumbersome in cases where list syntax would work. In dotted pair notation, the list ‘(1 2 3)’ is written as ‘(1 . (2 . (3 . nil)))’. For nil-terminated lists, you can use either notation, but list notation is usually clearer and more convenient. When printing a list, the dotted pair notation is only used if the cdr of a cons cell is not a list.

点对记法是一种准确表示car和cdr的常见语法。使用这种语法时，cons cell (a . b)表示它的car是对象a，而cdr 是对象b。由于cdr不一定需要是表，因此点对记法比表语法更为普遍。然而，在表语法可用时，如果使用点对记法，则显得较为笨拙。 在点对记法中，‘(1 2 3)’写作‘(1 . (2 . (3 . nil)))’。对于由nil终结的表，两种语法都可以使用，但表语法更为方便。当打印一个表时，点对记法仅在cons cell的cdr不是一个表时使用。

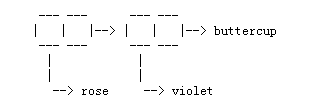
下面是用盒子演示点对记法的一个例子。此例子显示点对(rose . violet)：

Here's an example using boxes to illustrate dotted pair notation. This example shows the pair (rose . violet):



You can combine dotted pair notation with list notation to represent conveniently a chain of cons cells with a non-nil final cdr. You write a dot after the last element of the list, followed by the cdr of the final cons cell. For example, (rose violet . buttercup) is equivalent to (rose . (violet . buttercup)). The object looks like this:

借助于非nil终结的cdr，你可以组合点对记法和表记法来方便表示一个cons cell链。你在表的最后一个元素后写下一个点，并跟跟上最后一个 cons cell的cdr。例如，(rose violet . buttercup)和(code . (violet . buttercup))相等，这个对象看起来是这样的：

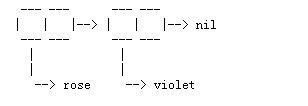


The syntax (rose . violet . buttercup) is invalid because there is nothing that it could mean. If anything, it would say to put buttercup in the cdr of a cons cell whose cdr is already used for violet.

语法(rose . violet . buttercup) 是无效的，这是因为它没有任何意义。如果非要说明其意义的话，可以说将buttercup放至 一个cons cell的cdr中，而该cons cell已经是violet的cdr了。

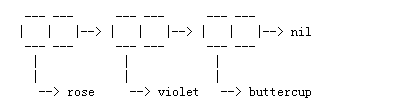
The list (rose violet) is equivalent to (rose . (violet)), and looks like this:

表(rose violet)和(rose . (violet))等价，它看起来是这个样子：



Similarly, the three-element list (rose violet buttercup) is equivalent to (rose . (violet . (buttercup))). It looks like this:

相似地，三元素表(rose violet buttercup)与(rose . (violet . buttercup)等价。 它看起来是这个样子：



2.3.6.3 Association List Type

关联列表类型

An association list or alist is a specially-constructed list whose elements are cons cells. In each element, the car is considered a key, and the cdr is considered an associated value. (In some cases, the associated value is stored in the car of the cdr.) Association lists are often used as stacks, since it is easy to add or remove associations at the front of the list.

关联表或alist是一种特殊构造的表，其元素是cons cells。在每个元素里，car被当作一个关键字，而cdr则被 当作关联值。（在某些情况下，关联值存储在cdr的car中。）关联表通常用作栈，因为在关联表的前面添加或删除关联很简单。

例如：

(setq alist-of-colors

'((rose . red) (lily . white) (buttercup . yellow)))

设置变量alist-of-colors为一个具有三元素的关联表。在第一个元素中，rose是关键字，而red是关联值。

要进步请了解关联表以及操作它的函数，请参考See Association Lists.一节。要了解另外一类用于处理大量具有关键字的的查找 表，请参考See Hash Tables.一节。

For example,

(setq alist-of-colors

'((rose . red) (lily . white) (buttercup . yellow)))

sets the variable alist-of-colors to an alist of three elements. In the first element, rose is the key and red is the value.

See Association Lists, for a further explanation of alists and for functions that work on alists. See Hash Tables, for another kind of lookup table, which is much faster for handling a large number of keys.