

Emacs 之魂（一）：开篇

作者：何幻

原文地址：<https://zhuanlan.zhihu.com/p/34106024>

程序员大部分的时间都是在和代码打交道，因此，对于文本编辑器一定不会陌生了。

编辑器是处理文本的工具。

就像趁手的兵器对武林高手的辅助作用一样，强大的编辑器也会使编码工作事半功倍，趣味十足。

可是，什么样的编辑器可以称之为强大呢？

江湖中，流传着关于两大编辑器的传说，

Emacs 是神的编辑器，而 Vim 是编辑器之神

有关最强编辑器的争论却一直没有结果，战争一触即发，

热爱 Emacs 的人们说，Emacs 是神用的，而 Vim 是人用的。

而热爱 Vim 的人们会说，Vim 毕竟是神，Emacs 只是个编辑器而已。

有些人可能会对文本编辑器不屑一顾，

因为目前有很多 IDE（集成开发环境：Integrated Development Environment）可用，

IDE 大大简化了程序员们的重复劳动，对代码进行编译调试非常方便。

Neal Ford 在《卓有成效的程序员》一书中提到，

最好寻找一个完美的编辑器，而不是 IDE。虽然公司的制度或者某些编程语言通常会告诉你，在代码编写方面，使用 IDE 是非常高效的方式，但我们仍然需要优质的文本编辑器来编辑纯文本文件。

以我个人的经验来说，如果以后潜在会使用多种语言进行编程，

就应该挑选并掌握一款文本编辑器，把它带在身边。

IDE 安装起来不太方便，并且和具体语言绑定的比较严重，
当仅仅使用特定某种语言进行编程时可以使用它。

以上两款强大的编辑器 Emacs 和 Vim，学习任何一种都是可以的，仅凭个人喜好决定，

最好都试用一段时间，然后听从自己的直觉。

其实，学习如何使用它们其实并不是最重要的事情，重要的是学习的过程中可以给我们带来什么。

本系列文章我们来探讨 Emacs，

之所以谈论它，是因为 Emacs 和 Lisp 有不解之缘，或者说 Emacs 有一颗 Lisp 的心。

Lisp 是一族函数式的编程语言，有众多方言，Emacs 使用了 elisp (Emacs Lisp) 。

很多文章都提到过 Emacs 的快捷键以及配置方式，

本系列文章并不打算写这些，而是通过 Emacs 介绍 elisp 这门语言，
来让我们一起欣赏 “解释器模式” 在编辑器中的伟大实现吧。

Emacs 之魂（二）：一分钟学会人界用法

上文提到了 编辑器之战，

据江湖传说，Emacs 被称为 “神的编辑器”，

Emacs 有着无与伦比的可扩展性和可定制性，简直变成了一个 “操作系统” 。

使用 Emacs 你可以收发电子邮件，

编辑远程档案，登录远程主机，登录 IRC 和朋友聊天，

当做计算器，管理目录，进行文件比较合并，浏览网站，

甚至还可以模拟其他编辑器，玩游戏，煮咖啡，等等，

最不济的话，用它还可以写代码，编辑文本文件。

Emacs 有这么强大的功能，是因为用户可以无限制的扩展它，

我们可以用 elisp 写代码，然后让 Emacs 做任何事情，

Emacs 有多强大，完全取决于它的使用者有多强大，
像这样使用 Emacs，我们称之为“神界用法”。

然而，在没有学会 elisp 之前，用它默认提供的功能就够了，我们可以称之为“人界用法”。

本文我们会发现，用一分钟学会人界用法，并不困难。

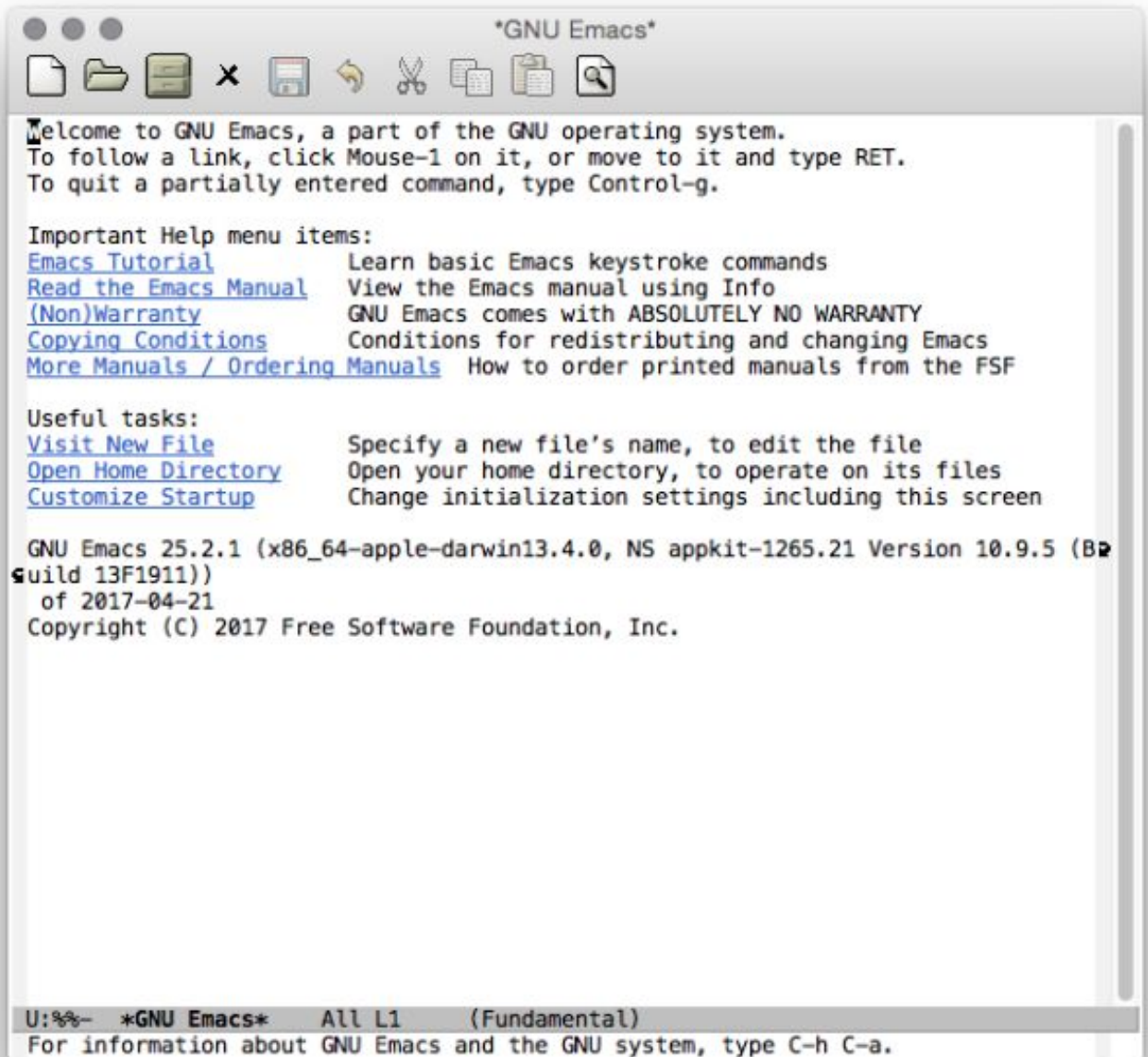
下面我们先下载安装 Emacs 吧，我想，这不应该记入在那一分钟之内。

Mac 用户可以到这里下载安装，[Emacs For Mac OS X](#)

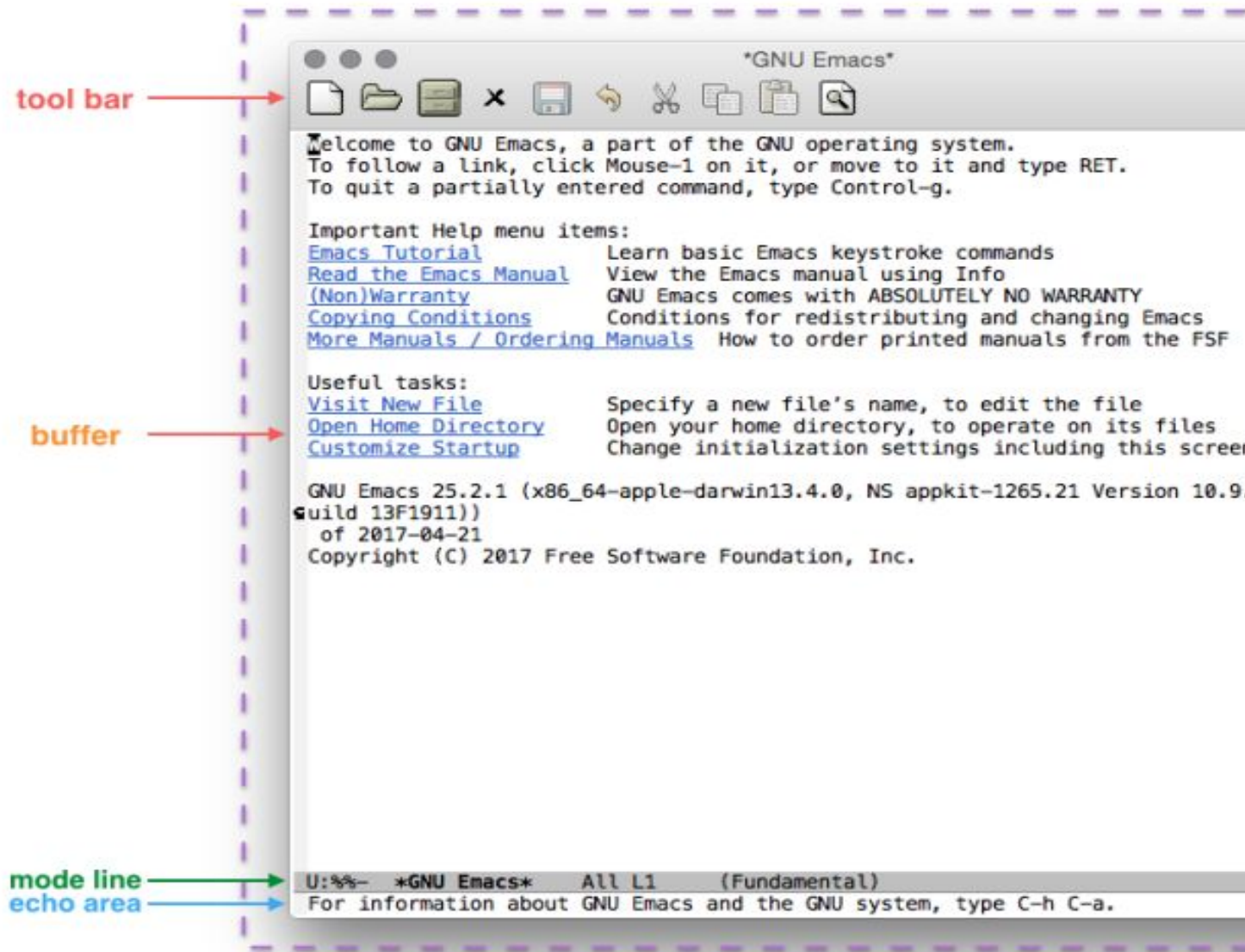
Windows 用户可以到这里下载，[Emacs For Windows](#)

也可以通过命令行来安装，[GNU Emacs: Download & installation](#)

安装完打开后，终于可以看到它的庐山真面目了，



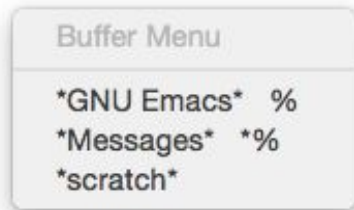
一分钟学会



如图所示，一个 Emacs 编辑器可以包含多个 frame，默认只打开一个，一个 frame 包含了 tool bar，buffer，mode line 和 each area，其中，buffer+mode line 合称一个 window，一个 frame 可以包含多个 window。

buffer 类似于其他编辑器中的标签，我们可以在一个 window 里面编辑不同的 buffer，

按住 **Ctrl** 键，在 **buffer** 中点击鼠标左键，就可以看到默认情况下 Emacs 加载了哪些 **buffer**，
我们可以选择一个 **buffer** 切换过去，然后使用同样的办法再切换回来。



tool bar 中包含了一些常用的功能，
打开文件，新建文件，打开文件夹，关掉当前 buffer，保存，撤销，剪切，复制，
粘贴，查找。

结合 tool bar，以及切换 buffer，我们已经在一分钟之内学会使用 Emacs 了。

快捷键

很多初学 Emacs 的同学被它的快捷键吓倒了，因为有些 Emacs 教程不喜欢人们使用鼠标，

其实有些场景，用鼠标可能会更快或者更直观，何乐而不为呢？

Emacs 有多强大，完全取决于它的使用者有多强大，因此**不必让快捷键束缚自己**，
例如，切换 buffer，点击 buffer 中的某个链接，等等。

然而，学会使用快捷键在大多数情况下都是高效的，
保存文件，上下左右移动光标，跳转到行首或者行尾，删除光标右边的一个字符，
这些都是平时用的最多的命令操作，总是通过点击 tool bar 或者把手移动到方向
键上是很不方便的。

Emacs 的快捷键非常之多，还可以自定义快捷键，
大部分 Emacs 教程，喜欢这样**罗列**它们。

C-f 后一个字符
C-b 前一个字符

C-p 上一行
C-n 下一行
M-f 后一个单词
M-b 前一个单词
C-a 行首
C-e 行尾
C-v 向下翻一页
M-v 向上翻一页
M-< 到文件开头
M-> 到文件末尾

C-x C-f "find"文件, 即在缓冲区打开/新建一个文件
C-x C-s 保存文件
C-x C-w 使用其他文件名另存为文件
C-x C-v 关闭当前缓冲区文件并打开新文件
C-x i 在当前光标处插入文件
C-x b 新建/切换缓冲区
C-x C-b 显示缓冲区列表
C-x k 关闭当前缓冲区
C-z 挂起 emacs
C-x C-c 关闭 emacs

其中, C 表示 Ctrl 键, M 表示 Alt 键, C-f 表示按住 Ctrl 然后按 f,
C-x C-s 表示先按 Ctrl+x, 再按 Ctrl+s,
C-x b 表示先按 Ctrl+x, 再按 b。

如何记快捷键

如果你和我一样, 仅仅看到上面那些快捷键就已经懵逼了, 不妨按下面的方法试一试,

- (1) 只记那些自己经常用到的快捷键, 练熟变成手指的条件反射
- (2) 把快捷键的功能写在前面, 键位写在后面, 按逻辑功能分类
- (3) 不知道用什么快捷键时, 去网上搜一下

有了这些经验之后，就不用记那么多快捷键了，
相信我，不用快捷键也慢不了多少，因为编程的瓶颈不在打字速度上。

此外，把功能写在前面有助于按逻辑功能分类，查找时也更方便，
有些快捷键基本上不会被用到，那还是**忘了它**吧，省得占用大脑内存。
以下是我总结的常用快捷键，

剪切: C-w

复制: M-w

粘贴: C-y

上一行: C-p

下一行: C-n

左移一个字符: C-b

右移一个字符: C-f

行首: C-a

行尾: C-e

文章开头: M-S-,

文章结尾: M-S-.

向上搜索: C-r

向下搜索: C-s

替换: M-S-5

撤销: C-x u

保存: C-x s

保存，不提示: C-x C-s

全选: C-x h

退出: C-x C-c

取消命令: C-g

切分成两个窗口: C-x 2

关闭其他窗口: C-x 1

切换窗口: C-x o

剪切光标到行尾: C-k
删除下一个字符: C-d
显示所有 buffer: C-mouse
关闭当前 buffer: C-x k

总结

本文介绍了 Emacs 的安装和“人界用法”，最后分享了一个记快捷键的小窍门。我们也实在没有必要因为在幼儿园中记不住座位的摆放位置而放弃学业。

Emacs 有多强大，完全取决于它的使用者有多强大，因此，努力锻炼自己才是 Emacs 用户的精髓，

从下文开始，我们开始学习 elisp，逐步探讨这门语言作为可编程的编程语言（programmable programming language）的真谛。

Emacs 之魂（三）：列表，引用和求值策略

回顾

上文我们介绍了 Emacs 的用法，发现一分钟学会使用它并不是难事，而且，我们没有让快捷键束缚住，因为 Emacs 的精髓在于 Emacs Lisp 中。

本文我们开始探讨 Emacs Lisp，不过在这之前我们还要先熟悉一下 Lisp 的特点和 Lisp 家族的成员，

随后本文重点分析和介绍了列表，引用和求值策略，

这几个概念，尤其是引用，对学习者来说非常容易引起困惑，

本文采用了不同的角度来描述这些概念。

1. 强大的 Lisp



1960 年，John McCarthy 发表了一篇计算机领域的文章，这是一篇“惊世之作”，

它的作用简直就像欧几里得《几何原本》对几何学的贡献一样。

John McCarthy 只用了一些简单的运算符和函数，构建出了一门图灵完备的编程语言，

称之为 Lisp，Lisp 是列表处理（List Processing）的简称。

这门语言的关键思想是，不论代码还是数据，都用统一的数据结构（列表）进行表示。

Lisp 语言具有很强的表达能力，我们可以用更少的代码做更多的事情。

通常而言，语言具有表达能力就必须提供丰富的内置功能和强大的扩展性。

语言的内置功能指的是语言默认提供的功能，它能减少程序员的重复劳动，帮助他们快速完成工作。

语言的扩展性，指的是当语言内置功能不能满足需要的时候，程序员可以怎样做。同时具有丰富的功能和强大的扩展性是很困难的，这需要在语言的设计阶段就考

虑好，

语言的内置功能越多，就会越复杂，扩展功能的与内置功能的一致性就很难被保证。

现代的高级编程语言，离不开编译器和解释器，

编译器将高级语言的代码转换成更底层的语言，例如 C 语言或者汇编，

解释器提供了一个运行时环境，直接解释执行高级语言的源代码。

一般而言，编译器是由语言的开发商提供的，使用者并不会参与到编译器的开发工作之中，

如果想要在语言中支持一等函数（first-class function），就必须让语言的开发商改写编译器，

如果需要增加新的类似 if-else 的控制结构，或者让语言支持面向对象编程，也要改写编译器才行。

因此，语言支持什么功能，以及源代码被如何编译，完全取决于开发商。

而 Lisp 语言则不同，它允许程序员对编译器进行编程，（元编程

Lisp 程序员可以决定代码被如何编译甚至如何被读取，像是半个编译器的开发者一样。

2. Lisp-1 和 Lisp-2

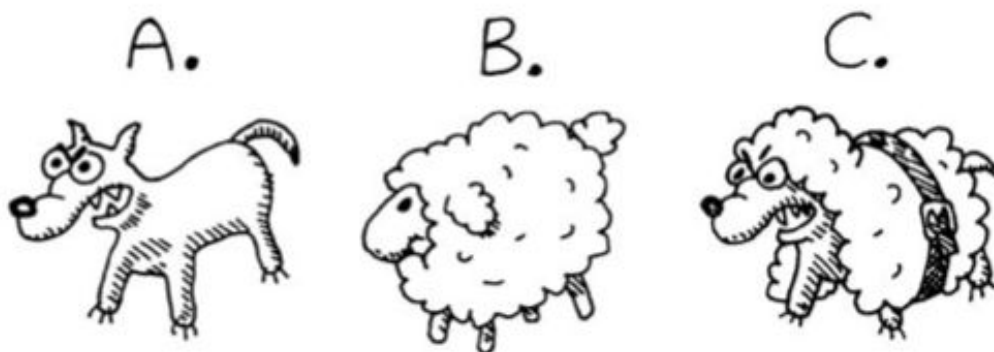
Lisp 语言构成了一个家族，具有成百上千种方言，

用的最多的几种是，Common Lisp，Scheme，Racket，和 Emacs Lisp。

其中 Scheme 的目标是简洁，Common Lisp 提供了强大的工业级支持，

Racket 提供了一种创造语言和设计实践的平台，Emacs Lisp 主要用于 Emacs 中。

• PERSONALITY TEST •
WHICH DO YOU MOST IDENTIFY WITH?



Emacs Lisp 更像 Common Lisp，它们都是 Lisp-2，
即同一个符号在不同的上下文中，可以分别用来表示变量和函数，
而 Scheme 和 Racket 则只能用来表示同一个实体，称为 Lisp-1。

出现 Lisp-2，主要是因为有效率方面的考虑，
在 Lisp-2 中，函数和变量分属不同的名字空间，在不同的环境中，由不同的求值器进行处理。

这样做也使语义更加复杂了，以后的文章中，我们会介绍 Emacs Lisp 中符号 (Symbol) 的概念。

3. 列表对象和它的文本表示

列表是 Lisp 语言中一种常用的数据结构，用来表示一批数据。

例如，由整数 1，2 和 3 构成的列表对象，Lisp 会将它打印为，(1 2 3)。

各个列表元素用空格分隔，用圆括号括起来。

然而，在 Lisp 代码中直接写 (1 2 3)，并不会创建一个列表对象，

因为 Lisp 程序也是用括号方式表示的，例如，(+ 1 2) 表示对整数 1 和 2 进行加法运算。

那么如何才能创建一个列表对象呢？

我们需要调用 list 函数，(list 1 2 3)，这段代码将会创建一个由整数 1，2 和 3

构成的列表对象，

这个列表对象打印为(1 2 3)。

注意，以上我们严谨的区分了 Lisp 对象和它的打印结果，

是因为对象和它的文本表示 (textual representation) 是不同的概念。

例如，在 C 语言中我们写，

```
int result = 1 + 2;
```

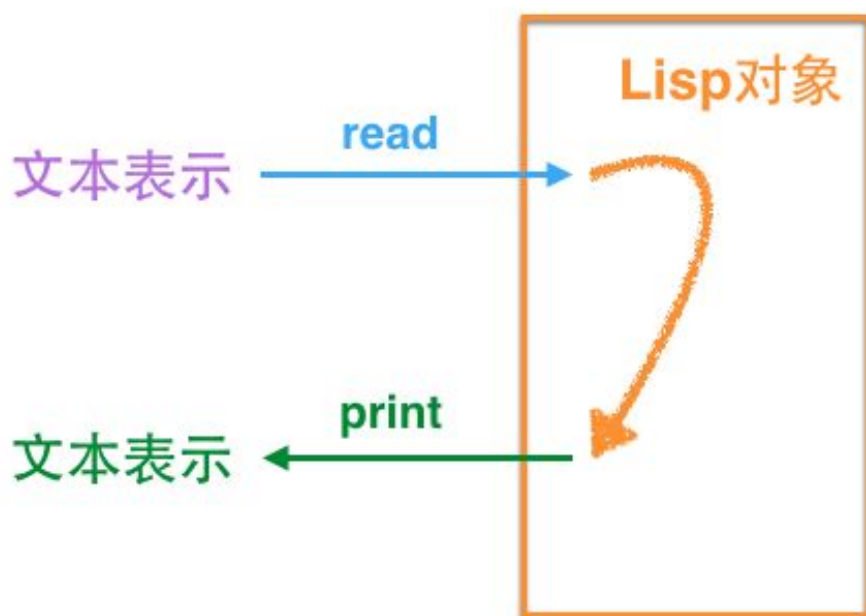
我们实际上是用“1”表示了整数 1，“1”只是一段文本，是印刷符号，而整数 1 是一个数学对象，

同样的，“+”是一段文本，它表示了加法运算符。

在 Lisp 语言中，我们用文本来写程序，而 Lisp 读取器得到的是 Lisp 对象，

经过对这些 Lisp 对象进行计算，得到了计算结果，也是一个 Lisp 对象，

最终，反馈给我们的是这个对象的文本表示。



4. 字面量和引用

在 Lisp 中，我们用文本 “1” 可以直接表示整数 1，用 “#t” 表示真值，类似的 “1” 和 “#t”，称之为对象的字面量表示（literal representation）。其它语言中，也提供了广泛的字面量表示法，例如，JavaScript 提供了数组和对象字面量，

```
const obj = {  
  x: 1,  
  y: [2, 3, 4]  
};
```

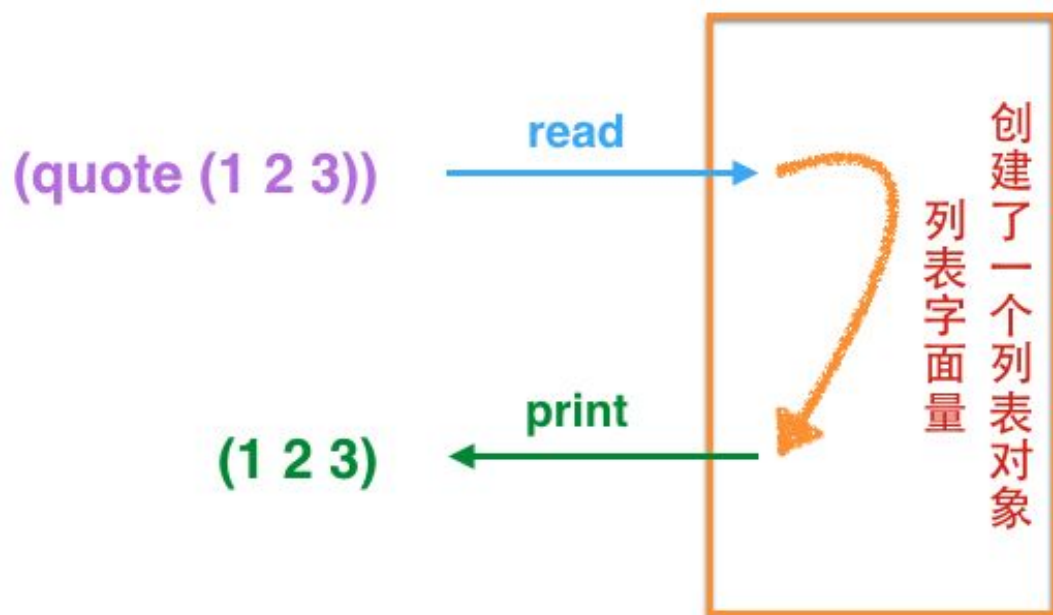
这段代码创建了一个 JavaScript Object，它的 x 属性值是 1，y 属性值是一个数组。

字面量表示法，使得我们不必调用 new Object 和 new Array 来创建它。

Lisp 中列表对象用的非常多，每次都使用 list 函数来创建是一件麻烦的事情，因此，Lisp 语言提供了列表对象的字面量写法，我们只需要调用 quote 就可以了。

```
(quote (1 2 3))
```

以上 Lisp 代码会创建一个打印形式为 (1 2 3) 的列表对象。



对于嵌套列表，使用 quote 是非常方便的，

```
(quote (1 (2 3) ((4 5) (6 7))))
```

像这样创建列表的方式，称为引用（quoting），

这不同于按引用调用（call by reference）中的“引用”（reference）。

quote 还有一个便捷的写法，就是用单引号来表示它，(quote (1 2 3)) 可以表示为，

```
'(1 2 3)
```

我们只需要在列表前加一个单引号即可，因为列表的右括号表明了它在引用这个列表。

5. quote 和 list

值得一提的是，引用并不保证每次都会重新创建列表。

例如，在 Emacs Lisp 中我们使用 defun 创建函数，

```
(defun foo ()  
  '(1 2 3))
```

然后，用以下方式进行函数调用，注意 foo 参数个数为 0 个，

```
(foo)
```

多次调用 foo，编译器可能返回同一个列表对象。

而 list 则不同，每次调用它会返回一个全新的列表对象，

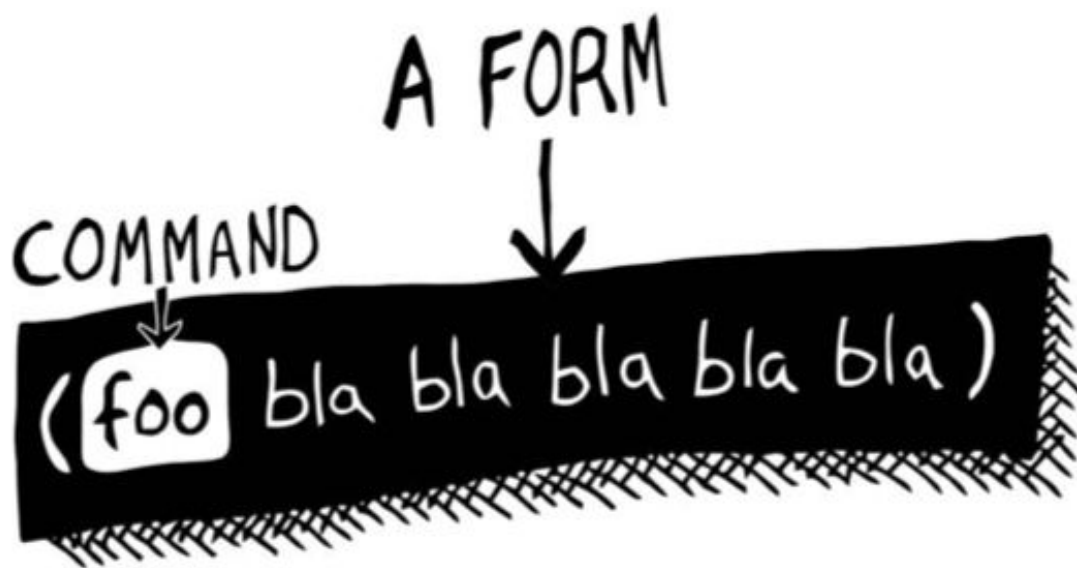
```
(defun foo ()  
  (list 1 2 3))
```

6. 求值策略

Lisp 代码是由表达式构成的，Lisp 程序的执行过程就是表达式的求值过程，

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

以上表达式的求值结果为 21。



在程序的列表表示法中，从左到右位于第一个位置的元素，是比较特殊的，它表示一个函数（function），一个宏（macro），或者一个内置的特殊命令（special form）。

位于其他位置的元素称为参数。

函数被调用的时候，它的每个参数都必须首先被求值，

例如，以上程序中`*`，`+`都是函数，

在调用乘法函数`*`时，它的参数`(+ 1 2)`和`(+ 3 4)`都首先要被求值，分别求值为 3 和 7，

然后再进行乘法运算，结果为 21。

而宏和内置的特殊命令，并不要求如此，它们有自己的对参数的求值策略。

其中“1”称之为自求值对象，对它进行求值将得到它本身，

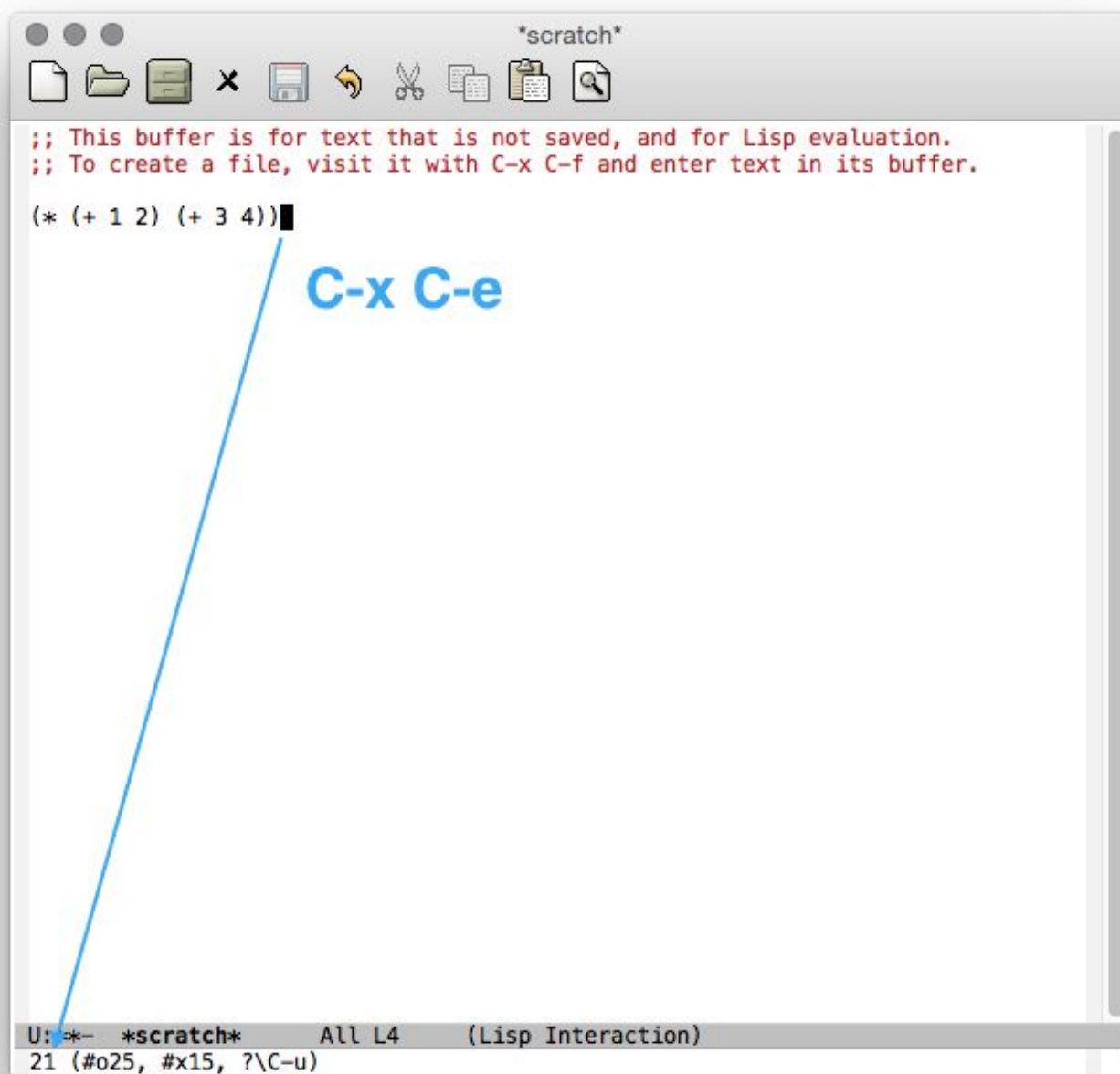
```
1  
(eval 1)  
(eval (eval 1))
```

其结果都为 1，其它的自求值对象还包括布尔值，字符串，向量，等等。

`(+ 1 2)`中 1 和 2 之前没有 quote，是因为它们是自求值对象，`(+ '1 '2)`和`(+ 1 2)`的计算结果是相等的。

7. 在 Emacs 中求值表达式

Emacs 可以直接求值文本中的 Lisp 代码，我们只需要将光标定位到列表尾部，然后按快捷键 `C-x C-e` 即可。（指的是按 `Ctrl+x`，然后再按 `Ctrl+e`



我们还可以试试 `quote` 和自求值对象，`1` 求值为 `1`，`'1` 求值为 `1`。然而 `"1` 却求值为 `(quote 1)`，是因为 `"1` 实际是 `(quote (quote 1))`，它表示用字面量方式创建了一个形如 `(quote 1)` 的列表对象。

下文，我们来讨论 Emacs Lisp 的控制结构和基本的数据类型，使用 Lisp 编程是一件有趣的事情。

参考

[The Roots of Lisp](#)

[Land of Lisp](#)

[Lisp in small pieces](#)

[An Introduction to Programming in Emacs Lisp](#)

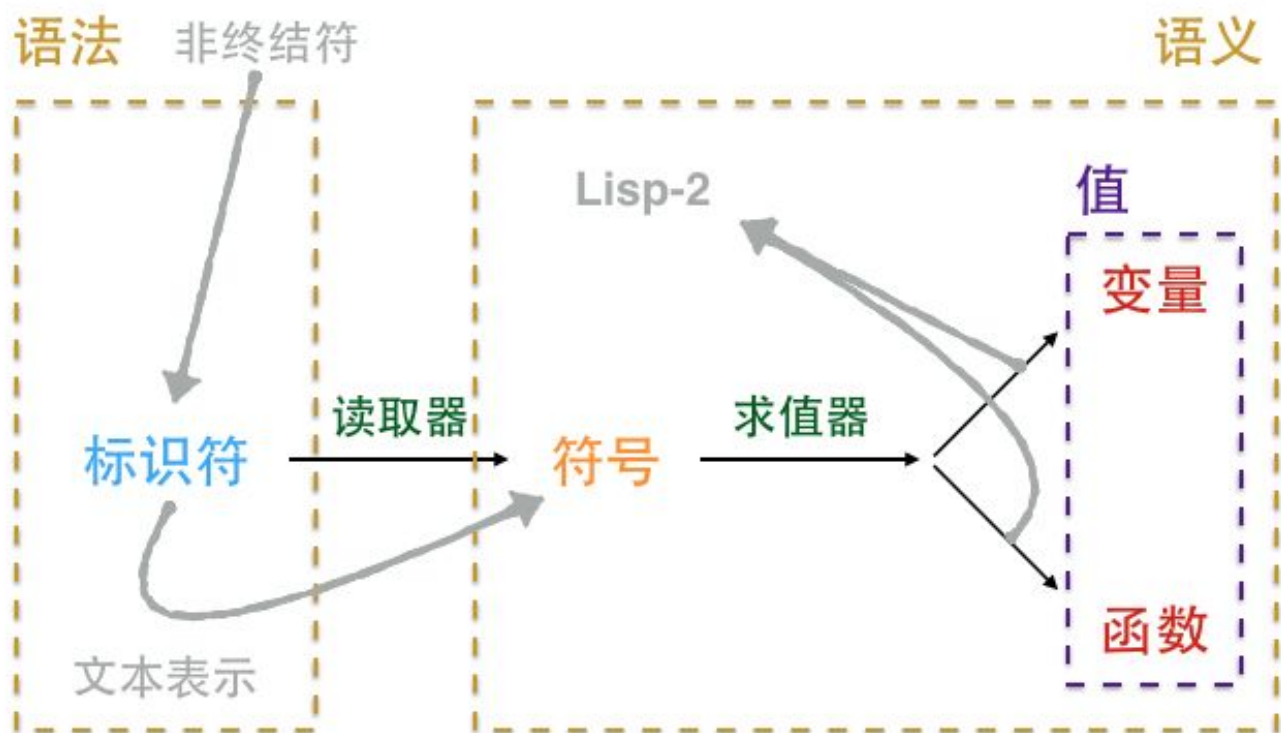
[GNU Emacs Lisp Reference Manual](#)

Emacs 之魂（四）：标识符，符号和变量

1. 符号

上文我们提到了 [Emacs Lisp](#) 是一种 [Lisp-2](#)，即同一个符号（symbol）在不同的上下文中，可以分别表示两种不同的值（value）：

变量（variable）或者函数（function），这里符号（symbol）实际上是一个 Lisp 对象，而它的文本表示（textual representation）称之为标识符（identifier）。



标识符，符号和变量，这三个概念如果不谨慎对待，就会造成混乱。其它编程语言可能没有“符号”的概念，这也是学习 Lisp 时容易困惑的原因之一。此外，这里“符号”特指 Lisp 语言的“Symbol”，不能用汉语字面意思来理解它。

标识符，是 Lisp 的上下文无关文法（context-free grammar）中的一个非终结符（nonterminal），它是一种词法结构，编译器前端（compiler front-end）在进行词法分析时会将标识符从字符流中识别出来。

符号（symbol）是一个 Lisp 对象，它是一个数据结构，由以下 4 个部分组成，

- (1) name: symbol 的名字
- (2) value cell: 作为一个**动态变量**，symbol 的值
- (3) function cell: 作为一个函数，它的函数值
- (4) property list: 属性列表

标识符直接在 **Lisp** 代码中出现，会被读取为一个符号（**symbol**），然后在不同的上下文中，**Lisp** 求值器会看情况取出 **value cell** 或者 **function cell** 的内容，作为该符号（**symbol**）的值（**value**）。

如果某一个函数接受符号（**symbol**）而不是它的值（**value**）作为参数，我们就得引用（**quote**）它，

即，我们使用引用，可以创建一个符号（**symbol**）字面量（**literal**）。

例如：**symbol-name** 函数可以用来获取符号（**symbol**）**x** 的 **name**，

```
(symbol-name 'x)  
"x"
```

结合上一篇，我们总结如下，

- (1) 直接写 `(foo bar bar)` 表示函数调用或者宏调用
- (2) 加引用 `'(foo bar bar)` 表示列表
- (3) 直接写 `x` 表示变量或者函数
- (4) 加引用 `'x` 表示符号（**symbol**）

如果只是这样的话，还很容易理解的，

可是 **value cell** 中只能保存**动态变量**，这一点理解起来就比较困难了。

“动态” 是什么意思呢？还要从变量的定义和类别说起。

2. 全局变量和局部变量

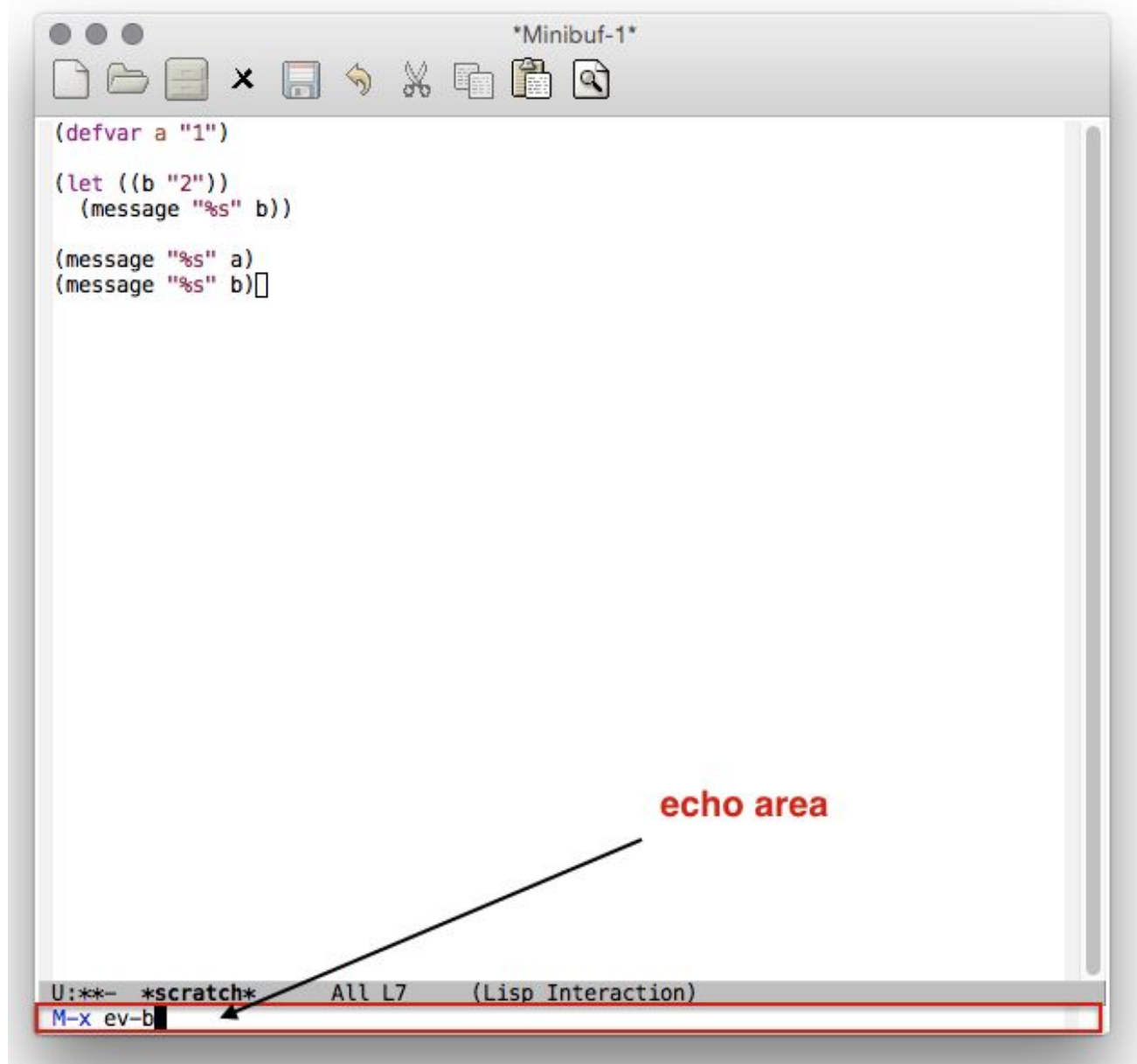
Lisp 提供了两种定义变量的方式，**defvar** 和 **let**，

其中 **defvar** 用来定义全局变量，**let** 用来定义局部变量。

例子：

```
(defvar a "1")  
(let ((b "2"))  
  (message "%s" b)) ; "2"  
(message "%s" a) ; "1"  
(message "%s" b) ; Error: Symbol's value as variable is void: b
```

以上程序中，我们用 `defvar` 定义了全局变量 `a`，和局部变量 `b`。
其中 `message` 用于在 Emacs 的 “echo area” 中输出内容，
`message` 的第一个参数是表示格式的字符串，第二个参数是待输出的内容。
Lisp 用分号表示注释。



为了执行这段程序，我们需要将它写到 Emacs 的 buffer 中，然后按 `M-x` 再输入 `eval-buffer` 回车，来求值整个缓冲区。
其中 `M-x` 表示按住 `alt` 键，然后再按 `x`，该快捷键命令会将光标定位到 `echo area`，等待用户输入一个函数名，

我们输入函数 `eval-buffer`，它用来求值当前 `buffer`，
它还有一个别名为 `ev-b`，可以记为 `M-x ev-b`。
注意，按 `M-x` 之后，我们不用输入 “`M-x`”，直接输入函数名 “`ev-b`”就可以了。
程序最终的执行结果如注释所示，变量 `a` 在整个程序中可用，而变量 `b` 只在 `let` 范围内可用。

3. 作用域和生存期

以上程序中，我们通过 `defvar` 和 `let`，让 `a` 的值为字符串“1”，`b` 的值为字符串“2”，
我们说，`defvar` 和 `let` 建立了两个绑定（binding），将 `a` 绑定为“1”，`b` 绑定为“2”。

The association between a variable and its value is called a binding.
——《Essentials of Programming Languages - P90》

变量除了可以分为全局变量和局部变量之外，还有另外两方面的属性，作用域（scope）和生存期（extent）。

作用域表示，在源代码文本中，绑定在什么地方（where）有效。生存期表示，在程序执行的过程中，绑定在什么时候（when）有效。

Emacs Lisp 支持两种形式的绑定，
动态绑定（dynamic binding）和静态绑定（lexical binding）。

动态绑定具有动态作用域和动态生存期，
动态作用域（dynamic scope），任何一段代码都可能访问变量的绑定，
动态生存期（dynamic extent），只有在绑定结构（例如 `let`）执行的过程中，绑定才有效。

静态绑定具有静态作用域（也称词法作用域）和无限生存期，
词法作用域（lexical scope），绑定在绑定结构的源代码文本范围中有效，
无限生存期（indefinite extent），某些情况下，绑定可能永远有效。

幸运的是，Emacs Lisp 同时支持这两种绑定方式，否则很难直观的理解它们，默认情况下 Emacs Lisp 支持动态绑定，我们还可以为 Emacs 启用静态绑定规则。

3.1 动态绑定

例子：

```
(defvar x 0)
(defun getx ()
  x)
(let ((x 1))
  (getx)) ; 1
(getx) ; 0
```

其中 `defun` 用于在 Emacs Lisp 中定义函数，以上代码定义了一个 `getx` 无参数函数，

`(getx)` 是对该函数的调用。

在对 `getx` 进行的第一次调用时，函数中引用了自由变量 `x`，Lisp 要寻找程序执行期间对 `x` 最近的绑定，

于是找到了 `let` 表达式中，`getx` 调用之前对 `x` 的绑定，为 1。

第二次调用 `getx` 时，`let` 表达式的执行已经结束了，它对任何变量的绑定都将销毁，

这时候再调用 `getx`，程序执行期间最近的对 `x` 的绑定，是 `(defvar x 0)` 对 `x` 的绑定，为 0。

在 Emacs Lisp 中，每一个符号（symbol）都有一个 value cell，表示变量的当前值（current dynamic value），当一个符号（symbol）被给定一个局部绑定时（dynamic local binding），Emacs 会把原来的 value cell 记录在一个栈上，然后把新值放入 value cell 中。当绑定结构（例如 `let`）执行完后，Emacs 进行弹栈操作，取出旧的值放回 value cell 中。

注意，其他语言中的全局变量并不是动态绑定，考虑以下 JavaScript 代码，

```
let x = 0;
```

```
function getX(){
  return x;
}

((x)=>{
  getX(); // 0
})(1);
getX(); // 0
```

JavaScript 的全局变量仍然是静态绑定，第一个 `getX` 被调用时，并不会携带 `x` 的任何信息过去。

`getX` 总是从源代码文本范围内寻找 `x`，JavaScript 对变量采用的是静态绑定。

3.2 静态绑定

例子：

```
; -*- lexical-binding: t -*-
(setq test (let ((foo "bar"))
  (lambda ()
    foo)))
(let ((foo "something-else"))
  (funcall test)) ; "bar"
(funcall test) ; "bar"
```

其中，`; -*- lexical-binding: t -*-` 是 Emacs 的文件变量（file variable），用于对当前文件或 buffer 启用静态绑定规则，它必须位于文件或者 **buffer** 的第一行。

在调用 `test` 函数时，函数中引用的自由变量 `foo`，总是从源代码文本范围内离该函数最近的位置寻找，

于是找到了 `(lambda () foo)` 外层 `let` 中绑定的 `"bar"`，

所以两次对 `test` 的调用，结果都是 `"bar"`。

在 Emacs Lisp 中，每一个绑定结构都会创建一个新的词法环境（lexical environment），在这个环境中，保存了变量名和它所对应值之间的对应关系（即，绑定关系），当 Lisp 求值器对某个符号（symbol）求值的时候，它首先从词法环境中寻找值，如果找到了，就用这个值。否则就认为这个符号

(symbol) 是一个动态变量，读取符号 (symbol) 的 value cell 作为变量的值。

4. 全局变量的动态性质

(1) 动态绑定变量的值总是从符号 (symbol) 的 value cell 中获取，而静态绑定变量的值从词法环境中获取。

所以，无法使用 symbol-value 获取静态绑定变量的值。

```
; -*- lexical-binding: t -*-
```

```
(let ((x 1))  
  (symbol-value 'x)) ; Symbol's value as variable is void: x
```

(2) 即使启用了变量的静态绑定规则，全局变量仍然是动态绑定的。let 并没有引入新的静态变量 x，而是，建立了局部动态变量 x，然后用局部动态变量遮挡了全局动态变量的值。

```
; -*- lexical-binding: t -*-
```

```
(setq test (let ((x 1))  
  (lambda ()  
    x)))
```

```
(funcall test) ; 1
```

```
; -*- lexical-binding: t -*-
```

```
(defvar x 0)
```

```
(setq test (let ((x 1))  
  (lambda ()  
    x)))
```

```
(funcall test) ; 0
```

以上两段程序都启用了静态绑定规则，第一段程序中的 x 是静态绑定的，

第二段程序中的 x 是全局变量，使用 defvar 定义了，所以它是动态绑定的。

在进行试验时，需要在全新的 buffer 中，分别测试，

否则 (defvar x 0) 一旦执行，即使再重新 M-x eval-buffer，x 的值已经被定义了。

参考

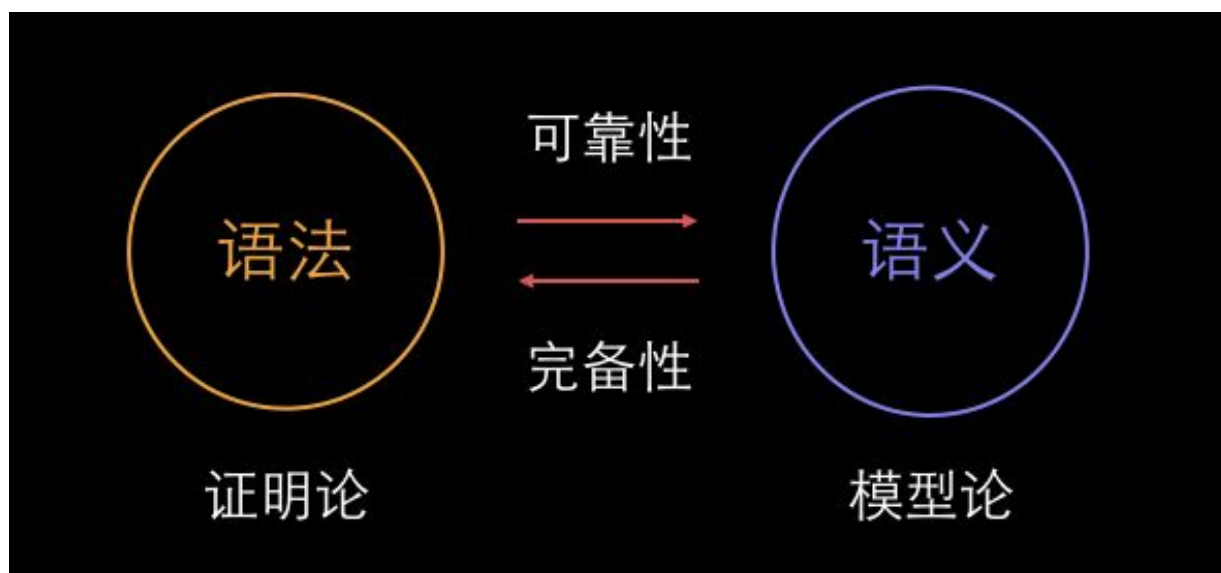
[GNU Emacs manual](#)

[GNU Emacs Lisp Reference Manual](#)

[Essentials of Programming Languages](#)

Emacs 之魂（五）：变量的“指针”语义

1. 语义学



在计算理论中，形式语义学是关注计算模式和程序设计语言含义的严格的数学研究领域。

语言的形式语义是用数学模型去表达该语言描述的可能计算来给出的。

提供程序设计语言形式语义的方法很多，其中主要类别有：

操作语义（operational Semantics），指称语义（denotational semantics），

公理语义（axiomatic semantics），代数语义（algebraic semantics）。

1.1 操作语义

将语言成分所对应的计算机系统的操作，作为该语言成分的语义，这样的建模方式称为操作语义学，

语言的操作语义应该是标准的，不应该依附于一个特定的计算机系统，因此，人们使用抽象的机器和抽象的解释程序来定义语言的操作语义。

操作语义学的基本思想来源于编译器和解释器，编译器或者解释器描述了程序语言的具体实现。

操作语义最早被用于定义 Algol 68 的语义，

1964 年 Peter Landin 使用了 SECD machine 这种抽象机器，定义了表达式的操作语义。

1980 年，Gordon Plotkin 提出了结构操作语义（structured operational semantics），它在更一般的数学结构上用归约关系（reduction relation）建立了语义的解释系统，

这种语义学具有结构化的特征，语言中复合成分的语义是由其子成分的语义复合而成的，

结构操作语义，对软件工程中结构化编程具有重要的指导意义。

1.2 指称语义

$$\begin{aligned}
\mathcal{E}[(\text{lambda } (I^* . I) \Gamma^* E_0)] = & \\
& \lambda \rho \omega \kappa . \lambda \sigma . \\
& \text{new } \sigma \in L \rightarrow \\
& \text{send}(\langle \text{new } \sigma | L, \\
& \quad \lambda \epsilon^* \omega' \kappa' . \# \epsilon^* \geq \# I^* \rightarrow \\
& \quad \text{tievalsrest} \\
& \quad (\lambda \alpha^* . (\lambda \rho' . C[\Gamma^*] \rho' \omega' (\mathcal{E}[E_0] \rho' \omega' \kappa')) \\
& \quad \quad (\text{extends } \rho (I^* \S \langle I \rangle) \alpha^*)) \\
& \quad \epsilon^* \\
& \quad (\# I^*), \\
& \quad \text{wrong "too few arguments"} \rangle \text{ in } E) \\
& \kappa \\
& (\text{update } (\text{new } \sigma | L) \text{ unspecified } \sigma), \\
& \text{wrong "out of memory"} \sigma \\
\mathcal{E}[(\text{lambda } I \Gamma^* E_0)] = & \mathcal{E}[(\text{lambda } (. I) \Gamma^* E_0)]
\end{aligned}$$

人们用程序设计语言编程，计算机系统用于加工数据，不同的计算机系统有不同的结构，因此对同一条命令的执行过程可能不同，但是最终结果应该是相同的。

指称语义学认为语言成分的含义是语言成分本身固有的，与计算机系统无关，所以，不应该将语言成分的执行过程看做它的语义，语言成分的语义应该是它的执行结果。

这种将最终结果看做语言成分语义的建模方式，称为指称语义学，这个最终结果称为该语言成分的指称。

语言成分的指称一般是一个数学对象，如整数，集合等等。

指称语义学是 Christopher Strachey 于 1964 年提出的，

后来 Dana Scott 创建了论域理论（domain theory）为指称语义学奠定了数学基础。

指称语义学方法在定义语言的语义时，先确定语言成分的指称，然后给出语言成分与其指称之间的映射关系，

而且确定复合语言成分指称的过程是语法制导的（syntax-directed），也称为

结构化的 (structural) ,
即语言成分的指称只依赖于它的子成分的指称。

1.3 公理语义

Example 1 while rule
 Prove $\{x \leq 10\} \text{ while } x < 10 \text{ do } x := x + 1 \text{ done } \{x < 10 \wedge x \leq 10\}$
 Solution:
 First, the while rule requires to prove:

$\{x \leq 10 \wedge x < 10\} x := x + 1 \{x \leq 10\}$
 $= \langle \text{Simplification} \rangle$
 $\{x < 10\} x := x + 1 \{x \leq 10\}$
 $\Leftarrow \langle \text{Inference rule: } := \rangle$
 $x < 10 \Rightarrow (x \leq 10) [x := x + 1]$
 $= \langle \text{Assignment} \rangle$
 $x < 10 \Rightarrow x + 1 \leq 10$
 \dots

while rule:

$$\frac{\{P \wedge B\} S \{P\}}{\{P\} \text{ while } B \text{ do } S \text{ done } \{P \wedge B\}}$$

 Inference rule $:=$

$$\frac{P \Rightarrow R [x := e]}{\{P\} x := e \{R\}}$$

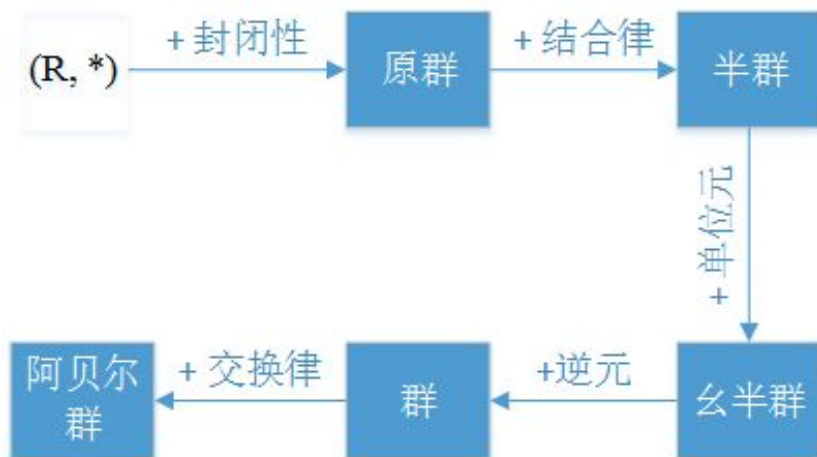
\dots
 $= \langle \text{Arithmetic} \rangle$
 $x < 10 \Rightarrow x \leq 9$
 $= \langle \text{Arithmetic} \rangle$
 true

公理语义在定义语义的时候，采用了数学中的公理化方法，
 语言的公理语义构成了一个由“公理”和“定理”组成的逻辑证明系统，
 它认为语言的语义，是由这个证明系统所能反映出来的一切性质。
 操作语义可以看做公理语义的一种有向形式。

1967 年，Robert W. Floyd 提出了论证一个程序是否具有某种性质的数学方法，
 1969 年，Tony Hoare 第一次用公理系统 (Hoare logic) 定义了一类程序设计
 语言的语义。

Hoare logic 使用带有前置条件和后置条件的归纳命题 (inductive proposition) ，作为描述语义的形式化工具。

1.4 代数语义



指称语义的研究方法建立在递归函数论基础之上，公理语义的研究方法建立在谓词逻辑基础之上，

代数语义学用代数方法研究计算机语言的语义，它建立在抽象代数的基础之上。

代数语义学描述了程序中不同种类（sort）的数学对象（object），以及这些对象之间的运算，它们构成了一种代数结构，

代数语义学，通过分析这种代数结构的性质，来描述程序的语义。

代数语义学源于人们对抽象数据类型（abstract data type）的研究，

泛代数（universal algebra）是一个用于研究抽象数据类型的数学框架。

在泛代数中，抽象数据类型的语法由代数项（algebraic term）描述，公理语义用项之间的等式集（a set of equations）描述，而指称语义对应于一个 Σ 代数，操作语义通过给等式设定方向来表示。

2. Lisp 的 Pointer semantics

2.1 Conceptual pointer

Lisp 语言提供了一个简洁的计算模型，是因为它在语义上的简洁性，

Lisp 语言中所有的值都可以“看做”指针，所有的内存都可以“看做”在堆（heap）中分配。

```
(defvar foo 5)
```

Lisp 会在堆中为 x 分配内存，让它包含一个指针，指向另一块初始化为 5 的内存，即，Lisp 中的值总是可以“看做”指向堆内存的指针。

我们可以用下图表示：

```
  +-----+
foo | *---+--->5
  +-----+
```

Lisp 语言的具体实现中，会采用不同的策略避免频繁分配内存的开销，但是在语言层面对用户是不可见的。

2.2 pointer semantics & value semantics

和 C 语言不同的是，Lisp 用户不用显式的释放内存，这使得心智负担大大降低了，Lisp 用户不用对指针解引用（dereference），因为默认每次都会这么做。

例如，算术加法函数 `+`，接受两个指针作为参数，

它会在进行加法操作之前，自动对参数进行解引用，并且返回一个指向加法结果的指针。

```
(+ 1 2)
```

当我们对上述表达式求值的时候，加法函数会得到两个指针，分别指向 1 和 2，求值结果会返回一个指向 3 的指针。

大部分编程语言除了具有 pointer semantics 之外，还具有 value semantics。

例如，它们会把 1，2 看做值，1 和 2 的每次出现，都是不同的一份拷贝。

Lisp 语言只有 pointer semantics，1 和 2 的每次出现都表示对唯一的数字对象的不同引用。

由于效率方面的考虑，具体实现可以做出任何调整，只要不影响 pointer semantics。

人们常说 Lisp，Smalltalk，Java 等这些语言没有指针，其实不太合理，而是应该说，所有的东西都是指针。（pointer semantics

把所有东西都实现为一个指针，代价是高昂的，我们必须把所有的指针，以及它指向的对象，全都分配在堆中。还必须使用额外的操作访问这些内存。

幸运的是，Lisp 的具体实现中，并不会这样表示它们，会对多种情况进行优化。例如，变量中会直接保存数字的二进制形式，而不是一个指向堆内存的指针。使用标签（tag）与那些具有相同二进制模式的指针进行区分。

2.3 例子

在 Lisp 语言中，一个 pair（又称为 cons cell），是一个在堆中分配的对象，它包含两个字段，每个字段都可以包含任意种类的值，例如，数字，字符，布尔值，或者一个指向其他堆内存的指针。

由于历史原因，第一个字段称为 `car`，第二个字段称为 `cdr`，pair 可以通过 `cons` 函数来创建。

```
(cons 22 15)
```

以上表达式创建了一个 pair，它包含了两个字段，其中 `car` 字段为 22，`cdr` 字段为 15。

我们可以用下图来表示：

```

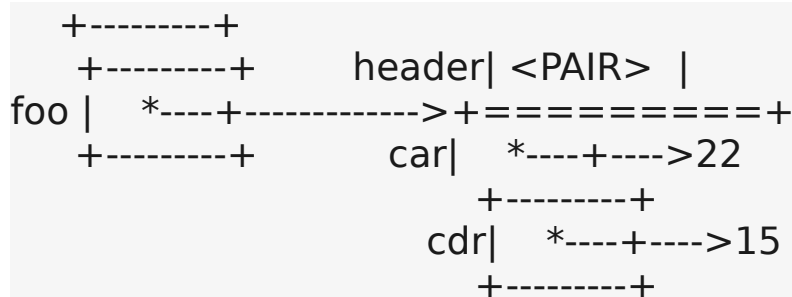
+-----+
header| <PAIR-ID> |
+=====+
car| *-----+----->22
+-----+
cdr| *-----+----->15
+-----+
```

其中，header 用于保存类型信息，用于表明在堆中分配了什么类型的对象，Lisp 用户不必关心 header 的值。

`car` 字段包含了一个指针，指向了 22，`cdr` 字段包含了另外一个指针，指向了 15。

```
(defvar foo
  (cons 22 15))
```


假设我们定义了一个全局变量，让它的值为一个 pair，于是我们可以用下图来表示：



变量 foo 指向了一个 pair，pair 中的两个字段分别指向了 22 和 15。

使用 car 函数和 cdr 函数可以得到 pair 中保存的两个字段的值，

```
(car foo)
> 22

(cdr foo)
> 15
```

参考

[形式语义学引论 - 周巢尘](#)

[程序设计语言理论基础](#)

[Syntax and Semantics of Programming Languages](#)

[An Introduction to Scheme and its Implementation](#)

Emacs 之魂（六）：宏与元编程

数据和代码

如果说 Lisp 语言有一个特性最能使人津津乐道的话，我想应该是它的宏系统（macro system）了吧，

在 Lisp 语言中，程序和代码的表现形式（textual representation）几乎一致，造就了它无与伦比的元编程能力。

这种对称性，使得 Lisp 语言可以像处理数据一样优雅的处理代码本身。

并且和其他语言不同的是，Lisp 的宏系统，并不是简单的文本操作，而是建立在语法对象（syntax object）基础之上。

前文提到过，我们直接写(foo bar bar)表示函数调用或者宏调用（macro call），

加引用'(foo bar bar)表示列表字面量，

直接写 x 表示变量或者函数，加引用'x 表示符号（symbol）。

如果我们把列表字面量和符号看做数据，把变量和函数调用看做程序，那么数据和程序的表现形式（textual representation）几乎是相同的，只差一个引用。

所以，如果一个函数能够处理数据（列表/变量），那么它也一定能够处理被引用的程序，

同理，如果一个函数能够返回一段数据（列表/变量），那么去掉引用之后（使用 eval），

也可以看做它是返回了一段程序。

例如，

```
(defun inc (var)
  (list 'setq var (list '1+ var)))
```

```
(inc 'x) ; (setq x (1+ x))
```

我们定义了一个 inc 函数，它接受 var 作为参数，返回了一个列表。

即，(inc 'x)的求值结果为(setq x (1+ x))，

其中，(setq x (1+ x))是一个列表。

我们可以通过 eval 直接把返回的列表当做程序来执行，

```
(defvar x 0)
(eval (inc 'x))
```

```
x ; 1
```

x 的值被修改了，变成了 1。

定义一个宏



我们只需要将上文的 inc 稍作修改，就可以把它转换成一个宏（macro），我们只需要将 defun 改成 defmacro 即可，

```
(defmacro inc (var)
  (list 'setq var (list '1+ var)))
```

现在 inc 就是一个宏（macro）了，它的使用方式和函数非常相似，

```
(defvar x 0)
(inc x)
```

```
x ; 1
```

我们看到，这里直接使用了 (inc x)，而不是 (inc 'x)，

并且，(inc x) 的作用和直接写程序 (setq x (1+ x)) 是一样的。

(inc x) 我们称之为宏调用（macro call），

而 (setq x (1+ x)) 我们称之为宏展开（macro expansion）后的程序。

编译器或者解释器会采用不同的策略进行宏展开，

一般而言，编译器会在求值程序之前，将代码中所有的宏（macro）进行展开，

即，将所有的宏调用 (inc x)，替换成它返回的那段程序 (setq x (1+ x))，

直到代码中不再包含宏（macro）为止，然后再进行编译。

一个简单的解释器实现，可能会一边执行程序一边进行宏展开操作，它会在运行时，通过判断符号（symbol）的类型，来决定进行函数调用还是宏调用。

这样可能会有助于理解宏的递归展开问题。

一个宏展开式中，可能还会包含其它的宏，也可能还会包含另一个宏的定义。

（以后的文章中，我们会介绍）

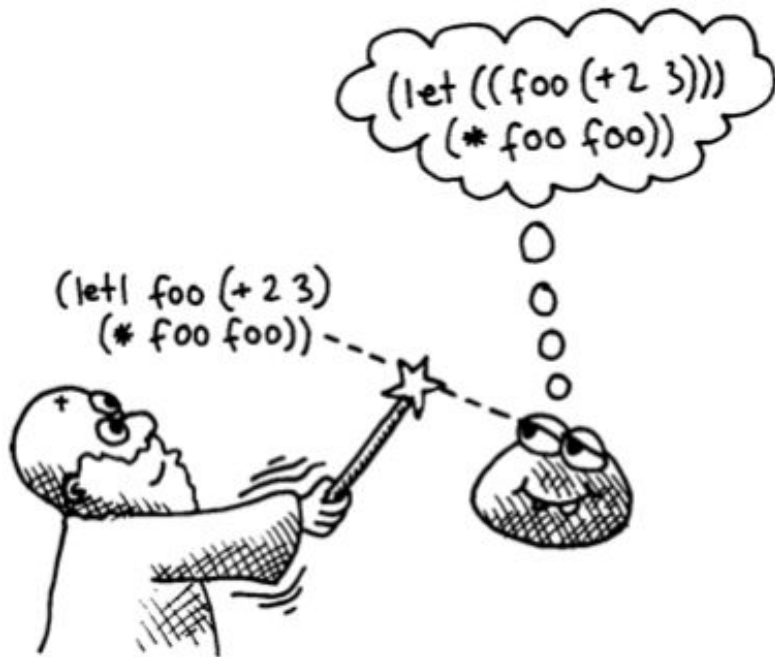
因此，在宏定义中，进行的具有副作用（side effect）的操作，

其执行时机并不是在运行时，而是在宏展开阶段，

而如果宏实参中包含了带有副作用的操作，那么它可能被展开到源代码中的多个位置，

从而被执行多次。

语法对象



在 Emacs Lisp 中，宏变量 `inc` 实际上是一个转换函数，

它将 `var` 转换成了 `(list 'setq var (list '1+ var))`，即把符号（symbol）转换成了一个列表对象。

宏变量的值与函数一样会保存在符号 (symbol) inc 的 function cell 中，因此，一个符号 (symbol) 不可能既表示一个函数又表示一个宏 (macro)。当 Lisp 解释器遇到一个符号 (symbol) 的时候，会判断它到底是一个变量，一个函数还是一个宏 (macro)。

```
(defun add1 (x)
  (+ x 1))
```

```
(defvar a 1)
(add1 a)
```

如果是一个函数，且当前进行的是函数调用(add1 a)，那么就会先求值它的实参，a 求值为 1，再将 add1 的形参 x 绑定为实参的值 1，再求值函数体，即，求值(+ x 1)，结果为 2。

```
(defmacro inc (var)
  (list 'setq var (list '1+ var)))
```

```
(defvar x 0)
(inc x)
x ; 1
```

如果是一个宏 (macro)，且当前进行的是宏调用(inc x)，那么它并不会像函数那样先求值函数体，而是直接将宏形参绑定为宏调用的实参值。

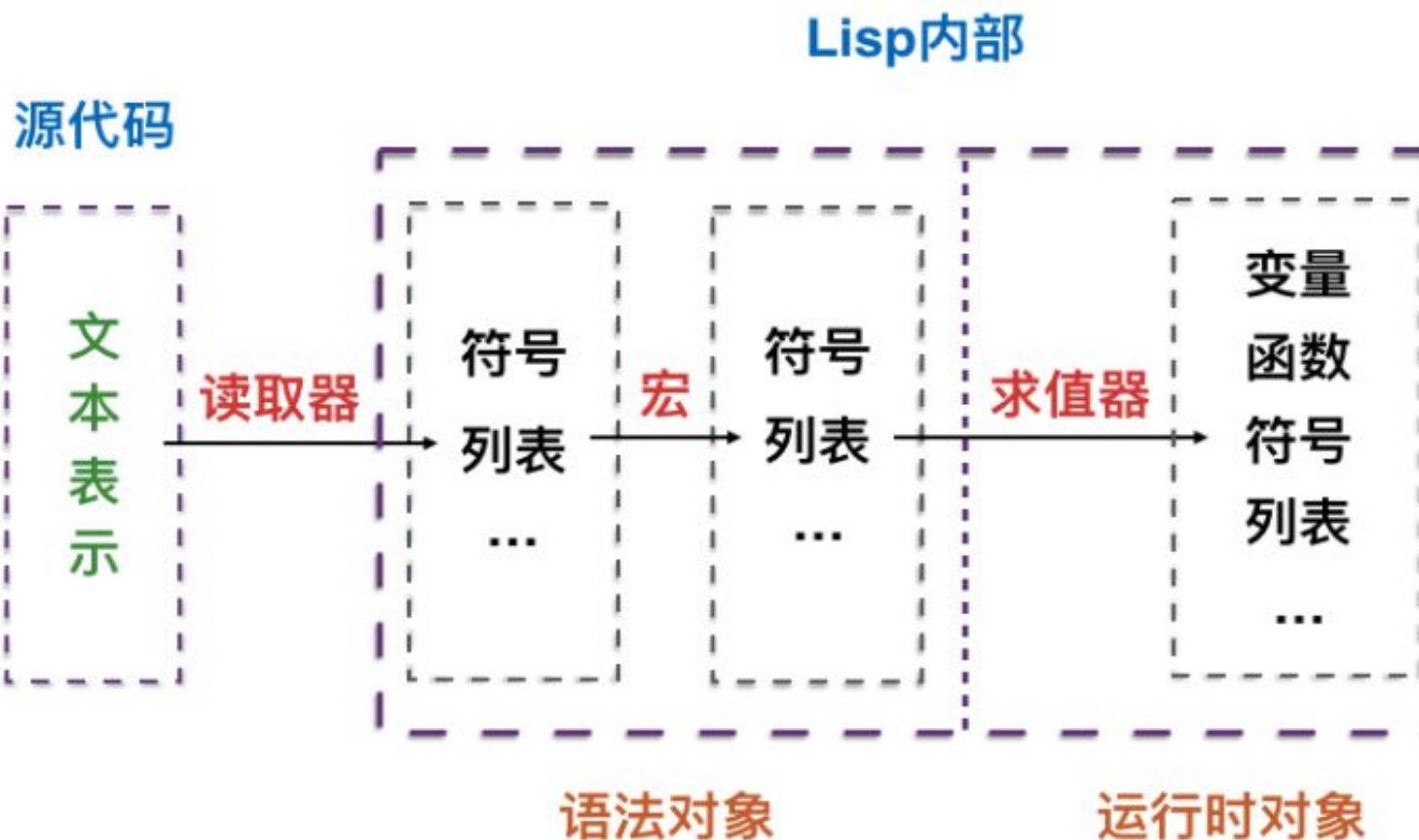
即，var 绑定为符号 (symbol) x。

值得注意的是，宏调用的实参，是一个符号 (symbol)，它是一个 Lisp 对象，而不是一个字符串，

宏 (macro) 所返回的结果，也是一个 Lisp 对象。

更明确的说，宏 (macro) 是一个针对语法对象 (syntax object) 的变换函数，它对读取器获得的语法对象 (syntax object) 进行变换。

在某些 Lisp 方言，例如 Scheme，这些语法对象 (syntax object) 包含了上下文信息，使用它们可以编写出强大而灵活的宏 (macro)。



这里容易引起混乱的是，在 Emacs Lisp 中，直接使用了符号和列表表示了语法对象，

而实际上语法对象是一个数据结构，在其内部包含了符号和列表的信息。这样做的好处是，在宏展开阶段宏（macro）接受和返回的都是语法对象，而在运行时阶段，处理的都是运行时对象了。

（例如：`syntax->datum` 和 `datum->syntax`）

通过以下程序我们可以验证，`var` 确实是一个符号（symbol）。

```
(defmacro inc (var)
  (message "%s" (symbolp var)) ; t
  (list 'setq var (list '1+ var)))
```

我们之前十分小心的区分了标识符，符号（symbol）和变量，是为了在类似这样的场景中保持清醒。

标识符经过 Lisp 读取器，在 Lisp 内部会变成一个符号（symbol），它是一个语法对象，

然后 Lisp 会对所有的宏（macro）进行展开，将这些语法对象绑定到宏形参上，对语法对象进行变换。

最后，求值器在运行时会求值这些符号（symbol），得到一个变量值或者函数值。

因此，编写宏（macro）可以看作是对编译器或者解释器进行编程，Lisp 允许用户在表达式被求值之前对它进行一些变换。

总结

本文初步介绍了 Lisp 的宏系统，展示了宏调用与函数调用之间的异同，我们发现 Lisp 的宏系统是建立在语法对象（syntax object）基础之上的，而不是简单的进行文本替换。

此外，由于 Emacs Lisp 的宏（macro）不是卫生的（hygienic），所以会和 Common Lisp 一样出现变量捕获问题。

下文我们开始介绍一些 Lisp 宏的常见陷阱和用法。

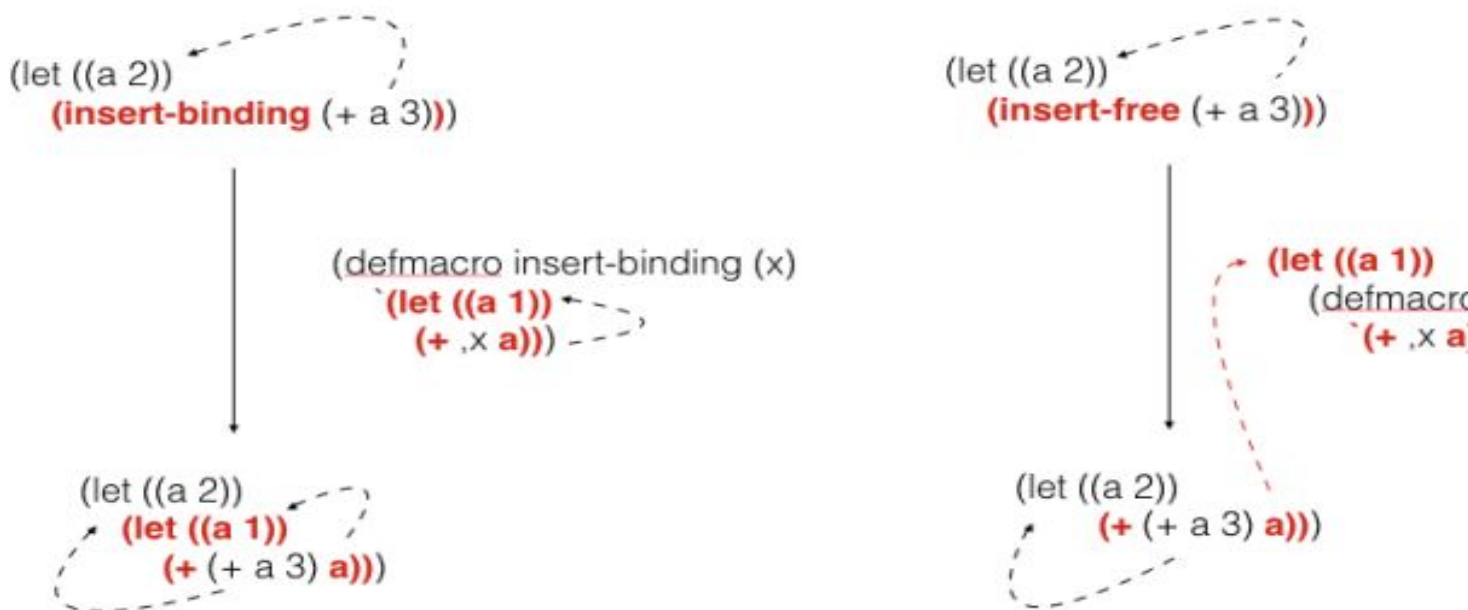
参考

[GNU Emacs Lisp Reference Manual](#)

[Chez Scheme Version 8 User's Guide](#)

[An Introduction to Scheme and its Implementation](#)

Emacs 之魂（七）：变量捕获与卫生宏



回顾

上文我们介绍了宏，它与函数是不同的，函数调用发生在程序执行期间，函数在调用之前，会先对它所有的实参进行求值，然后将形参绑定到这些实参的求值结果上，函数的返回值会作为函数调用表达式的值，Lisp 求值器不断的求值表达式，从而程序得以运行。

宏调用（macro call）发生在程序的编译期，或者说，宏调用发生在表达式的求值之前，在执行宏调用的过程中，宏形参直接绑定为实参所代表的语法对象（syntax object）上，宏调用的返回值，会进行表达式替换，将宏调用表达式替换为它的返回值，这个过程称为宏展开（macro expansion），之后在运行时，求值器就不会遇到宏了，所进行求值的只有被展开之后的表达式。

1. 交互函数

在介绍常用的宏之前，我们先介绍 Emacs 中交互函数（interactive function）的概念。

交互函数可以使用 `M-x` 在 echo area 中通过输入函数名进行调用（交互式调

用)，所以交互函数也称为命令（command）。

交互函数也可以被 Lisp 程序中的其他函数直接调用，这种调用方式称为非交互式调用。

Emacs 中函数定义 `defun` 包含以下几个部分，

```
defun name args [doc] [declare] [interactive] body ...
```

其中，`doc`，`declare` 和 `interactive` 都是可选的。

交互函数的定义中，具有 `interactive` 部分，

它是一个形如(`interactive arg-descriptor`)的表达式，用来指定该函数被交互调用时的行为，

对于非交互式调用，`interactive` 部分将失去作用。

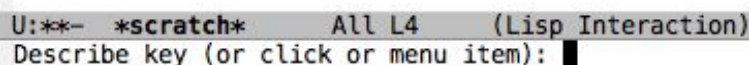
`arg-descriptor` 有三种可能的写法：省略，一个字符串，或者一个 Lisp 表达式。

具体情况可能会比较复杂，可以参考 [Using-Interactive](#)。

1.1 describe-key

`describe-key` 是一个交互函数，用来展示某个快捷键键相关的文档信息，

我们可以使用 `M-x describe-key` 来调用它，echo area 中会显示如下内容，等待我们键入一个快捷键，



U:~*~ *scratch* All L4 (Lisp Interaction)
Describe key (or click or menu item):

如果我们键入一个快捷键，例如 `C-a`，Emacs 就会展示出与 `C-a` 相关的文档信息了。我们还可以使用快捷键 `C-h k`，`C-h k` 相当于 `M-x describe-key`。

`C-a` runs the command `move-beginning-of-line` (found in `global-map`), which is an interactive compiled Lisp function in `'simple.el'`.

It is bound to `C-a`.

(`move-beginning-of-line ARG`)

Move point to beginning of current line as displayed.

(If there's an image in the line, this disregards newlines which are part of the text that the image rests on.)

With argument ARG not nil or 1, move forward ARG - 1 lines first.
If point reaches the beginning or end of buffer, it stops there.
To ignore intangibility, bind 'inhibit-point-motion-hooks' to t.

1.2 describe-function

describe-function 也是一个交互函数，用来展示某个函数（或者宏）相关的文档信息，它绑定到了快捷键 C-h f 上，
调用后，echo area 中会显示如下内容，等待我们输入函数（或者宏）的名字，



U:*** *scratch* All L4 (Lisp Interaction)
Describe function:

例如，when 相关的文档信息如下：

when is a Lisp macro in 'subr.el'.

(when COND BODY...)

If COND yields non-nil, do BODY, else return nil.

When COND yields non-nil, eval BODY forms sequentially and return value of last one, or nil if there are none.

它指出，when 是一个宏，并且定义在 subr.el 文件中。

鼠标左键点击 subr.el，会打开本地 subr.el.gz 文件中 when 的定义，如下，

（文件路径为：/Applications/Emacs.app/Contents/Resources/lisp/subr.el.gz

```
(defmacro when (cond &rest body)
```

```
  "If COND yields non-nil, do BODY, else return nil.
```

```
  When COND yields non-nil, eval BODY forms sequentially and return  
  value of last one, or nil if there are none.
```

```
  \ (fn COND BODY...)")
```

```
  (declare (indent 1) (debug t))
```

```
  (list 'if cond (cons 'progn body)))
```

可见，when 只是一个语法糖，最终会展开成 if 表达式。

subr.el.gz 文件中包含了很多常用的宏，

我们可以访问线上地址 [Github: emacs-mirror/emacs subr.el](https://github.com/emacs-mirror/emacs/blob/master/lisp/subr.el) 进行查阅。

2. 变量捕获

2.1 插入一个绑定

```
(let ((a 2))  
  (insert-binding (+ a 3)))
```



```
(defmacro insert-binding (x)  
  `(let ((a 1))  
    (+ ,x a)))
```

```
(let ((a 2))  
  (let ((a 1))  
    (+ (+ a 3) a)))
```

```
; -*- lexical-binding: t -*-
```

```
(defmacro insert-binding (x)  
  `(let ((a 1))  
    (+ ,x a)))
```

以上代码定义了一个宏 `insert-binding`，它将展开成一个 `let` 表达式，将 `x` 插入到一个 `a` 值为 1 的词法环境中。

其中，``(let ((a 1)) (+ ,x a))` 是反引用表达式，

下一篇文章中我们再详细讨论。

`(insert-binding 3)` 将展开成，

```
(let ((a 1))  
  (+ 3 a)) ; 4
```

然而，如果 `x` 中包含 `a`，就会引发歧义，例如，

```
(let ((a 2))  
  (insert-binding (+ a 3)))
```

上式会展开为，

```
(let ((a 2))
```

```
(let ((a 1))  
  (+ (+ a 3) a)) ; 5
```

我们看表达式(+ (+ a 3) a),

其中, 左边第一个 a, 来源于宏展开之前的词法绑定, 即,

```
(let ((a 2))  
  (insert-binding (+ a 3)))
```

而第二个 a, 来源于宏展开式中的词法绑定,

```
`(let ((a 1))  
  (+ ,x a))
```

在进行宏定义时, 我们并不知道 x 中有没有 a,

结果导致了, 宏展开式中的词法绑定意外捕获了 x 中的 a。

在本例中, x 就是(+ a 3), 其中 a 的值本来应该是 2,

结果展开后, 被宏展开式所捕获, 值变成了 1,

我们通过插入一个词法绑定, 完成了本例。

2.2 插入一个自由变量

```
(let ((a 2))  
  (insert-free (+ a 3)))
```



```
(let ((a 1))  
  (defmacro insert-free (x)  
    `(+ ,x a)))
```

```
(let ((a 2))  
  (+ (+ a 3) a))
```

```
; -*- lexical-binding: t -*-
```

```
(let ((a 1))  
  (defmacro insert-free (x)  
    `(+ ,x a)))
```

以上代码定义了一个宏 insert-free。

(insert-free 3)将展开为(+ 3 a)，其中 a 是自由变量，
a 的值取决于(insert-free 3)在何处被展开。

例如，

```
(let ((a 2))  
  (insert-free (+ a 3)))
```

将展开为，

```
(let ((a 2))  
  (+ (+ a 3) a)) ; 7
```

我们再来看表达式(+ (+ a 3) a)，

其中，左边第一个 a，来源于宏展开之前的词法绑定，即，

```
(let ((a 2))  
  (insert-free (+ a 3)))
```

而第二个 a，来源于宏展开式中的词法绑定，

```
(let ((a 1))  
  (defmacro insert-free (x)  
    `(+ ,x a)))
```

在进行宏定义时，虽然我们显式的将 a 绑定为 1，

但是 x 中包含的绑定，意外影响到了它，使得 a 的值变成了 2。

我们通过插入一个含自由变量的表达式，让它受展开式所处的位置影响。

2.3 hygienic macro

以上两个例子中，插入一个绑定会污染宏展开后的环境，而插入一个自由变量会被宏展开后环境所影响，

它们都有变量捕获问题，都不是卫生的 (hygienic)。

hygienic macro 通常翻译成“卫生宏”，是一种避免变量捕获的技术，

如果所使用的宏是卫生的，那么以上两个例子中，最后的求值结果应该都是 6，而不是 5 和 7。

卫生宏是一种语言特性，Scheme 中的宏是卫生的，而 Emacs Lisp 不是。

如果一个宏是卫生的，那么宏展开式中的所有标识符，仍处于其来源处的词法作用域中。

(1) 例如，根据 insert-binding 的定义，

```
; -*- lexical-binding: t -*-
```

```
(let ((a 1))  
  (defmacro insert-free (x)  
    `(+ ,x a)))  
(let ((a 2))  
  (insert-binding (+ a 3)))
```

将展开为，

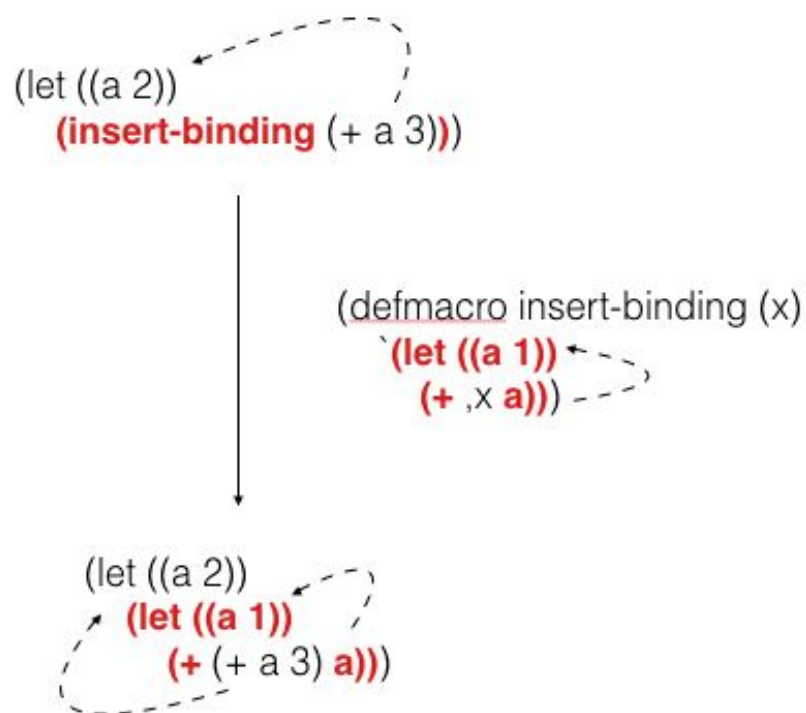
```
(let ((a 2))  
  (let ((a 1))  
    (+ (+ a 3) a)))
```

其中， $(+ (+ a 3) a)$ 中，

第一个 a ，来源于宏展开之前的词法环境，这个 a 的值为 2，

第二个 a ，来源于宏定义式，这个 a 的值为 1，

因此， $(+ (+ a 3) a)$ 求值为 6。



(2) 又例，根据 `insert-free` 的定义，

```
; -*- lexical-binding: t -*-
```

```
(let ((a 1))  
  (defmacro insert-free (x)  
    `(+ ,x a)))
```

```
(let ((a 2))  
  (insert-free (+ a 3)))
```

将展开为，

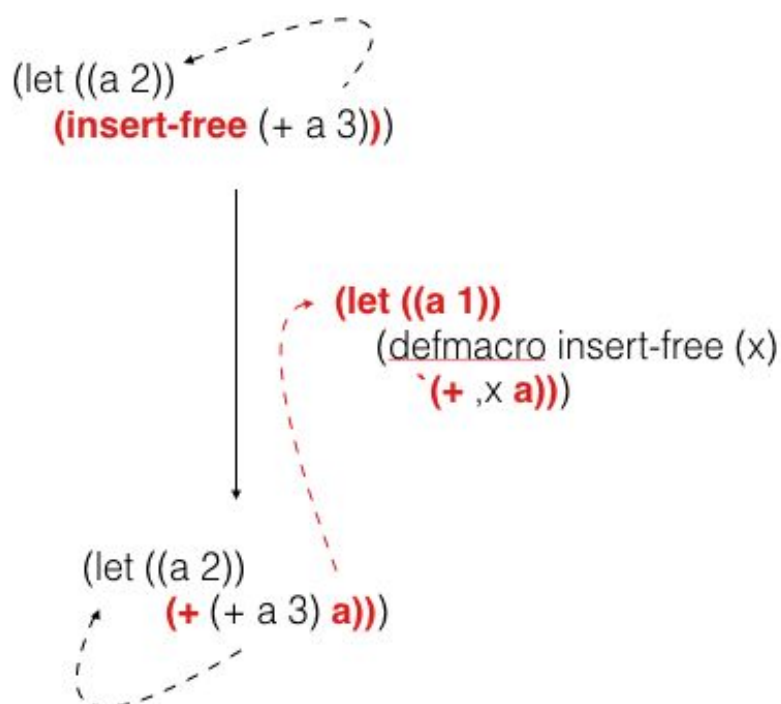
```
(let ((a 2))  
  (+ (+ a 3) a))
```

同理，`(+ (+ a 3) a)`中，

第一个 `a`，来源于宏展开之前的词法环境，这个 `a` 的值为 2，

第二个 `a`，来源于宏定义式，这个 `a` 的值为 1，

因此，`(+ (+ a 3) a)`的值也为 6。



总结

本文介绍了交互函数，介绍了如何查看一个函数或者宏的文档和定义，一些常用的宏，都可以通过查看 `subr.el` 来找到它们。

然后，我们介绍了两种与宏相关的变量捕获问题，引出了卫生宏的概念。

下文，我们继续讨论宏，来看一看展开为宏定义的宏之强大威力。

参考

[GNU Emacs Lisp Reference Manual](#)

[On Lisp](#)

[Let Over Lambda](#)

[The Scheme Programming Language](#)

Emacs 之魂（八）：反引用与嵌套反引用

1. 反引用

上文我们介绍了如何使用 defmacro 定义宏，

```
(defmacro inc (var)
  (list 'setq var (list '1+ var)))
```

我们定义了 inc 宏，(inc x) 会被展开为 (setq x (1+ x))，因此，

```
(defvar x 0)
(inc x)
```

```
x ; 1
```

宏做的是语法对象的变换操作，因此几乎每个宏最后都返回一个列表，

可是，类似上述 inc 宏那样，每次都使用 list 来创建列表，是一件麻烦的事情，所以，Lisp 提供了反引用（quasiquote/backquote），可以便捷的生成列表。例如，以上 inc 宏使用反引用来生成列表，可以修改为，

```
(defmacro inc (var)
  `(setq ,var (1+ ,var)))
```

可以看到，反引用 `(setq ,var (1+ ,var))` 与 (inc x) 的展开式 (setq x (1+ x)) 非常相像，

我们只需要将反引号去掉，然后将反引用表达式中的逗号表达式 ,var，替换为 var 绑定的值 x 即可。

2. 反引用表达式的求值规则

下面我们通过几个例子来说明反引用的使用方式，其中 => 表示“求值为”。

求值规则：

(1) 如果反引用表达式中不包含逗号,，那么它和引用表达式是一样的，因此反引用通常被看做是一种特殊的引用 (quote)

```
`(a list of (+ 2 3) elements)
=> (a list of (+ 2 3) elements)
```

(2) 反引用表达式中的逗号表达式会被求值

```
`(a list of ,(+ 2 3) elements)
=> (a list of 5 elements)
```

(3) 反引用表达式中的,@表达式，也会被求值，但是要求其结果必须是一个列表，

,@会去掉列表的括号，将列表中的元素放到,@表达式出现的位置

```
(defvar x '(2 3))

`(1 ,@x 4)
=> (1 2 3 4)

`(1 ,@(cdr '(1 2 3)) 4)
=> (1 2 3 4)
```

3. 生成宏定义的宏



以上，我们定义了宏 `inc`，

宏调用 `(inc x)`，会被展开为 `(setq x (1+ x))`。

在编写宏的时候，一个常用的思路是，

先考虑展开关系，即我们期望将 A 展开为 B，再根据这个线索编写相应的宏。

那么，我们可否编写一个宏，让它展开成 `(defmacro ...)` 呢？

是可以的，这是一种展开为宏定义的宏，它可以作为 `defmacro` 来使用。

考虑展开关系，我们期望将(create-inc)展开为

```
(defmacro inc (var)
  `(setq ,var (1+ ,var)))
```

于是，宏 create-inc 就应该被这样定义，

```
(defmacro create-inc ()
  `(defmacro inc (var)
    `(setq ,var (1+ ,var))))
```

我们来试验一下，

```
(create-inc) ; 定义了inc
```

```
(defvar x 0)
```

```
(inc x) ; 使用inc
```

```
x ; 1
```

我们还可以给 create-inc 加上参数。

考虑展开关系，我们将(create-inc-n y)展开为，

```
(defmacro inc-n (var)
  `(setq ,var (+ y ,var)))
```

那么 create-inc-n 应该怎么定义呢？事实上，

```
(defmacro create-inc-n (num)
  `(defmacro inc-n (var)
    `(setq ,var (+ ,',num ,var))))
```

第一次看到,',num 的时候，我非常惊讶，这到底是什么？

4. 嵌套反引用



嵌套反引用指的是，一个反引用表达式中嵌套出现了另一个反引用表达式。
在生成宏定义的宏中，嵌套反引用经常出现。

嵌套反引用表达式中，经常会出现类似`',num` 这样的表达式，
它不能被写成`num`，也不能被写成`,,num`，下面我们进行仔细的分析。

(1) `,num` 为什么不正确

先看一下展开关系，我们期望将`(create-inc-n y)`展开为，

```
(defmacro inc-n (var)
  `(setq ,var (+ y ,var)))
```

即，嵌套反引用表达式，应该按下述方式求值，

```
`(defmacro inc-n (var)
  `(setq ,var (+ ,',num ,var)))
=> (defmacro inc-n (var)
  `(setq ,var (+ y ,var)))
```

其中，`,var` 是不应该被求值的，因为这是内层反引用需要的，
如果我们将`',num` 写成`num`，那么它就和`,var` 一样不会被求值了，

```
`(defmacro inc-n (var)
  `(setq ,var (+ ,num ,var)))
=> (defmacro inc-n (var)
```

```
`(setq ,var (+ ,num ,var)))
```

这和我们期望的展开关系不同。

(2) ,,num 为什么不正确

写成,,num 在求值最外层反引用表达式的时候，确实会求值 num 的值，但是，在求值内层反引用表达式的时候，这个值还会被再求值一次。

(create-inc-n y)将被展开为，

```
`(defmacro inc-n (var)
  `(setq ,var (+ ,,num ,var)))
=> (defmacro inc-n (var)
  `(setq ,var (+ ,y ,var)))
```

可是，在进行宏调用(create-inc-n y)的时候，我们不应该关心 y 的值是什么，因为在宏展开阶段，y 可能还没有值。

而且，该展开式和我们预期的展开结果也不相同。

(3) ',num 是怎么来的

综上所述，我们需要在外层反引用表达式被求值的时候，求值 num，而在内层反引用表达式被求值的时候，不再继续求值 num 的值，因此，我们需要给 num 的值加上一个引用来“阻止”求值。

因此，(create-inc-n y)会被展开为，

```
`(defmacro inc-n (var)
  `(setq ,var (+ ',num ,var)))
=> (defmacro inc-n (var)
  `(setq ,var (+ ',y ,var)))
```

而内层反引用表达式被求值的时候，',y 将求值为 y。

所以，(inc-n x)将被展开为

```
`(setq ,var (+ ',y ,var))
=> (setq x (+ y x))
```

和我们期望的展开结果相同。

5. 嵌套反引用的求值规则

$$\sum_i \sum_j \sum_k v_{ijk} = \sum_i \left(\sum_j \left(\sum_k v_{ijk} \right) \right)$$

在生成宏定义的宏中，经常会出现嵌套反引用，

如果我们定义了另一个宏 other-macro 来生成 create-inc-n 的定义，

```
(defmacro other-macro ()  
  `(defmacro create-inc-n (num)  
    `(defmacro inc-n (var)  
      `(setq ,var (+ ,',num ,var))))))
```

那么，将出现三层嵌套反引用。

不过，不用担心，嵌套反引用也是有求值规则的，以下我们用两层嵌套反引用作为例子来说明。

求值规则：

(1) 嵌套反引用被求值的时候，**一次求值，只去掉一层反引用**，内层反引用不受影响，

```
`(defmacro inc-n (var)  
  `(setq ,var (+ ,',num ,var)))  
=> (defmacro inc-n (var)  
  `(setq ,var (+ ,',y ,var)))
```

(2) 嵌套反引用表达式中的逗号表达式，是否被求值，要根据情况来定，

如果最外层嵌套反引用总共有 n 层，那么一定不会出现包含大于 n 个逗号的表达式，

且包含逗号数目小于 n 的表达式不会被求值，**只有逗号数目等于 n 的表达式才会被求值。**

```
`(defmacro inc-n (var)  
  `(setq ,var (+ ,',num ,var)))  
=> (defmacro inc-n (var)  
  `(setq ,var (+ ,',y ,var)))
```

最外层嵌套反引用总共有 $n=2$ 层，

,var 表达式包含一个逗号， $1 < n$ ，不会被求值，

,',num 表达式包含两个逗号， $2 = n$ ，会被求值。

(3) 被求值的逗号表达式，其求值方式是，

去掉最右边的一个逗号，然后将表达式替换成它的值。

```
`(defmacro inc-n (var)
  `(setq ,var (+ ,',num ,var))))
```

```
=> (defmacro inc-n (var)
  `(setq ,var (+ ,',y ,var)))
```

,',num，去掉最右边的逗号,',num，然后将 num 替换成它的值 y，于是得到了,',y。

参考

[GNU Emacs Lisp Reference Manual](#)

[ANSI Common Lisp](#)

[On Lisp](#)

[Let Over Lambda](#)

Emacs 之魂（九）：读取器宏

1. 编译器宏



Lisp 源代码文本，首先经过读取器，得到了一系列语法对象，这些语法对象，在宏展开阶段进行变换，最终由编译器/解释器继续处理。

以下我们使用 `defmacro` 定义了一个宏 `inc`，

```
(defmacro inc (var)
  `(setq ,var (1+ ,var)))
```

它可以将 `(inc x)` 展开为 `(setq x (1+ x))`。

`inc` 宏可以看做对编译器/解释器进行“编程”，它影响了最终被编译/解释的程序。

因此，类似 `inc` 这样的宏，称为**编译器宏**（compiler macro）。

此外，还有一种宏，称为**读取器宏**（reader macro），

它在源代码的读取阶段，以自定义的方式，将文本转换为语法对象。

引用（quote）“`'`”，就是一个读取器宏，

它将源代码文本 `'(1 2)` 转换成 `(quote (1 2))`。

2. 用户定义的读取器宏

虽然，引用“`'`”是一个读取器宏，但它却不是由用户定义的，

支持用户自定义的读取器宏，是一个很强大的语言特性，

它可以让我们摆脱语法的束缚，创建自己的语言。

2.1 Common Lisp



(1) set-macro-character

在 Common Lisp 中，我们可以使用 set-macro-character，来模拟引用 “'” 的定义，

```
(set-macro-character #'  
  #'(lambda (stream char)  
    (list (quote quote) (read stream t nil t))))
```

当读取器遇到'a 的时候，会返回(quote a)。

其中 read 函数可以参考：[read](#)。

(2) set-dispatch-macro-character

我们还可以自定义捕获字符 (dispatch macro character) ，

例如，我们定义#?来捕获后面的文本，

```
(set-dispatch-macro-character #\# #\?  
  #'(lambda (stream char1 char2)  
    (list 'quote  
      (let ((lst nil))  
        (dotimes (i (+ (read stream t nil t) 1))  
          (push i lst))  
        (nreverse lst))))))
```

读取器会将#?7 转换成(0 1 2 3 4 5 6 7)。

(3) get-macro-character

我们还可以自定义分隔符，例如，以下我们定义了#{ ... }分隔符，

```
(set-macro-character #\}  
  (get-macro-character #\))  
  
(set-dispatch-macro-character #\# #\{  
  #'(lambda (stream char1 char2)  
    (let ((accum nil))  
      (pair (read-delimited-list #\} stream t))))
```

```
(do ((i (car pair) (+ i 1)))  
    ((> i (cadr pair))  
     (list 'quote (nreverse accum)))  
    (push i accum))))
```

读取器会将#{2 7}转换成(2 3 4 5 6 7)。

其中，get-macro-character 可以参考：[GET-MACRO-CHARACTER](#)。

2.2 Racket



在 Racket 中，我们可以通过创建自定义的读取器，得到一门新语言，例如，下面两个文件 language.rkt 和 main.rkt，

(1) language.rkt 模块创建了一个读取器，

```
#lang racket  
(require syntax/strip-context)  
(provide (rename-out [literal-read read]  
                     [literal-read-syntax read-syntax]))  
(define (literal-read in)  
  (syntax->datum  
   (literal-read-syntax #f in)))  
(define (literal-read-syntax src in)  
  (with-syntax ([str (port->string in)])  
    (strip-context  
     #'(module anything racket  
         (provide data)  
         (define data 'str)))))
```

(2) main.rkt 模块，就可以用新语法进行编写了，

```
#lang reader "language.rkt"  
Hello World!
```

然后，我们载入 main.rkt，查看该模块导出的 data 变量，

```
> (require (file "~/Test/main.rkt"))
```

```
> data
"\nHello World!"
```

在 `main.rkt` 中，

我们通过 `#lang reader "language.rkt"`，载入了一个自定义的读取器模块，该模块必须导出 `read`，`read-syntax` 两个函数。

这里，`read-syntax` 只是简单的获取源代码，导出到 `data` 变量中，最终返回了一个用于模块定义的语法对象(`module ...`)。

在本例中，它把 `"Hello World!"` 转换成了一个模块定义表达式，

```
(module anything racket
  (provide data)
  (define data "Hello World!"))
```

其中，`anything` 是模块名，`racket` 是该模块的依赖。

所以，当载入 `main.rkt` 后，我们就可以获取 `data` 的值了。

在实际应用中，我们还可以对源代码进行任意解析，创建自己的语言。

2.3 Emacs Lisp



Emacs Lisp 内置的读取器，并不支持自定义的读取器宏，为了实现读取器宏，我们需要重写 Emacs 内置的 `read` 函数，例如，`elisp-reader`。

Emacs 在启动时，会自动载入 `~/.emacs.d/init.el` 文件，然后执行其中的配置脚本，

因此，我们可以在 `init.el` 中调用 `elisp-reader`。

(1) 创建 `~/.emacs.d/init.el` 文件，

```
(add-to-list 'load-path "~/.emacs.d/package/elisp-reader/")
(require 'elisp-reader)
```

(2) 使用 git 克隆 elisp-reader 仓库到 ~/.emacs.d/package 文件夹，

```
git clone https://github.com/mishoo/elisp-reader.el.git
~/.emacs.d/package/elisp-reader
```

(3) 打开 Emacs，自动执行 init.el 中的配置，

(4) 在 Emacs 中定义一个读取器宏，然后求值整个 Buffer， (M-x ev-b)

```
(require 'cl-macs)
(def-reader-syntax ?{
  (lambda (in ch)
    (let ((list (er-read-list in ?} t)))
      `(list ,@(cl-loop for (key val) on list by #'cddr
                        collect `(cons ,key ,val))))))
```

(5) 测试 read 函数的执行结果， (C-x C-e)

```
(read "{ :foo 1 :bar \"string\" :baz (+ 2 3) }")
> (list (cons :foo 1) (cons :bar "string") (cons :baz (+ 2 3)))
(car { :foo 1 :bar "string" :baz (+ 2 3) })
> (:foo . 1)
```

源代码 { :foo 1 :bar "string" :baz (+ 2 3) } 被直接读取成了一个列表对象，

```
((:foo . 1) (:bar "string") (:baz (+ 2 3)))
```

对 car 函数而言，它看到的是列表对象，并不知道具体的语法是什么。

3. 总结

本文介绍了读取器宏的概念，Lisp 各方言中会对读取器宏有不同程度的支持，我们分析了 Common Lisp，Racket 以及 Emacs Lisp 的做法。

读取器宏直接作用到源代码文本上，用户定义的读取器宏可以对读取器进行“编程”，

借此可以支持自由灵活的语法，它是设计和使用 DSL 的神兵利器。

参考

[Common Lisp the Language, 2nd Edition: 8.4 Compiler Macros](#)
[ANSI Common Lisp: 14.3 Read-Macros](#)

Let Over Lambda: 4. Read Macros

[The Racket Reference: 17.3.2 Using #lang reader](#)

[Github: elisp-reader](#)