LISP 程序设计技术

严　勇　译

陈　屹　审校

科海培训中心

一九八八.二

前　　言

目前有关LISP 程序设计的书还很少，尤其是高级程序设计教材，基本上还是个空白。这本书为希望掌握 LISP 程序设计技巧的程序设计人员提供了丰富的材料，将有力地推动在这一领域的研究工作及实际的软件开发工作。高级程序设计技术是这本书的重点，但作者对LISP 语言的基本部分的阐述也是独具特色的。不同于一般的 LISP 教材，这本书以一种统一的观点从一个较高的角度描述 LISP 语言的各个组成部分，在不大的篇幅中对LISP语言进行了清晰、深入、系统的描述。这是一种新的尝试，实际上也为讲授LISP 语言的教师提供了参考。这本书的后两部分，从易到难逐步深入地介绍了用 LISP语言编程的种种基本技术，叙述简洁、脉络清晰，不同程度的读者都会有收获。这本书可作为LISP 语言高级课程的教材，也可以作为有一定程序设计经验的程序员进修的读物。此如那些实际进行软件开发的工作人员，特别是搞专家系统及决策支持系统的研制人员，这本书也可以当作手册来使用。建议予以出版。

编　者

12月4日

目　　录

第一部分　基本 LISP

在这一部分里，我们将系统地介绍 LISP 语言的基本内容。由于后面讲述高级程序设计的需要，我们在这里的侧重点是 LISP 语言的机制，旨在加深有一定程序设计经验的读者对于 LISP 语言的理解，因此在内容安排上有一定的考虑。出于完整性的考虑，我们没有略去 LISP 语言中的任何重要部分，但讲述方式与一般 LISP 语言课本不同，我们更多地强调对 LISP 语言的能力的剖析。对于接触 LISP 语言较少的读者，我们建议最好再有一本入门性的 LISP 教材作为辅助课本。

为了能对 LISP 语言有透彻的认识，我们从 LISP 语言数据结构与控制结构的数学基础入手。第一章我们谈 S- 表达式。S- 表达式理论本身有着很丰富的内容，本书中只涉及与 LISP 语言有关的部分。第二章介绍 LISP 语言：一种特殊的高级程序设计语言。我们以第一章的理论为基础，先给出 LISP 的核心部分，再从核心部分出发描绘 LISP 的全貌，最后深入到 LISP 的内部表示，讨论其运行机制。这一章里我们特别强调递归，因为这是 LISP 的能力的源泉。有关用 LISP 语言编写程序的基本方法及例子我们统一放在第三章中讲述。

第一章 S- 表达式

所谓 S- 表达式，指的是 Symbolic Expression，即符号表达式。它是一种数学结构，LISP 语言处理的数据是表示成这种结构，LISP 语言写出的程序也是这种结构。可以说，S- 表达式结构标志了 LISP 语言在句法上与其它面向数值处理的高级语言之间的差别。要透彻地理解 LISP 语言的机制，首先要理解 S- 表达式。

§1.符号处理

这一节里我们来看看符号处理的任务怎样由计算机来承担。符号处理与数值处理是完全不同的另外一个领域，而计算机的最基本的操作是二进制运算，或者更基本一些，01序列之间的变换。那么，符号处理的任务是如何表达为计算机能接受的形式？如何由计算机来操作？

首先是表达的问题。符号处理类似于处理词与句子，要点在于把计算机中的 0、1 看成代码，比如 010 视为字母 a 的代码，011 视为字母 b 的代码，001 视为字母 c 的代码，等等。由字母就能组成词，由词再组成句子。用句子就可以描述符号处理任务了。

然后是操作问题。符号处理的任务如何由机械的操作来完成？这里的精髓部分是“规则”概念。符号处理总是这样：通过一系列规则，变换一些符号串（句子）。

对于 LISP 语言，接受的合法句子就是 S- 表达式，而执行任务遵循的规则也写成 S- 表达式的样子。我们来看一个例子。

任务：对大学进行分类。

首先要把大学表示出来。我们用 S- 表达式写出两个句子，分别表示两个大学：

```
(UNIVERSITY1   (NAME   X)
               (AMOUNT-OF-STUDENTS    10,000)
               (AMOUNT-OF-DEPARTMENTS    23)
               (PHONE    282471)

(UNIVERSITY2   (NAME    Y)
               (AMOUNT-OF-STUDENTS    3,000)
               (AMOUNT-OF-DEPARTMENT11)
               (PHONE    890351)
```

把这些作为数据交给计算机。下一件事是规则，计算机需要知道规则才能做事。规则同样写成 S- 表达式的形状。规则的格式一般为：“满足一定条件，做或判断一定的事情”，用字母写就是“IF——THEN——”。假如我们要写这样一条规则：“如果一个大学系的数目超过15，学生数目超过6000，就算是一个复杂的大学”，那么可以这么写（我们称这条规则为CLASSIFY）：

```
(RULE    CLASSIFY
(IF      (GREATER AMOUNT-OF-DEPARTMENTS 15)
         (GREATER AMOUNT-OF-STUDENTS 6000))

(THEN    (IS COMPLEX)))
```

如果把这条规则交给计算机，作用于前面输入的数据，计算机就执行了一项符号处理任务：断定 UNIVERSITY1 是个复杂的大学。至于计算机如何把这条规则作用于那些数据，这正是 LISP 语言是如何运行的问题，第二章里我们再谈。

可以看出，透彻地研究怎样用 S- 表达式来表达各种各样的任务以及执行任务的方式是进行符号处理研究的前提，因而也是人工智能研究的前提。任务依领域不同而不同，任务表达的技术也随之而不同。本书的以后各章节中将随着程序设计技术的介绍而逐步介绍任务表达的技术，往往两者是融合在一起的。

LISP 语言集中了人们用S-表达式表达如何执行任务的经验。由于任务本身是表达成S-表达式的，因此 LISP 语言实际上是一整套处理 S- 表达式的手段。

我们在第一章里要做的，就是考察 S- 表达式的结构，以及在 S- 表达式上能有些什么样的处理手段，以期在进入第二章后对 LISP 语言能有更深入的理解。

§2.S- 表达式

S- 表达式由三个部分组成：

(i) 数。它们不是主要组成部分，但却是重要的；不管怎么说，任何任务中都可能会包含有数字。

(ii) 原子。这相当于词。例如，前面大学分类的例子中的 UNIVERSITY1，PHONE 等等都是原子。其它例子有：K#，ZYZ，ADEN，等。一个原子的构成原则是，它是一串符号，其中的第一个符号是字母或＃号星号横杠之类的字符；，但不能是数字。数字可以出现于原子的中间或末尾。通常，原子都用有意义的单词，那是为了便于记忆。原子不一定非要有意义。说明一点：有些书中把数也当作原子，称为数字原子，而把我们这里的原子称为文字原子。无论如何，两者在概念上是有区别的。

为了方便，我们规定两个特殊的原子：T 与 NIL。对它俩的处理是特殊规定的，因为其含义是确定的：T 代表“真”，NIL 代表“假”，或“空”。（其它原子的含义可以由用户随便规定）。以后我们会看到这两个原子的用途。

(iii) 点对。这是 S- 表达式的主要组成部分，它基于前两部分。我们逐步介绍：

首先是简单点对：其构成是由两个原子(或数)中间加一个点，然后两边加括号。例如：

(Z. Y)　　(A. 8)　　(88. 9)

(G#. UNIVERSITY1)

等等都是简单点对。

然后是复杂点由两个点对（简单的或复杂的）中间加一个点，两边加括号组成。例如：

((A. B) . (Z. Y))

((Z. Y) . (Z. Y))

(((A. B) . (Z. Y)) . (G#. UNIVERSITY1))

((((A. B) . C) . (D. E)) . (HELLO. (F. G)))

都是复杂点对。

区分简单点对与复杂点对只是为了引进点对概念时的方便。以后不再作此区分，而只是说点对。

组成点对的两个部分各有一个名称：左边的称为这个点对的首部，右边的称为尾部。

最重要也最常用的点对是表，我们按照同样办法分简单表和复杂表来介绍（对应简单点对与复杂点对）。

简单表：首部是原子，尾部是 NIL 的点对。例如：

(A. NIL)

(PHONE. NIL)

都是表。我们对于表有约定的书写方法，上两个例子写成：

(A)

(PHONE)

特别地，原子 NIL 也被看作是个表，称为“空表”，写成“( )”。一般地我们很少写“( )”。而只是写 NIL。什么间候它是原子，什么间候它是表，要根据上下文确定，不过在这个问题上不大会发生混淆。

稍复杂的表：首部是原子，尾部是表的点对。例如：

(C. (A. NIL)) 写成 (C A)

(B. (C. (A. NIL))) 写成 (B C A)

(A. (B. (PHONE. NIL))) 写成 (A B PHONE)

等都是表。

复杂表：首部、尾部都是表的点对。例如：

(Z. ((Y. (Z NIL)). NIL))

写成 (Z (Y Z))

((PHONE. (A. NIL)). ((Y. (Y. NIL)). (Z. NIL)))

写成 ((PHONE A) (Y Y) Z))

都是表。

以后不再区分简单的与复杂的表，一律都称作表。

表是点对的一种。(A. B) 这种点对不是。通常写表时不写成点对的形式，而写成上面讲到的那种约定形式。但是，看到 (A B C) 时，一定要记住最右端隐含了一个 NIL，它实际上是 (A. (B. (C. NIL))) 的缩写。理解表的原则是：每个右括号都暗示着一个 NIL。例如，这个表

((A Z) Y) ((B)))

隐含了五个 NIL，写成点对时是：

(((A. (Z. NIL)). (Y. NIL)). (((B. NIL). NIL). NIL))

我们前面举的大学表示的例子，就是用这种表写出的。实际上，LISP 语言涉及的S-表达式几乎都是表，我们之所以从点对过渡到表，是因为精细地使用 LISP 语言时，就会涉及隐在表后面的点对结构，如果对此没有透彻的理解，往往会在编写程序时遇上莫名其妙的麻烦。作为对照，我们简单地谈谈把表视为独立的结构时是如何给出的；我们实际上可以不涉及点对结构而直接给出表的构成：

表的构成是：

(i) 一对左右圆括号是表。

(ii) 若干原子用括号括起来是表。

(iii) 若干原子及表用括号括起来是表。

如果对表结构进行理论研究，不涉及点对表达式，这就可以作为表的定义了。读者可以与前面从点对过渡来的表定义相比较。

至此，S- 表达式的面貌都勾画出来了。表是 S- 表达式中的部分，但非常重要，以至于有时人们说 S- 表达式的时候，实际上指的是表。有一点需要注意，NIL 是表，也是原子，但 T 不是表，T 仅仅是个原子。

关于表我们有一些术语：

首层元素：作为点对，其首部是首层元素之一，然后是其尾部中的首部（表的尾部是表或 NIL），如此下去，直到尾部为 NIL。这一过程中，每个能充当首部的元素都算作表首层元素。看一些例子：

(A B C)

的首层元素是原子 A、原子 B、原子 C、

(Z NIL 8 Y)

的首层元素是原子 Z、原子 NIL，数 8，原子 Y。

((A B) (C T) NIL T)

的首层元素是表 (A B)，表 (C T)、原子 NIL，原子 T

(A ((B)) (C D))

的首层元素是原子 A、表 ((B))，表 (C D)

线性表：首层元素都是原子或数的表。

头部：视为点对表达式时的首部。例如：

(A B C)

的头部是原子 A

((A B) C)

的头部是表 (A B)

尾部：视为点对表达式时的尾部。例如：

(A B C)

的尾部是 (B C) 因为它写成点对表达式时是

(A. (B. (C. NIL)))

尾部是 (B. (C. NIL)) 写成表就是 (B C)

注意，(A B C) 的尾部不是"B C)"

另外，(A) 的尾部是 NIL。

表的书写方法提供了一种化简 S- 表达式书写的一种方法。化简 S- 表达式的原则是：若一个圆点右边紧邻左括号，就可以把这个圆点及左括号连同与这个左括号配对的右括号一起删除。例如

(A. (B. (C. D))) 可以化简为 (A B C. D)

((A. B) . (C. D)) 可以化简为 ((A. B) C. D)

(A. (CC. D) . E)) 可以化简为 (A (C. D) . E)

特别地，当我们遇到 NIL，就把它写成两个括号"( )"，而采用化简原则

(A. (B. NIL)) 可以写为 (A. (B. ( ))) 而进一步化简为 (A B)，我们把S-表达式的写法与表的约定写法联系起来了。

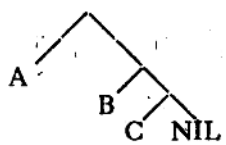
§3. 表达式的二叉树表示

第二节介绍 S- 表达式时只介绍了它的线性书写，很不直观，那主要是因为计算机只接受线性的符号串。与计算机打交道的 S- 表达式都是写成线性形式。但研究 S- 表达式的时候并不非要写成那么难读的形式，我们有更直观的写法：二叉树形式，在这种形式下，一个 S- 表达式的结构能清晰地表达出来。常常在思考有关 S- 表达式的问题时，我们借助这种表示法。

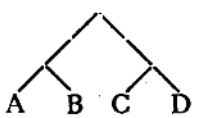
二叉树表示是二维的，先看看几个例子。

(A. B) 写成二叉树时是

(A B C) 写成二叉树时是



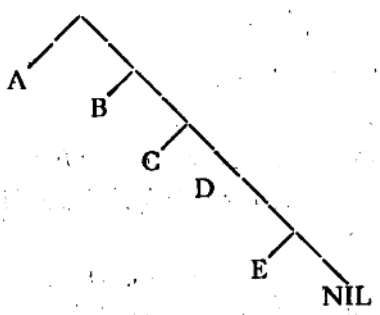
((A. B) . (C. D) 写成二叉树时是



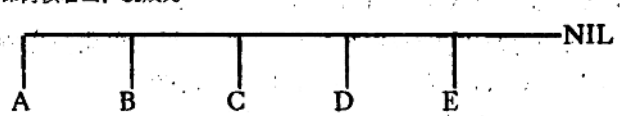
总的一个原则是，每个点都代表一分叉。由于 S- 表达式的构架是两个成份、一个点，所以表示出来的是两分叉的树，即二叉树。

注意表的二叉树表示，很有特点：

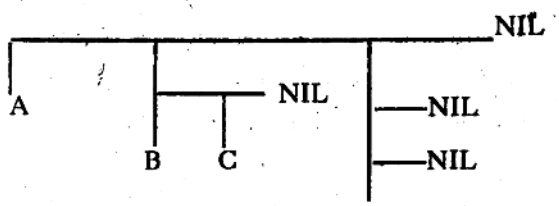
(A B C D E) 表示成



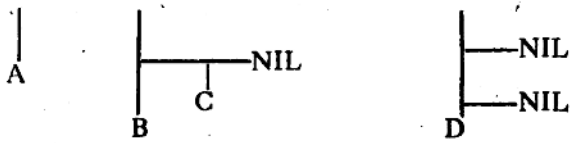
把这棵树横着画，就成为



再看表 (A (B C) ((D)))



这里，首层元素很清晰地表示出来：最上面一道横线领属的那些成份。上面那个例子中分别为：



就是原子 A、表 (B C) 、表 ((D))

原子的二叉树表示就是一个点，称为退化的二叉树。一般说来，我们需要二叉树的时候，都是非退化情形。

§4. S- 表达式上的基本操作

S- 表达上式能有什么样的结构操作，取决于 S- 表达式的结构特点，S- 表达式结构上的最大特点是它是递归规定的结构，这一点从 S- 表达式的定义中可以明显地看出来。因此，对 S- 表达式的操作（我们谈到操作时指结构操作）也相应地是一族递归函数。我们从函数概念谈起。

4.1 函数概念

通常对于加法我们说成是一个操作，经过操作 5+3 我们得到8。这件事还可以有另一种看法：由3与5对应到8。我们可以列一个这种对应的表：

3, 5 ⇒ 8
1, 1 ⇒ 2
1, 2 ⇒ 3
1, 3 ⇒ 4
.....

把这个表列全就把加法操作的所有可能性列出来了。当然这个表是无限的，不可能列举完全，数学上另有办法。我们这里强调的是这个例子里显示出来的“对应”概念。乘法的九九表也可以看成是用对应概念表达的乘法操作。

函数概念的核心就是“对应”。

提到一个函数 f，是指一个对应，以及两个集合：前一个集合的一个或几个元素对应着后一个集合的一个元素。

与函数概念有关的术语有自变量、值、定义域、值域、论域、完全函数、部分函数等。下面一一介绍。

自变量：前一个集合中的元素，有的函数要求一个自变量，即与后一个集合中每个元素

完成。这样的函数叫一元函数。有的函数要求多个自变量，即前一个集合里有多个元素元素对应后一个集合中的每一个元素。这样的函数视其要求自变量的数目分别为二元函数，三元函数，四元函数，等等，一元函数的例子有绝对值函数、开平方函数。二元函数的例子有加法函数、乘法函数。

值：后一个集合的元素。谈到一个函数的值时，指的是给定自变量的情况下，这个函数对应到的元素。函数的值必须是唯一的，同样的自变量只导致唯一一个值。

定义域：使函数有值的自变量范围。例如除法函数（一个二元函数），自变量〈2，0〉就不在定义域内，因为这样的自变量不对应到什么元素（2除以0是无定义的）。

值域：函数的值的范围。

论域：自变量的范围。或者说，论域界定了我们在什么范围内讨论一个函数。例如，我们可以在自然数的范围内谈加、减、乘、除，也可以在整数范围内谈，还可以在有理数范围内、实数范围内谈，不同的范围内效果不一样，（比如除法函与减法函数在不同的范围内定义域不同）。这范围的不同用术语讲就是论域的不同。

完全函数：定义域与论域相同的函数。

部分函数：定义域比论域小的函数。

注意，完全函数与部分函数这两个概念是相对而言的。同一个函数在不同的论域下会扮演不同的角色。

我们约定，当一函数的自变量取到定义域之外时，我们仍认为这时有个值，这个值记为"⊥"，念作 Bottom。它的意思是"无定义"。之所以把"⊥"当一个值看待，仅仅是为了便利。

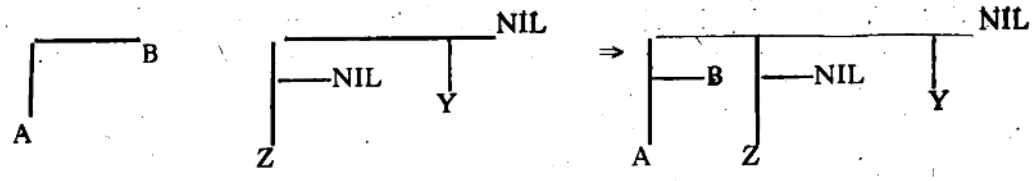
4.2 几个基本函数

把所有 S- 表达式的集合当作论域，那么值域也是 S-表达式的那些函数就实现了对 S-表达式的处理：把一个或几个 S-表达式处理成另一个 S-表达式。到底都有一些什么样的函数？回想 S- 表达式的二叉树表示，在其上的结构操作无非是取一棵子树，由两棵树合并造一棵新的树，判断两棵树的异同，判断是否退化。这几样操作是S-表达式上最基本的操作，它们对应着 S- 表达式上最基本的函数，我们一个一个来考察。

(i) cons 函数。名字来自 Constructure。这是个二元函数，在 S- 表达式论域上是个完全函数。它导致的对应是：两个 S- 表达式对应到这样一个 S- 表达式，其首部与尾部分别是那两个 S- 表达式（前一个自变量充当首部，后一个自变量充当尾部）。

从二叉树角度看，cons 函数导致的操作是把两棵树合并成一棵树。例如

cons [(A. B) ((Z) Y)] ; ⇒ ((A. B (Z) Y)



我们用希腊字母表示任意一个 S- 表达式，那么 cons 函数的定义可以表达为：

cons〔α，β〕⇒（α．β）

注意我们这里函数的书写方法：函数名字后跟自变量，自变量用方括号括起来，自变量之间用逗号隔开。一般数学书中也是这么书写函数，但不用方括号而用圆括号。我们这里用方括号是为了把这里讨论的函数与第二章要讲的 LISP 函数区分开。

看几个例子：

```
cons [A, B]          ⇨(A.B)
cons [A, NIL]          ⇨(A)
cons [A, (B)         ⇨(A  B)
cons [Z, (A  B)]          ⇨(Z  A  B)
cons [Y, (Z  A  B)]            ⇒(Y  Z  A  B)
```

细心的读者或许能发现 cons 函数对于表的作用。当 A 与 (B C) 通过 cons 得到 (A B C) 时，我们说把原子 A "并入" 表 (B C)。

Car 函数。名字来自 Contents of address part of register。念作〔ka：〕。

cdr 函数。名字来自 contents of decrement part of register 念作〔kulda〕。

这两个函数的名字由于历史的原因显得有些古怪。它俩都是部分函数，对于原子没有定义。car 将一个点对对应到其首部。cdr 将一个点对对应到其尾部，从二叉树角度看是取非退化树的左子树，cdr 是取右子树。

写成式子，car 与 cdr 的意思可以表达为。

car〔(α．β)〕⇒α

cdr〔(α．β)〕⇒β

若 Y 是原子，则：

car〔Y〕⇒⊥

cdr〔Y〕⇒T

看一些例子：

```
car [(A  B)]          ⇒  A
cdr [(A  B)]          ⇒ (B)
cdr [(A  B  C)]          ⇒(B  C)
car [(A  B)  C)]          ⇒ (A  B)
car [(A)]          ⇒ A
cdr [(A)]          ⇒ NIL
cdr [((A))]          ⇒ NIL
```

atom 函数。这是个一元全函数，功能是判断一个 S- 表达式是不是原子。atom 把一个 S- 表达式对应到原子 T 或原子 NIL，视这个 S- 表达式是否原子而定。atom 把原

子对应到 T，把非原子对应到 NIL。注意，这意味着 atom函数的值域是 {T, NIL}。从二叉树角度看，这个函数就是判断是否退化。

看一些例子：

```
atom [A]          ⇒ T
atom [UNIVERSITY1]        R  T
atom[(A. B)]          ⇒ T
atom[(A)]         ⇒ NIL
atom [NIL]        ⇒ T
atom [8]        ⇒ NIL
```

这种值域为 {T, NIL} 的函数我们也称作“谓词”。下一个基本函数也是一个谓词。

eP 函数。名字来自 equalty。它是个二元全函数，功能是判断两个原子是否相同。它把两个 S- 表达式对应到 T 或 NIL。当两个 S- 表达式都是原子且是同一个原子时，对应到 T，否则对应到 NIL。从二叉树角度看，eq 函数是判断两个退化的二叉树是否相同。

看一些例子：

```
eq [A,  A]          ⇒ T
eq [PHONE, PHONE]          ⇒ T
eq [(A.  B), (A.  B)]          NIL
eq [A.  B)]          ⇒ NIL
eq [A, (A)]          ⇒ NIL
eq [(A), (A)]          ⇒NIL
eq [5,5]          ⇒ NIL
eq [NIL,NIL]          ⇒ T
```

4.3 函数的复合

上面谈到的基本函数是些最基本的操作。由基本的操作可以构造较复杂的操作，即一些较复杂的函数。最简单的方法是几个操作的作用加起来。实现这一点是用函数的复合，也即由基本函数构造复合函数。

例如，car 与 cdr 复合。效果是先做 car 再做 cdr。这样，对 ((A. B) .C) 的操作结果就是 B，我们写成：

cdr〔car〔 ((A. B) .C) 〕〕⇒B

过程是car 得到结果 (A. B)，交给 cdr，得到 B。注意次序。

复合得到一个新的函数，我们可以为新函数取个名字，比如说 f。借助希腊字母（表示任意一个 S- 表达式）我们可以把函数的复合写成：

$$f[\alpha] \underset{af}{=} cdr[car[\alpha]]$$

或者干脆写成：

$$f \underset{df}{=} cdr \circ car$$

其中的“=”是用于构造新函数时使用的特殊符号，意思是“定义为”。另外，“o” 表示“复合”意思。

我们可以用复合的手段定义许多新的函数。看一些例子：

```
(1) f1=    conso (car, cdr)
        af
    f1[α]=    cons car [α], cdr [α]]
          af
于是：  f1[α]=α  (当 α 不是原子)
        f1 [(A  B)]=coms[car[(A  B)],
cdr[(A  B)]=cons[A, (B)]=(A  B)
(ii)  f2=    car · cons
        df
f2[A,B]=car[cons[α,β]]
  于是  f2 [&,  B]=&
        f2[(A), (B)]=car[cons[(A), (B)]]
=car[((A)  B)]=(A)
(iii)  f2=    atomo cons
         df
       f3 [α,  B]=    atom [cons[α,  B]]
       于是 f3[α,  B]=NIL
(iv)  f4=    atomoeg
        df
       f4[ &,  B]=    atom[eg [α,  B]]
             af
     于是  f4[α,B]= T
(v)  f5=    eg · (car, cdr)
       af
     f5[α]=    eg[car[α], cdr[α]]
   于是  f5[(B, B)]=eg[car[(B, B)]
   cdr[(B, B)]=eg[B, B)]=T
```

读者不难想象，类似的例子有很多很多。复合手段要想真正发挥威力，还需要有某种选择机制，能使我们在不同条件下做不同的事情。这是由函数 if提供给我们的。事实上，if 函数不但使我们能充分使用复合手段，也使其它的手段更有效力，在后面要讲到的递归手段中，if 函数是必不可少的。

if 函数很特别，它是个三元函数。当它的第一个自变量是 T 时，它的值是第二个自变量；当它的第一个自变量是 NIL 时，它的值是第三个自变量。也就是说，它的第一个自变量的定义域是 {T, NIL}，第二、三个自变量的定义域是 S- 表达式，因此 if 函数的值域也是 S- 表达式。由于 if 很特别，它是否是个全函数依赖于使用它的时候作什么规定。我们这里不再深究，读者不妨认为，它是个不完全函数。

通常 if 函数总是与其它函数复合着使用。看一个例子：

```
if[atom[A], cons[M, N], Car[M]]      ⇨(M, N)
```

由于 atom〔A〕得值 T，所以 if 的值是第二个自变量的值，它是 (M, N)。注意虽然第三个自变量无定义 (car〔M〕取值 ⊥)，if 仍然有值。再看一个例子：

```
if[cons[A,  B],  cdr[(A,  B)],  car[(A,  B)]]      ⇨⊥
```

这种情形很少发生，通常使用 if 时，其第一个自变量总是复合一个谓词的。

到现在，我们已经有能力用函数写出一条规则了。我们看一个例子。前面提到过“并入”，这是一种说法，意思是用 cons 函数作用一下。我们的规则要说的是：“对于一个 S- 表达式 α，如果它的首部是 UNIVERSITY（大学），就取出它的尾部。否则把 UNIVERSLTY并入α使之成为其首部。”用函数写出这条规则就是：

```
if[eq[Car[α], UNIUERSITY]
  cdr[α],
  cons[uNIVERSITY, α]]
```

我们就是这样进行符号处理的。

比复合更复杂也更有力的构造新函数的手段是递归。事实上，由于 S- 表达式结构上的特点，递归是构造 S- 表达式上函数的主要手段。同样道理，递归机制是 LISP 语言的特点所在，也是 LISP 语言发挥效力的关键。下一节里我们主要就是读用借归手段构造新函数。

§5. S- 表达式上的函数

5.1 函数的递归定义

我们来看看阶乘运算，直观上，阶乘的意思是：

$n_!=1\cdot 2\cdot 3\cdot 4\cdots\cdots n$

例如

$3_!=1\cdot 2\cdot 3=6$

$5_!=1\cdot 2\cdot 3\cdot 4\cdot 5=120$

象加法与乘法等运算一样，阶乘运算也能理解为一个函数，整数上的一元全函数。作为函数其定义可以写成：

```
factorial[n]=      ⎧          1          当 n=0
           df      ⎩ n.factorial[n - 1]     当 n>0
```

如果把乘法函数写成“*”，减法函数写成“—”，那么可以把定义阶乘函数 factorial 的右端写成如下的复合形式：

```
if[eq[n, 0], 1, *[n, factorial [ - n, 1]]]
```

但是这个复合形式中又出现了 factorial，只是接受的自变量不同了。关键之处就在于接受的自变量不同了。当我们写

```
factorial[n]=    if[eq[n, 0], 1,
           df
                  *[n, factorial[ - [n, 1]]]]
```