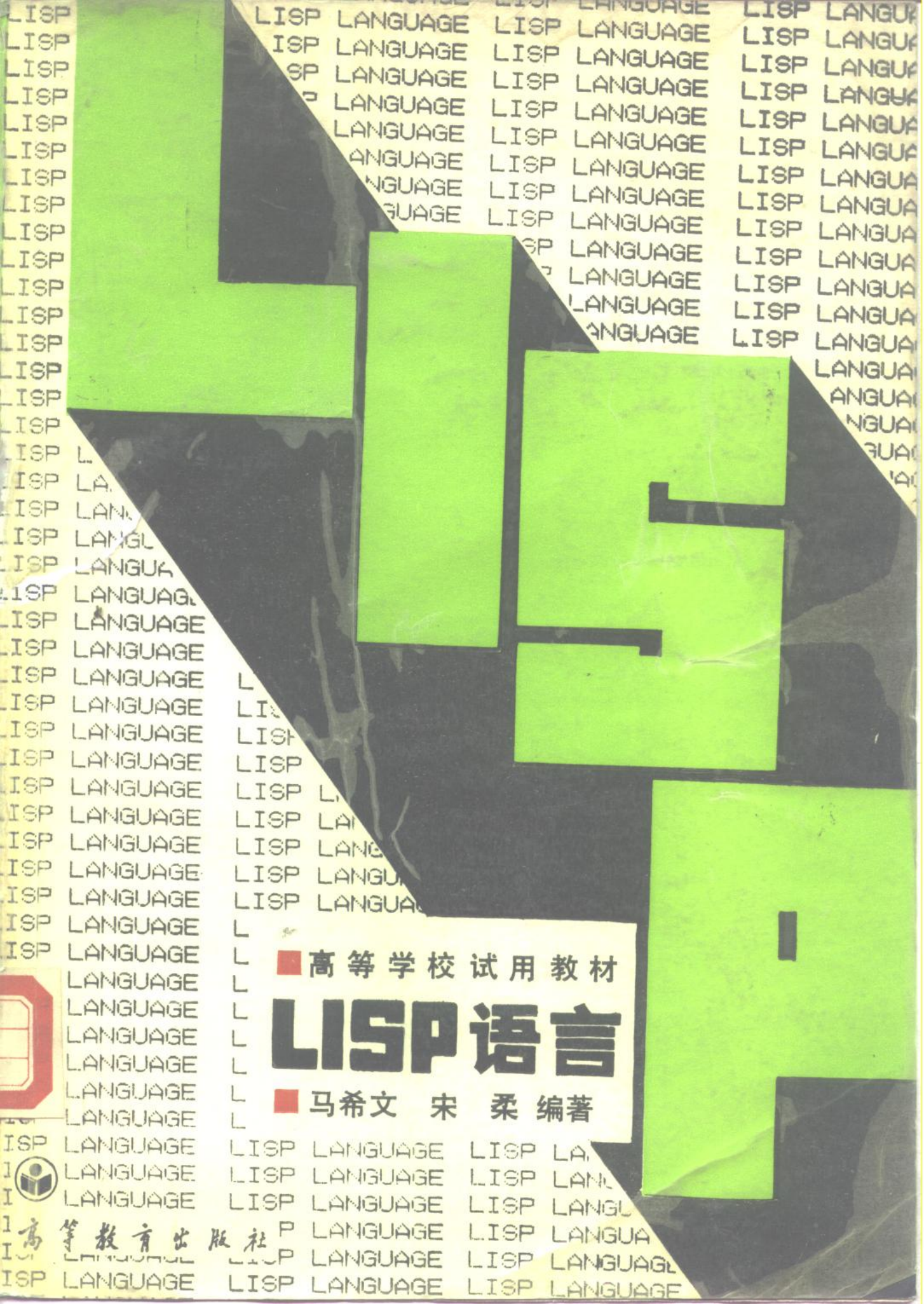

高等学校试用教材
LISP语言
马希文 宋 柔 编著
高等教育出版社

高等学校试用教材

LISP 语 言

马希文 宋 柔 编著

高等教育出版社

内 容 提 要

本书共十四章．一至三章介绍 LISP 的基本语法、语义，S 表达式的概念，以及 LISP 的核心部分 LISP 1；四至十章主要讨论 LISP 的函数程序设计方法；十一至十三章是 LISP 的非函数程序设计部分；十四章介绍 LISP 中的一些高级成分和高级应用．

本书以 LISP 语言为素材，注重训练学生的函数式程序设计的能力．书中用形式化的方法来表述 LISP 的语义，旨在提高学生的理论素质．同时，书中有大量的例题和习题，涉及到 LISP 的基本概念直至人工智能领域中的高级应用．仔细阅读这些程序并认真作习题，对学好本课程是十分有益的．

本书可用作高等院校计算机专业的教材或参考书，也可供从事人工智能工作的研究人员、工程技术人员参考．

高等学校试用教材

LISP 语言

马希文　宋柔　编著

*

高等教育出版社出版

新华书店北京发行所发行

北京印刷一厂印装

*

开本 787×1092　1/16　印张 20.25　字数 490 000

1990 年 7 月第1版　1990 年 7 月第1次印刷

印数 0001—2 100

ISBN7-04-002261-3/TP · 50

定价　4.40　元

前　言

本书是供计算机系科学生学习LISP语言的教材. 从1983年开始，作者先后在中国科学技术大学研究生院和北京计算机学院讲授LISP语言. 由于LISP语言是不断发展变化的，LISP语言的实现和应用也不断发展变化，因此我们的教学内容和所用的讲义也在不断地修改和更新. 现在对LISP教材的要求越来越迫切，在国家教委计算机软件教材编审委员会的鼓励下，我们在原讲义的基础之上，经过修改和补充，编写了这本LISP语言教材.

中国科学技术大学研究生院和北京计算机学院的有关师生对本课程的讲授提供了支持和帮助. 北京计算机学院王鑫使用本书原稿讲课时指出了其中的若干疏漏. 吉林大学庞云阶副教授审阅了全书. 对于以上各位，以及为本教材的成书提供帮助的其他所有同志，作者表示深切感谢.

LISP的理论和应用还在发展，因此教材也应不断发展. 本书内容虽然在教学中几度改进，但仍难免有各种问题，希望读者能把有关的意见和建议反馈给我们，这对我们今后的工作将大有裨益.

作者谨识

1987年11月于北京

目　录

绪　言

(一)

LISP 是 LISt Processing 的缩写，是“表处理”的意思，从历史上看，LISP 语言最初是 1960 年美国的 John McCarthy 提出来的．当时计算机语言刚刚兴起，FORTRAN, ALGOL 等语言也刚处在婴儿阶段．因此 LISP 语言可以称得上最早的计算机语言之一．

FORTRAN, ALGOL 这类语言都是数值计算语言，它是把用机器指令 (或汇编)编写数值计算程序时遇到的常用的数据结构和程序结构抽象为一些形式记法，使程序员得以摆脱程序设计中最繁琐的细节，从而提高工作效率．对于这些程序员来说，字符处理主要是用在自编格式输出的地方，在 FORTRAN 中则设计了专门的格式语句来处理这类问题．但格式语句所能做的事，当然不是一般的符号处理．

当时机器翻译和定理证明的研究已经起步，对符号处理语言的需求已经存在．LISP 语言把自己的处理对象规定为符号表达式，整数或实数则是符号表达式中极特殊的一个很小的子集．这个语言的出现，大大方便了要做符号处理的程序员，引起了大家的重视．于是，到了 1962 年，LISP 便作为一个实用系统出现了，这就是 LISP 1.5．

二十多年过去了，数百种算法语言不断地被提出，实现，绝大多数又都无声无息地消失了，或者只在一个很特殊的范围内被人使用．各种较早的语言，除了 FORTRAN 靠着强大的经济力量维持到今天外，只有 LISP 语言正得以广泛的使用，经久不衰．现在，在非数值计算的领域，特别是人工智能领域中，LISP 语言占有极重要的地位．

(二)

计算机语言的理论和实践是不断发展的．发展到一定阶段时，人们对语言有了一些新的认识，这样便需要对已有的语言进行修订、扩充．如果某种语言的基本框架容不下这种修订或扩充，这种语言就要被淘汰．然而 LISP 语言是以计算机的基础理论——递归函数论为背景的，所以它能顺利地通过所有阶段，得以扩充和完善．

例如数据结构．在 FORTRAN 语言中，结构化的数据只有一种即数组，亦即数的阵列，而且二维的数组并不是一维数组的阵列，也就是说不是递归定义的．因此数据结构的方式太贫乏，僵死．这种情况从计算能力的角度来说并不是很大的缺陷，因为各种结构化的数据归根结底总可以映象为数组而无需过多的计算开销 (从理论上说，只有自然数就足够了．但把结构化的数据映象为自然数，要用到 Gödel 编码，虽然理论上是可能的，但计算开销太大，实际上完全行不通)．但是程序的实践说明，这种映象工作不应交给程序员来做，而应由机器自动完成．于是出现了 PASCAL 这类具有递归数据结构的语言．

LISP 的数据结构虽然单纯，只有符号表达式这一种，但它适用于一切形式系统，因此任何结构化的数据都只是其特例．使用这种符号表达式进行计算时，数据的结构和规模可以不断改变，给程序员提供了极大的方便．

又例如，七十年代人们谈论软件危机．实质上是因为程序越来越大，越来越复杂，以致程序员很难凭朴素的智力来驾驭自己的程序．人们提出了许多解决方案．其中办法之一就是使程序（如何做）与其功能（做什么）一致起来．这导致了函数式程序设计和函数式语言的兴起．意味深长的是，LISP 语言早就作为一种函数式语言而存在多年了．后起的许多函数式语言，由于没有根本跳出 LISP 的框架，所以很少能取得社会地位．

在人工智能领域中，人们逐渐发现，由程序生成一些程序并在适当的时候执行这些程序是十分重要的．这种情况有时可以说成是“数据驱动”．如果用 PASCAL 或类似的语言来实现这一点，程序员实际上就得为此设计一种语言并写出它的解释程序．LISP 语言在这个问题上有着先天的优点．它具有自己解释自己的能力．当程序员需要的时候，他很容易写出一段程序来编制或改造另一段程序，而且随时又可以执行加工出来的程序．甚至一段程序还可以在运行过程中自己修改自己．这本是计算机固有的能力，但大多数语言都不是从计算的理论模型出发设计的，所以就损伤了这种能力．而 LISP 语言本身就是一个计算模型，其数据的形式和程序的形式是完全一致的，都是符号表达式，因此计算机的这种固有能力就充分表现出来了．在实用系统中，LISP 语言的支持环境，如编辑、排误等，都是用 LISP 实现的，而且可以嵌入在用户程序之中，作为一个标准函数来引用，使这些工具可以更加灵活地得到应用．

LISP 的这些优点，来自于它的理论的简单性与透彻性．因此，又可以以 LISP 为工具（或表达形式）展开对计算理论的研究，并把所得到的结果应用于程序生成，程序验证（或计算逻辑）的研究中去．事实上，这方面的研究有许多是针对 LISP 语言进行的，大多数又以 LISP 语言为工具．

（三）

一个语言有这样的先天优点，使它不可避免地具有早熟的性质．这也使它面临许多困难．这种困难归结到一点就是与现代计算机体系结构的矛盾．

现代计算机的内存是线性地组织起来的．要实现符号表达式，就要大量地使用指针，过多地消耗存储空间，并且影响运行效率．

现代计算机的控制机制是顺序式的，要实现 LISP 的递归式机制就要组织运行栈，这也会消耗存储空间，影响运行效率．

现代计算机的指令系统主要是为数值计算设计的，其大多数指令对于 LISP 的实现来说是没有价值的．而 LISP 所要求的结构转换，只好以指针的转递和重复间接取址来实现，这显然又不能互相配合．

为了解决这些问题，LISP 语言不得不向机器让步．LISP 1.5 增加的赋值、转向、顺序控制等机制，就是为了这个目的而设计的．

到了七十年代，微电子技术有了新的发展，于是人们开始设计专门的 LISP 机器来改善这种情况．这方面的工作正在发展之中，但是如何跳出顺序计算的巢臼，则并不是一个容易解

决的问题．看来，只有 LISP 机器得到充分的发展之后，LISP 语言早熟的弱点才能被消除，那时，LISP 语言才成为一个真正成熟的语言．

(四)

我们说 LISP 语言不是一个真正成熟的语言，在实践上主要是指实用软件很少有用 LISP 语言来编写的．常见的情况是，一个新的软件在实验室阶段用 LISP 语言写，以便利用 LISP 语言较灵活的优点加快研制过程；但到了实用阶段，一切都定型了，就改用别的语言重写一遍，以求软件的时空效率．

由于有这种情况，在程序的可移植性方面没有向 LISP 语言提出过认真的要求．加上 LISP 语言很容易由用户来扩充，所以到今天为止 LISP 不但未能标准化，而且标准化的呼声亦不强烈．这就不难想象，LISP 语言象汉语一样，有为数极多的方言．一个惯用某种方言的程序员可以完全看不懂另一个程序员用另一方言写的程序．当然，语言的核心部分是一致的，但是任何实际的程序都不可能只用核心部分写就，这就给学习 LISP 的人带来了一些困难．

(五)

面对这种方言混乱的情况，教学工作怎么办？教材应该怎样写？我们当然只能选用一种方言．在必要的时候，把其它方言的异同介绍一下．学生在实习时，必须在教师的指导下，了解所用的 LISP 方言的特点．在这方面，查阅文本是必要的，更重要的是上机试验．现行的 LISP 系统几乎都是对话式的，只要给学生提供充分的机时，他们就有机会对照文本、通过试验来了解所用的 LISP 系统的特点．这种通过试验来学习的能力是今后使用不同的 LISP 方言所必须的．

本书所依据的方言，是作者开发的 DCLISP．开发这个方言的目的之一正是为了教学．这个方言不是象 INTERLISP 那样有丰富的系统函数和高级的支撑环境，在这一点上这个方言可以说是朴素的．但这个方言有较多的机制，使学生可以学到更多的灵活使用 LISP 的方法．其中值得特别提到的是：

(1) 动态编译机制．多数 LISP 方言都有编译运行和解释执行两种方式，但由于环境中有矛盾，两者不能动态地改换．DCLLSP 克服了这个弱点，把两者的环境统一起来，使得程序进行中可以随时在两种方式之间变来变去，以求灵活性和效率更加协调．

(2) 可控制的约束机制．LISP 系统实现递归调用时有一套参数值约束机制，这在理论上是必须的，但在多数实际程序中，这种做法实际上是一种累赘．DCLISP 提供了一种手段使程序员可以干预这种约束机制，以提高时空效率．

(3) 按模式传递参数的机制．一般 LISP 方言传递参数都是把形参表与实参值的表顺次搭配来进行的．在 DCLISP 中，这种机制被扩大为把形参的模式与实参的值相匹配．这就是本书第四章讲的形参表达式以及第十四章讲的 PEXPR 表达式和 PLET 表达式的功能．使用这种方法既可以提高程序的可读性，又可以提高效率．

这些机制都是作者提出的，目前在其它方言中很少见到．

DCLISP 还提供了比较丰富的高级函数，其中有一些也是其它方言中少见的. 这使 DCLISP 具有某些泛函式语言的优点. 用这些高级函数写出的程序是结构性好，效率高，正确性比较有保障.

为了配合本书的教学，在有条件的情况下，直接采用 DCLISP 作为教学工具当然是有好处的.

(六)

现在许多学校的计算机系科都要讲授 LISP 语言. 学生学习 LISP 语言的目的何在呢?

当然，首先要把 LISP 语言当作非数值计算，特别是作为表结构变换的工具来学习. 另一方面，通过例题和习题，可以学到如何把朴素问题形式化的种种技巧. 但是，从作者的观点看，更重要的是应把它作为函数式语言来学习.

前面曾谈到函数式程序设计和函数式语言. 所谓函数式语言，就是用函数定义作为程序的说明部分，把函数计算作为程序的执行部分. 因此函数式语言实质上就是数学语言. 这样，用函数式语言所写的程序就便于程序员掌握了. 例如，令

$$f(x,y,z)=\begin{cases} z, & \text{当 } y=0 \\ f(x^2,k,zx), & \text{当 } y=2k+1,k\geqslant 0 \\ f(x^2,k,z), & \text{当 } y=2k,\quad k>0 \end{cases}$$

那么，用归纳法很容易证明:

$$f(x,y,z)=zx^y$$

于是计算 3^5 只用求 f(3, 5, 1) 就行了. 所以我们可以写出

```
def  f(x, y, z) =
          (y = 0) → z;
          odd (y) → f(x², y/2, xz);
                    f(x², y/2, z)
f(3, 5, 1)
```

这差不多就是某个函数式的程序了.

同样的程序用 PASCAL 语言来写，当然也可以使用函数的递归计算的机制，但更自然的办法是使用如下的含有循环语句的函数来编写程序:

```
FUNCTION  F (X, Y: INTEGER): INTEGER;
VAR       Z : INTEGER
BEGIN
      Z := 1;
      WHILE  Y > 0  DO
      BEGIN
      IF  ODD (Y) THEN  Z := Z * X;
      Y := Y  DIV  2;
```

```
            X: =X* X
            END;
        F: =Z
    END;
```

这个写法和数学公式之间有相当的差距.

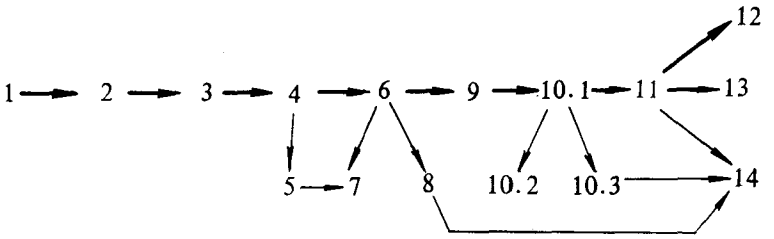
比较上面两个程序可以看出，函数式的程序较少(或较间接)地涉及计算顺序的细节，较多(或较直接)地涉及计算的目的. 这种特点在LISP语言中充分地表现了出来. LISP语言和PASCAL语言在这一方面的区别是很深刻的，程序员使用LISP语言编制程序时的思考方式与使用PASCAL语言也不相同. 因此，学习LISP语言的目的之一是要习惯这种思考方式. 这是一个技术全面的程序员所应有的能力. 从某种意义上说，即使要编一个PASCAL程序，先写出一个函数式的程序作为过渡也是有好处的.

因此，本书在内容安排上尽量突出LISP作为函数语言的特点，多用一些篇幅举例，多给学生一些练习机会，LISP语言中的非函数成分(如PROG, GO, SET等)则放在后面，讲得也比较简单. 这些成分对于学过PASCAL等语言的学生不会造成困难.

函数语言在教学上的一个方便之处就是容易作到严谨性和技巧性并重. 一般的程序语言教学总是把主要精力放在如何充分利用程序语言的各种机制方面. 这就使程序语言课程带有技术性的色彩. 有关程序语言的严谨理论则很难放在初级课程中. 但函数语言的理论机制比较简单，很容易在教学中兼顾数学上的严谨性. 本书很重视这一方面的选材和讲解. 学生在学习时如果认真对待这些内容. 对提高自己的理论素质可能会有裨益，对于今后涉足于计算机科学的理论领域更有好处.

(七)

本书共分十四章. 一至三章介绍LISP的基本的语法、语义，S表达式的概念，以及LISP的核心部分LISP1; 四至十章主要讨论LISP的函数程序设计方法; 十一至十三章是非函数程序设计部分; 十四章介绍LISP中的一些高级成分和高级应用(实现模式传递参数机制，实现逻辑型语言和泛函型语言). 各章节之间的联系如下图所示:



图中，用粗箭头连接起来的是主干部分，其余则是应用实例.

主干部分主要介绍LISP的各种基本成分，是本课程的基础性内容. 但其中也有一些比较深入的内容，如第三章§2全节，§3和§4的后半部分，第六章§2,以及第十章§1的后半部分. 略去这些内容或把它们往后安排，不会影响其余部分的教学. 此外，第十二章§3和§5主

要是应用实例.

程序设计方法需要在实践中学习，因此本书中安排了较多的应用实例. 第五章和第七章中的例子的规模是比较小的，其余一些应用实例的规模和难度比较大. 教师可根据实际可能，选一些作课上教材，其它一些留作课外阅读材料.

本书每章后面都附有若干习题. 打 * 号的习题是难度较大或程序量较大的. 仔细研读例题之后认真地做好习题，对学好这门课程是十分必要的.

第一章　LISP——函数型语言

§1　前缀表达式

LISP 语言 (指其核心部分) 是一种函数型语言. 它与传统的程序设计语言有许多不同之处. 最明显的就是: 在 LISP 语言中, 程序是由表达式组成的, 而不是由语句组成的. 我们把做为程序的表达式叫做程序表达式. 在 LISP 系统中, 计算的过程是靠对程序表达式的求值来实现的.

从具体形式来看, LISP 的程序表达式是一种前缀表达式.

在数学中, 四则运算符 "+", "−", "*", "/"写在两个运算项中间, 例如:

$$3+4 \qquad 5*2$$

而在 LISP 中, 要求把运算符写在运算项前面, 中间留下必要的空格, 并把它们都放在一个括号中. 因此, 上面两个算术表达式在 LISP 中应写成

(+ 3 4)　(* 5 2)

这就是前缀表达式. 如果打开 LISP 系统, 键入以上两个表达式之一, 系统就开始进行计算, 最后显示出计算的结果 7 或 10, 就是相应的表达式的值.

一个前缀表达式还可以充当更复杂的表达式的运算项. 例如 4−3*2 可以表达为

(− 4 (* 3 2))

这里 (* 3 2) 就是减法的第二个运算项. 同样, 4−3+2 则应写成

(+ (− 4 3) 2)

即把 (− 4 3) 的值当做加法的第一个运算项.

在许多 LISP 系统中, 不能直接使用 "+", "−", "*", "/" 做为运算符, 必须使用 "PLUS", "DIFFERENCE", "TIMES", "QUOTIENT". 此外应注意, 一般在 LISP 语言中只做整数运算, 因此 (QUOTIENT 15 4) 的值是 3, 而不是 3.75. 此外, 如果要计算除法的余数, 可以用 "MOD". 例如, (MOD 15 7) 的值是 1.

实际上, 以上这些运算在 LISP 语言中被看做是二元函数. 按照前缀表达式的写法, 应把函数名称写在自变量的前面, 中间留下必要的空格, 并把它们放在一个括号中. 例如, "SQUARE" 表示平方函数, 因此 (SQUARE 5) 的值就是 25; "ABS" 是绝对值函数, (ABS −41) 的值是 41.

按照这种看法, 如果在一个程序表达式中, 某些项目本身又是用程序表达式表示的, 就相当于数学中的复合函数了. 比如:

(H 2 (G 3))

就是数学中的 H (2, G (3)).

总之, 一个程序表达式就是一个 LISP 程序. LISP 程序的执行过程, 就是表达式的求值

过程，通常称做对表达式的求值.

§2 函数定义

LISP 语言提供一些函数可供用户直接使用. 这些函数称为系统函数 (又称 SUBR 型函数)，如 PLUS, DIFFERENCE, TIMES, QUOTIENT, MOD 等都是大多数 LISP 系统向用户提供的系统函数. ABS, SQUARE 等函数在有些 LISP 系统中规定为系统函数，有些 LISP 系统中则没有这些系统函数，用户需要时，得自己写出它们的定义.

例如要定义平方函数 SQUARE, 可以写

(DE SQUARE (X) (TIMES X X))

这是一个以 DE 开头的表达式，叫做定义表达式或 DE 表达式. 其中的第二项 SQUARE 是要定义的函数的名称，第三项 (X) 是这个函数的形式参数表 (简称形参表)，这个表中只有一个形式参数 (简称形参)，就是 X. 第四项 (TIMES X X)叫做函数的定义体，它说明这个函数的功能就是把自变量的值乘上自己，函数值就是自乘的结果. 这样，计算

(SQUARE 5)

就相当于在 X 取 5 为值的情况下，计算

(TIMES X X)

也就是计算

(TIMES 5 5)

的值，结果是 25.

函数的定义体也应该是一个表达式，与上节所说的程序表达式相比，这种表达式在应该出现常数的地方出现了做为变量的形式参数. 这种含有形式参数的表达式也叫做程序表达式. 极而言之，一个常数或一个形式参数也算做一个程序表达式，例如:

(DE SELF (X) X)

定义了一个函数 SELF，它的功能就是把自变量的值直接当作函数值. 例如 (SELF 6)的值就是 6.

下面的表达式

(DE SUM -OF-SQUARE (X Y) (PLUS (SQUARE X) (SQUARE Y)))

定义了函数 SUM -OF-SQUARE. 这个函数的形参表中有两个形参 X, Y. 定义体是

(PLUS (SQUARE X) (SQUARE Y))

其中又含有上面定义的函数 SQUARE,

这样，要计算表达式

(SUM-OF-SQUARE 3 4)

就相当于计算

(PLUS (SQUARE 3) (SQUARE 4))

这就又相当于计算

(PLUS (TIMES 3 3) (TIMES 4 4))

先得到

(PLUS 9 16)

再得到最后的结果:

25

引用一个函数的定义来计算一个表达式的值，叫做对这个函数的一次调用. 在调用一个函数时，每个形参都得到一个值，叫做它的约束值. 在上面的计算过程中，共有三次调用: 一次调用了SUM－OF－SQUARE,两次调用了SQUARE.后者是在计算SUM－OF－SQUARE的定义体时发生的,这种情况常常说成是"SUM－OF－SQUARE调用SQUARE".

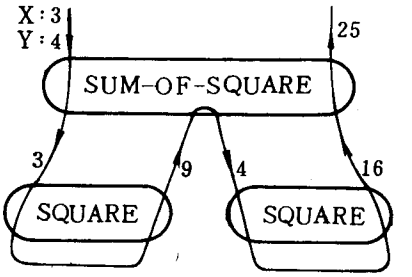


图 1－1

我们可以用一种调用关系图 (图 1－1) 表示这些关系:

在这个图上，分别标出了每次调用的函数名称，相应的形参约束值 (入口值) 和函数的值 (返回值). SQUARE 函数只有一个形参 X,在图上就不注明了，SUM－OF－SQUARE 有两个形参 X 和 Y,图上分别注明了它们的约束值 3 和 4. 把图上的箭头联接成一条曲线，表明调用的顺序.

§3 条件表达式

我们常常遇到分情况定义的函数. 例如绝对值函数可以定义为

$$|x|=\begin{cases} x, & \text{当 } x>0 \\ 0, & \text{当 } x=0 \\ -x, & \text{当 } x<0 \end{cases}$$

要用LISP 写出这个函数的定义，首先就遇到"x>0"，"x=0"，"x<0"应如何表达的问题. 在 LISP 中，这些也用前缀表达式的办法来写. 用"GREATERP"，"EQN"，"LESSP"分别表示"大于"、"等于"、"小于"，那么上面这三个关系应分别写成 (GREATERP X 0)，(EQN X 0)，(LESSP X 0).

GREATERP, EQN 和 LESSP 都是二元函数. 它们的函数值只有"真"、"假"两种值. 这种只取真假为值的函数通常叫做谓词. 在LISP 中，"真"记为 T,"假"记为 NIL. 因此,

```
(GREATERP 5 4)
(EQN 3 2)
(LESSP 5 5)
```

的值分别是 T, NIL 和 NIL. 这些也都是程序表达式.

绝对值函数的定义写作:

```
(DE  ABS  (X)
  (COND  ((GREATERP X 0) X)
         ((EQN X 0) 0)
         ((LESSP X 0) (DIFFERENCE 0 X))))
```

这里，作为定义体的程序表达式第一项是COND，这样的表达式叫做条件表达式或COND表达式．这也是一种程序表达式，但它的形式以及求值的步骤都有特殊的约定．

条件表达式的一般形式是

(COND 子句$_1$ … 子句$_n$)

每个子句又由两部分组成，其一般形式是

(检验条件 动作表达式)

而检验条件和动作表达式都是程序表达式．在上面的例子中，第一个子句的检验条件是(GREATERP X 0)，动作表达式是X，第二个子句的检验条件是(EQN X 0)，动作表达式是0，第三个子句的检验条件是(LESSP X 0)，动作表达式是(DIFFERENCE 0 X)．

对条件表达式求值的时候，顺次计算各子句中的检验条件的值，直到遇到一个值不是NIL的检验条件为止，然后，转而计算相应子句中的动作表达式的值，并把它的值做为条件表达式的值．如果每一个检验条件的值都是NIL，那么就规定条件表达式的值为NIL．(将来可以看到这种规定的好处．)

在上面的例子中，如果X的约束值是3，(就是说要计算(ABS 3)的值，)那么第一个子句的检验条件的值就是T，于是计算第一个子句的动作表达式的值，得到3，这就是条件表达式的值．如果X的约束值是 −5，那么第一、第二个子句的检验条件的值都是NIL，而第三个子句的检验条件的值是T，所以条件表达式的值就是这个子句中的动作表达式的值，即(DIFFERENCE 0 −5)的值，就是5．

这种做法可以更形式地叙述如下：

条件表达式的值是：

(1) 如果其中没有子句，则值为NIL．

(2) 如果第一个子句的检验条件的值不是NIL，则条件表达式的值就是第一个子句的动作表达式的值．(其余子句不予处置．)

(3) 以上两条都不满足时，条件表达式的值等于从中删去第一个子句后得到的新的条件表达式的值．

按照这个规定，要计算(ABS −5)的值，相当于求下面的条件表达式的值：

```
(COND ((GREATERP −5 0) −5)
      ((EQN −5 0) 0)
      ((LESSP −5 0) (DIFFERENCE 0 −5)))
```

这里第一个子句的检验条件值是NIL，按规定，这个表达式的值等于下面的条件表达式的值：

```
(COND ((EQN −5 0) 0)
      ((LESSP −5 0) (DIFFERENCE 0 −5)))
```

这时，第一个子句的检验条件(EQN −5 0)的值又是NIL，于是又要对下面的条件表达式求值：

```
(COND ((LESSP −5 0) (DIFFERENCE 0 −5)))
```

这一次第一个子句的检验条件(LESSP −5 0)的值是T，于是所求的值就等于下面的表达式的值：

```
(DIFFERENCE 0 −5)
```

结果是 5.

条件表达式各子句中的检验条件不一定非使用谓词不可. 任何程序表达式都可以用作检验条件. 我们规定只要检验条件的值不是 NIL，就都按同样的办法处理，这实际上就是把除了 NIL 以外的任何值 (例如 0 或别的数) 都看做检验的结果为真. T 和 NIL 都是常量. 有时要用 T 做为检验条件. 如果它前面的检验条件值都是 NIL，就会检验这个表达式，其结果当然为真，于是它后面的动作表达式的值就是条件表达式的值. 因此，用 T 作为检验条件，其效果相当于数学定义中的“否则的话 …”或者“在其它情况下 …”. 例如绝对值的定义也可以写成

$$|x|=\begin{cases}0 \quad -x, & \text{当 } x<0\\ x, & \text{否则}\end{cases}$$

那么，相应的LISP 定义就应写成

```
(DE  ABS  (X)
    (COND  ((LESSP  X  0)(DIFFERENCE  0  X))
           (T  X)))
```

值得注意的是，最后一个子句中的检验条件也可以写 0, 1 甚至 X. 但上面的写法最自然.

§4 递归定义的函数

阶乘函数可以定义为:

$$g(n)=\begin{cases}1, & \text{当 } n=0\\ n*g(n-1), & \text{否则}\end{cases}$$

用 LISP 语言则写成:

```
(DE  FACTORIAL  (N)
    (COND  ((ZEROP  N) 1)
           (T  (* N  (FACTORIAL  (SUB1  N))))))
```

其中 (ZEROP N) 相当于 (EQN N 0), (SUB1 N) 相当于 (DIFFE)RENCE N 1).

注意，这个定义的定义体内含有被定义的函数. 从数学的观点来看，这种定义是一种隐式定义，因为它并未明确地说明阶乘到底是什么，(甚至没有说明其定义域限于自然数,) 只是把阶乘定义为一个方程的解. 于是就发生了存在唯一性的问题. 研究这类问题超过了本书的范围.

但是从计算的角度来看，这种定义确实提供了一种计算的办法. 例如要计算 (FACTORIAL 4). 从定义,它应等于

```
(COND  ((ZEROP  4) 1)
       (T  (*  4  (FACTORIAL  (SUB1  4)))))
```

根据条件表达式的求值规则，它的值就是

```
(*  4  (FACTORIAL  (SUB1  4)))
```

而 (SUB1 4) 的值是 3, 因此，所求的值应该是如下的表达式的值:

```
(*  4  (FACTORIAL  3))
```

要计算这个表达式的值，又要求出 (FACTORIAL 3) 的值，再一次调用函数 FACTORIAL. 但这并不是无为地回到开始，因为这一次调用时，形参的约束值改变了.