CROMEMCO 微型计算机

软件资料汇编

清华大学计算中心译编

六

清华大学出版社

CROMEMCO 微型计算机

软件资料汇编

清华大学计算中心译编

（六）

清华大学出版社

1983

内 容 简 介

本书为“CROMEMCO 微型计算机软件资料汇编”的第六册。根据 CROMEMCO 公司提供的软件资料，本册选编了如下内容：Cromemco LISP 语言指令手册；Crememco C 语言参考手册；Cromemco Overlay Linker（覆盖连接程序）指令手册；Cromemco 系统诊断软件参考手册。

本书主要对象是该系统以及使用 M—5、M—8 系统的用户和程序员以及从事计算机软、硬件的工程技术人员和教学人员。

CROMEMCO 微型计算机

软件资料汇编（六）

清华大学计算[illegible]译

清华大学出版社出版
北京　清华园
清华大学印刷厂印刷
新华书店北京发行所发行，全国各地书店经售
☆
开本：787×1092　1/16　印张：16　字数：379 千字
1983 年 11 月第一版　1983 年 11 月第一次印刷
印数：1～32000
书号：15235.81　　定价：2.30 元

前　言

随着 Cromemco 微型计算机使用的不断推广，现有的“CROMEMCO 微型计算机软件资料汇编”已满足不了广大读者的需要，为此我们又选编了第六册。本册为用户又提供了几个新的软件工具。

LISP 是 LISt Processing 的缩写。它是一种高级的程序设计语言，在人工智能程序设计中得到越来越广泛的应用。本手册是为满足不同程度的读者需要而编写的，它概述了 LISP 语言的一般概念，并介绍了 Cromemco LISP 与其它 LISP 的不同点、各种函数及其使用方法。对一般用户，利用该手册就足以去使用 Cromemco LISP 软件系统，但对初学者来说，为了更有效地使用 LISP，请参考《人工智能程序设计》(Artificial Intelligencl Programmig)。人工智能程序设计包含了一些具体的应用，这将帮助读者增进对 LISP 的理解。

C 语言是一种很强的通用程序设计语言，获得成功的 Unix 操作系统以及与之有关的绝大部分软件都是用 C 语言写成的。本册主要介绍的是 Cromemco C 语言的特点和使用方法，以及它和参考语言的差别。关于参考语言，在 Brian W. Kernighar 和 Dennis M. Ritchie 写的“The C Programming Language”中已完整叙述，对 C 语言不熟悉的人可参考这本书。

覆盖连接程序 Overlay Linker 指导手册主要叙述了 Cromemco 复盖连接程序的使用方法。利用复盖连接的方法允许解释和执行那些比实际内存大的程序，这些程序可以是用 FORTRAN 语言、汇编语言或 C 语言写的。程序员首先按模块写出这样大的程序，然后用 OVRLINK 把模块建成一个复盖的树形连接结构。该程序的执行是靠装入到内存中的模块，这就大大提高了 Cromemco 计算机处理程序的能力。

Cromemco 系统软件主要叙述了软磁盘、硬磁盘及内存的测试诊断程序和使用方法，这给硬件维护工作带来了极大的方便。

本册是清华大学计算中心部分同志翻译和编写的。由于本册选编的资料比较新，很多方面我们都没实践过，错误在所难免，请读者批评指正。参加翻译和编写的有：王耆、汪原仁、孙宏昌、徐时新、有悦、王素明、范小安、范文建、俞军、林霞、林鄂华等。其中 LISP 指令手册由清华大学计算机科学与工程系控制与应用教研组陆玉昌审校，C 语言由软件教研组洪先龙审校、覆盖连接程序指令手册和系统诊断软件参考手册由吴企渊审校，全书由林鄂华同志统稿，特此表示感谢。

清华大学计算中心

1983.1

目　录

一、CROMEMCO LISP 指令手册

二、CROMEMCO C 参考手册

三、CROMEMCO Overlay Linker（覆盖连接程序）指令手册

一、CROMEMCO LISP
指 令 手 册

前　言

编写本手册以满足不同程度读者需要；包括从要了解 LISP 是 LISt Processing 的缩写的初学者，到要迅速了解此 LISP 与其它 LISP 不同之处的有经验的用户。

目录给出了每节所概括的内容的缩影。因为本 LISP 方言，正如所有其它 LISP 方言一样，具有其本身的特有风格。所有未来的用户必须阅读第二部分 CROMEMCO LISP 介绍。

随 CROMEMCO LISP 软件一起有两本 LISP 文献：人工智能程序设计(Artificial Intelligence Programming) 及 CROMEMCO LISP 手册。虽然一个积极的 LISP 用户也许发现利用本手册就足以去钻研此系统，而一个初学者要利用本手册去发现如何更有效地使用 LISP，就要花相当多的时间。人工智能程序设计一书包含一些具体的应用。这些应用将帮助初学者和有经验者增进对 LISP 的理解。

在 LISP 程序设计中要学的重要课程之一是“风格”。LISP 的能力与灵活性能导致程序设计量的增加，很容易写出难以捉摸的 LISP 编码。在人工智能程序设计中，风格的问题必须优先考虑。所有未来的 LISP 用户应把第四章中讨论的数据类型的定义牢记心中。

在 CROMEMCO LISP 表示的 LISP 方言与在人工智能程序设计一书中讨论的 LISP 方言不相同。前言余下的部分将着重注意一些不一致之处，以力图使某些困难的转变减到最少。

首先，回顾一下历史的记录。两个 LISP 都有同一祖先，即用于 DEC 的 PDP—6 的最早的 MacLISP。那个 LISP 是在 MIT 发展起来的。当 Stanford 接收 PDP—6 时，为了在 DEC 监控程序下运行，就改造了那个 LISP，完成某些修改和修饰之后，那个 LISP 就演变为 LISP1.6，也称为 Stanford LISP。Stanford LISP 输出到加州大学的 Irvine 校园时又变成了 UCI LISP。在 Irvine，UCI LISP 接受了另一个称作 BBN LISP 的 LISP 变种的编辑程序和调试程序，作了进一步的修改和增强。BBN LISP 很快就变成了所谓的 InterLISP。我们从 UCI LISP 得到了 LISP 变种，这个变种出现在人工智能程序设计一书中。这些转变估计花去了大约十年。

当时，MIT 的人们重写了 MacLISP。在 MIT 基于 LISP 的任务正变得十分巨大，有效的执行问题是很迫切的。新的实现，称作 BIBOP，用了大约五年的试验与老的 MacLISP 合成一体。与此同时，一个 MIT 的小组正在设计一个类似LISP的语言，称做 Muddle，它是一个称为 Planner 人工智能语言的实现工具。当它产生时，Muddle 借此变成了第一流的语言。它已经被发行并登记为 MDL，它包括了许多概念的结合，这些概念扩展了 LISP 的设计。MacLISP 的 MDL 和 BIBOP 版本都影响

了 CROMEMCO LISP。

在本 LISP 中另一个主要因素是 MIT 的 LISP 机的经验，这部机器和它的 LISP 语言也是一个联合，这次包括了结构上的平衡的联合。

虽然这两种 LISP 的祖先是同一个，但此后所走的路是十分不同。这两种 LISP 不论在非本质方面和本质方面都有不同。非本质的差别包括 LISP 库，一个 LISP 具有某些函数，而另一个 LISP 则没有。把 CROMEMCO LISP 中的定义函数表与人工智能程序设计一书附录（301～311 页）中的表做个比较，就能发现这些差别。例如，在人工智能程序设计一书的 GET 是 CROMEMCO LISP 中的 GETPROP，并且在特性表函数中自变量的次序是不同的。这些非本质的不同很容易通过下列手段来补救，即借助于定义的删除函数，根据你的爱好再定义现存的库函数。或者在你的程序设计中加上一些变化。例如，函数名“GETPROP”与其它特性表函数(ADDPROP，REMPROP 及 PUTPROP) 相配比“GET”更好。类似地，在 CROMEMCO LISP 中这些函数始终具有〈名字〉和〈特性〉作为它们头两个变量。

本质的差别要求更加注意，且概括如下：

首先，CROMEMCO LISP 不具有“PROG”特性。PROG 有助于做 (1) 局部变量的初始化，或 (2) 程序的迭代。作为替代方法，对于初始化，使用“&AUX”工具，而用“DO”的形式来表达迭代。假如确实要求 PROG，它可以表达成一个“CATCH/THROW”表达式的适当的集合。

例如，人工智能程序设计一书中图 1.1 能表达为：

```
(DE ADD (N) (DO ((N N (SUB1 N))      ;  把一个局部参数 N 初始化为实际参
                                     ;  数。每循环一次减 1。
            (SUM 0 (PLUS SUM N)))
                                     ;  把 SUM 初始化为 0，每次迭代用
                                     ;  (PLUS SUM N) 代替 SUM。
            (((EQUAL N 0) SUM))))
                                     ;  当 N=0 时，以 SUM 退出 DO。
```

注意，在这两种 LISP 之间，注释的规定是不同的。要注意，在 CROMEMCO LISP 中没有实现特别“括号”]。通过对 LISP 的编辑程序的了解能较好地完成括号的平衡对称。

其次，CROMEMCO LISP 没有 LEXPR，而用 &REST 代替。两种 LISP 都有 FEXPR。但 FEXPR 实际上的需要是很少的，通常，宏指令提供了相应的要求。

当你已学习第五章的控制流时，你应该已经十分熟悉 CROMEMCO LISP 和宏指令。因此很容易消化这一章。

第六章 LISP 中的 I/O，从字符串的讨论开始。由于 CROMEMCO LISP 提供了第一流的串目标，这个讨论中的许多内容是过时的。正如大多数语言那样，输入及输出问题与具体实现有关。CROMEMCO LISP 提供了很强的输入/输出软件包。我们建议你理解和使用这些性能，而不是把 CROMEMCO LISP 映射到第六章的性能上去。

第七章，编辑 LISP 表达式，是一个如何把 LISP 作为一个系统实现的语言来使用的有趣例子。我们鼓励你去读完这一章，并把你的 CROMEMCO LISP 代码与人工智能程序设计一书中的 LISP 代码相比较一下。

此时，你对 LISP 的结构应该有一个适当的了解，至此再去阅读第四章，然后再继续进行人工智能程序设计一书的其余部分。

第一部分　LISP 的一般介绍

LISP 是第二个最老的高级程序设计语言，仅晚于 FORTRAN。最初实现的努力开始于 1958 年，在 J. Mccarthy 指导下进行的，J. Mccarthy 是当前斯坦福大学人工智能实验室主任。在那时他与 M. Minsky 一起是 MIT 人工智能项目共同奠基人。J. Mccarthy 关心的是需要一种用于表达人工智能问题的明确的符号表示法。这些问题不同于传统的计算概念。在这个新概念中他们宁愿注重结构的相互关系，而不是简单的数字量。当然，任何非数字问题能够变化为一个“等效的”数字量。可是，问题语句和它的解答的大部分逼真性在变换中可能丧失。Mccarthy 确认目标的表达和操作必须在更为抽象和更为原始的水平上来处理。一个例子将有助于正确地说明这个讨论，

这些设想的一项早期基础性的试验包括处理代数表达式的算法的设计。例如，代数化简可以把 $2*(x+6*y)+x$ 改写为 $3*(x+4*y)$。(代数系统的详尽的讨论参考 1979 年 8 月 BYTE 杂志上 D. R. Stoutemyer 的文章“LISP-based symbolic Math systems”）。这个算法的设计包括两个问题的解决：代数表达式的表示法和处理这种表示法的算法的详细说明。

1. 表示法问题

如何能在保持特性的情况下把代数表达式译成代码呢？这些特性对于这种符号管理算法是重要的。我们能够给表达式的每一个成分赋一个数字，然后把表达式编码为这些数字的向量。（回想起来，这是 1958 年的事，而 FORTRAN 是仅有的高级语言）。假设适当的约定能有区别的表示系数的数字和表示象 *、+、(、及）等成分的数字，我们便发现，我们的算法花费大量时间试图去找到表达式的成分：“在 $2*(x+6y)+x$ 中，请问哪一个是 * 的第二个操作数？”在这个问题中，我们需要一个表示法，它使相互间关系表示得更清楚。LISP 引入了符号表达式，一种非常通用的，抽象的符号，与自然数字相比较，它具有诱人的理论特征，而且在传统的计算机上也有一个自然的和有效地表示法。这种人为的，即实践与理论很好地结合是 LISP 特有的性质之一。

后面我们将更详尽的讨论符号表达式和它们的表示法。而现在，我们把我们的注意力限制在它们的应用上。对于我们的问题，我们选择把代数表达式表示为称作“表”的特殊类型的符号表达式。一个表包含零个或多个元素。空表可以用一个封闭的圆括号表示，即（）；一个非空的表可以包含另一个表作为其元素，以及包含原子（非表的）元素。把这些原子元素称作为原子或符号。对该例子来说，一个原子或者是一个数，正象在大多数其它语言一样；或者可以是一个非数字的目标，称为文字原子。某些 LISP，包括 CROMEMCO LISP 称为文字原子符号。在大多数其它语言中文字原子通常称为标识符，即字母和数字（或者特殊的字符）串，在标识符中这样串的第一个字符是一个

字母。回想一下其它语言，标识符出现在语言的句法中，而不是出现在数据目标中，下面是 LISP 的文字原子：

CROMEMCO　　ROCKET　　TIMES　　A

到目前为止，我们已经说过 () 代表空表，而表可以用原子或表作为元素。但还没有叙述如何把目标表示成表的元素。给定元素 e1，e2，e3，我们能建立几个表；其中之一是（e1，e2，e3），另一个是（e2，e1，e3）。人们可以用逗号分隔这些元素以建立表，并用相应的括号括起这个组合体。正如例子所表明的那样。元素的次序很重要。这些表不是元素的集合，而是序列。略去逗号可简化表示法，例如可写成（e1 e2 e3）。

正如前面指出的那样，这些 LISP 数据结构是一些有趣的抽象的目标，然而当前我们主要关心的事是在解决复杂问题时它们的有效范围。特别是，我们能怎样利用这些数据目标来表示代数表达式？例如，我们可以把表达式 6y 表示成一个表（TIMES 6y），此处我们把 TIMES 和 y 分别写作乘法操作和 y 的表示法。应注意，表的第一个元素表示操作，而表的其余元素表示操作数。接着表达式 x+6y 应表示为（PLUS X (TIMES 6 Y)）。注意，该表示法仍然能分清哪一个分量是操作，哪一个分量是操作数。最后，2(x+6y)+x 被写成

(PLUS (TIMES 2 (PLUS X (TIMES 6 Y)))X)

这种表示法本身特别简单：每一个表的第一个元素总是表示一个操作；表的余下部分的元素，或是表，在此情况下他们表示复杂的子表达式，或是数或标识符，在这种情况下他们表示数或者原始表达式的变量。给出这种表示法之后，我们继续研究我们的算法。

2. 设计算法

如何写出算法？这个算法把我们所希望获得的处理过程译成代码。算法的概念超越了一个特定程序设计语言的任何想法。我们应该在句法考虑上尽可能不受限制地叙述我们的算法。在这个水平上，我们的思想不应该受特定语言句法上的过时所限制。当我们的领域变得更复杂时，这个自由变得十分重要。在解的形成中，任务的进一步分解是十分有用的。可以把一个算法看成由两个分离的部分组成：逻辑体现了问题中的各元素之间的相互关系，控制成分指定元素如何使用。换句话说，逻辑成分对知识编码，而控制成分包含着应用知识的技术。

由于我们的许多知识是从相当抽象的数据结构中得到的，所以程序设计语言的主要任务是提供一整套操作数据结构的手段。随着是一套附加的控制结构。因为控制流常常以数据结构为基础，控制结构趋向于补充数据结构。LISP 的控制结构是条件表达式和递归，在代数化简问题上我们需要这样结构。

LISP 的条件表达式类似于其它语言的 If-then-else 结构。条件表达式的应用适应于当我们遇到的数据目标可能是几种形式中的一种时。例如，多项式中项可能是一个变量，一个常数或一个常数与一个变量的乘积。我们的算法将包括一个条件表达式，它测试这些变式的具体值，并去执行相应的动作。