LISP-130 解 释 系 统

宋国宁 齐继光 张士杰 庄国强 王天然

（中国科学院沈阳自动化研究所）

〔提要〕本文详细地介绍了我们研制成功的 LISP-130 解释系统。本系统以 LISP1.5 文本为主要依据，并吸取了其他 LISP 系统的特点，构成94个系统函数，以这94个函数为基础，可以方便地定义出各种新函数。

LISP 语言是人工智能、模式识别及其他求解非数值问题的基本语言。国外越来越多的计算机开始配备 LISP 类型的语言，成为系统软件的基本组成部分。

随着国内计算机应用领域的不断扩大，非数值应用的比重越来越大，迫切需要适合于非数值计算的语言。同时考虑到国产 DJS-130 机已被广泛使用，为此研制了适用于DJS-130小型机的LISP独立解释系统。本系统于1980年11月鉴定通过。

§1 总体方案概述

LISP-130是以 LISP 1.5 为主要依据，并考虑到国产 DJS-130 小型机的特点而配置的一种语言，由于130 机内存容量较小（32K），因此在考虑总体方案时，既要照顾到使用时灵活方便，也要考虑到解释系统不能太庞大，以致挤掉内存自由空间。为此我们从如下几方面设法：

（1）按递归方式设计解释程序。LISP 语言的重要特点是递归。因此采用递归方式进行解释是很自然的，同时也使程序大为精练，节省了内存空间。

（2）基本函数与常用的非基本函数尽量纳入到系统函数中，本系统共有94个系统函数使得不常用函数可以方便地用系统函数来定义，使得本语言的表达能力不因系统小而削弱。

（3）根据具体工作方式不同采用两种不同的数据结构。

（4）实行存储动态分配，使得已用过的单元可以反复地被利用，从效果上等于大大扩充了自由空间的容量。

（5）尽量简化管理程序。由于常规管理程序对 LISP 单用户来说是很不经济的，因此另外为 LISP 配备了极为简单的专用管理程序。

经过以上努力，我们把解释系统连同管理程序压缩在 8K 以内，其中留出了 1K 左右的扩充余地。

LISP 源程序有二种不同格式，为了使解释程序能灵活地适应两种不同类型源程序格式，在开始正式解释前要区别是什么类型的源程序。是 eval 型还是 evalquote 型。若遇（〈函数〉〈表达式〉…〈表达式〉）则按 eval 型处理，调用解释程序 eval。

若遇〈函数名〉（〈实元〉…〈实元〉）即第一个表达式是原子（函数名），则按 evalquote 型处理，调用解释程序 apply。

由于 eval 与 apply 是解释系统两个主要递归程序，它们是互相嵌套调用的，因此虽然增加了处理二种类型源程序的灵活性，却并不增加系统的开销。

在源程序运行方式上也有二种，一种是人机交互式，一种是成批处理式。对于简单的程序，可以直接从电传打入表达式，立即可以获得结果。若程序较大，则用成批处理式，由纸带输入源程序较合适。在总体方案中，兼顾到二种运行方式，并允许两种方式之间的互相切换，从而增加灵活性。例如“走迷宫”程序中先用纸带输入程序的定义部分（这是程序的主要部分），然后切换到人机交互式，通过电传调用已定义的函数，以便随机地指定迷宫的出发点与目标点，使程序的运行显得更为灵活。这种灵活性在调试程序阶段更为必要。下面流程示意图说明解释系统各入口的相互关系。

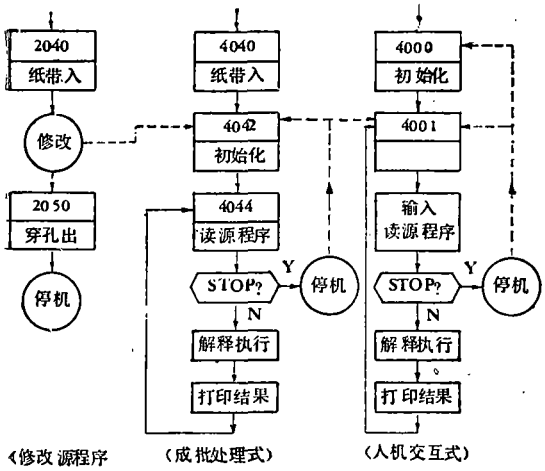

图 1

一个实用的解释系统必须具备对源程序的修改能力，但它本身并不属于解释程序范围。因此我们在专用管理程序的键盘命令中增加了简单的源程序编辑功能，可以任意修改、插入、删去一段源程序，并可穿成纸带保存。

§2 基本数据结构

LISP 程序运算过程主要是处理各种各样的表格。由于各种表不是固定不变的，因此最方便的是采用拉链式表结构。130 机字长只有16位，所以我们采用二个相邻内存单元作为基本单位。其中一个内存单元用作 cdr，另一个用作 car，例如 (A(BC) D) 的内存图象为：

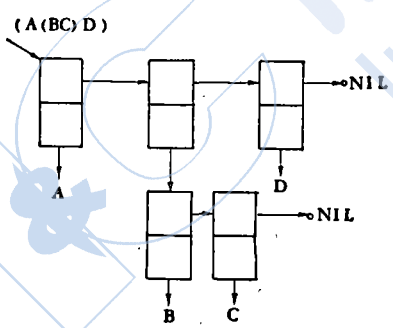

图 2

这与常规的表示法稍有不同，我们采用这种结构的理由是：在表处理过程中，cdr 运算要比 car 运算多，因此为 cdr 运算提供更多的方便是合适的(通过间接寻址就实现 cdr 运算)。

在解释系统内部，为了有效利用内存，我们采用了二种不同类型的数据结构，一种是顺序的，一种是拉链的。

顺序结构（如堆栈、字母表）：堆栈是一种下推表，也可以用拉链方式来构造，这样做虽然灵活，可以与自由空间互通有无，但本身的时空效率都较低。由于堆栈的工作方式比较简单（压入与弹出），不需要一般的表处理运算，因此权衡得失还是采用顺序方式为宜，开辟一个内存区域专作堆栈用，在解释系统中设置两个堆栈：中间结果栈与返回地址栈，为了使这二个栈互通有无，设计成一个自顶向下，一个自底向上。

字母表（A、B…E）也采用顺序结构，因为它是固定不变的（长度26）。通过字母表，指向一串串的原子链表。

拉链结构：(原子链表、原子特性表、配对表)。

原子链表：它犹如一本花名册，将所有字符原子按第一个字母分成26条链排列，由于原子是在运行中不断增加的，而且事先也难以估计会在哪个字母链上增加些什么原子，因此它采用拉链方式，查找原子时，首先找准开头字母，确定是哪张链表，然后在链表上按先后次序查对，常用原子排在前面，容易查到，这种查表方式简单、效率高。

原子特性表：原子是符号表达式的最基本元素，但原子本身又是用特性表来描述的，我们设计的特性表的结构如下：

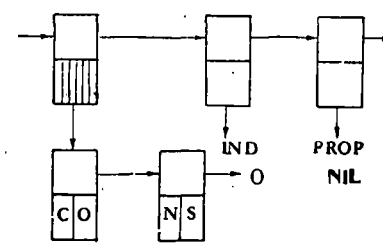

图 3

表头的 car 部分的反码（图中用阴影线表示反码）指向一张特殊的表，这表由拼写该原子名称的 ASCII 码组成，例如 CONS，4 个字母需占 2 个单元，表长亦为 2。由于表头的 car 取了反码，它是一个负数，因此就可以利用这一特征来标志这是一个原子。表头的 cdr 部分则仍以原码的形式指向其他特

性，它由指示符及相应的特性值组成，(例如：SUBR 及入口地址；EXPR 及 λ 表达式……) 这样的结构可以提高查找特性表的速度，同时为动态存贮分配提供较为单纯的表结构形式。

配对表 ASS，在解释程序中起着特殊重要的作用，通过它将变量及其值，局部函数及其定义联系起来。每当进入一个自定义函数，总要相应地在配对表上增加新的对偶，而完成自定义函数计算后，就自动恢复，因此在解释过程中，配对表是不断地变化的。通过它的进退伸缩（与堆栈配合）有条不紊地为函数的解释提供正确的信息。

§3 输入程序 RS

输入程序的功能是从源程序区读入一个完整的符号表达式，并把它转变成表结构。这是一个按递归方式工作的子程序，根据定义，

$$\langle\text{表达式}\rangle = \begin{cases} \langle\text{原子}\rangle \\ (\quad) \text{ 即 NIL} \\ (\langle\text{表达式}\rangle\cdot\langle\text{表达式}\rangle) \\ (\langle\text{表达式}\rangle\cdots\langle\text{表达式}\rangle) \end{cases}$$

工作过程如下：

（1）若读入的第一个符号是原子，则去查找原子链表，若找到，则按此地址返回，若找不到，则在链表上增加一个新原子，并以此新原子的地址返回。

（2）若读入的第一个符号是“(”，则继续读下一个符号，若为“)”，则返回 NIL 值。

（3）否则，调用本身，读入一个子表达式，并拉成链表，该链表的 car 部分指向刚刚读入的那个子表达式。

（4）继续读下一个符号，若为“)”，则拉链结束，使 cdr 部分指向 NIL，并按表头地址值返回。

（5）若为“.”则调用本身，读入下一个子表达式，使 cdr 部分指向刚读入的这个子表达式，并按表头地址返回。

（6）否则，调用本身，读入下一个子表达式并延伸链表，使其 car 部分指向刚读入的这个子表达式。

重复（4）、（5）、（6）步骤，直至遇“)”返回。

在工作过程中，还进行必要的词法分析，凡是拼写错误，括号不配对等等都在这一阶段被检查出来。为了方便使用，避免弄错括号，我们提供了一个特殊符号“：”，它等价于“任意多的自动配对的右括号”。例如：

(A(B:　　等价于　(A(B))

(A(B(C(D:　等价于　(A(B(C(D))))

另外，在本解释系统中可将 (QUOTE A) 写成 'A，其中 A 可以是原子或符号表达式。

§ 4 解释程序 EVAL

这是一个通用的解释程序，与后面介绍的 APPLY 一起构成解释系统的核心部分。它的功能是计算一个表达式（称为 form）的值。

根据语法公式：

$$\langle\text{表达式}\rangle = \begin{cases} \langle\text{原子}\rangle = \begin{cases} \langle\text{数}\rangle \\ \langle\text{常量}\rangle \\ \langle\text{变量}\rangle \end{cases} \\ \langle\text{表}\rangle = \begin{cases} (\langle\text{一般函数}\rangle\langle\text{子式}\rangle\cdots\langle\text{子式}\rangle) \\ (\langle\text{特殊函数}\rangle\ \langle\text{子式}\rangle\cdots\langle\text{子式}\rangle) \end{cases} \end{cases}$$

解释过程如下：

（1）若表达式为原子，则进一步分是什么原子。

① 若为数（或 T，NIL）则原封不动引用，即 form 之值就是 form 本身，返回。

② 若为字符原子，则先查原子特性表，看是否有指示符 APV（永久值），若有，即确定为常量，按相应的特性值返回。

③ 若查不到指示符 APV，则查配对表 ASS，若查到，即确定为变量，取相应的配对值返回。

④ 若查不到配对值，则此变量无定义，打印出错，并作相应处理。

（2）若表达式是表，则析出表头，看成是函数 (fn = (car form))，而剩余部分看成是变元表。(args = (cdr form))，并进一步查是什么性质函数。

① 若为特殊函数，则直接调用另一解释程序 APPLY，而不对变元表 args 作任何处理。

② 若为一般函数，则先将变元表中的各元素分别计算出来，变成一张实元表，即：(EVAL (子式)，…EVAL (子式))，这就要求多次递归调用本身，分别计算各子式之值，然后将它们联成一张新表，代替原来的变元表，然后再调用 APPLY，去处理函数及其实元表。

由于递归调用 EVAL 计算子式，这就实现了对

函数嵌套结构的解释。例如计算：(CONS (CAR X) (CDR Y))

首先调用 EVAL，析出函数 CONS。CONS 是一般函数，于是二次调用 EVAL，计算子式 (CAR X)，(CDR Y)；于是又析出函数 CAR，CDR，它们也是一般函数，于是又去计算子式 X，Y，而 X，Y 是原子，就去查特性表或配对表，找到相应的值，然后返回调用 APPLY 求出 (CAR X)；(CDR Y) 最后调用 APPLY 求出(CONS (CAR X)(CDR Y))。可见，单靠 EVAL 是不能工作的，必须与 APPLY 配合，交错调用。只有最简单的情况，遇到原子时，才能直接求出它的值。

§5 解释程序 APPLY

这也是一个通用解释程序。它的功能是对函数及实元表 (fn, Args) 进行解释求值，根据语法公式：

$$\langle\text{函数}\rangle = \begin{cases} \text{原子} = \begin{cases} \langle\text{系统函数}\rangle \\ \langle\text{自定义函数}\rangle \\ \langle\text{局部定义函数}\rangle \end{cases} \\ \langle\text{表}\rangle = \begin{cases} (\text{LAMBDA}(\text{X}\cdots)\langle\text{表达式}\rangle) \\ (\text{LABEL}\langle\text{函数名}\rangle(\text{LAMBDA}\cdots)) \\ (\text{FUNARG}\langle\text{函数}\rangle\cdot\text{ASS 表}) \\ (\text{其它表达式}) \end{cases} \end{cases}$$

解释过程如下：

（1）若函数为原子，则查特性表，进一步分类：

① 若为系统函数，则转相应解释子程序入口。

② 若为自定义函数，则取其定义（λ 表达式）代替原来的函数，并再次调用自身进行处理。

③ 若特性表上查不到，则查 ASS 表。若查到即为局部定义函数，改用其定义(也是 λ 表达式)代替原函数，并再次调用自身进行处理。

④ 若 ASS 表上查不到，则函数无定义，打印出错，并转相应处理。

（2）若函数为表，则析出表头，进一步分类：

① 若为 LAMBDA，则进行虚元与实元的配对，添加到 ASS 表上，然后调用 EVAL 计算虚元表后面的表达式。

② 若为 LABEL，则将函数名与后面的 λ 表达式配对，添加到 ASS 表上，然后转①按 LAMBDA 类处理。

③ 若为 FUNARG，则以〈函数〉后的 ASS 代替原来的ASS，并以当前的〈函数〉代替原来的函数，并调用本身进一步处理。

④ 若为其他表达式，则先调用 EVAL 计算出这个表达式之值，并以此值作为函数名，再次调用 APPLY 自身。这一功能使得函数本身可以是计算出来的结果，增加了灵活性。

由上可见，只有当函数是系统函数时，才能转到子程序入口，并可求得值。遇到其他类型的函数，只是反复递归调用本身或 EVAL。不能直接获得结果。

§6 关于特殊函数的解释

特殊函数有：QUOTE、FUNCTION、DEFINE、COND、…，等等。它们与一般函数的区别在于不事先对它们的变元进行计算。有时直接引用变元（如QUOTE），有时在引用变元同时，附加一些现场信息（如 FUNCTION），有时在函数解释子程序的内部对变元进行计算（如 COND、AND、OR、…），有时则是副作用是主要的（如 DEFINE）。

现分述如下：

（1） QUOTE。最简单，直接将变元原封不动地变成函数值就行。

（2） FUNCTION。它的功能是引用函数名作为 MAP 型函数的实元。在简单的情况下，可用 QUOTE 来取代它。但当引用函数中包含有自由变量、且该变量又与外层调用的约束变量重名时，就会发生混淆。为此，在引用函数名同时，要附加当时的现场信息，即当时配对表的状态，所以，解释时将函数名和 ASS 表配对，并冠以特殊表标记 FUNARG，组成一张新表：《标志 FUNARG〉〈函数名〉·ASS 表），将此表作为 FUNCTION 函数的值。当进入 EVAL 解释时，就自动析出函数名，并按 ASS 表内容临时改变现场（见 EVAL 解释）。

（3）COND。它有任意多的变元，每个变元一般是由二个元素的表所组成。根据 COND 的含义，应先计算第一个变元中的第一个元素（递归调用 EVAL），若不为 NIL，就接着计算第二个元素（又调用 EVAL），并以此值作为 COND 的值返回。若第一个变元的第一个元素之值为 NIL，则跳过第二个元素，对第二个变元的第一个元素进行计算（调用 EVAL），若不为 NIL，就接着计算第二个变元的第二个元素，并以此值作为 COND 值返回。否则，又跳过第二个元素，对第三个变元的第一个元素进行计

算，……，直至所有变元处理完毕，都得 NIL，就按 NIL 值返回，由于 COND 中调用 EVAL，而 EVAL 中又可能调用 COND，所以，它必须是递归的。

（4）DEFINE。它是以副作用为主的伪函数，它只有一个变元，且为如下形式：

((函数名 (λ…))
(函数名 (λ…))
⋮
(函数名 (λ …)))

解释时，在各函数名的原子特性表上设置 EXPR 指示符，并以相应的 λ 表达式作为它的值。并将各函数名析出，组成一张表，作为 DEFINE 的值，其目的是为了打印出这张表告诉用户，哪些函数已经有了定义。由于 DEFINE 的格式容易写错，故在解释过程中，要作较多的语法检查。

（5）其他函数，如 AND、OR、… 亦属于特殊函数，它们的解释过程都比较简单，故从略。

§7 解释程序 PROG

LISP 语言中引入 PROG 特性的目的是吸收其他语言中用“过程”来描述算法的优点，使得 LISP 语言更为灵活、方便。它的解释过程如下：

（1）先造局部变元表：将各局部变元与NIL 配对，添加到 ASS 表上。

（2）造标号表：扫视整个 PROG 程序的顶层，凡是以单独原子出现者，均认为是标号，并将它们联成一张标号表。

（3）从局部变元表后开始解释执行。凡遇标号，跳过不做，对非标号的表达式，调用 EVAL 进行计算，进行检验：

① 若为一般表达式，则顺序往下执行。

②若为 (RETURN…)则返回，并以计算结果值作为 PROG 之值。

③ 若为 (GO〈标号〉)则查标号表，若找到该标号，则把控制转向计算标号后的表达式，若找不到相应标号，则标号无定义，打印出错并转相应处理。

由于 PROG 允许重进入，因此它的解释程序也是递归的。

各类函数的解释程序是多种多样的，上面叙述的是通用的，有代表性的。其它函数的解释程序就不一一赘述了。

§8 存储动态分配（废料收集程序GC）

在函数的解释过程中，要不断消耗自由空间，例如 CONS，每次调用它就消耗掉一对内存单元（用作联结两个变元组成新的表达式），但经过若干次运算后，往往只有少数单元要继续保留，大多数单元时过境迁，不再有用了，需要在适当的时候，由系统收回，这一工作由废料收集程序完成。

在系统初始生成时，就将自由空间连成一张很长的链表，称为自由空间表 (FSL)，每当解释系统需要提供新单元时，它就从表头“卸下”一对内存单元，并将 FSL 表指针往后拨一步。一直到全部自由空间表消耗殆尽，就调用废料收集程序，重新构造自由空间表。

鉴别某个单元是否为废料的准则是：凡是有用单元必定与系统某些表结构联结在一起的，反之就是废料。联系有用单元的表结构有：

（1）原子链表上的原子以及原子特性表延伸出来的表结构；

（2）中间结果堆栈中的指针指向的表结构；

（3）某些特殊变量我们称为锁变量（如 form，fn，Args，Ass，Res，…）所指向的表结构。

废料收集程序的工作过程如下：

（1）顺藤摸瓜，打上标记。即沿上述三类表结构的走向按先 car 后 cdr 的顺序一个不漏地打上标记，这是一件很费时间的工作，当这工作完毕时，凡是有用单元都打上了标记，它们可能分散分布于内存的各处。而凡是废料则没有标记。

（2）顺序搜索，收集废料。从自由空间的起始地址开始一直到末地址，依次检查标记，凡已有标记者就清除这个标记（准备下次打标记用），凡是没有标记者就视为废料，将它拉成一张链表，即新的自由空间表。

§9 输出程序 PS

输出程序与输入程序相反，它的功能是将计算得到的表结构转变成字符串，由打印机输出，工作过程如下：

（1）若表达式是原子，则找出相应的打印名（打印名的 ASCII 码）送缓冲区。

（2）若不是原子而是表，则先往缓冲区送出字

符"("。

（3）对表的 car 部分调用 PS 本身，送出一个子表达式（它可以是原子，也可以是表）。

（4）考查表的 cdr 部分，若为 NIL，送出字符")"作为结束。

（5）若 cdr 部分指向一般原子，先送出字符"."，然后送出该原子的打印名，最后补上字符")"。构成如下形式的表$(S_1\ S_2\cdots S_{n-1}\cdot S_n)$作为结束。

（6）若 cdr 部分指向表，则送出"空格"。并重复执行(3)(4)(5)(6)直至 cdr 出现 NIL 或其他原子为止。

由于在处理 cdr 部分时调用了本身，因此这也是一个递归的子程序。

等全部表达式解释完毕，字符送往缓冲区后，启动打印设备。

§10 出错处理及程序调试

在解释过程中，若发现词法及语法错，就立即打印出错地址，通过这个信息标志出错的性质。在人机交互式运行时，打印出错后，返回 NIL 值或 0 值，程序继续往下执行。这是因为电传输入时很容易打错，出错不停机。在成批处理时打印出错后停机，以便进一步查找原因。可以用键盘命令 QD6440，就进一步打印出出错现场的其他信息，如 form，Fn，Args。

为了便于调试程序，系统设有跟踪函数 trace 与解除跟踪函数 untrace。

函数 trace 的作用是在要跟踪函数的原子特性表上设置指示符 trace。

函数 untrace 的作用则相反，删去指示符 trace。

当解释程序 APPLY 遇到原子函数时，要检查是否有 trace 标志，若有，则在解释前安排打印变元值。在解释后，则安排打印结果值。

§11 结束语

经过对系统函数的反复测试及各类程序的运行，证明本系统是可靠的。

由于采用了合理的数据结构、递归技术、内存的动态分配和简化管理等技术，使得本系统能以较少的开销而获得较全的功能。

本系统能适应 EVAL 和 EVALQUOTE 两种不同格式的源程序，并能以两种不同方式（人机交互式和成批处理式）运行，从而增加了使用的灵活性。

本系统具有较完善的词法检查和语法检查功能，这就增加了运行的可靠性，又便于调试程序，发现错误。

由于本系统所要求的硬设备是所有的 DJS-130 机都具备的，所以，便于推广使用。

参 考 文 献

〔1〕 Waite, W. M., Implementing Software for Non-numeric Application, Prentice-Hall, Inc., Englewood Cliffs, N. J. 1973.

〔2〕 McCarthy, J., Abrahams, P.W., Edwards, D. J., Hart, T.P. and Levin, M. I., LISP 1.5 Programmer's Manual, The M.I.T Press, Cambridge, Mass. 1962.

〔3〕 Siklossy, L., Let's, Talk LISP, Prentico-Hall, Inc., Englewood Cliffs N. J. 1976.

〔4〕 宋国宁，LISP 语言及其在人工智能中的应用(一)、(二)、(三)，信息与控制 №.4～6，1980.