微型计算机软件资料汇编

第 三 册

机械工业部计算中心
合肥工业大学微型机应用研究所 编译

GRAPHICS

muLISP

rDBMS

CP/M

机械工业部仪表局情报室
《仪表工业》编辑部

微型计算机软件资料汇编（第三册）

机械工业部计算中心 编译
合肥工业大学微型机应用研究所

•

机械工业部仪表局情报室《仪表工业》编辑部编辑出版
轻工业出版社印刷厂排版印刷
北京市建华书刊发行社发行

•

1984年10月 北京
代号：8402

内 容 提 要

《微型计算机软件资料汇编》第三册包括"BASIC-80使用手册"和"muLISP/muSTAR-80参考手册"。

Microsoft BASIC-80是目前功能最强、适应面最广的BASIC语言之一。尤其是它的字串处理和文件处理的功能优异，为进行数据管理提供了强有力的工具。"BASIC-80使用手册"除对BASIC-80语言作了详尽的介绍外，还提供了不少关于程序设计的知识和技巧，无论对熟悉BASIC语言的人，还是对初学者，都有一定的参考价值。

muLISP/muSTAR-80人工智能开发系统是用于微型计算机的一个高级软件包，可用于各种人工智能研究开发。"muLISP/muSTAR-80参考手册"介绍了muLISP数据结构、内存管理和原始定义函数以及muSTAR-80辅助程序的使用，可作为使用muLISP语言的参考资料。

微型计算机软件资料汇编

第三册

机械工业部计算中心
合肥工业大学微型机应用研究所 编译

机械工业部仪表局情报室
《仪表工业》编辑部

编 译 出 版 说 明

本资料汇编收集了近期从国外引进的微型计算机软件，包括CP/M操作系统及其支持程序、高级程序设计语言、数据库管理系统和应用软件包，可以在Zilog Z80系列、Intel 8080系列微型机上使用，并已在H/Z89微型机上验证。

收集在资料汇编中的有：

微型机操作系统CP/M2.2；

小型关系数据库管理系统CONDOR SERIES/20；

高级语言 COBOL-80，PASCAL/MT+，FORTRAN-80，MBASIC，PL/I-80，C，muLISP；

编辑和字处理系统；

分类/合并程序；

库存管理程序；

图形软件包；

远程终端仿真程序等

这些资料大部分以使用手册的形式提供，可作为微型机用户手册，也可供计算机系统软件和应用软件人员以及大专院校有关专业师生学习参考。

本资料汇编由机械工业部仪器仪表工业局组织，机械工业部计算中心和合肥工业大学微型机应用研究所编译，刘运基、康兴钨审校，并请旅美学者赵鉴芳教授指导审定。在编译出版过程中，得到许多同志的大力协助，谨在此表示谢意。

由于编译者水平所限，难免有错漏之处，敬请读者指正。

本资料汇编共分六册，由机械工业部仪表局情报室《仪表工业》编辑部陆续出版。

目　录

BASIC-80 使用手册

muLISP/muSTAR-80人工智能系统参考手册

BASIC-80使用手册

第一章　系统介绍和基础知识

本章包括：1.“装配指南”和BASIC-80程序设计语言的基础知识（BASIC-80是适合于8080和Z80微处理器的扩充内容最丰富的BASIC语言之一）；2. BASIC-80对硬件及系统软件的各项要求；3.面向用户的BASIC-80工作环境的说明。

第一节　装配指南

此节供Microsoft MBASIC-80解释程序和BASIC编译程序之用

1　磁盘目录

你接收到的磁盘应载有下列文件：

1) Microsoft MBASIC-80解释程序源盘：

MBASIC · COM

PI · BAS

MBASIC · COM是BASIC解释程序，它具有的命令和功能在本参考手册中有详细的介绍。PI · BAS是一个用BASIC语言写的实例程序，它用于计算π的值。PI · BAS将有助于熟悉解释程序的工作过程。

2) Microsoft MBASIC编译器源盘Ⅰ：

BASCOM · COM

BASLIB · REL

BASIC编译器存放在文件BASCOM · COM中，它具有各种命令和功能，在“BASIC编译器用户手册”中说明。BASLIB · REL是BASIC编译器系统程序库，可以通过库管理程序（即LIB · COM，在编译器源盘Ⅱ中）来修改它。

3) MBASIC编译器源盘Ⅱ：

L80 · COM

M80 · COM

CREF · COM

LIB · COM

PI · BAS

PI · REL

“Microsoft实用手册”的第二部分，定义了MACRO-80汇编器〔M80 · COM〕的使用和操作。在该实用手册的第三部分论述了交叉调用程序CREF · COM，第四部分论述了连接装配程序L80，第五部分论述了库管理程序LIB · COM。

PI · BAS是一个实例程序，用来计算π值，它帮助你学会如何编译、连接和执行一个程序。PI · REL是一个浮动目的文件，它是对PI · BAS进行编译而产生的。

基于你收到的存储载体可能是其他类型，你可以把上述文件复制到一块或多块磁盘中。

2. PI · BAS的输出实例

PI BAS程序的输出实例清单于下。注意，由于处理数据所使用的算法不同，因而解释程序和编译程序所产生的结果可能有所区别。

π的近似值——双精度、二项式定理展开式计算（解释程序结果）

N	边数	边长	不定近似值	过剩近似值
3	8	0.765366911888l2	3.06146764755249	4.95931573036713
4	16	0.39018064737320	3.12144517898560	3.87800677621650
5	32	0.19603428244591	3.13654851913452	3.47739260077205
6	64	0.09813534468412	3.14033102989197	3.30237067197655
7	128	0.04908246546984	3.14127779006958	3.22030812114884
8	256	0.02454307302833	3.14151334762573	3.18054350336212
9	512	0.01227176748216	3.14157247543335	3.16096780640274
10	1,024	0.00613591633737	3.14158916473389	3.15125708956375
11	2,048	0.00306796119548	3.14159226417542	3.14641880958168
12	4,096	0.00153398059774	3.14159226417542	3.14400368450104
13	8,192	0.00076699029887	3.14159226417542	3.14279751177684
14	16,384	0.00038349514944	3.14159226417542	3.14219477240231
15	32,768	0.00019174757472	3.14159226417542	3.14189348940372
16	65,536	0.00009587385284	3.141594'0994263	3.14174501554227
17	131,072	0.00004793689368	3.14159226417542	3.14166756506744
18	262,144	0.00002396846321	3.14159440994263	3.14163205998885
19	542,288	0.00001198423161	3.14159440994263	3.14161323485294
20	1,048,576	0.00000599211580	3.14159440994263	3.14160382236958

π的近似值——双精度、二项式定理展开式计算（编译程序结果）

N	边数	边长	不定近似值	过剩近似值
3	8	0.76536686473018	3.06146745892072	4.95931523537420
4	16	0.39018064403226	3.12144515225805	3.87800673496263
5	32	0.19603428066912	3.13654849054594	3.47739256563251
6	64	0.09813534865484	3.14033115695475	3.30237081249040
7	128	0.04908245704582	3.14127725093277	3.22030755454287
8	256	0.02454307657144	3.14151380114430	3.18054396821973
9	512	0.01227176929831	3.14157294036709	3.16096827709498
10	1,024	0.00613591352593	3.14158772527716	3.15125564133382
11	2,048	0.00306796037257	3.14159142151120	3.14641796432625
12	4,096	0.00153398063749	3.14159234557012	3.14400376602075
13	8,192	0.00076699037514	3.14159257658487	3.14279782442605
14	16,381	0.00038349519462	3.14159263433856	3.14219514270746
15	32,768	0.00019174759819	3.14159264877699	3.14189387407905
16	65,536	0.00009587379921	3.14159265238659	3.14174325781772
17	131,072	0.00004793689962	3.14159265328899	3.14166795419967
18	262,144	0.00002396844981	3.14159265351459	3.14163030351872
19	524,288	0.00001198422491	3.14159265357099	3.14161147846025
20	1,048,576	0.00000599211245	3.14159265358509	3.14160206600152

3. 磁盘的使用

磁盘的装入

打开磁盘驱动器门，将磁盘标签面朝向打开的门，插入磁盘，然后小心地关上驱动器门。

磁盘的保管

由于磁盘易于损坏，所以在使用磁盘时必须采取如下的保护性措施：

1） 磁盘不使用时，应将其插入纸套中。

2） 让磁盘远离磁场、磁性纸装订机以及被磁化的剪刀、螺丝起子和大功率电子仪器。因为各种磁场能使磁盘上记录的数据丢失。

3） 更换已损坏或过分陈旧的纸套。

4） 只能在磁盘标签上写字，而且只能使用沾水笔尖，不能使用铅笔或元珠笔，因为它们可能损坏磁盘的表面。

5） 磁盘应远离热源或具有污染性的物质。

6） 磁盘不要暴露在阳光、烟雾或湿度很大的环境中。

7） 不要触摸磁盘表面。因为触摸可能会使数据遭到破坏。

写保护

磁盘可以加上写保护，从而使数据不能写入到磁盘中（所有交付的磁盘都具有写保护功能）。一块磁盘写保护的方式，取决于磁盘的大小。

在5.25英寸磁盘的侧边有一个写保护缺口。当缺口用标签或不透明的纸带贴上时，数据就不能再写入到磁盘中。

一块8英寸磁盘的写赋能缺口在盘的侧边。如果写赋能缺口露出，则数据就不能写入到磁盘中。为使8英寸磁盘写赋能，要用标签或不透明的纸带将缺口贴上。

4. 制备工作盘

按照CP/M手册中的规定，要制备一个工作盘，首先要接通计算机电源，然后从CP/M源盘1上将CP/M导入内存。

如果有两个或多个相同尺寸的驱动器，那么就可以使用DUP·COM文件来复制MBASIC源盘。如果仅有一个驱动器，则可以按如下方式：

1） 使用程序FORMAT·COM对用来拷贝的空白磁盘进行初始化。

2） 使用PIP·COM程序来复制MBASIC源盘。

注：所有源盘都加上了写保护，以保证有一个精确的软件源本。因此源盘被复制后应存放在一个安全的地方，而将拷贝盘作为日常编程之用。

第二节 系统介绍

1. 手册简介

本BASIC-80手册是BASIC-80语言的主要参考书。章节是根据功能来编排的。例如，如果需要了解有关字串的资料，那么只要查阅第五章“字串”。

本BASIC-80手册还包括了“装配指南”和一张使用卡片，该指南告诉你怎样拷贝BASIC解释程序工作盘，而使用卡片中列出了许多常用资料，故应把它放在手边以便查找。

2. 硬件要求

运行BASIC-80解释程序时，对硬件的要求如下：

1) 8080或Z80微处理器，

2) 具有48K的RAM，

3) 一个软盘驱动器，

4) 一个终端显示器，

5) 可选项——一台硬拷贝设备〔例如行式打印机——译者注〕。

以上是最低要求的硬件结构。我们建议配有多个磁盘驱动器，若计划编制大型程序，那么无疑将需要一台硬拷贝设备。

3. 系统软件的要求

BASIC-80解释程序必须在CP/M(2.0或2.0以后的版本)操作系统支持下运行。

4. 制备磁盘

BASIC-80解释程序分布在5.25英寸或8英寸磁盘中。"装配指南"中有关于制备工作盘的资料。

除了作拷贝用途之外，一律不要使用BASIC-80源盘，源盘应存放在安全的地方，"装配指南"中还有更多的有关磁盘保管的资料。

5. BASIC-80的初始化

BASIC-80解释程序以绝对二进制形式存放在磁盘上，其文件名为MBASIC·COM。BASIC-80可以直接装入内存使用。为装入BASIC-80，可在CP/M提示符后面键入：

```
MBASIC
```

上述命令将把MBASIC装入内存。MBASIC被装入内存后，将显示某些启动信息.该信息类似下列形式：

```
BASIC-80 Rev·5.2
〔CP/M Version〕
Copyright 1977,78,79,80(c) by Microsoft
Created: 11—Aug—80
15430 Bytes free
```

请注意：系统版本号码、文件创立日期、内存的自由字节数可能与此不同。

当把一个程序文件名附到BMASIC 这个命令串后面时，该程序可以自动运行。例如，如果你想将解释程序装入内存并运行SAMPLE·BAS程序，则可键入下面的命令串：

```
MBASIC␣SAMPLE
```

这里MBASIC和SAMPLE 之间的空格是需要的（在本手册中符号␣指明一个空格）。在上面命令中假定文件名的补缺值是·BAS。若指定的文件名没有找到，则将显示"File not found"（文件未找到）的信息，并返回到CP/M命令状态。

第三节 基础知识

1. 操作方式

解释程序装入内存之后，BASIC-80将显示字符"OK"，这表明BASIC-80已处于命令

工作方式，或称命令状态。

在命令状态中，一旦用回车符RETURN结束指令，BASIC-80解释程序就立即执行，在命令状态中输入的命令和语句前面都不要行号。算术和逻辑运算的结果可以被立即显示并保存下来以备后用，但指令本身在执行以后就丢失了。这种状态对于调试程序和将BASIC-80作为一个计算器以完成快速计算是有用的，该计算一般不需要一个完整的程序。

如果一个程序行由一个行号开始，则BASIC-80认为你想存储该程序行以供以后执行。这称为间接方式或程序方式。存在内存中的程序，输入RUN命令后则被执行。

2. 行格式

在BASIC-80程序中，程序行格式如下（方括号中是可选项）：

nnnnn BASIC-80语句〔:BASIC-80语句〕

按照程序员的意图，若干条BASIC-80语句可以放在同一行中，但每条语句之间要以冒号分隔。

一条BASIC-80程序行要以行号开头、回车符结尾，每行最多包含255个字符。

利用终端键盘上的LINE FEED键，可以将一逻辑行分开到若干物理行上去。LINE FEED键允许你在下一个物理行中继续输入同一逻辑行，而不必用回车键RETURN。

3. 行号

每一条BASIC-80程序行的起始都有一个行号，行号决定程序行存入内存的次序，也供分支跳转和编辑查找时使用。行号必须在0～65529范围内。在EDIT、LIST、AUTO和DELETE命令中可以使用一个〔·〕代表当前行。

4. 字符集

BASIC-80字符集由字母、数字和特殊字符所组成。字母可以是大写字母或小写字母，数字是0～9。

BASIC-80也识别以下的特殊字符和终端键：

符号	名称
␣	空格
;	分号
=	等号或赋值号
+	加号
−	减号
*	星号或乘号
/	斜杠或除号
↑	上箭头号或乘幂号
(	左圆括号
)	右圆括号
%	百分号
#	数值符（或镑）
$	美元符号
!	惊叹号
〔	左方括号

]	右方括号
,	逗号
.	句号或十进制小数点
'	单引号(撇号)
"	双引号
:	冒号
&	和
?	问号
<	小于
>	大于
\	右斜杠或整除符号
@	At符号
_	下划号
DELETE	删除最后输入的一个字符
ESC	从编辑状态子命令 I 中退出
TAB	光标跳到下一表格位置，光标每八格一跳
LINE FEED	光标移动到下一个物理行
RETURN	结束一行
CTRL	控制键

5. 控制字

BASIC-80 中有下列控制字：

CTRL—A	在一行输入时进入编辑状态
CTRL—C	中断程序执行并返回到BASIC-80状态
CTRL—G	使终端响铃
CTRL—H	退格，删除最后键入的字符
CTRL—I	表格符(TAB)，光标每八位一跳
CTRL—O	程序运行当中暂停输出，下一个 CTRL-O则重新恢复输出。
CTRL—R	重新显示当前语句行
CTRL—S	暂停程序执行
CTRL—Q	在 CTRL—S 之后，用以恢复程序运行
CTRL—U	删除当前语句行

若要输入这些控制字，则需要在按住CTRL 键的同时按入字母键。例如：要输入CTRL—G，则按住CTRL键不放，再按字母键G。

第四节　BASIC-80编程

本节使你了解如何编制BASIC-80程序，并说明 BASIC-80 程序设计中的一些特点。

1. 装入BASIC-80解释程序

在使用BASIC-80解释程序之前，必须把它装入计算机内存，它是一个绝对二 进 制 文

件，即该程序可以直接被计算机执行。在按下列步骤操作之前，必须先导入操作系统，如果不了解如何操作，则应查阅有关操作系统手册。

调用解释程序的CP/M文件名是MBASIC·COM，因而要把BASIC-80解释程序装入内存，可在CP/M系统提示符后面键入：

A〉MBASIC

（不要键入A〉，因为它代表 CP/M 系统提示符。并记住按 RETURN 键来结束这一行。）

这里假定MBASIC·COM文件驻留在当前补缺盘中，若该文件没有存放在当前补缺盘中，那么就要先键入驱动器名然后再键入文件名。例如：若A：是当前补缺盘，而 BASIC-80文件驻留在驱动器B：中，则要用下面的命令来装入BASIC-80解释程序：

A〉B:MBASIC

BASIC-80装入内存后，荧光屏上将显示有关启动的信息，可使用的自由存储单元总数以及BASIC-80的文本编号也将被显示。记下可使用的存储单元总数，这对于编制一个复杂的大型程序来说，无疑是一个关键性的信息。

当BASIC-80以上述方式装入内存时，它将对工作环境作下述约定：

不能同时打开三个以上的磁盘文件，

可使用全部有效内存，

随机记录大小为128字节。

当然，你可以通过使用某些开关去改变这些约定。

可被同时打开的磁盘文件的数目范围为0～15，开关/F：用来指定同时打开的文件 的 最多个数。BASIC-80将在内存中为每一个由开关/F：规定的文件建立一个文件缓冲区，这将减少可使用的存储单元的数量。例如要建立 5 个文件缓冲区，可采用如下命令：

A〉MBASIC␣/F:5

注意！在MBASIC和/F:5间有一个空格，如果未键入这个空格，CP/M系统将认为这个开关是文件名的一部分。

还可以通过开关 /M：来规定 BASIC-80 使用的内存的最高极限位置。在某些场合中，为了预留汇编语言子程序的空间，设置的最高存储位置要处于CP/M BDOS 的位置之下，但希望不要低得太多。在所有场合中最高存储位置都应在BDOS的起始位置之下（BDOS的起始地址在存储单元 6 和 7 之中）。如果/M：开关被省略，则可使用的存储空间最高可达 到BDOS的起始位置。

注：文件的数目和最高存储位置，可用十进制、八进制（加前缀＆O）或十六进制(加前缀＆H)数来表示。

通过使用开关/S：，还可以改变随机文件记录的大小。记录大小的补缺值是128字节，一个记录最多为256字节。例如要指定记录大小为200字节，则可以使用以下的命令：

A〉MBASIC␣/S:200

以上三个开关的任何组合都可以用在一条命令行中，例如命令：

A〉MBASIC␣PAYROLL·BAS

表明使用全部存储空间，可同时打开的文件数目为 3 。又如，装入和 运 行

INVENT·BAC：

A〉MBASIC↵/M:32768　INVENT·BAC

表明可使用低端32K的存储单元，可同时打开3个文件。

BASIC-80解释程序装入内存后，就可以开始编制程序。

2. 编制BASIC-80程序

一个BASIC-80程序由若干行语句所组成。语句即是对BASIC-80的指令。每一程序行用一个行号开始，后面跟随一条或多条BASIC-80程序语句。行号指出语句执行的顺序，该顺序也可通过某些语句来改变。

BASIC-80程序行的格式是：

行号	语句关键字	语句正文	行终止符
100	LET	x=x+1	〈RETURN〉

BASIC-80程序中的每一程序行，必须由行号开始，行号必须在0～65529正整数范围内。BASIC-80行号是一种标志，用来区别程序中各行，因此，每一个行号在程序中必须是唯一的。

在BASIC-80程序中，每一程序行要用一个回车符来结束，即键入键盘上的RETURN键。

可以使用象1、2、3、4这样的连续行号。如：

```
1↵x=1
2↵y=2
3↵z=x+y
4↵END
```

然而，实际应用的是以10为增量的递增行号，这便于在已编好的语句行间插入新的语句。

```
10↵x=1
20↵y=2
30↵z=x+y
40↵END
```

另外一个非常有用的东西是让BASIC-80自动产生行号，这可用AUTO命令来实现。例如，若键入命令AUTO100,10，则BASIC-80会由行号100开始自动编号，每行递增10。你所要做的只是在产生的行号后面键入BASIC-80程序行。

3. BASIC-80程序的运行

BASIC-80程序编好后，通常需要去运行这个程序。运行是通过RUN命令来实现的。下面的命令使BASIC-80立即执行内存中的现行程序：

```
RUN
```

这个命令使程序从最低的行号开始连续执行（除非执行的顺序被象GOTO这样的语句所改变）。在RUN命令中也可以规定程序执行开始行号。例如下面的命令使程序从行号100开始执行：

```
RUN↵100
```

RUN命令也可以用于执行一个存在于磁盘上的BASIC-80程序。比如假设ALBUM·BAS文件存储在现行补缺盘上，下面的语句就可使ALBUM·BAS程序投入运行：

RUN"ALBUM"

注意：在上述文件名字串中，没有指定驱动器和文件的扩展名。在这种情况下，则假定是现行补缺驱动器，扩展名是·BAS。

另外，要保证在文件名字串中只使用大写字母。BASIC-80依赖CP/M进行文件处理，而CP/M系统中大多数公用程序不能识别用小写字母写的文件名。因此，以小写字母作为文件名来存储文件显然是不恰当的。因为CP/M不能识别小写字母的文件名，因而也就不能删除或更改这类文件名。虽然小写字母为名称的文件能在BASIC-80中被删除，但在BASIC-80程序语句中使用文件名时，也一定要用大写字母。

这并不是说，使用小写字母作为文件名有原则上的错误，而是使用小写字母作为文件名可能会产生意外的结果。也许有人希望用小写字母作为文件名，以使文件更难被删除或被改名，这样就为一些重要的程序提供了一个额外级别的保护。

4. BASIC-80程序的调试

有时，BASIC-80程序并不象你预期的那样被执行，这通常是由于存在语法错误或逻辑错误。语法错误是易于觉察的，因为BASIC-80解释程序不仅能指出语法错误，而且还指出错误程序行，并自动进入编辑状态；但是，逻辑错误是难于发现的，然而系统提供了若干条语句使得逻辑查错变得较为容易些。

当BASIC-80发现语法错误时，它将在产生的错误行上自动进行编辑状态。此时，可以按L键以显示这一行（L是一个BASIC-80编辑子命令，有关编辑程序的详细说明请参看第九章"编辑"）。

语法错误的产生往往是由于拼错了关键字或使用了不正确的语句结构。请记住BASIC-80解释程序要求所有关键字都用空格分隔这一点。改正语法错误的简便方法主要是查阅参考手册。

在发现语法错误的时候，可以查阅参考手册中的有关章节，这可以利用索引来查找。在发现并纠正了错误之后，应该记住这种错误是如何产生的，并避免重犯类似错误。

BASIC-80解释程序的内部特性使得调试一个BASIC-80程序非常方便。它提供若干条语句帮助对BASIC-80程序进行查错，但你首先要弄清错误的性质。

一个程序中的错误，可能引起一个错误输出的发生，或使程序执行转移到错误语句，计算的结果可能是错误的或无法理解的。程序中的错误还会显示出状态错误标志，所以在要了解程序为什么这样做之前，你必须知道程序目前正在做什么。

可以相信，在绝大多数情况下(99.99%)是程序本身有错误，而BASIC-80解释程序出错是极不可能的。BASIC-80解释程序是适合于8080/z80微处理器的最广泛最透彻的BASIC语言之一，它本身是非常可靠的。所以如果产生了问题，总可以认为是由用户程序中的错误所引起的。

一旦确定了程序正在做什么，就可以着手去弄清为什么程序不能正确地执行。例如假设一个程序分支转移到一个行号，而这个行号却是你所不希望的。系统提供了跟踪标志以跟踪程序的流程，TRON语句用来设置跟踪标志，而TROFF语句用来撤消跟踪标志。

在跟踪程序运行时，行号被依次显示出来，它们被置于方括号([ ])内。最好是打印一张程序清单，以便对照这张清单来进行跟踪。

另外一项重要技术是在程序中设置断点。你可以用STOP语句去暂时停止程序的执行，

然后输入命令去打印每个变量的值，也可以给变量赋于新值，最后你可以用 CONT 命令或GOTO命令让程序继续运行。

虽然可以打印和改变变量的值，但在使用了STOP语句中止执行后，不能改变原来的BASIC-80程序。如果在暂停时修改程序，那么所有已经存入的变量值都将丢失，所有打开的文件都将被关闭。

5. BASIC-80程序的保存

当一个BASIC-80程序的编制或调试工作告一段落时，你一定希望将当前程序拷贝到磁盘上，这可以用SAVE命令来实现。SAVE命令的一般格式是：

SAVE"〈文件名〉"

〈文件名〉必须是一个合法的CP/M文件名。如果没有给出驱动器号，则约定为当前补缺驱动器；如果没有给出文件扩展名，则补缺值为·BAS。例如，要想保存程序 GAME·BAS，则可以使用下面语句：

SAVE"C:GAME·BAS"

注意，该文件将保存在驱动器C:上，文件扩展名·BAS可以省略，而由系统提供补缺值。文件名一定要用大写字母。BASIC-80解释程序一般以紧缩二进制形式来保存文件，然而程序也可以ASCII码形式来保存。例如：

SAVE"C:GAME", A

这个命令将文件以ASCII码形式存储在驱动器C:中，其文件名为GAME·BAS。也可以以带保护的形式来存储程序，这样使得程序不能被列出或编辑，这只要在文件名后面附加一个字母P。例如：

SAVE"C:GAME", P

这个文件将以二进制编码形式存储。当带保护的文件以后被运行或装入时，则将不能列出这个程序或对它进行编辑。

6. BASIC-80程序的装入

若想在某一个BASIC-80程序中开始下一阶段的工作，必须将它从磁盘调入内存。这可使用LOAD命令来实现。LOAD命令的一般格式为：

LOAD"〈文件名〉"

例如，要想装入PAYROL·BAS程序，则可使用下面的命令：

LOAD"PAYROL"

注意该文件的扩展名被省略了，BASIC-80约定文件扩展名是·BAS。另外，驱动器号也被省略了。在这种情况下，约定为当前补缺驱动器。

文件名规定采用大写字母，这个规定适用于所有包含文件名的字串常量或变量。

同样也可采用LOAD命令来执行程序，这只要附加一个字母R在文件名的后面。例如：

LOAD"PAYROL", R

这种LOAD命令的格式将把程序装入内存并立即执行它，好象输入了RUN命令一样。所有已经打开的文件保持打开，以供刚装入的文件使用。

7. 在硬拷贝设备中开列BASIC-80程序

在程序设计中，有时会需要BASIC-80程序的硬拷贝清单。把 BASIC-80 程序送到硬拷贝设备和把程序送到控制台非常相似，这可使用LLIST命令。

LLIST命令的格式为：

LLIST

它将当前的程序清单送到硬拷贝设备中，同时我们也能够规定列出行号的范围，例如，为了列出单独一行，可以使用命令：

LLIST 100

它仅仅将100行号的语句列出。也可以指定一个行号范围：

LLIST 100-500

这将列出从行号100到500（包括100和500）的所有程序行的内容。

LLIST命令将输出直接送到CP/M设备 LST:。可以将几台不同的物理设备赋值给这台逻辑设备。有关这方面的操作请参阅CP/M手册。

第二章　表达式

表达式就是一串由BASIC-80解释程序求值的符号。表达式由数值变量或字串变量、数值常量或字串常量以及各种函数组成。表达式中的操作数可以是一个，也可以由算术运算符、逻辑或关系运算符将多个操作数结合在一起。本章将说明组成表达式和计算表达式的各种规则。

第一节　常　　量

常量是BASIC-80执行期间所使用的真实的值。常量有两种类型：字串常量和数值常量。

1. 字串常量

字串常量是一个包括在两个引号之中，由1到255个字符所组成的序列。字串常量举例如下：

"HELLO"

"25,000.00"

"Number of Employees"

2. 数值常量

数值常量可以是正数或负数。BASIC语言中的数值常量不能有逗号。数值常量有五种类型：

1) 整数

整数的范围在－32768和＋32767之间。整数不能带十进制小数点。

2) 定点数

定点数是正的或负的实数，定点数带有十进制小数点。

3) 浮点数

浮点数是正的或负的以指数形式表示的数（类似于科学计数法）。浮点数是由一个可带符号的整数或定点小数（尾数），附加上字母E和一个可带符号的整数（指数）所组成。浮点数允许的范围是10^{-38}到10^{+38}。例：

235.988E－7 ＝·0000235988

2359E6＝2359000000

注：双精度浮点数用字母D替代字母E。

4) 十六进制数

十六进制数由十六进制数字带上前缀＆H所组成。例：

＆H76

＆H32F

5) 八进制数

八进制数由八进制数字带上前缀＆O或＆所组成。例：

＆O 347

&1234

6） 单精度和双精度数值常量

定点和浮点数可以是单精度数，也可以是双精度数。如用双精度，则用十六位数字存储，也用多至十六位数字输出。

单精度常量可以是下列形式中的任一种：

(1) 七位或少于七位的数字，

(2) 带E的指数形式，

(3) 尾随一个感叹号（！）。

双精度常量可以是下列形式中的任一种：

(1) 八位或更多的数字，

(2) 带D的指数形式，

(3) 尾随符号（#）。

例：

单精度数	双精度数
46.8	345692811
-7.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

第二节 变 量

变量是BASIC－80程序中用以代表数值的一个名称。变量的值可以由程序员明确给定，也可以由程序中的计算结果来赋入。变量在被赋值之前，它的值约定为零。

变量名和类型标志符

BASIC-80的变量名可以为任意长，但是只有开始的40个字符是有效的。变量名中的字符，可以是字母和数字，也允许有十进制小数点，但变量名的第一个字符必须是字母。

变量名不能是保留字，但BASIC-80允许保留字是变量名的一部分。如果一个变量名以FN开始，则假定它要调用一个用户自定义函数。保留字包括所有的BASIC-80命令、语句、函数名和操作符。

变量可以表示数值或字串，字串变量名要以一个美元符号（$）结尾。例如：A$＝"SALES␣REPORT"。$是一个变量类型标志符，它宣称这个变量是表示字串的。

数值变量名可以宣称为整数、单精度或双精度数。

这些变量类型的标志符分别是：

% 整数变量

! 单精度变量

双精度变量

数值变量类型的补缺型式为单精度数。

2. BASIC-80变量名举例

PI# 表示一个双精度值。

MINIMUM！　表示一个单精度值。

LIMIT%　表示一个整数值。

宣称变量类型还有另一种方法，BASIC-80语句 DEFINT、DEFSTR、DEFSNG 和 DEFDBL 可以在程序中宣称变量名的类型。这些语句的详细讨论可参阅第四章“程序语句”。

3. 数组变量

数组就是一组或一个方阵的数据，这些数据可以由同一个变量名来代表，数据中的元素可通过由整数或整数表达式作为下标的数组变量名来查找。数组变量名中的下标个数和数组的维数相同。

例如：V(10)表示了一维数组中的一个元素，T(1,4)表示了两维数组中的一个元素，依次类推。数组最大的维数是255，每维中最多允许有32767个元素。有关更详细的资料参看第六章“数组”。

第三节　类型的转换

一旦需要，BASIC-80就对数值常量进行类型转换。转换的规则和实例具体说明如下。

如果使一个数值常量，赋值给另一类型的数值变量，那么该常数就将以变量名所宣称的类型进行存放（若让一个字串变量赋于某一数值，或反之让一个数值变量赋于某一字串，则将产生“Type mismatch”（类型不匹配）的错误）。例：

```
10A%=23.42
20PRINT  A%
RUN
23
```

在表达式计算过程中，所有的算术运算或关系运算中的操作数都被转换到同一的精度等级，即均转换到各操作数中最高的精度等级，运算的结果也是如此。例：

```
10  D#=6#/7
20  PRINT D#
RUN
 .8571428571428571
```

在上述实例中，运算是以双精度进行的，结果D#也是一个双精度数值。例：

```
10  D=6#/7
20  PRINT D
RUN
  .857143
```

在这个例子中，计算用双精度进行，结果是D（单精度），于是双精度数值经过舍入处理成为单精度数值，然后打印出来。

当一个定点数转换成整数时，小数部分要按四舍五入进行取整。例：

```
10 C%=55.88
20 PRINT  C%
```

```
RUN
 56
```

如果一个双精度变量被赋于一个单精度值，那么仅仅是前面的七位数字（经过取整）有效。因为一个单精度值只提供了七位数字的精度。

被打印出来的双精度数和原来的单精度数之差的绝对值，将小于原来单精度值的6.3E-8倍。例：

```
10  A=2.04
20  B#=A
30  PRINT A; B#
RUN
2.04   2.039999961853027
```

第四节　表达式和运算符

表达式可以是单个的字串、数值常量、变量，或常量、变量和运算符的组合，其结果产生一个单值。

运算符用来对数值进行算术或逻辑运算，BASIC-80提供的运算符有四种类型，即：

1. 算术运算符

运算符	操　作	表达式举例
∧	求　幂	X∧Y
－	求　反	－X
*，/	乘法、浮点除法	X*Y
＋，－	加法、减法	X+Y

表 2-1　算术运算符

使用括号可以改变运算进行的次序，括号内先算，在同一对括号内运算根据常规进行。比如，表达式：

```
A*  (Z－ ((Y+R)/T))↑J+VAL
```

计算的顺序是：

```
Y+R=e1
(e1/T) =e2
Z－e2=e3
e3↑J=e4
A*e4=e5
e5+VAL=e6
```

整除和取模运算

在BASIC-80中，还能使用两个附加的算术运算符：整除和取模运算。

整数除法用右斜线（＼）表示。在除法进行之前操作数按四舍五入变成整数。例如：

```
10＼4=2
25.68＼6.99= 3
```

整除的运算优先级别在乘法和浮点除法之后。

取模运算用算符MOD表示。它给出了一个整数值，也就是两整数相除的余数。例如：

```
10.4 MOD  4=2 (10\4=2得余数2)
25.67 MOD 6.99=5 (26\7=3得余数5)
```

取模运算的优先级别在整除之后。

溢出和除数为零

在计算算术表达式时，若遇到以零作为除数的情况，则将显示错误信息“Division by zero”（被零除），而把带着被除数符号的机器无穷大值（即1.70141E+38）作为除法的商，然后计算继续进行。

若在求幂运算中产生了零的负指数幂，则也将显示错误信息：“Division by zero”（被零除），并把正的机器无穷大值作为幂运算结果，然后运算继续进行。

若运算中发生溢出，那么错误信息“Overflow”（上溢出）将被显示出来，并以带正确代数符号的机器无穷大值作为结果，计算继续进行。

2. 关系运算符

关系运算符用于两个值的比较。比较的结果可以是真（−1），也可以是假（0）。比较的结果常用来决定程序的流程。

算符	测试关系	表达式
=	等于	X=Y
<>	不等于	X<>Y
<	小于	X<Y
>	大于	X>Y
<=	小于等于	X<=Y
>=	大于等于	X>=Y

表 2-2 关系运算符

等号也常用来给变量赋值。

当算术运算符和关系运算符组合在同一个表达式中时，总是首先执行算术运算。例如：

```
X+Y< (T-1) /Z
```

如果X加Y的值小于T减1除以Z的值，则关系表达式的值为真。又如：

```
IF SIN (X) <0     GOTO 100
IF I MOD  J<>0   THEN  K=L+1
```

3. 逻辑运算符

逻辑运算符用于进行多重关系的判断、位操作和布尔运算，逻辑运算按位产生“真”（非零值）或“假”（零值）的结果。在表达式中，逻辑运算在算术运算和关系运算之后进行。逻辑运算把它的操作数转换成整数，且运算的结果也是整数。操作数的范围必须在−32768到32767之间（注：当在逻辑运算中使用变量时，要把变量宣称为整数类型，可用整型数标志符%或用DEFINT语句。参阅第四章有关DEFINT的讨论），否则会发生“Overflow”（溢出）错误。

逻辑运算的结果由表2-3决定，表中算符按优先级别排列。

操作符	举　例	说　　明
NOT	NOT A	A的逻辑非。如A为真，则NOT A为假。
AND	A AND B	A和B的逻辑积。只当A和B值同时为真，A AND B 的值才为真；A和B的值只要有一个为假，A AND B的值就为假。
OR	A OR B	A和B的逻辑和。只要A和B的值有一个为真，则A OR B的值就为真；只有当A和B的值均为假时，A OR B的值才为假。
XOR	A XOR B	A和B的逻辑异或。只有当A或B中有一个为真时，A XOR B值才为真，否则为假。
IMP	A IMP B	A逻辑隐含B。仅当A是真值，并且B是假值时，A IMP B才为假，否则为真。
EQV	A EQV B	A 逻辑等于B。若A和B的值同时为真，或同时为假，则A EQV B值为真，否则为假。

表 2-3　逻辑运算符

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	XOR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

表 2-4　逻辑运算真值表

在进行逻辑运算时，参加运算的操作数被变换成带符号的二的补码形式的整数，范围从－32768到＋32767（如果操作数不在此范围内，将出错）。若两个操作数都是0或－1这

样的值，则逻辑运算的结果也是0或－1。逻辑运算是按位进行的，即结果的每一位都是由两个操作数的相应位的值所决定的。在二进制数中，位15是最高有效位，而位0是最低有效位。

这样，我们可以使用逻辑运算来测试一个字节，以求得一个特征位的状态。例如，可以使用AND操作来屏蔽机器I/O状态字中的某些位，而OR操作常用于合并两个字节，以产生某一个特殊的二进制值。下面的例子将有助于理解逻辑运算是如何进行的（在下例中，二进制数前面开头的零均未写出）：

63 AND 16=16

$63=111111_2$，$16=10000_2$，所以63 AND 16=16

15 AND 14=14

$15=1111_2$，$14=1110_2$，所以15 AND 14=14

－1 AND 8=8

$-1=1111111111111111_2$（补码），$8=1000_2$，所以－1 AND 8=8

4 OR 2=6

$4=100_2$，$2=10_2$，所以4 OR 2=6（110_2）

10 OR 10=10

$10=1010_2$，所以1010_2 OR $1010_2=1010_2$（10）

－1 OR －2=－1

$-1=1111111111111111_2$，$-2=1111111111111110_2$，所以，－1 OR －2=－1。十六个零每位求反则为十六个1，它就是－1的二的补码。

NOT X=－（X＋1）

任一整数的二的补码，等于原数每位求反加1。

6 IMP 2 =－5

$6=110_2$，$2=10_2$，所以6 IMP 2=－5

3 EQV 4 =－8

$3=11_2$，$4=100_2$，所以3 EQV 4=－8

关系表达式的逻辑运算

逻辑运算符和关系运算符一样，能够用来决定程序的流程。逻辑运算符能够连接两个或更多的关系表达式，而产生一个真值或假值来用于判断。例：

IF D<200 AND F<4 THEN 80

IF F>10 OR K>Q THEN 50

IF NOT P THEN 100

关系表达式运算的结果，可以为真（－1），或为假（0），然后这些结果被作为逻辑运算中的操作数。

4. 函数

表达式中使用的函数，是对操作数进行某种预定的运算。BASIC-80中有许多驻留在系统中的内部函数，象求平方根（SQR）和正弦（SIN）等函数。所有的BASIC-80内部函数在第七章“函数”中有详细的说明。

BASIC-80也允许由程序员自己定义函数。用户定义函数的构筑和引用也请参阅第七章“函数”。

第三章　命令状态语句

不管什么时候，当提示符“OK”显示在屏幕上时，BASIC-80就处于命令状态之中。在这种状态中，BASIC-80对输入命令立即作出响应。

很多命令在命令状态中可以使用，它们是：

AUTO	EDIT	LOAD	RESET
CLEAR	FILES	MERGE	RUN
CONT	LIST	NEW	SAVE
DELETE	LLIST	RENUM	SYSTEM

这些命令（除了CONT）也可以用在程序中。

1. AUTO（自动编行号）

格式：　AUTO ␣〈行号〉，〈增量〉

AUTO命令有启动自动编行号的功能。由于AUTO命令自动产生行号，所以我们只需键入程序的正文。

AUTO初始行号是〈行号〉，后面的每一行号逐个以〈增量〉递增。如果没有指定行号及增量，补缺值均为10。如果〈行号〉后面跟一个逗号，但是〈增量〉不指定，则约定使用上文所规定的增量值。

如果AUTO产生一个已经用过的行号，那么在行号后面就会打印出一个星号，这是为了告诉用户再输入字符将会替换已有的内容。然而，如在星号后面立即打一个回车，则原有的行被保留，并且出现下一个行号。

AUTO 用 CTRL-C 来终止，打 CTRL-C 的这一行不保留。打了 CTRL-C 后BASIC-80回到命令状态。例：

AUTO　100, 50　产生行号 100, 150, 200……

AUTO　产生行号 10, 20, 30, 40……

AUTO　500　产生行号 500, 510, 520……

2. CLEAR（变量初始化）

格式：CLEAR，〈表达式1〉，〈表达式2〉

CLEAR命令可将所有的数值变量置零，将所有字串变量置为空串。CLEAR命令也可用于设置存储区高限和BASIC-80所占有的堆栈空间的总数。

〈表达式1〉是一个存储单元（十进制表示）。如果它被指定，就设置了BASIC-80可以使用的存储高限。

〈表达式2〉预留了BASIC-80使用的堆栈空间，补缺值是256字节或现有存储单元的1/8，这要看哪个较小。

注意：在以前的Microsoft BASIC文本中，〈表达式1〉指定了字串存储区的大小，〈表达式2〉设置可用内存的高限。在本文本中，BASIC 5.0释放器动态地分配字串空间，所以没有必要规定字串存储区的内存总数。只有在没有任何空余内存供BASIC-80使用时才出现“Out of string space”（字串空间不够）的错误信息。例：

CLEAR　将所有数值变量置零，将所有的字串变量置为空串。

CLEAR，32768　置32768为BASIC-80使用的最高存储单元。

CLEAR，2000　分配2000个字节为堆栈空间。

CLEAR，32768，2000　置32768为BASIC-80使用的最高存储单元，并且分配2000个字节为堆栈空间。

3. CONT（继续执行程序）

格式：CONT

本命令用于在打了CTRL-C或执行了STOP或END语句后继续执行程序，也可以在出现错误之后继续执行程序。

程序将在中断行之后开始执行。如果断点发生在INPUT语句中的提示符后面，则程序继续执行时将会重新显示提示符（？或提示字串）。

CONT通常连同STOP用来查错。程序暂停执行后可以用命令状态语句来检查和修改变量，然后用CONT或命令状态语句GOTO来继续执行程序，用GOTO语句重新执行程序可从指定的某行开始。

如果程序中断后已经进行了编辑，则CONT命令无效。在程序中断后对程序作了改动，CONT命令也是无效的。这时，若还用CONT命令，那么错误信息“Can't continue”（不能继续执行）就会出现在屏幕上。

4. DELETE（删除程序行）

格式：DELETE ␣〈行号〉—〈行号〉

DELETE 语句用来删除内存中的程序行。

BASIC-80在执行DELETE以后总是回到命令状态。若没有〈行号〉，“Illegal function call”（非法函数调用）错误就会发生。例：

DELETE 40　　删除40行

DELETE 40-100　　删除40～100行（包括第40行及第100行）

DELETE -40　　删除到40行（包括第40行）

5. EDIT（进入编辑状态）

格式：EDIT ␣〈行号〉

EDIT 语句可使系统在指定的行号上进入编辑状态。

在编辑状态中，可以编辑一行中的某些部分而不必重新打入整行。一进入编辑状态，BASIC-80显示被编辑的行号，接着是一个空格，然后等待编辑状态子命令。

编辑状态子命令可以根据下列功能分类：

1）移动光标

2）插入文本

3）删除文本

4）寻找文本

5）替换文本

6）结束和编辑状态重启动

编辑状态子命令不显示在终端上。有些编辑状态子命令可用一个整数开头，该整数表示命令执行的次数，当开头的整数没有被指定时，它约定为1。

编辑状态子命令在第九章“编辑”中解释。

6. FILES（列出文件名）

格式：FILES “〈文件名〉”

FILES 命令用来列出驻留在磁盘中的文件名。

“〈文件名〉”必须遵照标准的CP/M命名规则。如果省略〈文件名〉，就列出当前主驱动器盘上所有的文件名。“〈文件名〉”可以是包含问号（？）的字串，该问号用来匹配文件名和扩展名中的任何字符。星号（*）用来匹配整个的文件名或扩展名。例：

FILES　　列出当前磁盘中的所有文件名。

FILES “*.BAS” 列出带扩展名 .BAS 的文件名。

FILES “B:*.*” 列出驱动器B:上的所有文件名。

注意最后一个例子，驱动器的指定用大写字母给出。MBASIC内所有有关的磁盘驱动器名都必须用大写字母给出，用小写字母指定驱动器名时将会产生“Bad file name”（错误文件名）的错误。

7. LIST（在终端上列出程序）

格式：LIST ␣〈行号〉—〈行号〉

LIST命令用来列出内存中当前的全部或部分程序，并显示在终端上。

执行LIST以后，BASIC-80总是回到命令状态。如果省略行号，则从最小的行号开始，一直显示到程序末尾，除非键入CTRL-C来中止显示过程。

如果指定某一行，那么，只有这一行显示在终端上。例：

LIST　　列出全部程序。

LIST 500　　列出第500行。

LIST 150-　　列出从第150行到程序结尾的全部程序。

LIST -100　　列出从最低行号到第100行的程序。

LIST 150-400　列出150行后到400行的程序，包括第150行和第400行。

8. LLIST（在打印机上打出程序）

格式：LLIST ␣〈行号〉—〈行号〉

LLIST命令可全部或部分地将内存中的现行程序打印在行打印机上。LLIST的可选项与LIST相同，BASIC在执行LLIST以后总是回到命令状态。

LLIST 采用132个字符的宽行打印机。见LIST举例。

9. LOAD（从磁盘中装入程序文件）

格式：LOAD “〈文件名〉”，R

LOAD 命令用来把文件从磁盘装入内存。

“〈文件名〉”是程序的CP/M文件名，补缺的扩展名为 .BAS。

LOAD 在装入指定的程序前将关闭所有打开的文件，并删除当前驻留在内存中的所有变量和程序行。

如果用上了R可选项，那么程序一装入内存就立即开始运行，而且所有原已打开的文件保持打开状态。

R可选项还可以用来连接几个程序（或一个程序中的几个模块）。信息可以通过临时磁盘数据文件在程序之间传递。例：

LOAD "STARTRK", R

LOAD "B:GAME1 .BAS"

注意：BASIC-80不把文件名转换成大写字母。因此，在指定CP/M文件名的所有语句中应该用大写字母来表示文件名。如果在目录中建立了小写的文件名，那么它只能被BASIC-80访问。

10. MERGE（归并程序）

格式：MERGE "〈文件名〉"

MERGE命令将把磁盘程序合并到内存中的现行程序中去。

"〈文件名〉"是磁盘程序的CP/M文件名。补缺的文件扩展名为.BAS。文件必须是以ASCII码形式存储的，否则，将产生"Bad file mode"（错误文件类型）的错误。

如果磁盘文件中有与内存中程序相同行号的行，那么磁盘中文件的行会复盖内存中相对应的行。合并可以认为是把磁盘中的程序行"插入"到内存的程序中去。

在执行MERGE命令以后，BASIC-80总是回到命令状态。例：

MERGE "PROG1"

MERGE "B:TEST.BAS"

11. NEW（删除现行程序）

格式：NEW

NEW命令用来删除内存中的现行程序，并清除所有的变量。在NEW命令执行后，所有的数值变量置为零，而字串变量则置为空串。

在执行NEW命令后，BASIC-80总是回到命令状态。

12. RENUM（对程序重新编号）

格式：RENUM〈新行号〉，〈老行号〉，〈增量〉

RENUM命令将程序重编行号。

〈新行号〉是所要新序号中的第一个行号，它的补缺值为10。〈老行号〉是现有的程序行号，重新编号就从这行开始，补缺值为原程序的第一个行号。〈增量〉是新的行号序列中每行之间的递增量，补缺值为10。

RENUM命令也可使GOTO、THEN、ON/GOTO、ON/GOSUB和ERL等语句中的原行号修改成相应的新行号。如果这些语句中某一句后面出现了不存在的行号，那么错误信息"Undefined line xxxxx in yyyyy"（未定义的行号）就会打印出来。错误行号（xxxxx）没有被RENUM修改，而行号yyyyy被修改了。

RENUM命令不能用来改变程序行的次序或建立比65529大的行号。在这两种情况下，都会发生"Illegal function call"（非法函数调用）的错误。例：

RENUM	对全部程序重新编号，新行号序列中的第一行行号是10，每行增量是10。
RENUM 300, 50	对全部程序重新编号，新行号序列中第一行行号是300，每行增量是50。
RENUM 1000, 900, 20	对900行起的行重新编号，新行号以1000开头，每行增量是20。

13. RESET（换磁盘）

格式：RESET

RESET命令允许把一个新的磁盘换到当前主驱动器中去。RESET不能带驱动器名，试图给一个驱动器名就会产生错误信息“Syntax error”（语法错误）。

应该在用一新的磁盘代替原来磁盘以后，才发出RESET命令，如果在换盘前发出了RESET命令，BASIC-80就会读出原磁盘的目录资料。

RESET命令的作用只是将新盘中的目录资料读入内存，而不关闭已打开的文件。例：

RESET

14. RUN（执行程序）

格式1：RUN〈行号〉

RUN命令的第一种格式用来执行当前在内存中的程序。

如果指定〈行号〉，程序就从这一行开始执行。如果没有〈行号〉，RUN命令就会从最低行号开始执行，BASIC-80在执行RUN后总是回到命令状态。例：

RUN 40　从第40行开始执行内存中的现行程序。

RUN　　执行内存中的现行程序，从最低行号开始执行。

格式2：RUN“〈文件名〉”，R

RUN命令的第二种格式用来把磁盘中的BASIC-80程序装入内存并运行它。R是任选的，如果用上它，所有已打开的数据文件保持打开。

“〈文件名〉”是磁盘上的文件名，补缺扩展名是.BAS。

“〈文件名〉”必须是一个正确的CP/M文件名，并用引号引起来。

RUN在装入指定程序以前，关闭所有打开的文件，并删除内存中的当前内容。然而，用上R可选项，则所有已打开的数据文件保持打开。例：

RUN “PROG1”　　装入并执行PROG1.BAS

RUN “B:GAME”, R　装入并执行B:GAME.BAS，已打开的数据文件保持打开。

15. SAVE（把程序写入磁盘）

格式：SAVE“〈文件名〉”，A

SAVE“〈文件名〉”，P

SAVE“〈文件名〉”

SAVE命令可把当前在内存中的程序写入磁盘。

“〈文件名〉”是一个放在引号内的字串，它符合CP/M文件名的组成规则，扩展名未写，则会补上.BAS。如果〈文件名〉已存在于磁盘上，那么这个文件将会被重写。

A可选项可把文件以ASCII码形式存储起来，否则BASIC-80将采用压缩二进制形式。虽然ASCII码形式在磁盘中占据了较大的空间，但是有些磁盘操作命令要求文件用ASCII码形式表示。例如，MERGE命令就要求使用ASCII码形式的文件。

P可选项以编码二进制形式存储文件以保护文件。当被保护的文件被RUN（或被装入）时，要想列出或编辑是不可能的。例：

SAVE “COM2”, A

SAVE “PROG”, P

16. SYSTEM（执行CP/M热启动）

格式：SYSTEM

SYSTEM 命令可关闭全部文件，然后执行 CP/M 热启动。 CTRL-C 使系统回到BASIC-80命令状态，而SYSTEM命令则使系统回到CP/M。例：

```
SYSTEM
A〉〔CP/M提示符〕
```

（假定A是现行补缺磁盘）

第四章　程序语句

程序语句可根据功能分为四类：数据类型定义、赋值和内存分配、控制以及I/O（非磁盘）。本章将对这些语句进行说明。

注意：这些程序语句也能用来作为命令状态语句。

第一节　数据类型定义语句

DEF语句宣称用某个范围内的字母打头的变量的类型。然而，用类型标志符宣称变量类型比DEF语句有更高的优先级。

如果没有数据类型宣称语句，BASIC-80假定所有未带类型标志符的变量都是单精度型变量。

1. DEFINT（宣称变量是整数型）

格式：DEFINT ␣〈字母范围〉

DEFINT语句用来宣称某个范围内的变量名是整数型。

整数型数据比单精度型或双精度型少占内存，然而被声明的整数型变量只能赋予－32768到＋32767范围内的整数值，包括这两数本身。例：

DEFINT I-N　所有以字母I、J、K、L、M、N打头的变量都定义为整数型变量。

2. DEFSNG（宣称变量是单精度型）

格式：DEFSNG ␣〈字母范围〉

DEFSNG语句用来宣称某一范围内的变量名是单精度型。

单精度型变量用7位数字精度存储，而用6位数字精度打印出来。例：

DEFSNG A-D　所有以字母A、B、C、D打头的变量都定义为单精度型变量。

3. DEFDBL（宣称变量是双精度型）

格式：DEFDBL ␣〈字母范围〉

DEFDBL语句用来宣称某一范围内的变量名是双精度型。

双精度型变量用17位数字精度存储，而用16位数字精度打印。例：

DEFDBL X-Z, A　所有以字母X、Y、Z和A打头的变量都定义为双精度型变量。

4. DEFSTR（宣称变量是字串型）

格式：DEFSTR ␣〈字母范围〉

DEFSTR语句用来声明某一范围内的变量名都是字串数据型。一个字串是一系列的字符——字母、空格、数和特殊的字符等，最长可达255个字符。例：

DEFSTR S　所有用S开头的变量都是字符串变量。

第二节　赋值和内存分配语句

1. DIM（建立数组）

格式：DIM〈带下标的变量表〉

数组定维说明语句用来设置数组下标的最大值并相应地分配内存。

如果一个数组变量名使用前没有DIM语句，那么它的下标最大值约定为10。如果用了一个比最大值还要大的下标，"Subscript out of range"（下标越界）的错误就会发生。下标的最小值总是为零，除非另外用OPTION BASE语句指定。

DIM语句将数组的各元素初始值置零。例：

```
10 DIM A(20)
20 FOR I=0 TO 20
30 A(I)=I+1
40 NEXT I
```

2. OPTION BASE（设置数组下标的最小值）

格式：OPTION BASE n

OPTION BASE语句用来宣称数组下标的最小值，补缺值为0，它可以变为1。OPTION BASE语句必须出现在DIM语句之前，如果OPTION BASE语句在一个数组定维说明语句后出现，"Duplicate definition"（重复定义）的错误就会发生。例：

```
OPTION BASE 1
```

有关数组内存分配详细情况，见第六章"数组"。

3. ERASE（从程序中删去数组）

格式：ERASE〈数组名表〉

ERASE语句用来删去程序中的数组，数组可以在它们被删除后重新定维，或者把以前分配给它们的内存空间供作他用。

如果没有首先删除某个数组而重新加以定维，那么"Duplicate definition"（重复定义）的错误就会发生。而如果试图删除一个还没有用DIM语句定义的数组，"Illegal function call"（非法函数调用）的错误就会发生。例：

```
10 DIM A(40)
20 ERASE A
30 DIM A(50)
```

4. LET（给变量赋值）

格式：LET〈变量〉＝〈表达式〉

LET语句将一个表达式的值赋给变量。

注意，词LET是任选的，因为将一个表达式的值赋给变量时，等于号就足够表达意思了。例：

```
10 LET D=12
20 LET SUM=X+Y+Z
```

或

```
10  D=12
20  SUM=X+Y+Z
```

5. REM（插入注释）

格式：REM〈注释〉

REM语句允许注释性的说明插入程序中。

REM语句是不被执行的，但当列程序时它按照键入的样子显示出来。

REM语句可以从GOTO或GOSUB语句转入，并且从REM语句后面的第一行可执行语句开始，程序继续执行。

也可以把注释添在一行程序的后面，此时在注释之前加一个单引号。例：

```
10 REM THIS IS A REMARK
20' THIS IS ALSO A REMARK
```

6. SWAP（交换变量的值）

格式：SWAP〈变量〉，〈变量〉

SWAP语句用来交换两个变量的值。

无论哪一类型的变量（整数型、单精度型、双精度型、字串）都可以被交换，但是这两个变量必须是相同类型的，否则就会发生“Type mismatch”（类型不匹配）的错误。例：

```
10 A$="ONE" : B$=" FOR " : C$="ALL"
20 PRINT A$; B$; C$
30 SWAP A$, C$
40 PRINT A$; B$; C$
    RUN
      ONE FOR ALL
      ALL FOR ONE
    OK
```

第三节　控制语句

对程序设计员来说，有两种类型的控制语句可供使用，一种控制程序执行次序，另一种用于带条件判断的程序执行。

1. 执行次序

执行次序控制语句用来改变程序行执行的次序。通常，执行从最低行号开始，并且依次往下执行，一直到最高行号为止。

执行次序控制语句让BASIC-80程序员能设计并运行程序的任何逻辑序列。

1) END（程序运行结束）

格式：END

END语句可终止程序的执行，关闭全部文件并回到命令状态。

END语句可以放在程序的任何位置以终止程序执行。与STOP语句不一样，END并不显示BREAK信息。END语句是否放在程序的结尾是任意的。在执行完END语句后，

BASIC-80总是回到命令状态。例：

```
520 IF K>1000 THEN END
```

2) FOR/NEXT（重复执行循环）

格式：FOR〈变量〉=X TO Y〔STEP Z〕

```
      ……
      NEXT〔〈变量〉〕
```

这里X、Y、Z可以是常量、变量或数值表达式。

FOR/NEXT 语句允许循环体里的一系列语句反复执行给定的次数。

〈变量〉被用来作为循环计数器。第一个数值表达式（X）是计数器的初值，第二个数值表达式（Y）是计数器的终值，第三个数值表达式（Z）是计数器每次的增量。

在FOR/NEXT循环执行之前，这三个数值要先计算出来。首先算出终值，接着算出初始值，然后循环计数器就被赋上此初始值。

试图在循环执行时改变这三个值是非法的。计数器也不应该随意乱改，否则循环将不按你所期望的执行。

在这三个数值算出以后，将检查一下初始值是否超过终止值。如果循环的初始值超过终止值，循环将不会执行。(如果STEP值是负的，初始值必须大于终止值，否则循环将不执行。)

一旦进入循环，则执行跟在FOR后面的程序行，一直执行到遇见NEXT语句为止。然后循环计数器按指定的STEP值递增，并应检查一下循环计数器的值是否大于终止值。如果不大于终止值，BASIC-80转回到FOR语句之后的语句，反复执行上面的过程。如果循环计数器的值大于终止值，则继续执行跟在NEXT后面的语句。FOR语句和NEXT语句之间的语句构成FOR/NEXT循环的范围。

如果STEP未被指定，则增量约定为1。如果STEP是负值，循环计数器的值每循环一次要递减一次。循环一直执行到循环计数器的值小于终止值为止。例：

```
10 FOR J=5 TO 1 STEP-1
20 PRINT J;
30 NEXT J
RUN
 5  4  3  2  1
OK
```

在这个循环范围内的语句将被执行5次。在这个例子中，5是初始值，1是终值，—1是增量。注意，这里初始值比终值大，因为增量值是负值，所有这些是有效的。另外，30行中变量J可从NEXT语句中省去。

```
10 FOR J=5 TO 1
20 PRINT J;
30 NEXT J
RUN
OK
```

在上面这个例子中，循环范围内的语句不执行，因为初始值比终值大，又省略了STEP值，因而步长约定为1。

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
OK
```

在这个例子中，循环执行10次，首先要定终值，终值(I＋5)是10，接下来定初始值，初始值是1。然后循环计数器被置上与初始值相等的值，因为步长值被省略掉，所以增量约定为1。

嵌套循环　FOR/NEXT循环可以嵌套，也就是一个FOR/NEXT循环可以放在另一个FOR/NEXT循环之中。

当循环被嵌套时，每个循环必须有各不相同的变量名作为它的计数器，内循环的NEXT语句必须放在外循环NEXT语句之前。如果嵌套循环在同一处结束，单独一条NEXT语句就可以用来作为所有循环的结尾。

NEXT语句中的变量可以省略，在这种情况下，NEXT语句与最近的FOR语句搭配。如果NEXT语句在它对应的FOR语句之前被遇到，那么“NEXT Without FOR”(有NEXT缺FOR)的错误就会发生，而且程序停止运行。

有效嵌套

```
FOR J=1 TO 10
  FOR I=1 TO 5
  NEXT I
NEXT J
```

无效嵌套

```
FOR J=1 TO 10
  FOR I=1 TO 5
  NEXT J
NEXT I
```

注意有效嵌套的形式，内循环完全被包含在外循环之中。

3) GOSUB/RETURN (转移到子程序)

格式：GOSUB 〈行号〉

　　RETURN

GOSUB/RETURN语句用来将控制转移到某个子程序，并从子程序返回。〈行号〉是这个子程序第一行的行号。

子程序可以在程序中调用任意次。一个子程序也可以在另一个子程序中被调用，这种子程序嵌套只受内存容量的限制。

子程序中的RETURN语句使BASIC-80返回到最近一次调用了此子程序的GOSUB后面的那条语句。子程序内可以包含多条RETURN语句。

子程序可以出现在程序中的任何位置。然而，把子程序与主程序隔离开来是一种较好的程序设计方法。为了防止意外地进入子程序，子程序前面可用STOP、END或GOTO等语句，让程序控制绕过子程序。例：

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
```

```
35 REM
40 REM THIS IS THE SUBROUTINE
45 REM
50 PRINT "SUBROUTINE␣" ;
60 PRINT "IN␣" ;
70 PRINT "PROGRESS"
80 RETURN
RUN
  SUBROUTINE IN PROGRESS
   BACK FROM SUBROUTINE
OK
```

4) GOTO (无条件转移)

格式：GOTO〈行号〉

GOTO语句可使程序无条件地跳出正常顺序而转移到指定的某一行去继续执行。

如果〈行号〉这一行是一条可执行语句，那么这条语句以及那些跟随在它后面的语句就被执行。如果它是一条不可执行的语句，就从〈行号〉后的第一条可执行语句开始执行。

如果〈行号〉未预先指定，就会显示出"Undefined line number"(未定义的行号)的错误信息。例：

```
10 GOTO 30
20 PRINT  "LINE 20"
30 PRINT  "LINE 30"
40 END
RUN
  LINE 30
OK
```

5) ON/GOTO和ON/GOSUB (定值和转移)

格式：ON〈表达式〉GOTO〈行号表〉

ON〈表达式〉GOSUB〈行号表〉

根据表达式计算得到的值，ON/GOTO语句和ON/GOSUB语句将控制转移到几个指定行号中的一个。〈表达式〉的值必须是正的，而且小于255。如果〈表达式〉的值不是整数，小数部分将四舍五入。

〈表达式〉的值决定了控制将转到行号表中的哪一行去。例如，如果表达式的值是3，则将往行号表中的第三个行号转移。

如果表达式的值是零或者比行号表中行号个数还要大，那么BASIC-80将继续执行下一条可执行语句。如果这个值是负数或比255大，则"Illegal function call"(非法函数调用)的错误就会发生。

在ON/GOSUB语句中，行号表中的每一行号应该是每个子程序的开头行号。例：

```
10 L=4
20 ON L GOTO 50, 60, 70, 80
```

```
30 END
50 PRINT "LINE 50" : GOTO 90
60 PRINT "LINE 60" : GOTO 90
70 PRINT "LINE 70" : GOTO 90
80 PRINT "LINE 80" : GOTO 90
90 STOP
RUN
LINE 80
OK
```

在这个例子中，L=4.所以控制就转移到行号表中的第四个行号去,这里,第四个行号是80。如果L>4或L=0，程序就会转移到30行去执行。

6) STOP（暂停执行）

格式：STOP

STOP语句用来暂停程序执行并回到BASIC-80命令状态。

STOP语句可以用在程序的任何位置，当遇见STOP时，下列信息就被打印出来：

```
Break in nnnn
```

跟END语句不一样，STOP语句不关闭任何文件。

BASIC-80在执行STOP语句后总是回到命令状态。可以用CONT命令使程序继续往下执行。例：

```
10 PRINT "LINE 10"
20 STOP
30 PRINT "LINE 30"
40 END
RUN
LINE 10
BREAK IN 20
OK
CONT
LINE 30
OK
```

2. 条件判断执行

条件判断执行语句用来有选择地执行一条语句或一组语句。这条语句或这组语句的执行必须满足一定的条件。

1) IF/THEN/ELSE（条件判断执行）

格式：IF〈表达式〉THEN〈语句〉ELSE〈语句〉

IF〈表达式〉GOTO〈行号〉ELSE〈语句〉

IF/THEN/ELSE语句根据表达式的结果作出一个决定程序流程的判断。

如果〈表达式〉结果是真的（即不为零），就执行THEN语句。THEN后面既可以跟一个要转去的行号，也可以跟一条或几条要执行的语句。如果要执行多条语句，这些语句之

间必须用冒号（:）隔开。

如果表达式的结果是假的（即为零），THEN语句就不被理睬，而执行ELSE语句（如果有的话）。ELSE后面既可以跟要转去的行号，也可以跟一条或多条要执行的语句。如果要执行的是多条语句，它们之间必须用冒号（:）隔开。

关键字THEN可被GOTO语句替换。在这种情况下，如果表达式的结果是真的，则程序转移到GOTO语句中指定的语句行号。例：

```
IF I THEN ? "I IS NOT ZERO" ELSE ? "I IS ZERO"
```

如果I不是零，这语句就会打印出"I IS NOT ZERO"。如果I值是零，就会打印出"I IS ZERO"信息。

```
IF X=A GOTO 100 ELSE PRINT "NOT EQUAL"
```

如果X=A，这语句就会转移到100行。如果X不等于A，就会打印出"NOT EQUAL"。

```
IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

这语句将根据变量（IOFLAG）的值把输出送到终端或行打印机。如果IOFLAG是零，输出送到行打印机；如果IOFLAG不是零，输出送到终端。

注意：在命令状态中，如果IF/THEN语句后面跟了一个行号，就要产生"Undefined line number"（未定义行号）的错误，除非预先已用间接方式，键入了含有这个行号的语句。

当用IF来检测浮点运算的值是否相等时，要记住浮点数在机器内部表现的值可能与打印出来的值不完全一样，因此检测必须考虑到精确度带来的影响。例如，要比较一个单精度型变量A和1.0是否相等，用：

```
IF ABS (A-1.0) <1.0E-06 THEN……
```

如果A值是1.0并且相对误差小于1.0E-6，那么检测结果就是TRUE（真值）。

IF语句嵌套　IF/THEN/ELSE语句可以嵌套，但必须保证使用相同个数的IF和ELSE，每一个ELSE与最靠近的还未被匹配的THEN匹配。在下面例子中，操作员用换行键入的方式（不是用RETURN键，而是用LINE FEED键——译者注），把ELSE语句包含在20行之中。例：

```
10 INPUT A
20 IF A=C THEN IF A=B THEN
    〈操作员打换行〉
   PRINT "A=B A=C" ELSE PRINT "A NOT=B"
    〈操作员打换行〉
   ELSE PRINT "A NOT=C"
30 PRINT A
```

这条嵌套的IF语句首先看看是否A=C，如果A不等于C，就执行第二个ELSE，打印出"A NOT=C"的信息，并且继续执行30行。

如果A=C，执行第一个THEN，这就导致另一个检测。这时，A与B比较，如果A不等于B，执行第一个ELSE，打印出"A NOT=B"的信息，并继续执行第30行。

如果A=B，执行第二个THEN，产生"A=B A=C"的信息，它被打印在终端上，

打印完这信息以后，继续执行第30行。

2) WHILE/WEND（条件判断循环）

格式：WHILE〈表达式〉

〈循环语句〉

WEND

WHILE…WEND语句用来在所给的条件为真时，反复执行一系列语句。

如果表达式不是零（即真值），〈循环语句〉就被执行，直到碰上WEND语句为止，然后BASIC-80又返回到WHILE语句，再检查〈表达式〉，如果它仍然不等于零（真值），以上过程又重复一遍。如果表达式的值是零（假值），则开始执行WEND下面的语句。

WHILE/WEND循环可有任意层嵌套。每个WEND与最近的WHILE相匹配。如果一条WHILE语句没有被匹配，就会出现“WHILE Without WEND”（有WHILE而无WEND）的错误；如果一条WEND语句没有被匹配，就会出现“WEND Without WHILE”（有WEND而无WHILE）的错误。例：

```
10 I=1
20 WHILE I
30 PRINT "WHILE/WEND LOOP"
40 I=0
50 WEND
60 END
RUN
WHILE/WEND LOOP
OK
```

第四节 I/O语句（非磁盘）

1. DATA（存储常数）

格式：DATA〈常数列表〉

DATA语句用来存储数值和字串常数。这些常数是用READ语句赋给变量的。

DATA语句是不可执行语句，它们可以放在程序中的任意位置。一条DATA语句可以包含许多常数，但不能超过一个逻辑行。在一个程序中，DATA语句数目不限。

READ语句会依照行号的次序从DATA语句中读数据，不管一条DATA语句中有多少数据项，也不管这些DATA语句放在程序的什么地方，所有的数据被认为构成连续的一列。

〈常数列表〉可以包含任意型式的数值常量。即：定点数、浮点数或整数（不允许数值表达式进入列表）。

只是当字串常量中包含逗号、冒号或有效打头空格及结尾空格时，才用双引号把它们引起来，否则引号可以不要。

READ语句中所列的变量的类型（数值或字串）必须与DATA语句中相应的常量类型相一致。

用RESTORE语句可以使READ语句又从头开始读DATA语句。例：

```
10 DATA 12, 3, HELLO,"GOOD BYE", 34
20 DATA 1, 2, 3, 4, 5
```

2. INPUT（来自键盘的输入）

格式：INPUT〔〈"提示"〉；〕〈变量表〉

INPUT语句用来在程序执行过程中，从键盘输入数据。

当碰到INPUT语句时，程序执行暂停，并打印出一个问号表示程序等待键入数据。

如果INPUT语句中包含〈"提示"〉，那么这提示会显示在问号之前，然后应在键盘上键入所需要的数据(可以用以逗号代替提示与变量表之间的分号的办法来消除问号)。

如果关键字INPUT后直接跟上一个分号，那么用户打入的回车，不等效于回车/换行序列（在一般情况下，用户键入的回车在机器中的效果是一个回车/换行序列——译者注)。

键入的数据赋给变量表中的变量。键入数据项的多少须与列表中的变量个数相等，键入的各数据项必须用逗号分开。

列表中的变量名可以是数值变量名，也可是字串变量名（包括带下标的变量)，每项键入数据的类型必须与变量名的类型一致。键入到INPUT语句的字串不需要用引号引起来。

在INPUT时，数据项太多或太少，或者键入了错误类型的数据（即用字串代替了数值等)，都会出现错误信息"？Redo from start"(重做)，输入值并没有赋给变量，一直到键入的数据能被接收为止。

注意：以前的MBASIC文本对非法INPUT操作的处理与此有所不同。例：

```
10 INPUT "ENTER VALVE"; X
20 PRINT X
30 END
RUN
ENTER VALVE？〔你键入〕·5
 5
OK
```

3. LINE INPUT（输入整行）

格式：LINE INPUT〔〈;〉〈"提示"〉;〉〈字串变量〉

LINE INPUT语句用来输入一整行的字串（不超过255个字符)，不用分隔符。

〈"提示"〉是一个文字串，它在接收数据之前被显示在屏幕上，不显示问号，除非它是提示中的一部分。所有提示之后到回车之前的输入字符都被赋值给〈字串变量〉。

如果关键词LINE INPUT后面直接跟上一个分号，那么用户键入的用以结束输入的RETURN不等效于回车/换行序列。

打CTRL-C可以跳出LINE INPUT，BASIC-80回到命令状态，并打印"OK"。而用CONT命令又可以重新执行这一条LINE INPUT语句。例：

```
10 LINE INPUT "NAME？--"; J$
20 PRINT J$
30 STOP
RUN
```

```
NAME？ - -〔你打入〕JONES.JACK L.
JONES.JACK L.
OK
```

4. LPRINT（输出数据到行打印机）

格式：LPRINT〈表达式序列〉

LPRINT语句用来在行打印机上打印数据。

LPRINT语句与PRINT语句相比，除了向行打印机输出外，其余都相同。

LPRINT打印宽度的补缺值是132个字节。

5. PRINT（在终端上输出数据）

格式：PRINT〈表达式序列〉

PRINT语句用来向终端输出数据（可用问号代替PRINT语句中的关键词PRINT）。

如果〈表达式序列〉省略，则打印出一个空行；如果有〈表达式序列〉，那么表达式的值被打印在终端上。序列中的表达式可是数值和字串表达式，字串常量应该用引号引起来。

打印位置　每一个打印项的位置由标点符号来确定，这些标点符号用来隔开要打印的各项。BASIC-80把一行分为若干区域，每14个空格为一个打印区。

在表达式序列中，逗号（，）使下一个值打印在下一个打印区的开头处，分号（；）使下一个值紧跟在上一个值后面打印。在表达式之间打一个空格或几个空格与打一个分号有同样的作用。

如果表达式序列用一个逗号或一个分号来结束，那么下一个PRINT语句在同一行上开始打印，并按上述的规则安排位置。如果表达式序列结尾处没有逗号或分号，那么在行结尾处必有一个回车。如果要打印的数据超过终端的行宽，BASIC-80会转移到下一个物理行去继续打印。

被打印的数字值后面总是跟一个空格。正数前面总是有一个空格，负数前面总是有一个负号。

用6位或少于6位（格式不固定）的精度表示的单精度数也以不固定的格式输出。例如，10^(－6)输出为0.000001，而10^(－7)用1E－7输出。

用16位或少于16位（格式不固定）的精度表示的双精度数也以不固定的格式输出。例如，1D-16输出为0.0000000000000001，1D-17输出则为1D-17。例：

```
10  X=5
20  PRINT  X+5, X-5, X* (-5), X ^ 5
RUN
10  0   -25   3125
OK
```

在上面这个例子中，PRINT语句中的逗号使各个值打印在每一打印区的开头处。例：

```
10  FOR  X=1  TO  5
20  J=J+5
30  K=K+10
40  ?  J; K;
50  NEXT X
```

```
RUN
5    10    10    20    15    30    20    40    25    50
OK
```

在这个例子中，PRINT语句中的分号使每个值紧接着打印在前一个值后面（不要忘记：数字后面总是跟一个空格，正数前面也有一个空格）。在40行中，问号用来代替PRINT。

6. READ（从DATA语句中读数据）

格式：READ〈变量表〉

READ语句用来从DATA语句中读数据，并把它们赋值给变量。

READ语句总是必须和DATA语句配合使用。READ语句把DATA语句中的常量赋给READ语句中的变量。

赋值是一对一的。READ语句中的变量可以是数值或字串，但必须与读到的值类型一致。如果数据类型不一致，则会产生"Type mismatch"(类型不匹配）的错误。

一条READ语句可以在一条或多条DATA语句中读数据（依次读取），而几条READ语句也可以在同一条DATA语句中读数据。

如果〈变量表〉中变量的个数超过DATA语句中常量数据的个数，"Out of data"(数据不够读）的错误就会产生。

如被指定的变量个数少于DATA语句中的数据个数，那么后面的READ语句就从第一个还没有被读过的数据开始读。如后面没有READ语句，那么对多余的数据将不予理睬。

为了从头开始重读DATA语句，可以用RESTORE语句。例：

```
10  FOR  I=1  TO  10
20  READ  A(I)
30  NEXT  I
40  DATA  3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

这个程序段从DATA语句把值读入数组A。执行完以后，A(1)的值是3，依次类推。

7. RESTORE（读指针复位）

格式：RESTORE〈行号〉

RESTORE用来让DATA语句中的读指针复位。这样，可以重新读取这些数据。

执行RESTORE语句后，下一条READ语句就会从程序中第一条DATA语句中的第一项开始读数。如果指定〈行号〉，下一条READ语句就会从这指定的DATA语句中的第一项开始读数。例：

```
10  READ  A, B, C
20  RESTORE
30  READ D, E, F
40  DATA  57, 68, 79
```

此程序段中对变量A、B、C分别赋值57、68、79。在20行中RESTORE语句将会使指针复位，结果30行中READ语句对变量D、E、F也分别赋值57、68、79。

8. WRITE（向终端输出数据）

格式：WRITE〈表达式序列〉

WRITE语句用来把数据输出给终端。

如果〈表达式序列〉省略，就会输出一个空行。如包含〈表达式序列〉，那么表达式的值输出到终端。序列中的表达式可以是数值和字串表达式，每两项之间用逗号隔开。

当输出多个被打印的项目时，每一项都有一个逗号与后一项隔开。打印的字串也加有引号。在打印序列的最后一项后，BASIC-80会插入一个回车/换行。

WRITE语句输出数值的格式与PRINT语句相同。例：

```
10  A=80 : B=90 : C$ = "BASIC-80"
20  WRITE A, B, C$
RUN
80              90              "BASIC-80"
OK
```

第五章 字 串

一个字串由一连串的字符组成，最长可达255个字符，字符包括字母、空格，数字和特殊符号。一个字串常量是指由双引号括起来的一串字符，一个字串变量可以用字串标志符$加在一个变量名后面来宣称，也可以用DEFSTR语句来宣称。

Microsoft BASIC-80提供了有关字串操作的完整功能，可以对字串进行比较、打印、显示和字串连接等基本操作。还有一些字串函数对BASIC-80程序员来说也是十分有用的。

这一章包括下面几个部分：

字串的输入/输出

字串操作

字串函数

第一节 字串的输入/输出

字串常量可以按照与数值常量同样的方法输给一个程序。INPUT语句也适用于字串，通常在键盘上输入字串时，字串不需要用引号括起来。例如：

```
10 INPUT "YOUR NAME";J$
20 PRINT "HELLO";J$
RUN
YOUR NAME? (you type) JOHN
HELLO JOHN
OK
```

然而在INPUT语句中，如果输入的字串包含逗号、冒号或头尾带空格，则这个字串必须用引号括起来。例如：

```
10 INPUT "YOUR NAME";J$
20 PRINT J$
RUN
YOUR NAME? (you type)"JONES, JOHN"
JONES, JOHN
OK
```

LINE INPUT语句可被用来输入包含逗号、冒号或头尾带空格的字串，而不必将字串用引号括起来。例如：

```
10 LINE INPUT "YOUR NAME";J$
20 PRINT J$
RUN
YOUR NAME (you type) JONES, JOHN
```

```
JONES, JOHN
OK
```

第二节　字串操作

字串可以用“+”号连接。例如：

```
10 X$ = "FIRST"
20 Y$ = "AND"
30 Z$ = "LAST"
40 PRINT X$ +Y$ +Z$
RUN
FIRST AND LAST
OK
```

用来比较数值的关系运算符=、〈 〉、<、>、<=、>=同样可以用来对字串进行比较。

字串是从左向右逐个字符按ASCII码值进行比较的。ASCII码值小的字符排在前面。

例如，字串“Z$”排在字串“Z*”的前面，因为“$”(ASCII码值为十进制36) 比“*”号 (ASCII码值为十进制42) 小。

不同长度的字串比较时，如果能比的各个字符都相同，则短的字串排在前面。字串比较中，每个字符，包括空格和任何不可显示字符都是有效字符。例如“AB”排在“AB␣”前面，因为字串“AB␣”中最后有一个空格。

一个字串常量无论是用在赋值语句中还是用在比较表达式中，都必须用双引号将字串括起来。例如：

```
Z$ = "STRING CONSTANT"
IF Z$ = "NUMERIC CONSTANT" THEN PRINT Z$
```

第三节　字串函数

下面这些字串函数可供BASIC-80程序员使用：

函数	意义
ASC (X$)	字串转换成ASCII码值
CHR$ (I)	ASCII码值转换成字串
HEX$ (X)	十进制转换成十六进制
INKEY$	从终端读入一个字符
INPUT$(X,Y)	读字符 (可读若干个)
INSTR(I,X$,Y$)	检索子串
LEFT$(X$, I)	返回最左面的字符
LEN (X$)	字串长度
MID$(X$,I,J)	返回子串

MID$(X$,I,J)=Y$	取代部分字串
OCT$ (X)	十进制转换成八进制
RIGHT$(X$,I)	返回最右面的字符
SPACE$(X)	返回空格串
STR$(X)	将数值转换成字串型式
STRING$(I,J)	建立字串
STRING$(I,X$)	建立字串
VAL (X$)	将字串转换成数值型式

表 5-1 字串函数

1. ASC（将字串转换成ASCII码值）

格式：ASC (X$)

该函数返回字串X$的第一个字符的ASCII码(十进制)。如果X$是空串，则返回错误信息 "Illegal function call"(错误函数调用)。例如：

```
10 X$ = "TEST"
20 PRINT ASC (X$)
RUN
84
OK
```

上例中，字串X$的第一个字母是T，T的ASCII码（十进制）是84。

2. CHR$（ASCII码值转换成字串）

格式：CHR$ (I)

CHR$函数返回ASCII码值等于I的字符（ASCII码表在附录B中列出)。CHR$通常用来把一个特殊字符送到终端。例如，发送BEL字符可用语句PRINT CHR$ (7)。例如：

```
PRINT CHR$ (66)
B
OK
```

3 HEX$（十进制转换成十六进制）

格式：HEX$ (X)

该函数把一个十进制的自变量转换成用十六进制表示的字串，X在转换以前被四舍五入成一个整数。例如：

```
10 INPUT X
20 A$ =HEX$ (X)
30 PRINT X;"DECIMAL IS";A$;"HEXADECIMAL"
RUN
? 32
32 DECIMAL IS 20 HEXADECIMAL
```

4. INKEY$（从键盘上读入一个字符）

格式：INKEY$

INKEY$函数或者返回一个单字符的字串（即从终端读入的字符），或者返回一个空串（当未从终端接收到字符时）。屏幕不显示输入字符，而且任何字符都可以输入计算机，只有CTRL-C例外，它将使BASIC-80返回到命令状态。例如：

```
10 X$=INKEY$
20 IF X$=CHR$(32) THEN STOP
30 GO TO 10
```

这个例子将从键盘上读入字符，直到键入一个空格（十进制ASCII码值为32）为止。

5. INPUT$（读字符）

格式：INPUT$ (X, Y)

INPUT$函数返回一个含有X个字符的字串。这个字串从终端上或从文件号Y中读入，如果终端作为输入设备，那么字符不被显示，而且所有控制字符都可以输入，只有CTRL-C是例外，它中断INPUT$函数的执行。例如：

```
10 OPEN "I",1,"DATA DAT"
20 IF EOF(1) THEN 50
30 PRINT INPUT$(1,1)
40 GOTO 20
50 END
```

上面这个例子将打印出DATA.DAT文件的全部字符。又例如：

```
10 X$=INPUT$(1)
20 IF X$= "P" THEN 500
30 IF X$= "S" THEN 700 ELSE 10
```

这个例子将从键盘读入一个字符，如这个字符是P，程序转移到500行上去执行；如这个字符是S，程序转移到700行上去执行；如该字符既不是P，也不是S，则程序返回第10行。

6 INSTR（检索子串）

格式：INSTR (I, X$, Y$)

该函数在X$中寻找Y$第一次出现的位置，并返回该值。I是可选项，它是一个偏差量，用来预置寻找的起始位置。I必须在1～255之间。如果I>X$的长度，或X$是空串，或找不到Y$，INSTR返回0，如果Y$是空串，INSTR返回I或1。

X$和Y$可以是字串变量，字串表达式或文字串。例如：

```
10 X$= "ABCDEB"
20 Y$= "B"
30 PRINT INSTR(X$, Y$), INSTR(4,X$,Y$);
RUN
 2  6
OK
```

7. LEFT$（返回靠左边的字符）

格式：LEFT$ (X$, I)

LEFT$函数返回一个字串，它由X$最左方的I个字符组成。I必须在0～255之间。如果I大于X$的长度，则返回整个字串X$，如果I等于0，则返回长度为0的空串。例如：

```
10 A$= "BASIC-80"
20 B$=LEFT$(A$,5)
30 PRINT B$
RUN
BASIC
OK
```

8. LEN（返回字串长度）

格式：LEN (X$)

该函数返回字串X$中字符的个数，不可把显示字符和空格都算在内。例如：

```
10 X$= "ABC DEF"
20 PRINT LEN(X$)
RUN

OK
```

9. MID$（返回子串）

格式：MID$ (X$, I, J)

该函数返回一个字串，字串内容为字串X$中第I个字符开始向右的J个字符。I和J的范围为0～255。如果J参数省略，或第I个字符的右边少于J个字符，则从第I个字符开始向右的全部字符被返回。如果I大于X$的长度，则返回一个空串。例如：

```
10 A$= "GOOD"
20 B$= "MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$, 8, 8)
RUN
GOOD EVENING
OK
```

10. MID$（取代部分字串）

格式：MID$ (X$, I, J) =Y$

MID$函数的这种形式是用字串Y$代换字串X$中的一部份。

从字串X$中第I个字符开始，用字串Y$代换。字串Y$实际用来代换的字符个数可以用可选项J来确定。

无论J参数省略与否，字串代换以后不会改变字串X$原来的长度。例如：

当A$= "1234567" 时

执行语句	对应结果
MID$ (A$, 3, 4)="ABCDE"	12ABCD7
MID$ (A$, 5)="ABCDE"	1234ABC
MID$ (A$, 1, 2)="A"	A234567

11. OCT$（十进制转换成八进制）

格式：OCT$ (X)

该函数将返回一个和十进制自变量对应的由八进制表示的字串。X在OCT$(X)转换

前四舍五入成一个整数。例如：

```
PRINT OCT$ (24)
 30
OK
```

12 RIGHT$ (返回串右边的字符)

格式：RIGHT$ (X$ I)

该函数将返回字串X$中最右边的I个字符。如果I等于X$的长度，则返回整个字串。如果I等于0，则返回长度为0的空串。例如：

```
10 A$="DISK BASIC-80"
20 PRINT RIGHT$ (A$, 8)
RUN
BASIC-80
OK
```

13. SPACE$ (返回空格串)

格式：SPACE$(X)

该函数将返回长度为X的空格串。表达式X被四舍五入为一个整数，而且范围必须为0～55。例如：

```
10 FOR I=1 TO 5
20 X$=SPACE$ (I)
30 PRINT X$; I
40 NEXT I
RUN
   1
     2
       3
         4
           5
OK
```

14 STR$ (返回字串形式)

格式：STR$ (X)

该函数返回数值X的字串表示形式。例如，如果X=45.3，则STR$ (X) 就是表示"␣45.3"这个字串。前面留有一个空格是用来表示45.3的符号的。算术运算可以在数值X上执行，但不能在字串STR$ (X)上执行。例如：

```
PRINT STR$ (100)
100
PRINT STR$ (-100)
-100
```

15. STRING$ (建立字串)

格式：STRING$ (I, J) 或STRING$ (I, X$)

该函数返回一个长度为I的字串，字串中的字符是ASCII码值J所对应的字符，或是字串X$中的第一个字符。I和J必须用十进制表示，范围是0～255。例如：

```
PRINT STRING$ (10, "*")
**********
PRINT STRING$ (15, 65)
AAAAAAAAAAAAAAA
```

16. VAL（返回数值形式）

格式：VAL (X$)

VAL函数将返回字串X$的数值表示形式。该函数扫描字串X$时跳过X$中所有的空格，tab键和换行。如果X$的第一个字符不是+、-、&和数字，那么VAL (X$) =0。&表示八进制数，VAL函数将把八进制数转换成十进制数返回。如果X$兼含有数字和字母，则仅仅前面的数字字符被转换成数值表示形式。例如：

```
PRINT VAL ("100 FEET")
100
PRINT VAL ("FEET 100")
0
PRINT VAL ("&100")
64
PRINT VAL ("-3")
-3
```

第六章　数　组

这一章介绍建立和使用数组的方法。数组是数据项的简单的顺序列表，它可以是一维向量数组，也可以是由行和列组成的表格。

数组项可以是字串和数值。每一项称为一个“元素”。在这一章里将通过举例来帮助理解数组的概念。

本章还列举了几个示范程序用来说明数组的操作。如加法、乘法、转置和其他数值数组的操作。

第一节　数组操作

1. 数组的宣称（DIM）

一个数组被调用以前，要由数组宣称语句（即DIM语句）对数组宣称。DIM 语句指定一个数组中可含元素的最多个数。DIM语句的格式如下：

DIM〈名字〉(〈整型表达式〉)

其中：〈名字〉是在BASIC-80中有效的变量名。〈整型表达式〉是有效的整型表达式，求值时，被四舍五入成一个正整数。该值表示称为〈变量名〉的数组可含元素的最多个数。维数可多达255，每一维中可含元素最多达32767。例：

```
DIM A (3),D$ (2, 2, 2)
DIM Q1(R+T)
DIM Z# (100)
```

一个数组可以不经宣称而直接使用，BASIC-80在处理未用 DIM 语句宣称的数组元素时，规定最大下标为10，也就是说，数组可以不经DIM语句宣称而建立起来。

2. 数组下标

数组中的每一个元素可以由跟在数组名后面的数组下标来唯一确定。数组下称可以是一个整型表达式。例：

```
A (1),D8 (I, J, K)
Q1 (2)
Z# (55)
```

用一个负值的下标来调用数组某元素，将导致“Illegal function call”（非法函数调用）的错误。如果调用数组时下标值大于 DIM语句中宣称的最大下标或调用时使用了太多或太少个数的下标，都会导致“Subscript out of range”（下标越界）的错误。

3. 选择基准语句

数组元素的最小下标值约定为0。宣称为A (10) 的一个数组实际上是一个含有11个元素的数组A (0)～A (10)。OPTION BASE语句用来改变数组下标最小值的补缺值，从0改为1。下面的例子说明了OPTION BASE语句的用法。例：

```
OPTION BASE 1
```

DIM A (10)

这段程序建立了一个含有A（1）～A（10）的10个元素的数组。OPTION BASE语句必须出现在DIM语句之前，或者在任何带下标变量被调用之前。在数组建立以后再使用OPTION BASE语句会导致“Duplicate definition”(重复定义)的错误。

4. 向量数组

向量数组就是一维数组。这类数组可以由DIM语句建立，也可以由BASIC-80自行建立（补缺形式)。假定数组A已建立为补缺形式的数组，BASIC-80将分配存储单元如下：

数组元素	带下标变量
元素＃1	A（0）
元素＃2	A（1）
元素＃3	A（2）
元素＃4	A（3）
元素＃5	A（4）
元素＃6	A（5）
元素＃7	A（6）
元素＃8	A（7）
元素＃9	A（8）
元素＃10	A（9）
元素＃11	A（10）

表6-1 数组存储分配

变量A（9）将对应向量数组A的第10个元素（如果用OPTION BASE语句将最小下标值置成1，则A（9）将对应数组的第9个元素)。

5. 多维数组

宣称二维数组的方法和宣称向量数组相同，只是要宣称行和列两个尺寸。例如，宣称一个3×3的数组，可用下面给出的语句：

OPTION BASE 1

DIM A (3, 3)

上面的语句执行以后，BASIC-80将给数组开辟9个存储单元（注意，最小下标值已由OPTION BASE语句置成为1)。

上例中数组A的存储分配如下：

行＼列	1	2	3
1	A(1, 1)	A(1, 2)	A(1, 3)
2	A(2, 1)	A(2, 2)	A(2, 3)
3	A(3, 1)	A(3, 2)	A(3, 3)

表6-2 多维数组存储分配

BASIC-80分配数组元素存储单元时是按行分配的，所以在从左向右地观察上面的数组时，会发现第二个下标值的变化比第一个频繁。

建立字串数组的方法和建立数字数组的方法相同。可以用DIM语句来宣称一个字串数

组。例如，

DIM A$(100)

这条语句建立了一个含有101个元素的字串数组。存取数组某个元素时，只需要在变量名（数组名）后面跟上一个数组下标即可。如，

A$ (20) = "ASTRING ARRAY"

第二节　矩阵运算

下面收集了一些子程序，它们对进行矩阵运算是很有用的。这些子程序用在你的主程序里时行号可以改变，从而使子程序和主程序可以取得一致。

1. 矩阵输入子程序

```
5000 'SUBROUTINE NAME -- MATIN 2
5010 'ENTRY I%=#OF ROWS, J%=#OF COLUMNS
5020 DIM MAT (I%, J%)
5030 FOR K%=1 TO I%
5040 PRINT "INPUT ROW #", K%
5050       FOR L%=1 TO J%
5060       INPUT MAT (K%, L%)
5070 NEXT L%, K%
5080 RETURN
```

上面这个子程序将从终端接收数据，并将数据赋给一个二维数组MAT 进入该子程序时，整数变量I%应该含有该矩阵的行数，J%应该含有矩阵的列数。

```
5000 'SUBROUTINE NAME -- MATIN 3
5010 'ENTRY          I%=SIZE OF DIMENSION #1
5020 '               J%=SIZE OF DIMENSION #2
5030 '               K%=SIZE OF DIMENSION #3
5040 DIM MAT (I%, J%, K%)
5050 FOR L%=1 TO I%
5060       FOR M%=1 TO J%
5070             FOR N%=1 TO K%
5080 READ MAT (L%, M%, N%)
5090 NEXT N%, M%, L%
6000 RETURN
```

上面的子程序从DATA语句中读取数据，并将数据赋给一个三维数组MAT。进入该子程序时，整数变量I%应含有第一维的元素个数，J%应含有第二维的元素个数，K%应含有第三维的元素个数。数据应开列在DATA语句中。

2. 标量乘（乘以单个变量）

```
5000 'SUBROUTINE NAME -- MATSCALE
5010 'ENTRY --          I%=SIZE OF DIMENSION #1
```

```
5020 '                    J%=SIZE OF DIMENSION # 2
5030 '                    K%=SIZE OF DIMENSION # 3
5040 '       A--ORIGINAL ARRAY
5050 '  X--SCALAR FACTOR
5060 '  B--NEW ARRAY
5070 FOR L%=1 TO K%
5080   FOR M%=1 TO J%
5090     FOR N%=1 TO I%
6000     B (N%, M%, L%) =A (N%, M%, L%)*X
6010     NEXT N%
6020   NEXT M%
6030 NEXT L%
6040 RETURN
```

这个子程序把一个三维数组A中的每一个元素都乘以变量X的值，形成一个新的三维数组B。进入该子程序时，I%应含有第一维的元素个数，J%应含有第二维的元素个数，K%应含有第三维的元素个数，X应被赋值而作为乘法的因子。数组A和B都应该预先用DIM语句定义。

3. 矩阵转置

```
5000 'SUBROUTINE  NAME - MATTRANS
5010 'ENTRY I%=#OF ROWS, J%=#OF COLUMNS
5020 'TRANSPOSE A INTO B
5030 FOR K%=1 TO I%
5040   FOR L%=1 TO K%
5050   B (L%, K%) =A (K%, L%)
5060   NEXT L%
5070 NEXT K%
5080 RETURN
```

这个子程序将把一个二维矩阵A转置成另一个二维矩阵B。进入该子程序时，I%应含有行数，J%应含有列数。数组A和B应预先用DIM语句定义。

4. 矩阵加法

```
5000 'SUBROUTINE  NAME - MATADD
5010 'ENTRY -             I%=SIZE OF DIMENSION # 1
5020 '                    J%=SIZE OF DIMENSION # 2
5030 '                    K%=SIZE OF DIMENSION # 3
5040 'ARRAY A+B=C
5050 FOR L%=1 TO K%
5060   FOR M%=1 TO J%
5070     FOR N%=1 TO I%
5080     C (N%, M%, L%) =B (N%,M%,L%) +A (N%,M%,L%)
```

```
5090      NEXT N%
6000    NEXT M%
6010  NEXT L%
6020  RETURN
```

数组A、B、C必须预先用DIM语句定义。

5. 矩阵乘法

```
5000  'SUBROUTINE  NAME — MATMULT
5010  'ENTRY  — ARRAY A MUST BE D1% BY D3% ARRAY
5020  '          ARRAY B MUST BE D3% BY D2% ARRAY
5030  '          ARRAY C MUST BE D1% BY D2% ARRAY
5040  FOR I%= 1 TO D1%
5050    FOR J%= 1 TO D2%
5060    C (I%, J%) = 0
5070      FOR K%= 1 TO D3%
5080      C (I%, J%) =C (I%, J%) +A (I%,K%)*B(K%,J%)
5090      NEXT K%
6000    NEXT J%
6010  NEXT I%
```

这个子程序将二维数组A乘以二维数组B，生成数组C。

第七章　函　数

BASIC-80提供一整套内部函数供程序员使用。其中一组是算术函数，它们用符号名来表示。当其被调用时，将返回一个单值，这个值或是整数型或是单精度型。算术函数的自变量用圆括号括起。

BASIC-80还拥有一组特殊函数。这些特殊函数在调用时，都有各自的调用条件。

BASIC-80还可让用户自己来构造和调用一个函数。

第一节　算术函数

可供BASIC-80程序员使用的算术函数有：

函数	定义
ABS (X)	绝对值
ATN (X)	反正切
CDBL (X)	转换成双精度
CINT (X)	四舍五入为整数
COS (X)	余弦
CSNG (X)	转换成单精度
EXP (X)	e的X次幂
FIX (X)	截去自变量的小数部份
INT (X)	小于等于X的最大整数
LOG (X)	x的自然对数
RND (X)	0～1之间的随机数
SGN (X)	X的符号（+、－、或0）
SIN (X)	正弦
SQR (X)	平方根
TAN (X)	正切

表 7-1　算术函数

1. ABS（绝对值）

格式：ABS (X)

返回表达式X的绝对值。例：

```
PRINT ABS (7 * (－ 5 ))
 35
OK
```

2. ATN（反正切）

格式：ATN (X)

返回X的反正切值，结果为弧度，并在$-\pi/2 \sim \pi/2$之间。表达式X可以是任何数值类

型，但ATN求值总是按单精度进行。例：

```
10 X=3
20 PRINT ATN (X)
RUN
 1.24905
OK
```

3. CDBL（转换成双精度）

格式：CDBL (X)

CDBL函数将X转换成双精度数。例：

```
10 X=454.67
20 PRINT X;CDBL (X)
RUN
 454.67   454.6700134277344
OK
```

4. CINT（转换成整数）

格式：CINT (X)

CINT函数将X转换成整数。X的小数部分被四舍五入，返回最接近于X的整数。如果返回值不在－32768～32767之间，"Overflow"（溢出）的错误就会发生。例：

```
PRINT CINT (45.67)
 46
OK
```

5. COS（余弦）

格式：COS (X)

返回X的余弦值，X须用弧度表示。余弦计算用单精度进行。例：

```
10 X=2 * COS (.4)
20 PRINT X
RUN
 1.84212
OK
```

6. CSNG（转换成单精度）

格式：CSNG (X)

将X转换成单精度数。例：

```
10 A#=975.3421#
20 PRINT A#;CSNG (A#)
RUN
 975.3421   975.342
OK
```

注："#"号用来标志双精度数据类型。

7. EXP（e的幂）

格式：EXP (X)

返回e的X次幂，e是自然对数的底(2.71828···)。X必须小于等于87.3365。如果EXP溢出，则显示错误信息“Overflow”（溢出）。例：

```
10 X= 5
20 PRINT EXP (X-1)
RUN
  54.5982
OK
```

8. FIX（截去自变量的小数部分）

格式：FIX (X)

返回X的整数部份。FIX和INT的主要区别在于：FIX 仅仅是简单地截去X数的小数部分，而INT会将一个负数X变成一个小于等于X而又最接近于X的整数。例：

```
PRINT FIX (58.75)
  58
OK
PRINT FIX (-58.75)
-58
OK
```

9. INT（取整）

格式：INT (X)

返回一个小于等于X的最大整数。当X是负数时也是一样。例：

```
PRINT INT (99.89)
  99
PRINT INT (-12.11)
-13
```

10. LOG（自然对数）

格式：LOG (X)

返回X的自然对数值。X必须大于0。如果X≤0，则出现错误信息“Illegae function call”（非法函数调用）。例：

```
PRINT LOG (45/7)
 1.86075
```

11. RND（随机数发生器）

格式：RND (X)

返回一个0～1之间的随机数。如果用某一段程序生成一串随机数，那么程序每次运行时，都将生成同样的一串随机数，除非在随机数发生器中重新“埋种”。RANDOMIZE 语句就是用来重新“埋种”的。

如果X＜0，随机数序列重新开始；当X＞0或省略时，将产生序列中的下一个随机数；X＝0将重复上一个生成的随机数。例：

```
10 RANDOMIZE PEEK (11)
```

```
20 FOR I=1 TO 5
30 PRINT INT (RND*100);
40 NEXT
RUN
  24  30  31  51  5
OK
```

注：这个示范程序每次被执行时将产生不同序列的随机数。

12. RANDOMIZE（在随机数发生器中重新“埋种”）

格式：RANDOMIZE〈表达式〉

RANDOMIZE语句用来在随机数发生器中重新“埋种”。〈表达式〉被用来作为随机数“种子”。如果〈表达式〉被省略，BASIC-80将挂起执行的程序，用提示语句询问此值：

```
Random Number Seed (−32768 to 32767)?
```

这时输入的值将用来作为随机数“种子”。

如果随机数发生器没有重新“埋种”则程序每次被执行时，RND函数返回同样的随机数序列。为了改变这种情况，可把RANDOMIZE语句放在程序的头上，并在每次运行时改变它的自变量。

13. SGN（表达式符号）

格式：SGN (X)

根据X的值给出其符号。

如果X＜0，SGN (X) 返回－1；如果X＝0，SGN (X) 返回0；如果X＞0，SGN (X) 返回1。

可以利用SGN (X) 函数来构成下面这样的分支语句：

```
ON SGN (X) + 2 GOTO 100, 200, 300
```

如果X是负值，程序将转移到100行；如果X是零，程序将转移到200行；如果X是正值，程序将转移到300行。例：

```
10 INPUT X
20 ON SGN (X)+2 GOTO 50, 60, 70
50 PRINT "NEGATIVE":GOTO 10
60 PRINT "ZERO":GOTO 10
70 PRINT "POSITIVE":GOTO 10
RUN
? −10
NEGATIVE
?  0
ZERO
?  10
ROSITIVE
OK
```

14. SIN（正弦）

格式：SIN (X)

返回X的正弦值。X必须用弧度表示。SIN(X)用单精度计算。例：

```
PRINT SIN(1.5)
 .997495
OK
```

15. SQR（平方根）

格式：SQR(X)

返回X的平方根值。X必须大于等于0。如果X＜0，显示错误信息“Illegal function call”（非法函数调用）。例：

```
10 X=25
20 PRINT X, SQR(X)
RUN
 25          5
OK
```

16. TAN（正切）

格式：TAN(X)

返回X的正切值，X必须用弧度表示，TAN(X)用单精度计算。如果TAN溢出，显示错误信息“Overflow”（溢出）。例：

```
PRINT TAN(10)
 .64836
OK
```

第二节　数学函数

有些函数可以利用BASIC-80的内部运算函数计算而得，现提供于下面：

函数	BASIC-80 求法
正割	SEC(X)=1/cos(X)
余割	CSC(X)=1/SIN(X)
余切	CTG(X)=1/TAN(X)
反正弦	ARCSIN(X)=ATN(X/SQR(−X＊X+1))
反余弦	ARCCOS(X)=−ATN(X/SQR(−X＊X+1)+1.570796
双曲正弦	SINH(X)=(EXP(X)−EXP(−X))/2
双曲余弦	COSH(X)=(EXP(X)+EXP−X))/2
双曲正切	TGH(X)=(EXP(X)−EXP(−X))/(EXP(X)+EXP(−X))
双曲正割	SECH(X)=2/(EXP(x)+EXP(−X))
双曲余割	CSCH(X)=2/(EXP(X)−EXP(−X))
双曲余切	CTGH(X)=(EXP(X)+EXP(X))/(EXP(X)−EXP(−X))

表 7-2　数学函数

第三节　特殊函数

下面这些特殊函数可供BASIC-80程序员使用。

函数	意义
FRE(X)	自由内存空间数目
INP(I)	从通道口输入
LPOS(X)	打印头的位置
NULL(X)	设置空字节的数目
OUT I, J	从通道口输出
PEEK(I)	从内存读一字节
POKE I,J	写一字节到内存
POS(X)	当前光标位置
SPC(X)	打印空格
TAB(I)	表格控制
VARPTR(X)	变量指针
WAIT I,J,K	通道口状态
WIDTH I	预置终端行宽
WIDTH LPRINT I	预置打印机行宽

表 7-3　特殊函数

1. FRE（返回自由内存空间量）

格式：FRE(0)　　FRE(X$)

该函数返回BASIC-80没有使用的内存空间的总字节数。自变量为虚拟自变量。

FRE（" "）迫使系统在返回自由空间值之前进行某些内务整理，时间约需1～2分钟。通常BASIC-80不自动进行内务整理，除非所有空间都被占用。例：

```
PRINT FRE(0)
```

2. INP（从I/O口输入字节）

格式：INP(I)

INP返回从口I读入的字节。I必须在0～255间。和INP相对的函数是OUT。例：

```
10  A=INP(255)
```

3. LPOS（返回打印头位置）

格式：LPOS(X)

返回行打印机打印头在缓冲区里的现行位置。这一位置并不一定和打印头的实际物理位置相对应。X是一个虚拟自变量。例：

```
100IF LPOS(X)>60  THEN LPRINT CHR$(13)
```

4. OUT（输出一个字节到I/O口）

格式：OUT I,J

OUT语句送一个字节到一个输出口上。I和J必须是整数表达式，在0～255之间。I是输出口号码，J是被传送的数据。例：

```
100 OUT32, 100
```

5. PEEK（检查某内存单元中的内容）

格式：PEEK(I)

返回从内存单元I中读出的字节。返回值是个十进制整数，在0～255之间，I在0～65536之间。和PEEK相对的函数是POKE。例：

```
PRINT PEEK (34000)
 234
OK
```

注：在PEEK内存单元34000时，不一定得到这个结果。

6 POKE（改变内存单元内容）

格式：POKE I, J

POKE I, J

POKE函数将改变某内存单元中的内容。I和J必须是整型表达式。

I是将改变内容的内存单元的地址，在0～65535之间。

J是放入内存单元I中的数据，必须在0～255之间。

POKE和PKKE能高效率地进行数据存储，用这二个函数可以安装汇编语言子程序，向汇编语言子程序传送参数，或从子程序中传回结果。例：

```
POKE 34000, 1
OK
```

7. POS（返回当前光标位置）

格式：POS(I)

返回当前光标位置。光标最左位置为1。I是虚拟自变量。例：

```
IF POS(I)>60 THEN PRINT CHR$(13)
```

8. SPC（打印空格）

格式：SPC(I)

SPC函数用来在终端或行打印机上打印空格。整数变量I指定了被打印的空格数。I必须在0～255之间。SPC函数仅仅能用在PRINT和LPRINT语句中。例：

```
PRINT "OVER"; SPC (15); "THERE"
OVER                    THERE
OK
```

9. TAB（表格控制）

格式：TAB(I)

TAB语句使光标或打印头移到第I格的位置。如果光标或打印头的现行位置已经在第I格的右边，那么TAB语句将使光标或打印头转到下一行的第I格位置上。

最左边一格位置为1，最右边一格位置为宽度减1。I必须是不带符号的整型表达式，范围为1～255。TAB函数只可以用在PRINT和LPRINT语句中。例：

```
10 PRINT "NAME";TAB(10);"AMOUNT"
20 READ A$, B$
30 PRINT A$;TAB(10);B$
```

```
40 DATA "WILLIAMS", "$20.00"
RUN
NAME                AMOUNT
WILLIAMS            $20.00
```

10. VARPTR（变量指针）

格式1：VARPTR(〈变量名〉)

格式2：VARPTR(#〈文件号〉)

对于格式1，VARPTR 函数返回一个地址值，该地址表示〈变量名〉中变量在内存中的存储单元地址。必须预先对〈变量名〉赋值。否则执行该函数时会发生"Illegal function call"（非法函数调用）的错误。

任何类型的变量名（数字、字串、数组等)都允许，返回结果是在－32768～32767 之间的一个整数。如果返回的是负数，把它加上65536就得到实际的地址。这返回地址（我们假定为A）对于不同类型的变量有不同的含义。

注：下面这些具体例子的结果取决于你的系统有多少内存，BASIC-80已占用的内存有多少，等等。

如果〈变量名〉是一字串变量，则：

A——含字串长度

A＋1——含字串实际起始地址低字节（LSB)

A＋2——含字串实际起始地址高字节（MSB)

存储字串的实际地址可以这样计算：

实际地址＝(A＋2)＊256＋(A＋1)

这个地址很可能处于内存的高地址区域，即字串变量存储区。

如果这个字串是一个常量（一串文字)，那么A将表示这个字串常量所在的程序行的实际存储区域(记住：A仅仅是有关信息的地址，必须用 PEEK(A)来获得A中的实际内容)。例：

```
X$="ABC"   〔you type〕
OK
PRINT VARPTR(X$)  〔you type〕
-23927
OK
```

如果〈变量名〉是一个整数值，则：

A——含有 2 字节整数的低字节（LSB)

A＋1——含有 2 字节整数的高字节（MSB)

要显示此整数的信息(2的补码的十进制表示法)，可用PRINT PEEK(A)和PRINT PEEK(A＋1)。例：

```
I% = 1000 〔you type〕
OK
PRINT VARPTR(I%)  〔you type〕
-29121
```

OK

如果〈变量名〉是单精度数值，则：

A——含有值的低字节（LSB）

A+1——含有值的次高字节（NEXT MSB）

A+2——高字节（MSB），但其中最高位为符号位。

A+3——用加128形式表示的幂指数（即将实际指数值加上128作为存储形式）。

如果〈变量名〉是双精度数值，则：

A——含有值的低字节（LSB）

A+1——次高字节（NEXT MSB）

A+2——更高字节

A+3——更高字节

A+4——更高字节

A+5——更高字节

A+6——最高字节（MSB），但最高位是符号位。

A+7——用加128形式表示的幂指数。

双精度和单精度数值是按规范化的指数形式存储的，小数点被规定在MSB之前。指数存成加128的形式（即将128加到指数上去）。MSB的最高位用来表示符号，如果是正数，这位为0，如果是负数，这位为1。例：

```
10 A = 23.4
20 B#=23.12345678
30 PRINT VARPTR(A), VARPTR(B)
RUN
-23888          -23880
```

对于格式2来说，VARPTR函数返回指定的随机文件的FIELD缓冲区的地址。例：

```
10 OPEN "R", 1, "OUT DAT"
20 FIELD#1, 128 AS JUNK$
30 PRINT VARPTR(#1)
RUN
-2345
OK
```

11 WAIT（监控通道口）

格式：WAIT I,J,K

I是正被监控的通道口号（十进制整数），K和J是整型表达式。WAIT函数在监控某输入设备口状态的过程中将程序挂起，直到该输入设备口上的状态变为某特定的状态（按位表示）为止。从口上读入的数据先和K进行XOR（异或）运算，然后和J进行AND(与)运算。

如果结果为0，BASIC-80回头再读通道口上的数据，如果结果不为0，则继续执行下一条可执行语句。如果K省略，其补缺值为零。I、J、K必须在0～255之间（记住：所有的数都是十进制数，除非前面带有&H、&O或&）。例：

WAIT 20,6

执行暂停，直到通道口20的第一位或第二位等于1时，再继续执行下一条语句（第0位是最低位，第7位是最高位）。又例：

WAIT 10,255.7

执行暂停，直到通道口10的高五位中的任何一位等于1或者低三位中的任何一位等于0时，再继续执行下一条语句。

12 WIDTH（置行宽）

格式：WIDTH〔LPRINT〕〈整数表达式〉

WIDTH函数可用来设置终端或行打印机的行宽。LPRINT可选项是用于行打印机设置行宽的。

〈整数表达式〉规定了显示打印的每一行中字符个数。补缺行宽对于终端是72，对于行打印机是132 。

如果〈整数表达式〉为255，则行宽不限，这时，BASIC不再插入回车。然而，用POS或LPOS函数来检测光标或打印头的位置时，超过255以后，返回值为0。例：

WIDTH 80 设置终端行宽为80个字符。

WIDTH LPRINT 96 设置打印机宽为96个字符。

第四节 用户自定义函数

有时在同一程序的几个不同地方都要执行某些顺序完全相同的程序语句或数学公式的运算，BASIC-80允许程序员自己定义函数，然后象引用诸如ABS SIN或SQR这些标准系统函数一样地去引用这些函数。

用户有时还需要用汇编语言去编写一个专用的程序段，BASIC-80提供了从BASIC程序中调用汇编语言程序的方便。

DEF FN（定义函数）

格式：DEF FN〈名称〉（〈变量表〉）=表达式

DEF FN语句用来定义一个隐函数。

〈名称〉必须为合法的变量名，它以FN打头构成一个函数名。变量表中的变量都是“哑”名，它代表函数调用时要传递的参数。

自变量个数不限，等号右边可以为任何正确的表达式，整个函数定义语句的长度限制在一个逻辑行（255个字符）内。

用户自定义函数可以为任何类型，函数类型由放在函数名之后的一个类型说明符（%、!、#或$）来定义；或者，这些代表函数名的字符也可用函数DEFSTR、DEFSNG等来定义；未加定义的数值变量则约定为单精度数据类型。

一旦规定好函数类型，则表达式的值在返回时要进行相应的转换。在函数调用中，如果函数类型、参数等不匹配，则会出现“类型不匹配”的错误。DEF FN不能用于直接命令方式。例：

```
10 DEF FNAB (X,Y) =X+Y
20 SUM=FNAB(10, 20)
```

```
30 PRINT SUM
RUN
30
OK
```

第五节　汇编语言程序

采用下述两种方法，都可以调用一个汇编语言程序。第一种方法是利用USR函数，第二种方法是利用CALL语句。

更详细的说明，参见附录E“汇编语言子程序”。

1. DEF USR（定义USR子程序的入口地址）

格式：DEF USR〈数字〉＝〈表达式〉

DEF USR语言用来定义多至10个汇编语言子程序的入口地址。

〈数字〉是汇编语言子程序的编号，它可以为0～9中的任一个数。如果这个〈数字〉被省略，则约定它为0。

表达式的值即为汇编语言子程序的首地址。如这个数前面没有数字进制的基数标志符，它就是十进制数。前面带有&H的数为十六进制数，前面带有&O或&的数为八进制数。例：

```
DEFUSR1＝&H22
DEFUSR2＝45000
DEFUSR5＝ADDRESS
```

2. USR（使用汇编语言函数）

格式：USR〈数字〉(X)

USR用来使用一个汇编语言子程序。〈数字〉必须在0～9范围之内，且应和DEF USR语句相对应。如〈数字〉被省略，则约定它为0。X是传给汇编语言子程序的参数。例：

```
Z＝USR1(B/2)
A＝USR2(1.23)
C＝USR5(ARG1)
```

注：有关如何定义和使用USR函数的详细说明，参见附录E。

3. CALL（调用汇编语言子程序）

格式：CALL〈变量名〉[(参数表)]

CALL语句用来调用一个汇编语言子程序。

汇编语言子程序在内存中的首地址将赋给〈变量名〉，这个地址应在CALL语句被执行之前就被指定。〈变量名〉不能是数组变量名，〈参数表〉包括传递给汇编语言子程序的参数。

CALL语句的执行过程和Microsoft的FORTRAN，COBOL以及编译BASIC中的CALL是一样的。有关这个调用过程的说明见附录E“汇编语言子程序”。例：

```
110 MYROUT＝&HD000
120 CALL MYROUT (I,J,K)
```

第八章　特殊功能

BASIC-80提供了一些特殊功能，其中一种是“错误陷阱”，它能在程序执行时有效地进行检错。另一种是PRINT USING 语句，它允许程序员自己规定数值或字串输出的格式。

还有一种重要功能是跟踪，它允许程序员一行一行地去跟踪运行一个程序。

BASIC-80还向程序员提供了一种覆盖管理技术，这是用CHAIN语句 和COMMON语句实现的。

第一节　错误陷阱

BASIC-80允许程序员去编写错误检测程序和弥补错误的处理程序，或者给出一个较为完整的错误原因的说明。这是通过使用 ON ERROR GOTO、RESUME 和ERROR语句以及ERR和ERL变量而实现的。

1. ON ERROR GOTO（陷阱赋能）

格式：ON ERROR GOTO〈行号〉

使用ON ERROR GOTO语句，可以使陷阱赋能，并指明了错误处理子程序的第一行。

一旦错误陷阱被赋能，任何检测出的错误，包括命令状态错误（如语法错误）都将使控制跳转到错误处理子程序。如果〈行号〉没有，则会产生一个“未定义行号”（Undefined line number）的错误信息。

为了撤去错误陷阱，必须执行 ON ERROR GOTO 0语句。如遇到再出现的错误，则将显示错误信息并停止程序的运行。出现在捕捉错误子程序中 的 ON ERROR GOTO 语句将使得 BASIC-80 程序停止运行并将这一错误信息显示出来。我们建议：如果你没有错误补救措施，那么在所有的错误捕捉子程序中都应加上一条 ON ERROR GOTO 0 语句。

如果在错误处理子程序运行当中又出现了错误，那么，错误信息将被显示出来，并且程序终止运行。错误陷阱无法捕捉错误处理子程序中的错误。例：

```
10 ON ERROR GOTO 1000
```

2. RESUME（继续执行）

格式：RESUME

```
RESUME 0
RESUME NEXT
RESUME 〈行号〉
```

RESUME 语句用在错误处理程序执行完毕以后，继续执行原程序。

根据从何处开始继续运行程序，可决定采用下述四种形式中的某一种：

RESUME或RESUME 0　从发生错误的这条语句起继续运行程序。

RESUME NEXT　从发生错误的那条语句的下一句开始继续运行程序。

RESUME〈行号〉　从〈行号〉指明的那条语句继续往下运行。

凡不在错误捕捉程序中的RESUME语句会产生一个“无错误而重新启动”(RESUME Without error) 的错误信息。

错误陷阱举例：

```
100  ON ERROR GOTO 500
200  INPUT "WHAT ARE THE NOMBERS TO DIVIDE"; X, Y
210  Z=X/Y
220  PRINT "QUOTIENT IS"; Z
230  GOTO 200
500  IF ERR=11 AND ERL=210 THEN 520
510  STOP
520  PRINT "YOU CAN'T HAVE A DIVISOR OF ZERO!"
530  RESUME 200
```

3. ERROR (产生错误信息)

格式：ERROR〈整型表达式〉

ERROR语句用于模拟产生一个BASIC-80错误，或者用来由用户定义一个错误信息码。

〈整型表达式〉的值必须大于0小于255，如果〈整型表达式〉的值等于BASIC-80中已有的某一错误信息代码，则这个ERROR语句将模拟产生那个错误，并将错误信息显示出来。

为了定义自己的错误信息代码，整型表达式的取值要大于已有的BASIC-80错误代码。(最好使用最大可能的有效数值，这样在更多的错误代码编入BASIC-80时，可以确保不相矛盾)。这样，在错误捕捉程序中，这种用户自定义错误信息代码就可以很方便地加以使用。

如果一个ERROR语句中所标明的代码未经定义错误信息，则BASIC-80的响应信息是“不可显示错误”(Unprintable error)。不在错误捕捉程序中使用ERROR语句将导致错误信息的产生，并使运行停止。例：

```
LIST
10 S=10
20 T=5
30 ERROR S+T
40 END
OK
RUN
String too long in line 30
```

或者用命令方式：

```
OK
ERROR 15 (你打入这一行)
String too long (BASIC-80显示这一行)
OK
```

4. ERR和ERL变量

当进入错误处理子程序时，错误信息代码被赋给变量ERR，出错的那一行的行号被赋给变量ERL。变量ERR和ERL通常用于IF/THEN语句，以指明错误捕捉程序的流程。

如果产生错误的那条语句是直接命令，则ERL的值为65535。想测试错误是否出现在直接命令中，可用 IF65535=ERL THEN…。在其他情况下，可用：

IF ERR=错误信息代码 THEN…

IF ERL=行号 THEN…

如果行号不在关系符的右边，就不可能用 RENUM 命令对它重新编号。而因为 ERL 和ERR都是保留的变量名，所以二者都不可以出现在LET语句（赋值语句）中的等号的左边。

一进入错误处理子程序，错误代码就赋给变量ERR。错误代码以及它们所代表的含义都列在表8-1之中。有关错误信息的更为详尽的讨论，请参见附录A“错误信息”。

5. 错误信息代码

一般错误信息

代码	错误信息	说明
1)	NEXT WITHOUT FOR	循环语句中有 NEXT 而无 FOR
2)	SYNTAX ERROR	语法错误
3)	RETURN WITHOUT GOSUB	子程序返回，但没有子程序调用语句
4)	OUT OF DATA	数据不够读
5)	ILLEGAL FUNCTION CALL	非法函数调用
6)	OVERFLOW	溢出
7)	OUT OF MEMORY	内存不够
8)	UNDEFINED LINE NUMBER	未定义的行号
9)	SUBSCRIPT OUT OF RANGE	数组下标越界
10)	DUPLICATE DEFINITION	重复定义
11)	DIVISION BY ZERO	被零除
12)	ILLEGAL DIRECT	非法直接语句
13)	TYPE MISMATCH	数据类型不匹配
14)	OUT OF STRING SPACE	字串存储区不够
15)	STRING TOO LONG	字串太长
16)	STRING FORMULA TOO COMPLEX	字串表达式太复杂
17)	CAN'T CONTINUE	不能继续运行
18)	UNDEFIED USER FUNCTION	用户函数未定义
19)	NO RESUME	没有重新起动语句
20)	RESUME WITHOUT ERROR	无错误而重新起动
21)	UNPRINTABLE ERROR	不可显示的错误
22)	MISSING OPERAND	缺操作数
23)	LINE BUFFER OVERFLOW	行缓冲器溢出
26)	FOR WITHOUT NEXT	循环语句中有FOR而无NEXT
29)	WHILE WITHOUT WEND	有 WHILE 而无 WEND

30) WEND WITHOUT WHILE　　有WEND而无WHILE

表8-1　错误信息代码

磁盘错误信息

代码	错误信息	说明
50)	FIELD OVERFLOW	定段溢出
51)	INTERNAL ERROR	内部错误
52)	BAD FILE NUMBER	错误文件号
53)	FILE NOT FOUND	文件未找到
54)	BAD FILE MODE	文件类型错误
55)	FILE ALREADY OPEN	文件已打开
57)	DISK I/O ERROR	磁盘读写错误
58)	FILE ALREADY EXISTS	文件已存在
61)	DISK FULL	磁盘已满
62)	INPUT PAST END	越过文件末尾读入数据
63)	BAD RECORD NUMBER	错误记录号
64)	BAD FILE NAME	错误文件名
66)	DIRECT STATEMENT IN FILE	文件中有直接语句
67)	TOO MANY FILES	文件太多

表8-1(续) 错误信息代码

第二节　格式输出

PRINT USING语句用于按规定格式输出信息，这在打印工资支票或结算单这类应用之中是很有用的。

PRINT USING（格式输出）

格式：PRINT USING〈格式串〉；〈表达式序列〉

PRINT USING语句用于以规定格式打印字串或数字。

〈表达式序列〉包括要显示的字串表达式和数值表达式，它们之间用分号分开。〈格式串〉是一串字符或变量，它们是规定的描述格式的字符，这些字符（见后）确定了要显示的字串或数值的各个域以及它们的格式。

1. **字串域**

当PRINT USING语句用于显示字串时，下述三种格式符可用来规定字串各个域的格式：

· "!"——规定只显示字串的第一个字符。

· "\n spaces\"——规定字串的前2+n个字符将被显示，如果\\之间不留空格，则显示串的前两个字符，如果\\之间有一个空格，则显示字串的前3个字符，依此类推。如果字串长度超过定义域范围，则多余的字符被截去。如果定义域范围比字串长，则字串在域内左对齐，域内不足部分填以空格。例：

```
10 A$="LOOK":B$="OUT"
```

```
20 PRINT USING "!"; A$;B$
30 PRINT USING "\  \";A$;B$
40 PRINT USING "\    \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT !!
```

· "&" ——定义域长度可变，以"&"规定字串域，则输出和输入一样。例：

```
10 A$ = "LOOK"; B$ = "OUT"
20 PRINT USING "! " ;A$
30 PRINT USING "&" ;B$
RUN
L
OUT
```

2. 数字域

当PRINT USING用来显示数字量时，可用下述几个格式串来规定各个域的格式：

· "#" ——代表相应的各个数位。数位一般都要填满。如果要显示的数字的位数少于格式规定，那么这个数在域内将向右对齐，左边以空格填满。

· "." ——小数点可以插在格式串中的任何位置上，如果格式串规定小数点前面有一位，那么显示时这一位总会有一个数（必要时补0）。数进行必要的四舍五入。例：

```
PRINT USING "##.##"; .78
 0.78
PRINT USING "###.##"; 987.654
 987.65
PRINT USING "##.##    "; 10.2,5.3,66.789, .234
 10.20    5.30    66.79    0.23
```

上面最后一个例子中，在格式串末尾加了三个空格，目的是使所显示的数彼此分开。

· "+" ——加在格式串首部或尾部的"+"号，可在一个数显示时，把该数的符号（正或负）也在这个数的前面或后面显示出来。

· "−" ——加在格式串尾部的"−"号，用以在显示一个负数时，在其后面加上一个负号。如果被显示的数是正数，则在其后面加一个空格。例：

```
PRINT USING "+##.##"; -68.95, 2.4, 55.6, -.9
 -68.95  +2.40  +55.60  -0.90
PRINT USING "##.##-"; -68.95, 22.449, -7.01
 68.95-  22.45  7.01-
```

· "**" ——格式串前面的双星号（**）表示在被显示的数值域的空格部分用"*"去填满。同时"**"本身还定义了两个数字位。例：

```
PRINT USING "**#.#"; 12.39, -0.9, 765.1
 *12.4  *-0.9  765.1
```

· “$$”——两个$符号用来在一个格式化的数的左边加上一个“$”符号,“$$”本身还定义了两个数位，其中一位显示$本身。指数型的数不能用$$，负数也不能用$$，除非该负数的负号放在数的右边。例：

```
PRINT USING "$$###·##"; 456.78
␣$456.78
```

· “**$”——格式串前面的**$综合了前两种符号的功能：一是打头空格用*去填满,一是在数的前面加一个$号。“**$”本身还定义了三个数位,其中一位是$位。例：

```
PRINT USING "**$##.##"; 2.34
***$2.34
```

· “,”——如在格式串小数点的左边加一逗号，则在被显示的数字串中，自小数点向左，每隔三位就打一个逗号。如逗号位于格式串的末尾，则在被显示的数中，逗号位置和格式串一样。逗号本身也定义了一数位。对于指数型（^^^^）的数，逗号不起作用。例：

```
PRINT USING "####,.##"; 1234.5
1,234.50
PRINT USING "#####.##,"; 1234.5
1234.50,
```

· “^^^^”——在格式串后面放置四个向上的箭头，用来规定指数格式，四个箭头的位置用来显示E+××。小数点位置可任意规定，显示时有效数字部分向左对齐，指数写成规范形式。除非在格式串的最前面放置一个“+”号或在其尾部放一个“-”号，数在显示时，小数点左边那一位上总有一个空格或负号。例：

```
PRINT USING "##.##^^^^"; 234.56
2.35E+02
PRINT USING "·#####^^^^ "; 888888
8889E+06
PRINT USING "+·##^^^^"; 123
+.12E+03
```

· “_”——在格式串中的底划线，表示将它后面的那个字符作为文字输出。例：

```
PRINT USING "_!##.##_!"; 12.34
!12.34!
```

这个文字本身也可以是底划线，要显示它，须在它前面再加一条底划线。

· 错误信息——如果一个被显示的数比定义的数字域大，则会在这个数的前头显示一个百分号（%）。如果一个数经过了四舍五入而超出了格式规定，则也会在这个经过四舍五入后的数前显示一个百分号。例：

```
PRINT USING "##.##"; 111.22
%111.22
PRINT USING ".##"; .999
%1.00
```

如果所定义的数位超过24，就会显示一个“非法函数调用”(Illegal function call)的错误信息。

第三节　跟踪运行

为了帮助查错，有两条语句可用来跟踪程序的运行：

TRON/TROFF（设置/撤消跟踪标志）

格式：TRON　　TROFF

TRON/TROFF语句用于跟踪程序的运行。

为了帮助查错，可用TRON语句（既可用直接命令方式也可用间接方式执行）设置跟踪标志，从而使程序运行时，每一条执行过的语句行号都显示在屏幕上，行号均放在方括号内。TROFF语句撤消跟踪标志（NEW命令也有此功能）。例：

```
TRON
OK
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K+10
40 PRINTJ; K; L
50 K=K+10
60 NEXT
70 END
OK
RUN
〔10〕〔20〕〔30〕〔40〕 1  10  20
〔50〕〔60〕〔30〕〔40〕 2  20  30
〔50〕〔60〕〔70〕
OK
TROFF
OK
```

第四节　覆　　盖

BASIC-80提供了两条语句，CHAIN和COMMON，用来进行程序覆盖。在程序运行时，利用这两条语句，可以将其他程序合并运行，还可以把某些或所有的变量从一个程序传递到另一个程序中去。

1. CHAIN（调用覆盖过程）

格式：CHAIN〔MERGE〕〈文件名〉〔,〔〈行号表达式〉〕〔,ALL〕〔,DELETE〈范围〉〕〕

CHAIN语句用来调用一个程序并将变量从现行程序传递到调用程序中去。

〈文件名〉指被调用的程序名。例：

CHAIN "PROG1"

〈行号表达式〉是一个行号或一个表达式，通过这个表达式可以求出被调用程序中的某一行号，这个行号是被调用的程序运行的起点，如果它省略，则从第一行开始运行。例：

CHAIN "PROG1", 1000

注意〈行号表达式〉不受 RENUM 命令的制约。

如有ALL选择项，则现行程序中的所有变量都被传递到被调用的程序中去，如果 ALL 选择项省略，则现行程序中必须含 COMMON 语句，以确定哪些变量要传递。例：

CHAIN "PROG1", 1000, ALL

如果程序中包括有 MERGE 选择项，那么它将使用覆盖管理技术将一个子程序并入现行程序，也就是说，MERGE 操作是用现行程序和被调用程序来进行的。被调用的程序必须是ASCII码文本。例：

CHAIN MERGE "OVRLAY", 1000

在一个覆盖模块调入之后，通常还希望删去它，以便覆盖上新的模块。为了达到这一目的，可用 DELETE（删除）选择项，在〈范围〉内的行号，受 RENUM 命令的制约。例：

CHAIN MERGE "OVRLAY@", 1000, DELETE1000—5000

如果MERGE选择项省略，那么 CHAIN 语句并不向被链接程序提供变量类型定义和用户自定义函数。也就是说，任何 DEFINT、DEFSNG、DEFDBL、DEFSTR 或 DEFFN 语句的定义的共用变量都必须在被链接的程序中重新定义。

2. COMMON（传递变量）

格式：COMMON〈变量表〉

COMMON 语句用于将变量传到被链接的程序中去。

COMMON 语句和 CHAIN 语句一起用于链接过程。COMMON 语句可以出现在程序中的任何地方，但我们建议应把它放在程序的开头。同一变量不允许出现在几条 COMMON 语句之中。数组变量名后面必须附上（ ）。如果所有变量都要传递，则使用带ALL选择项的CHAIN 语句，而省去 COMMON 语句。例：

```
100 COMMON A, B, C, D ( ), C$
110 CHAIN "PROG3", 10
```

第九章　编　辑

在编辑状态中，不必重新打入某一整行，就可以对这一行中的某一部分进行编辑，一进入编辑状态，BASIC-80即给出要编辑的那一行的行号，然后是一个空格，再等待编辑状态中的子命令。

常用的编辑状态子命令用来对一行文字进行插入、删除、替换或搜索等操作，这些子命令终端不显示。某些编辑状态子命令前面可用一个整数来规定这个子命令的执行次数，如果没有这个整数，则约定为1。

编辑状态子命令可按下面的功能分类：

1. 移动光标
2. 文本插入
3. 文本删除
4. 文本寻找
5. 文本替换
6. 编辑状态的结束和重新启动。

在编辑状态中，如果BASIC-80接收到一个不认识的命令和非法的字符，机器就发出铃声（CTRL-G），对这一命令或字符将不予理睬。打入下列命令即可进入编辑状态：

EDIT〈行号〉

这里，行号是待编辑的那一行的行号，如果没有行号，将会产生一个“未定义的行号”（Undefined line number）的错误信息。

所需行号被显示出来后，接着是一个空格，现在，光标位于这一行第一个字符位置上。

请键入下面这行程序：

```
100 FOR J=1 TO 10:PRINT J:NEXT
```

我们将用这行程序来验证各种编辑状态子命令。

第一节　移动光标

·n Space Bar（空格键）

在编辑状态中，空格键用于光标右移。例如，现在对刚才键入的第100行程序进行编辑．一进入编辑状态，这一行的行号即显示在屏幕上：

```
100_
```

按一下空格键，光标移过一格，这行程序中的第一个字符被显示出来。如果这个字符是一个空格，那么就在屏幕上显示一个空格。你可以连续按空格键，直到第一个非空格字符显示出来。这时，屏幕上是：

```
100 F_
```

还可以使光标一次移动许多格，为此，应先键入一个需要移动的空格数．再按空格键。例如，为了使光标一次移动5格，先键入5，然后再按空格键，字符就会象逐个移动光标那样

显示出来：

```
100 FOR J=_
```

（你显示的内容不一定就是这样，因为这取决于你原先在这行程序中插入了多少空格。）

·BACK SPACE

在编辑状态中，按BACK SPACE键可使光标左移一格，在光标左移过程中，字符不被删除掉。我们再回到刚才的例子：

```
100 FOR J=_
```

如果现在光标位于等号之后，则只要按一下BACK SPACE键，就可把光标移到等号的下面，即：

```
100 FOR J=
```

第二节　文本插入

·I（插入）

I子命令用于在现行光标位置前插入文本，被插入的字符依次显示出来。要终止插入操作，应按一下ESC键。如果按下RETURN键，其效果等于先按ESC键，再按RETURN键。

利用空格键将光标移到10中0的右边：

```
100 FOR J=1 TO 10_
```

现在假定要把上句中的10改成100，按下I键（不能按RETURN键），系统进入插入状态。为了达到将10改成100的目的，键入一个0，则屏幕显示：

```
100 FOR J=1 TO 100_
```

这样，插入就完成了。按下ESC键，则退出插入状态。现在如再按RETURN键，则所有的更改都被保存下来。系统又回到BASIC-80命令状态。如果你显示这100行，则屏幕将显示：

```
100 FOR J=1 TO 100:PRINT J:NEXT
```

在插入状态中，可用BACK SPACE键删去光标左侧的字符。

如果一行已达到255个字符，而你还插入，则机器发出铃声，插入的字符也不再显示出来。

·X（行延伸）

X子命令用于延伸一行程序，它先将光标移至这行的末尾，并自动进入插入状态，而后就象在I命令中那样进行文本插入操作。当你要结束插入操作时，可按ESC键，或者按回车键（RETURN），返回BASIC-80命令状态。

例如，要延伸刚键入到程序中的第100行，可先进入编辑状态，屏幕显示：

```
100_
```

这时，按下X键，屏幕上就显示整个一行，光标跳到这一行的末尾：

```
100 FOR J=1 TO 100:PRINT J:NEXT_
```

现在，系统处于插入状态。用户可在这一行末尾添加上其他的程序语句。例如，键入：PRINT "ALL DONE" 和一个回车（RETURN）。这一行就包含了这一新语句。如果

你显示第100行，就可以看到：

```
100 PRINT J=1 TO 100:PRINT J:NEXT:PRINT"ALL DONE"
```

第三节　文本删除

·nD（删除）

nD子命令用于删除光标右侧的n个字符，被删去的字符被夹在两条右斜线之间。光标位于被删去的最后一个字符的右侧。如果开始时光标右侧的字符少于n个，则这一行余下的部分全被删去。

例如，让先前键入的第100行程序进入编辑状态，再用空格键将光标移到FOR语句的末尾，则屏幕上可以看到：

```
100 FOR J=1 TO 100:_
```

这时，键入8D，光标右侧的8个字符就被删去，屏幕上可以看到：

```
100 FOR J=1 TO 100:\PRINT J:\
```

注意：夹在两条右斜线之间的是被删去的字符。

现在，按下回车键（RETURN），系统又返回到BASIC-80命令状态。如果用LIST命令显示第100行程序，可以看到，PRINT J语句已从这行程序中被删去了。

·H（删除和插入）

H子命令删除光标右侧的全部字符，并自动进入插入状态。对于在一行后部进行替换操作来说，H命令是很有用的。比如，你想把第100程序的最后一条语句更改一下，首先必须让100行程序进入编辑状态，然后用空格键将光标移到NEXT语句的右边，屏幕上可看到：

```
100 FOR J=1 TO 100:NEXT:_
```

按H键，再按STOP，然后再按回车（RETURN），将这一更动保存下来，并使系统返回到BASIC-80命令状态。

现在，显示第100程序，如果你是按这一节所讲的办法做的，则屏幕上可以看到：

```
100 FOR J=1 TO 100:NEXT:STOP
```

第四节　文本寻找

·nS〈字符〉（字符搜索）

搜索子命令用于搜索〈字符〉第n次出现的位置，并将光标置于那个位置之前，光标当前所在位置不在搜索范围之内。如果要搜索的那个字符找不到，则光标就停止在这一行的末尾。在搜索过程中，光标所经过的那些字符全被显示出来。注意，仅仅是光标右侧的那些字符才包括在搜索范围之内。

例如，用上一节改过以后的第100行作为例子，首先进入编辑状态，再键入2S:，这一命令表示搜索第100行程序中第2个冒号（:）所在的位置，屏幕显示：

```
100 FOR J=1 TO 100:NEXT_
```

这时可以执行你想要执行的任何子命令，比如你想要在NEXT语句之后写入一个循环

计数变量，首先要进入插入状态，然后键入一个空格和变量J。再按ESC键，则系统退出插入状态，最后按回车（RETURN），退回到BASIC-80命令状态。此时，你如果要显示第100行程序，就可在屏幕上看到（当然，假定你是按照本节所介绍的编辑步骤操作的）：

```
100 FOR J=1  TO  100:NEXT J:STOP
```

·nK〈字符〉（搜索并删除）

搜索并删除子命令类似于搜索子命令，只有一点不同，即在搜索过程中光标所经过的那些字符都被删去了。光标位于找到的那个字符之前，所有被删去的字符都夹在两条右斜线之间。

例如，对目前形式的第100行程序进行编辑，先进入编辑状态，然后键入2K:，这个命令删去这一行中第二个冒号之前所有的字符，屏幕上将可以看到：

```
100\FOR J=1 TO 100:NEXT J\_
```

第二个冒号也要删去，所以键入D，则屏幕上可以看到：

```
100\FOR J=1 TO 100:NEXT\\:\
```

这时，再按回车键（RETURN），并用LIST显示第100行，则可以看到：

```
100  STOP
```

第五节　文本替换

·nC（更改）

更改子命令用于更改从光标现在位置开始的规定数目的字符。如果你只键入C，前面不给数字，计算机则认为你只改一个字符。如果你在键入C之前还打入了一个数n，则计算机认为你要更改从光标起的n个字符。

例如，再重新键入原先的那行(第100行）程序：

```
100 FOR J=1 TO 100:PRINT J:NEXT
```

然后再使这行程序进入编辑状态，你在屏幕上可以看到：

```
100_
```

现在，我们假定你打算把FOR/NEXT循环体中的终值100改成150，首先应该用空格键把光标移到100中的第一个0位置上。如果光标移过了，只要按BACK SPACE键，就可以使光标返回。

```
100 FOR J=1 TO 1_
```

此时，键入C，BASIC-80将认为你只要改一个字符，再键入5，然后按回车。如果显示第100行，在屏幕上就可以看到更改后的这行程序是：

```
100 FOR J=1 TO 150:PRINT J:NEXT
```

第六节　编辑状态的结束和再启动

·RETURN（保存更改内容并退出编辑）

如果按回车（RETURN），则一行中未显示的部分都将显示出来，你所进行的全部更改都被保存下来，计算机返回到BASIC-80命令状态。

·E（保存更改内容并退出编辑）

E子命令和RETURN效果一样，不同的只是一行中未显示的部分不再显示出来。

·Q（取消更改内容并退出编辑）

Q子命令使系统又返回到BASIC-80命令状态，但是不保存在编辑过程中的任何更改的内容。

·L（显示一行）

L子命令显示一行中的剩下部分（保存已进行的更改），并使光标重新回到这一行的开始位置上，系统仍处于编辑状态。L命令常常在你一进入编辑状态时，用来显示这一行。例如：

```
EDIT 100
100_
〈键入L〉
〈BASIC-80响应：〉
100 FOR J=1 TO 100:NEXT:STOP
100_
```

·A（取消更改内容并重新启动）

A子命令可让你对某一行重新进行编辑，已进行的任何更改都作废，而重新恢复原先的内容，光标重新定位在这一行的开始位置上。注意不要在执行任何其他子命令的过程中使用A子命令。如果你在执行其他子命令，比如插入命令，则要按ESC键，然后再按A键。在下面这个例子中，操作员首先显示原先的程序行，然后在插入状态下改子程序，而后又决定推倒重来，故用A子命令去恢复原先的程序行：

```
EDIT 100
100_
〈键入L〉
100 FOR J=1 TO 100:NEXT:STOP
100_
100 FOR J=1 TO 100_〈操作员按1，再按0〉
〈操作员按ESC键〉
100 FOR J=1 TO 1000:NEXT:STOP
100_
〈操作员按A〉
100_
〈操作员按L键，注意，原先程序行的内容已被恢复〉
100 FOR J=1 TO 100:NEXT:STOP
100_
```

第七节　其他编辑状态功能

·SYNTAX ERROR（语法错误）

在程序执行过程中发现有语法错误时，BASIC-80将自动地使产生错误的这行程序进入编辑状态，例如：

```
10 K=2(4)
RUN
SYNTAX ERROR IN 10
OK
10_
```

当结束对这一行的编辑，按回车（RETURN）或E子命令后，BASIC-80要将这一改过的语句重新插入原程序中，这使得所有变量的值都丢失，所有打开的文件都被关闭。要想保存这些变量的值供验证结果，首先应用Q子命令退出编辑，BASIC-80将重新返回到命令状态，而全部变量的值都被保存下来。

·CTRL-A

若想在键入某一行时进入编辑状态，可以按CTRL-A，BASIC-80将响应一个回车，一个感叹号〈!〉和一个空格，光标停在这一行的第一个位置上。此时你可以用任一编辑状态子命令进行操作。

·当前行编辑

当你想编辑当前行时，可以使用句点（·）来代表当前行。于是，命令：

```
EDIT·
```

将在当前行进入编辑状态。行号标记（·）总是代表当前行。

第十章　BASIC-80磁盘文件操作

BASIC-80提供了几组语句供生成程序和数据文件，并进行文件操作。

文件操作命令对程序文件的操作是很有用的，其中某些也可用于数据文件。

文件管理语句用于打开和关闭数据文件，检查文件的结束，以及获得有关文件大小的信息。

顺序存取语句用于存取顺序文件。顺序存取文件容易使用，但数据必须顺序地存取。

随机存取语句用于对随机文件进行存取和操作。随机存取文件比顺序存取需要较多的步骤，但文件中的记录可按任意次序读取。

第一节　文件操作命令

本节复习对程序和数据文件进行操作的命令和语句。这些语句和命令在第三章“命令状态语句”中已讨论过。

1. FILES〔“(文件名)”〕

FILES命令是将现行盘上的文件名列表。如果有可选项（文件名），则在指定盘上的此文件名均被列出。

2. KILL “文件名”

KILL命令从盘上删除文件。“文件名”可以是一个程序文件，也可以是一个顺序或随机存取的数据文件。如果“文件名”是一个数据文件，则在它被删除以前必须先行关闭。

3. LOAD “文件名”〔,R〕

LOAD命令将程序从盘上装入内存。可选项R使程序立即运行。LOAD命令会清除内存的现行内容，并在装载前关闭所有文件。然而如果有R，则被打开的数据文件保持打开状态，因而程序可以链接或分段装载，并对同一数据文件进行存取。

4. MERGE “文件名”

MERGE命令从盘上将程序装载到内存，但不清除内存的现行内容，盘上的程序行号与内存中的程序行号合并。如果二者有同样的行号，则仅保留盘上的程序行。在MERGE命令后，“合并成”的程序保留在内存中，而BASIC-80回到命令状态。

5. NAME “旧文件” AS “新文件”

想改变一个磁盘文件名，可用NAME语句，NAME “旧文件”AS“新文件”。NAME命令可用于程序文件、随机文件或顺序文件。

6. RESET

当你在现行主驱动器（指补缺驱动器——下同）中换上一块盘时，RESET命令会读出新盘上的目录信息。RESET并不关闭原来盘上已打开的文件。因此，应该在原来盘上的文件都已关闭，并换上新盘以后使用RESET。

7. RUN “文件名”〔,R〕

RUN“文件名”命令将程序从盘上装入内存并执行之。RUN命令在将程序装入前，清除内存现行的内容并关闭所有文件。然而，如果可选项R也包括在命令中，则所有的已打开的数据文件保持打开状态。

8. SAVE“文件名”〔,A〕

SAVE将内存中的程序写入磁盘。可选项A使程序存成ASCII码形式（否则，将采用压缩的二进制形式）。

保护文件

如果要用二进制编码形式保存程序，可采用SAVE命令中的可选项P。例如：

```
SAVE "MYPROG", P
```

用这种文法保存的程序不能显示或编辑。

第二节　文件管理语句

BASIC-80提供了一整套用于磁盘文件管理的I/O语句，这些语句列表如下：

语句	功能
OPEN	打开盘文件并对盘文件分配文件号。
CLOSE	关闭盘文件，收回文件号。
EOF	如果文件已结束，则返回－1（真值）。
LOF	返回文件最后一个存储区域中的记录号（存储区域是指磁盘上一个目录项所管辖的存储区，其大小为16k。——译者注）
LOC	对于随机文件，返回供存取的下一个记录的记录号；对于顺序文件，返回已存取的扇区的总数。

表 10-1　文件管理语句

OPEN语句用于给盘文件分配文件号。同时，OPEN语句还对文件存取方式进行定义（顺序或随机存取方式）。

CLOSE语句完成OPEN语句相反的功能。它从盘文件收回文件号。

如果顺序文件已经结束，则EOF函数返回－1（真值）。EOF函数也可用于随机文件，以决定最后的记录号。

LOF函数将返回文件最后一个存储区域中的记录总数。

在随机文件中，LOC函数将返回供存取的下一个记录。在顺序文件中，它返回文件自打开以来存取的记录的总数。

这些语句将在下面讨论。用这些语句写的一个详细的程序实例，见附录F。

1. OPEN（打开盘数据文件）

格式：OPEN“方式”，〔#〕〈文件号〉，“〈文件名〉”〔，〈记录长度〉〕

这里：

“方式”是一个字串，它的第一个字符必须是下列方式特征字符之一。

O　指顺序输出方式

I　指顺序输入方式

R　指随机输入/输出方式

这个字串称为"方式特征字串"。

〈文件号〉是一个整型表达式，它表示文件所用的文件号。随后的输入/输出操作都使用这个文件号。

〈文件号〉不能超过BASIC-80初始化时所设置的可同时打开的文件总数。如果在初始化时没有进行设置，则BASIC-80将假定最大值为3。可参阅第一章"系统介绍和基础知识"，以对初始化过程有更多的了解。

"〈文件名〉"是完整的CP/M文件名。扩展名不补缺，因此文件名必须包括扩展名。如果没有指明驱动器，则指现行主驱动器。

〈记录长度〉是一个整型表达式，如果命令中包括它，即定义了随机文件的记录长度。其最大的记录长度为256字节。记录长度的补缺值为128字节。如果希望记录长度超过128字节，必须在BASIC-80初始化时加以说明。这个记录长度的可选项只能用于随机文件，试图定义顺序文件记录的大小将会出现"语法错误"。

OPEN语句将一个文件号连上某一个文件，OPEN语句也定义文件操作的方式（顺序或随机存取）。随后的输入/输出操作将使用分配给该文件的文件号。例如，假定文件用下列语句打开：

```
OPEN "I", 2, "SAMPLE.DAT"
```

此语句将文件号2分配给SAMPLE.DAT文件。因未说明驱动器名，因此BASIC-80将假定SAMPLE.DAT文件存放在现行主驱动器盘上。这个文件的方式特征字串为"I"——顺序输入。

如果SAMPLE.DAT不存在现行主驱动器盘上，则因为不能从一个不存在的文件上输入数据而出错。现在，假设存在这个文件，从这个文件输入数据，要用下述语句：

```
INPUT#2, 〈变量表〉
```

注意这个INPUT#语句使用了文件号2，而文件号2是分配给SAMPLE.DAT文件的（这只是INPUT#语句的常见形式，有关INPUT#语句的详细讨论在本章后部）。

现假定用下述的OPEN语句：

```
OPEN "O", 3, "B:OUTPUT.DAT"
```

此语句将把文件号3分配给文件OUTPUT.DAT。因为文件名中也指明了驱动器B:，故BASIC-80将在驱动器B:上生成这输出文件。如果这文件原已存在驱动器B:上，则该文件将被破坏，文件的所有原先的内容将丢失。现在，要向这个文件输出数据，可用下述语句：

```
WRITE#3, 〈变量表〉
```

这WRITE#语句使用了文件号3，而文件号3已经分配给文件B:OUTPUT.DAT。因此，〈变量表〉所表示的数据将被写入文件B:OUTPUT.DAT（WRITE#语句在本章后部将详细讨论）。一个文件也可打开成为随机输入／输出文件。可用一条OPEN语句打开一个文件兼作随机输入和随机输出。例如，下述语句将打开一个随机输入/输出文件：

```
OPEN "R", 1, "RANDOM.DAT"
```

现在，RANDOM.DAT文件已作为随机输入/输出而打开。如果RANDOM.DAT还不存在，那么它将被生成在现行主驱动器盘上。现在，可向这个文件进行随机输入或随机输出。注意这个OPEN语句没有指明记录大小，因此，BASIC-80将使用补缺值128字节。

也可用 OPEN 语句来定义不同大小的记录(但只限随机存取文件)。

例如，打开随机存取文件 RANDOM.DAT，并宣称记录的大小为 32 字节，可用下述语句：

```
OPEN "R", 1, "RANDOM.DAT", 32
```

现在记录的大小将为32字节，而 CP/M 扇区的大小为 128 字节，因此，4个记录将存于一个 CP/M 扇区中。也可在进行初始化时，用/S 开关来定义记录的大小（见第一章“系统介绍和基础知识”中的初始化过程)。

请特别注意下面这一点：即文件打开时定义的方式必须与文件存取的方式相吻合。例如，执行下述语句

```
OPEN "I", 1, "TEST.DAT"
```

TEST.DAT 文件已打开作为顺序输入文件，并分配了文件号1。现在，如想对这个文件输出数据，将是非法的，并且将产生错误信息。例如：

```
WRITE#1, "HELLO THERE"
```

此 WRITE#语句使用了文件号 1。因为原先 OPEN 语句已将文件号 1 定义为顺序输入，所以此 WRITE#语句是非法的，并将产生一个错误信息。

然而，这个规则有一个例外。在特定情况下，某几条顺序输入/输出语句可用于随机文件。在随机文件中使用顺序输入/输出语句的条件将在本章最后部分解释。

2. CLOSE（关闭盘数据文件）

格式：CLOSE〔#〕〔〈文件号〉〕

CLOSE 语句用于终止对盘数据文件的输入/输出操作。

〈文件号〉是文件被打开时的文件号。不带参数的 CLOSE 语句将关闭所有打开的文件。

假设下面 OPEN 语句出现在程序中：

```
OPEN "O", 1, "ARTIST.DAT"
```

并有一系列的输出语句涉及到这个文件。当文件输出结束时，必须用 CLOSE 语句关闭它：

```
CLOSE#1
```

这语句将使文件号 1 与文件 ARTIST.DAT 脱离，任何涉及文件号 1 的操作现在将无效。而文件 ARTIST.DAT 随后可用同一个或另外的文件号来重新打开。例如：

```
OPEN "I", 3, "ARTIST.DAT"
```

文件 ARTIST.DAT 现在于文件号 3 连接，并定义为顺序输入文件。现在，对这个文件的输入操作将有效。当输入结束时，这个文件也必须用 CLOSE 语句关闭：

```
CLOSE#3
```

这个文件还可重新被打开：

```
OPEN "R", 3, "ARTIST.DAT"
```

文件号 3 重新与文件 ARTIST.DAT 连接。但这一次，文件定义为随机输入/输出文件。

CLOSE 语句将最后仍留在缓冲区中的数据写入文件。这一点在本章的后部要详细谈。

END 语句和 NEW 命令将自动地关闭所有的盘文件。编辑或修改一个程序，也将自动地关闭所有打开的盘文件。但 STOP 语句不关闭盘文件。

3. EOF（检查文件的结束）

格式：EOF（〈文件号〉）

〈文件号〉是由前面的 OPEN 语句分配给盘数据文件的文件号。

如果一个顺序文件已结束，EOF 函数将返回-1（真值）。

EOF 函数可用来检查顺序文件是否结束。EOF 函数应和 INPUT#语句及 LINE INPUT#语句一起使用，以避免 "Input Past end"（越过文件末尾读入数据）的错误。

EOF 函数也可用于随机文件。如果在随机文件的终点处执行 GET 语句，EOF函数将返回-1(真值)。这可用来确定随机文件的大小。例：

```
10    OPEN "I", 1, "DATA"
20    IF EOF(1) THEN 100
30    INPUT#1, A$
40    GOTO 20
        ⋮
100   PRINT "END-OF-FILE REACHED"
```

4. LOF（返回记录号）

格式：LOF（〈文件号〉）

〈文件号〉是原先 OPEN 语句分配给盘数据文件的文件号。

LOF 函数返回文件最后一个存储区域中的记录总数。如果文件不超过一个存储区域，则 LOF 返回文件的实际长度（参考 "CP/M 应用程序手册" 中关于存储区域的详细内容)。例如：

```
110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"
```

5. LOC（返回记录号）

格式：LOC（〈文件号〉）

〈文件号〉是原先 OPEN 语句分配给盘数据文件的文件号。

当用于随机文件时，LOC 函数返回现行记录号。现行记录号比最后存取的那个记录的号数大 1。对一个文件首次进行存取，现行记录号是 1。允许最大的记录号是32767。

当用于顺序文件时，LOC 函数返回自文件打开以来存取的扇区总数(每扇区128字节)。例：

```
10  OPEN "I", 1, "TEST.DAT"
20  OPEN "R", 2, "RANDOM.DAT"
200 PRINT "SECTORS READ--"; LOC(1)
210 PRINT "NEXT REC#--"; LOC(2)
```

第三节　BASIC-80 顺序输入/输出

顺序文件比随机文件易于生成，但它在存取数据时灵活性与速度都受到限制。被写入顺序文件的数据是一项挨着一项保存的（顺序地），用什么次序写入则读出时也按同样次序读出。数据按 ASCII码字符串形式保存。

3.1　顺序存取语句

语句	功能
INPUT#	从顺序文件输入数据
LINE INPUT#	从顺序文件输入整行数据
PRINT#	写数据到顺序文件
PRINT# USING	用一定格式写数据到顺序文件
WRITE#	写数据到顺序文件（分隔符自动地被插入）。

表 10-2　顺序存取语句

1. INPUT#（从顺序文件中输入数据）

格式：INPUT#〈文件号〉,〈变量表〉

INPUT#语句用于从一个顺序的盘文件读出数据，并赋值给程序度量。数据将被顺序地读出。当文件被打开时，在文件的开头设有指针。数据每次从文件读出，指针将逐渐推进。想重新从文件开头读取数据，则必须关闭这个文件再重新打开。

〈文件号〉是文件定义为输入文件时所用的文件号。〈变量表〉包含用输入数据赋值的变量名（变量的数据类型必须与变量名类型相匹配，读一个字串到数值变量中是非法的）。

（1）数值输入

数据项在顺序文件中的排列与响应 INPUT 语句所需要的排列一样。对数值而言，开头空格被忽略。

遇到不是空格、回车或换行的第一个字符，就假定为数值的开始，数据用空格、回车、换行或逗点结束。

例如，假定有下述数据这样地存在文件上：

（注意：b 代表空白或空格——ASCII码32）

```
bb2.1234b-123.234bb456〈回车〉
```

那末，输入语句：

```
INPUT#1, X, Y, Z
```

或者输入语句的序列：

```
INPUT#1, X:INPUT#1, Y:INPUT#1, Z
```

将赋值如下：

```
X＝2.1234
Y＝－123.234
Z＝456
```

在2.1234值前面的二个空格是打头空格，因此被忽略。下一个字符是 2，这是第一个数值域的开头。

BASIC-80 输入/输出处理器现扫描第一个数值域的结尾。在2.1234及－12.234之间的

空格b是结束符，故当BASIC-80遇到这个空白b时，它假定第一个数据域已结束了。第一个数值域赋值给变量表中的第一项，即变量X。

BASIC-80输入/输出处理器现在扫描第二个数值域的开头，负号（－）是第二个数值域的开头。BASIC-80输入/输出处理器现在扫描第二个数值域的结束符，这是在－123.234和456之间的逗号。因此，当BASIC-80遇到这逗号时，就假定第二个数值域已结束。第二个数值域赋值给变量表中的第二项，即变量Y。

BASIC-80输入/输出处理器现在扫描第三个数值域的开头。数字4是第三个数值域的开头。BASIC-80输入/输出处理器将再扫描第三个数值域的终止符，这是在456后面的回车符。因此，当BASIC-80遇到这个回车时，它假定第三个数值域已结束，这第三个数值域赋值给变量表中的第三项，即变量Z。

到此，所有在变量表上的三个变量都已赋值，故INPUT#语句的执行已完成，继续执行下一条语句。

（2）字串输入

当BASIC-80扫描顺序数据文件中的字串时，开头的空格、回车及换行都被忽略。遇到第一个不是空格、回车、或换行的字符，即假定为字串的开头。

字串被认为不带引号，并用一个逗号、回车或换行来结束（或者已读满255个字符）。

如果第一个字符是引号，则该字串就被认为是一个带引号的字串，它将包括在第一个引号到下一个引号之间的所有字符。逗号、空格及回车等字符均可包括在此字串中，一个带引号的字串中不能再出现引号。

例如，假定数据象下面这样地存在盘文件上：

（b代表一个空白或空格——ASCII码32）

BEITON, HARBOR, MI "49022"〈回车〉

那么，语句：

```
INPUT#1, A$, B$, C$
```

将赋值如下：

```
A$=BENTON
B$=HARBOR
C$=MI "49022"
```

注意在上述例子中，逗号用作分隔符，所有三个字串都被认为是不带引号的。

在最后的字串域中，引号作为字串的一部分来考虑，这是因为字串用字母M开头，并用回车来结尾。

假定一个逗号插入MI及"49022"之间，则盘上的数据存储成下列形式：

```
BENTON, HARBOR, MI, "49022"
```

现在这里总共有4个字串域。前面是三个没有引号的字串域，而最后一个是带引号的字串域。这4个域可用下面语句输入：

```
INPUT#1, A$, B$, C$, D$
```

变量将赋值如下：

```
A$=CENTON
B$=HARBOR
```

C$ =MI

D$ =49022

变量D$不包括引号，因为引号是用来分隔字串域的，因此它不代表数据。

2. LINE INPUT# (从顺序文件输入整行)

格式：LINE INPUT#〈文件号〉,〈字串变量〉

LINE INPUT# 语句用于从一个顺序的盘文件读入一个完整行（如没有分隔符，最多为255个字符），赋值给一个字串变量。

〈文件号〉是用 OPEN 语句所定义的文件号。该文件必须定义成顺序输入文件（I 方式）。〈变量〉是将要赋值的变量名。

LINE INPUT# 读入顺序文件中直到回车为止的所有字符。随后跳过回车/换行（如果遇到的话），第二个 LINE INPUT# 语句又读入下一个回车前的所有字符。

如果没有发现回车，LINE INPUT# 将读满255个字符。这 255 个字符将赋值给字串变量。

当一个数据文件的每个域都是用回车来终止时，或者当一个用 ASCII 码形式保存的 BASIC-80 程序被另外一个程序当作数据读出时，LINE INPUT# 语句是特别有用的。

例如，假定下述程序保存在盘文件中：

```
10 OPEN "O", 1, "LIST"〈回车〉
20 INPUT C$〈回车〉
30 PRINT#1, C$〈回车〉
40 CLOSE#1〈回车〉
```

那么，语句：

```
LINE INPUT#1, Z$
```

可以用来重复地读每个程序行，每次一行。

3. PRINT# 和 PRINT# USING (写数据到顺序文件)

格式：

```
PRINT#〈文件号〉,〈表达式序列〉
PRINT#〈文件号〉, USING〈字串格式〉;〈表达式序列〉
```

PRINT# 语句用于将数据写入一个顺序的盘文件。〈文件号〉是当文件作为输出文件而打开时所用的文件号。在〈表达式序列〉中的表达式是将写入文件的数值和(或)字串表达式。

PRINT# 不将数据压缩存到盘上。数据写入磁盘的样子，正如用 PRINT 语句在终端上显示一样（PRINT 语句已在第四章“程序语句”中讨论）。所以，要注意盘上数据的分隔，以便能从盘上正确地读入。

在表达式序列中，数值表达式应该用分号来分开。

例如：

```
PRINT#1, A; B; C; X; Y; Z
```

如果逗号用作分隔符，则插在打印域中的附加的空格也将同样地被写入盘中。

字串表达式序列也用分号来分开。为了让字串以正确的格式存在盘上，在表达式序列中

必须另加分隔符。

例如，设A$＝“CAMERA”及B$＝“93604-1”

语句：

PRINT#1，A$；B$

会将CAMERA93604-1写到磁盘上。因为这里没有分隔符，输入时不能作为二个独立的字串。要纠正这个问题，必须把一个明确的分隔符插入PRINT#语句中：

PRINT#1，A$；“，”；B$

写入磁盘后，可得到：

CAMERA，93604-1

这就可被读回到二个字串变量中。

如果字串本身包括逗号、分号、有效的打头空格、回车或换行等，则应把它们用引号(CHR$（34））括起来写入磁盘。

例如，设A$＝“CAMERA，AUTOMATIC”及B$＝“93604-1”

语句：PRINT#1，A$；B$

将按如下形式将数据写入磁盘：

CAMERA，AUTOMATIC93604-1

而语句INPUT#1，A$，B$

将输入“CAMERA”到A$，“AUTOMATIC 93604-1”到B$。要正确地在盘上分隔这些字串，可加上双引号（CHR$(34)）再写入磁盘。例如语句

PRINT#1，CHR$(34)；A$；CHR$(34)；CHR$(34)；B$；CHR$(34)

将会按如下形式把数据写入磁盘：

“CAMERA，AUTOMATIC”“93604-1”

而语句INPUT#1，A$，B$

会将“CAMERA，AUTOMATIC”输入给A$，“93604-1”输入给B$。

PRINT#语句也可用USING可选项来控制盘文件的格式。例如：

PRINT#1，USING“$$###.##，”；J；K；L

格式串中最后的逗号用来分隔文件中的各项数据。若想完整地了解PRINT USING语句，请参见第八章“特殊功能”。

注意：WRITE#语句将自动地在顺序文件的数据项之间插入合适的分隔符。

4. WRITE#(写数据到顺序文件)

格式：WRITE#〈文件号〉，〈表达式序列〉

WRITE#语句用于将数据写入顺序文件。

〈文件号〉是OPEN语句中所定义的文件号，文件必须定义为顺序输出文件（O方式)。序列中的表达式是字符串或数值表达式，它们必须用逗号隔开。

WRITE和PRINT#之间的不同是WRITE#语句在把数据写到磁盘上去时，自动地在各项间插入逗号，而且引号照样写入作为分隔符。因此，用户不需要再设置另外的分隔符。写入磁盘文件变量表的最后一项的后面，还有一个回车/换行。

例如，设A$＝“CAMERA”及B$＝“93604-1”，则语句WRITE#1，A$，B$会在磁盘上写成下述形式：

"CAMERA", "93604-1"

而后面的INPUT#语句，象INPUT#1，A$，B$，将会输入"CAMERA"到A$，"93604-1"到B$。

注意：在顺序输出的大多数场合，我们都推荐使用WRITE#语句。使用顺序文件时，问题大多出在数据之间没有插入适当的分隔符。WRITE#语句免除了插入分隔符的麻烦，从而也就避免了顺序输入/输出时的大量错误。

在某些个别的顺序输出中，当WRITE#语句无法提供足够的灵活性时，可以考虑使用PRINT#或 PRINT# USING 语句。但必须注意确保所有的数据项要用正确的分隔符来分开。

3.2 顺序存取技术

1. 生成及存取一个顺序文件

下列步骤用于生成一个顺序文件和存取文件中的数据：

（1）打开顺序输出文件

```
OPEN"O", 1, "DATA.DAT"
```

这一步将 DATA.DAT 的文件号定义为1，既然已定义为O方式，则文件仅供顺序输出。在文件名中没有指明的驱动器，就是指主驱动器。

如果一个DATA.DAT 文件已存在于主驱动器盘上，则该文件的内容将丢失。这是由于当一个文件被打开作为顺序输出文件时，BASIC-80输入/输出处理器将把 EOF 标志移至文件的开头，因此该文件原先的内容不能再被读取。

（2）将数据写入文件

```
WRITE#1, A$, B$, C$
```

这一步骤假定某些字串值已预先赋给字串度量 A$、B$及C$。WRITE#语句将把数据带上分隔符号写入文件中，因此它不需要再插入任何分隔符。

PRINT#语句也能写数据到顺序文件中，但它必须在数据项间插入分隔符。因此在顺序输出的大多数应用场合，用 WRITE# 语句更为方便。

（3）关闭文件

```
CLOSE#1
```

这语句将把缓冲区中的所有残余数据写入盘文件，然后终止输出操作。文件在被打开作为顺序输入之前，必须先行关闭。

（4）重新打开作为输入文件

```
OPEN "I", 1, "DATA.DAT"
```

文件号 1 再次连上DATA.DAT文件，这次，文件打开作为顺序输入文件。

（5）读数据

```
INPUT#1, X$, Y$, Z$
```

数据将从DATA.DAT文件中读出，并赋值给字串度量 X$ Y$及Z$。

注意：上述例子忽略了顺序输入/输出处理中输入/输出缓冲区的动作。但是，实际上BASIC-80是按块（128字节）读写的，故每条 INPUT#或WRITE#语句可能不一定需要一次磁盘存取操作。

对于顺序输出，每个WRITE#或PRINT#语句将把数据放在缓冲区内。当缓冲区满

后，一次写入盘文件。

对于顺序输入，每次将128个字节读入缓冲区内，然后BASIC-80输入/输出处理器将在缓冲区中寻找数据，以对INPUT#语句中的变量进行赋值。

2. 加数据到一个顺序文件

一旦一个现存的顺序文件被打开作为输出文件（“O”方式），文件现行的内容将被破坏。因此，要分几个步骤把数据加到一个现有的顺序文件后面。下面就是加数据到一个现存的“DATA.DAT”文件后面的过程。

（1）将文件“DATA.DAT”打开作为顺序输入文件

```
OPEN "I", 1, "DATA.DAT"
```

这一步骤使文件号1连上数据文件DATA.DAT。这文件定义为顺序输入文件，因为在文件名中没有指明驱动器，BASIC-80将假定是现行主驱动器。如果文件DATA.DAT在现行主驱动器盘上没有找到，将产生一个“File not found”（文件未找到）的错误。

（2）打开第二个文件“TEMP.TMP”作为顺序输出文件

```
OPEN "O", 2, "TEMP.TMP"
```

文件TEMP.TMP将用作一个暂时的工作文件。整个过程完成后，此文件将被重新命名，并包含原有数据和新加的数据。

（3）在“DATA.DAT”中读数据并且写到“TEMP.TMP”中

```
INPUT#1, A$, B$, C$
WRITE#2, A$, B$, C$
```

此步骤必须重复执行，直至在文件#1中的数据全部读完。

（4）关闭“DATA.DAT”且删除它

```
CLOSE#1
KILL "DATA.DAT"
```

此文件已不再需要了，因为信息已从该文件中拷贝到文件TEMP.TMP中去了。

（5）将新数据写到“TEMP.TMP”中

```
WRITE#2, A$, B$, C$
```

赋值给字串变量A$、B$及C$的新数据将被写入盘文件。

（6）关闭文件

```
CLOSE#2
```

这一步骤终止对文件#2的输出操作。

（7）将文件“TEMP.TMP”改名为“DATA.DAT”

```
NAME "TEMT.TMP" AS "DATA.DAT"
```

现在盘上有了一个仍称为“DATA.DAT”的文件，它包括所有原先的和新加的数据。

第四节　BASIC-80随机输入/输出

生成及存取随机文件比顺序文件需要更多的步骤，但用随机文件有它的优点。其中之一是随机文件在盘上占用较少的存储空间，因为 BASIC-80 用压缩的二进制码形式存储随机文件中的数值，而顺序文件却把数据存成一串 ASCII字符。

随机文件的最大优点是数据可以随机存取，即在盘的任何地方存取——不需要象顺序文件那样读过所有信息，这是因为信息是保存在称为记录的一个个单元中，存取也是通过这些单元进行，而一个记录是编了号的。

存在随机文件中的所有数据必须是字串形式。

要在随机文件中存储数值，必须先将数值转换成字串，BASIC-80 提供了几个函数将数值转为字串形式。这些函数（MKI$、MKS$、MKD$）将在本章后部介绍。

4.1　随机存取语句

语句	功能
FIELD	设置随机文件缓冲区。
LSET	将数据装入随机缓冲区(左对齐)。
RSET	将数据装入随机缓冲区(右对齐)。
GET	读随机记录。
PUT	写随机记录。
MKI$	将整型数转换成 2 字节的字串。
MKS$	将单精度数转换成 4 字节的字串。
MKD$	将双精度数转换成 8 字节的字串。
CVI	将 2 字节的字串转换成整型数。
CVS	将 4 字节的字串转换成单精度数。
CVD	将 8 字节的字串转换成双精度数。

表 10-3　随机存取语句

1. FIELD（设置随机文件缓冲区）

格式：FIELD＃〈文件号〉，〈段宽度〉AS〈字串变量〉

FIELD 语句在随机文件缓冲区中为变量分配空间。

〈文件号〉是在OPEN语句中分配给随机文件的文件号。

〈段宽度〉是分配给〈字串变量〉的字符（字节）数。

例如语句 FIELD＃1，20 AS N$，10 AS ID$，40 AS ADD$

将会在随机文件缓冲区中，分配前面20个字节给字串变量N$，紧接着的10个字节给 ID$，再后面的40个字节给ADD$。FIELD 语句并不向随机文件缓冲区写任何数据，它仅定义缓冲区中的各段。

FIELD语句只能用于已打开的随机输入/输出（“R”方式）文件。FIELD语句必须放在任何随机文件输入/输出操作的前面。

在一条 FIELD 语句中分配给各变量的总字节数，不得超过文件被打开时所规定的记录长度。否则，将产生“Field overflow”（定段溢出）的错误（记录补缺长度是 128）。

如果所规定的记录长度小于128，则BASIC-80输入/输出处理器得照顾到将记录分块及拼合的问题。例如，如果在OPEN语句中规定记录长度为32字节，BASIC-80输入/输出处理器得把每个物理记录（扇区）分成4块逻辑记录。用户程序不必管这些逻辑记录的分块和拼合问题。

如果规定记录长度大于128，BASIC-80也得照顾到记录的分块及拼合的问题。大于128的记录长度必须在BASIC-80初始化时就用/S开关来设置，最大的记录长度为256字节。

在早先的Microsoft BASIC文本中，用户程序必须要考虑记录的分块及拼合问题。

同一文件可执行数条FIELD语句，而且所有的FIELD语句同时有效。例如，下述FIELD语句可用于设置32字节的随机缓冲区：

```
FIELD#1, 16 AS F1$, 16 AS F2$
```

此FIELD语句分配随机缓冲区中开头16个字节给变量F1$，后面的16个字节给变量F2$。然而，另一条FIELD语句可以用来重新定义缓冲区：

```
FIELD#1, 32 AS BUFF$
```

故变量BUFF$对应缓冲区中所有32个字符，F1$仍将对应前面16个字符，而F2$仍将对应后面16个字符。

不要在INPUT或LET语句中使用已用于定段的变量名。一旦变量名用于定段，它将指向随机文件缓冲区中的特定地址，而如果随后一句带有该变量名的INPUT或LET语句一旦被执行，变量的指针就转到字串存储空间。举例：

```
FIELD #1, 128 AS IBUFF$
FIELD #4, 10 AS A$(1), 10 AS A$(2), 10 AS A$(3)
FIELD #2, I AS STUFF$
```

（注意：在执行这条语句前，变量I必须赋值为整型数。）

2. LSET/RSET（将数据装入随机缓冲区）

格式：

```
LSET〈定段度量〉＝〈字串表达式〉
RSET〈定段度量〉＝〈字串表达式〉
```

LSET/RSET语句是特殊的赋值语句，它将字串表达式赋值给FIELD语句中出现的变量（定段度量）。

LSET/RSET语句将数据从内存装入随机文件缓冲区，这一步是PUT语句的准备工作，将数据装入随机缓冲区的唯一办法是用LSET/RSET语句。

如果〈字串表达式〉的字节数比分给〈定段变量〉的字节要少，那么LSET让字串在段中靠左对齐，右边加满空格；RSET让字串在段中靠右对齐，左边加满空格。

LSET与RSET之间仅有的不同是LSET是靠左对齐而RSET是靠右对齐。如果字串太长，一律从右边截掉。

数值在进行左对齐或右对齐之前，必须转换成字串。几个特殊的随机输入/输出函数已提供来完成这个转换（关于MRI$、MKS$及MKD$函数的讨论见本章后部）。举例：

```
150 LSET A$=MKS$(AMT)
```

```
160 LSET D$=DESC$
170 LSET V$= "LEFT-JUSTIFY AND PLACE IN BUFFER"
180 RSET G$= "RIGHT-JUSTIFY AND PLACE IN BUFFER"
```

这里，字串变量A$、D$、V$及G$必须已出现在前面的 FIELD 语句中。

3. GET（读随机记录）

格式：GET〔#〕〈文件号〉〔，〈记录号〉〕

GET 语句用于从随机盘文件读记录到随机缓冲区。在执行 GET 语句以前，被存取的文件必须已打开为随机文件。

另外，随机文件缓冲区必须已用 FIELD 语句定义好。如果随机文件缓冲区未被定义，则在执行GET语句后仍然无法取得数据。

〈文件号〉是文件打开时所定义的文件号。如果〈记录号〉省略，则现行记录被读入缓冲区。现行记录号比上一次存取的记录号大1。文件刚打开进行第一次存取时，现行记录号是1。最大可能的记录号是32767。

如果想要GET一个记录，其记录号大于文件中最后的记录号，虽然不会出现错误，但缓冲区中将填满NUL字符（ASCII0）。LOF函数可用于防止发生这种情况。举例：

```
GET #1, 100
GET #2,
GET FILE, IREC
GET #5,REC
```

4. PUT（写随机记录）

格式：PUT〔#〕〈文件号〉〔,〈记录号〉〕

PUT语句用于从随机缓冲区写一个记录到随机盘文件。在执行PUT语句前，被存取文件必须已打开为随机文件。

另外，必须先用 FIELD 语句定义好随机文件缓冲区。如果随机文件缓冲区未曾定义，则在执行PUT语句前无法将数据装入缓冲区。

〈文件号〉是文件被打开时所定义的文件号，如果〈记录号〉省略，则数据被写入现行记录。现行记录号比上一次存取的记录号大1。文件刚打开进行第一次存取时，现行记录号是1，最大可能的记录号是32767。

如果〈记录号〉大于文件结尾记录号，〈记录号〉则变成文件新的结尾记录号。存储空间将分配给这个新的结尾记录，以及记录号比结尾记录号低的所有新记录。举例：

```
PUT #1
PUT #2, 43
PUT I, J-1
PUT I, 4
```

5. MKI$、MKS$、MKD$（将数值转换成字串）

格式：MKI$（〈整型表达式〉）

MKS$（〈单精度型表达式〉）

MKD$（〈双精度型表达式〉）

这三个转换函数（MKI$、MKS$、MKD$）用于把数值转换成字串。任何装入随机

文件缓冲区中的数值都必须先转换成字串。

MKI$函数用于把一个整数转换成2字节的字串。整数必须在整数值允许范围内，如果不是这样，则将产生一个"Illegal function call"（非法函数调用）的错误。非整数的小数部分将被截去。

MKS$函数用于把一个单精度数转换成4字节的字串。MKD$函数用于把一个双精度数转换成一个8字节的字串。

这些函数并不将数据装入随机缓冲区。因此当一个数值转换成一个字串后，仍必须用语句将它装入随机文件缓冲区中。另外，随机文件缓冲区必须先用FIELD语句定义好。

如果事先未定义随机文件缓冲区，那么在执行GET语句后仍无法读取数据。另一方面数据必须用LSET或RSET语句装入随机缓冲区。

举例来说，将整型变量IV%转换成一个字串，并把它赋值给定段变量FV$，可用下述单条语句：

```
LSET  FV$=MFI$(IV%)
```

变量FV$必须已出现在前面的FIELD语句中。举例：

```
90   AMT=(K+T)
100  FIELD#1, 8  AS  D$, 20  AS  N$
110  LSET  D$=MKS$(AMT)
120  LSET  N$=A$
130  PUT  #1
```

6 CVI、CVS、CVD(将字串转换成数值形式)

格式：CVI(〈2字节字串〉)
CVS(〈4字节字串〉)
CVD(〈8字节字串〉)

CVI、CVS和CVD函数用于将字串转换为数值，这些函数一般是用于对从盘文件中所读出的字串进行转换。数据总是以字串形式存在随机文件中，因此，要读出存在随机文件中的数值量，必须把它从字串形式转换过来。

CVI函数把2个字节的字串转换成一个整数。如果字串的长度大于2个字节，则只有字串中前面两个字符被用来进行转换。如果字串长度小于2字节，那么将会产生"Illegal function call"（非法函数调用）的错误。

CVS函数把一个4字节的字串转换成一个单精度数。如果字串长度大于4个字节，则只有字串中前面4个字符被用来进行转换。如果字串长度小于4字节，则将产生"Illegal function call"（非法函数调用）的错误。

CVD函数把一个8字节的字串转换成一个双精度数。如果字串长度大于8字节，则在字串中只有前面8个字符被用到。如果字串长度小于8字节，那么将产生"Illegal function call"（非法函数调用）的错误。举例：

```
PRINT CVS(A$)
A#=CVS(BUFF$)
I=I+CVI(I$)
```

4.2 随机存取技术

1. 生成一个随机存取文件

生成一个随机文件需要下述步骤：

（1）将文件打开作为随机文件

```
OPEN "R", 1, "FILE.DAT", 32
```

在这个例子中，方式特征字符为"R"——随机存取。文件号1赋给文件 FILE.DAT。因为文件名没有包括驱动器名，所以就是指现行主驱动器。这个例子也规定了记录长度为32个字符（字节）。如果记录长度省略掉，则长度补缺值是128字符（字节）。

（2）设置随机文件缓冲区

```
FIELD#1, 20 AS NAM$, 4 AS A$, 8 AS P$
```

用 FIELD 语句在随机缓冲区中为变量分配空间，这些变量将写入随机文件。FIELD语句中用了文件号 1，它已打开作为随机文件。对顺序输入或输出的文件，不能使用 FIELD语句。

FIELD 语句将给变量 NAM$ 分配随机文件缓冲区的前面20个字节，紧接着的 4 个字节分配给变量 A$，再后面的 8 个字节分配给变量 P$。

（3）把数据装入随机缓冲区

```
LSET    NAM$=X$
LSET    A$=MKS$ (AMT)
LSET    P$=TEL$
```

用 LSET 可把数据装入随机缓冲区。数值装入缓冲区前必须先转换成字串。要实现这一点 可使用"转换函数"：MKI$ 将一个整数、MKS$ 将一个单精度数、MKD$ 将一个双精度数分别转换成字串。

在这一步中，单精度变量 AMT 首先被转换成字串，然后赋给变量 A$。变量 A$ 已出现在先前的 FIELD 语句中，这条语句分配了 4 个字符（字节）给变量 A$。

（4）写数据到盘上

```
PUT# 1
```

用 PUT 语句将数据从缓冲区写到盘上。因为这条 PUT 语句没有指明记录号，故数据被写入现行记录。现行记录号比最后存取的记录号大 1。文件刚打开进行第一次存取时，现行记录号是 1。

在 INPUT 或 LET 语句中不能使用定段字串变量。这会使得原来指向随机文件缓冲区的变量指针指到字串存储空间中去。

2 在随机文件中读取数据

对随机文件进行读取操作，需要下列程序步骤：

（1）将文件打开成为随机文件

```
OPEN "R",#1,"FILE.DAT",32
```

这一步打开随机文件"FILE.DAT"。这个文件现在能用文件号 1 来进行存取。

（2）设置随机文件缓冲区

```
FIELD#1, 20 AS NAM$, 4 AS A$, 8 AS P$
```

用 FIELD 语句在随机文件缓冲区中为变量分配空间。这些变量将用于在文件中读取数据。在这个例子中，20个字节分配给字串变量 NAM$，4 个字节分配给字串变量 A$，

而 8 个字节分配给字串变量 P$。

注意：对同一个随机文件进行输入或输出操作，程序中通常只用一条 OPEN 语句和一条 FIELD 语句。

（3）读数据到缓冲区

```
GET#1
```

用 GET 语句将所需的记录读入到随机缓冲区中，因为没有指明记录号，故现行记录将被读入。现行记录号比最后存取的记录号大 1。文件刚打开进行第一次存取时，现行记录号为 1。

（4）在缓冲区中读取数据

程序现在可以对缓冲区中的数据进行操作了。数据必须用“转换函数”从字串形式转换回来。在这些转换函数中，CVI、CVS、CVD分别将字串转换成整数、单精度数和双精度数。

```
PRINT NAM$
AV=CVS(A$)
DP#=CVD(P$)
```

4.3 附加性质

在 GET 语句之后，INPUT# 和LINE INPUT# 语句可以从随机文件缓冲区中读入字符，PRINT#、PRINT# USING 及WRITE# 语句也可在 PUT 语句前把字符存入随机文件缓冲区。

在使用 WRITE# 语句的情况下，BASIC-80将缓冲区空间用空格填满（如果需要的话），然后插入一个回车。任何想越过缓冲区末尾去读写的企图，将导致“Field overflow”（定段溢出）的错误。

第十一章　BASIC-80提要

这一章将对BASIC-80程序设计语言的重要概念、观点、关键字等进行概括，各种内部函数和字串函数也包括在内。

一、缩写词

缩写	功能
?	代替 PRINT
'	代替 REM
.	“当前行”，代替 LIST、EDIT 等命令中的行号

二　数据类型标志符

字符	数据类型	示例
$	字串	ZDS$，WLW$
%	整数	I%，VALUE%
!	单精度数	V!，FLAG!
#	双精度数	DP#，PL#
D	双精度数（指数计数法）	1.23456789D-12
E	单精度数（指数计数法）	1.23456E+23

三、算术运算符

操作符	含义
+	加
−	减
*	乘
/	除（浮点）
\	整除
^	乘幂

四、字串操作符

操作符	含义	示例
+	连接（使字串连接在一起）	"A"+"B"+"C"

五　关系运算符

操作符	（用于）数值表达式	（用于）字串表达式
<	小于	领先
>	大于	随后
=	等于	相同
<=或=<	小于等于	领先或相同
>=或=>	大于等于	随后或相同
<>或><	不等于	不相同

六、逻辑运算符

操作符	功能
NOT	按位非
AND	按位与
OR	按位或
XOR	按位异或
IMP	按位隐含
EQU	按位相等

七 命令

命令/功能	示例
AUTO〈行号〉 〈增量〉 由〈行号〉开始自动进行编号，每次增加一个〈增量〉。	AUTO AUTO 10 AUTO 5,5
CLEAR 置数值变量为零，字串变量为空串。	CLEAR
CLEAR,〈表达式〉 与CLEAR相同，但〈表达式〉设置了BASIC-80可用的内存最高界限。	CLEAR,32768
CLEAR 〈表达式1〉,〈表达式2〉 与CLEAR,〈表达式〉相同，但〈表达式2〉设置了BASIC-80可用的堆栈空间。	CLEAR,32768,2000
CONT 在BREAK或STOP之后，使程序继续运行。	CONT
DELETE〈行号〉 在现行程序中，删除指定行号的那一行。	DELETE 100
DELETE-〈行号〉 删除现行程序中从头直到〈行号〉所指定的行号为止的每一行。	DELETE-500
DELETE〈行号〉—〈行号〉 删除现行程序中从第一个行号开始到第二个行号为止的每一行。	DELETE 10-1000
EDIT〈行号〉 在指定的行号上进入编辑状态。	EDIT 100
FILES〈"文件名"〉 列出现行磁盘上的文件名。	FILES"*.BAS"
LIST 从第一行开始列出在内存里的现行程序。	LIST
LIST〈行号〉 列出指定行号的那一行。	LIST 100
LIST〈行号〉—〈行号〉	LIST 10-100

列出从第一个行号开始到第二个行号为止的所有程序行。

LLIST

列出在内存里的整个或部分的现行程序，并在打印机上打印出来。可选项与LIST命令相同。

```
LLIST
LLIST 500
LLIST 150-
LLIST-100
LLIST 150-400
```

LOAD〈"文件名"〉.〔R〕

从磁盘上装入一个程序进入内存。R是可选项，如果用上它，程序装入内存以后，立即开始运行。

```
LOAD"B:GAME"
LOAD"PROG.ASC",R
```

MERGE〈"文件名"〉

将一个磁盘文件并入内存程序。

```
MERGE"B: TEST.BAS"
```

NEW

删除现行程序并清除所有变量。

```
NEW
```

RENUM〈nn〉,〈mm〉,〈ii〉

对程序行重新编号，将原先的〈nn〉行作为〈mm〉行，并从〈mm〉开始，以〈ii〉为递增量向后编号。

```
RENuM
RENUM 300, 5
RENUM 1000,900,20
```

RESET

在主驱动器换盘。

```
RESET
```

RUN〈行号〉

从指定行号开始执行现行程序。如果行号不指定，就从最小行号开始执行。

```
RUN 100
RUN
```

RUN〈"文件名"〉.R

从磁盘上调入一个程序并执行它。可选项R使所有的数据文件保持打开。

```
RUN"PROG1"
RUN"B:GAME",R
```

SAVE"文件名".A

SAVE"文件名".P

将现行程序存入磁盘。如果选用A，该文件就是以ASCII码格式存盘；如果选用P，该文件就是以保护的格式存盘；如果两种皆不使用，该文件就是以压缩二进制码形式存盘。

```
SAVE"COM2",A
SAVE"TEST1"
SAVE"INVEN",P
```

SYSTEM

关闭所有文件，CP/M热启动。

```
SYSTEM
```

八 编辑状态子命令

命令	功能
RETURN	结束编辑，并返回命令状态。
〈i〉SPACE BAR	光标右移〈i〉格。

〈i〉 BACK SPACE	光标左移〈i〉格。
L	显示程序行的剩余部分，并将光标移到该程序行的下一行的起始位置。
X	显示程序行的剩余部分，光标移至本行的结尾，并自动进入插入状态
I	在光标的现行位置开始插入文字。用 ESC 键退出插入状态。
A	编辑更改作废，并将光标返回到起始位置。
E	结束编辑，保存所有的更改，并返回命令状态。
Q	结束编辑，所有的更改作废，并返回命令状态。
H	删除本行的剩余部分，并自动进入插入状态。
〈i〉D	删除指定数目的字符，从光标现行位置开始往后删。
〈i〉C	用键入的〈i〉个字符，去更换（代替）现行光标位置后面的〈i〉个字符。
〈i〉S〈C〉	光标移动到字符〈C〉第〈i〉次出现的位置上，搜索从现行光标位置开始
〈i〉K〈C〉	删除从现行光标位置到字符〈C〉第〈i〉次出现的位置之间的所有字符。

九 格式打印域界标志符

数值标志符	功能	示例
#	数值域	# # #
.	十进制小数点位置	# # . # #
+ -	打印在数字前面的符号（加表示正，减表示负）	+ # # . # #
-	仅当打印负值时，才让负号打印在数的后面。	# # . # #-
* *	用星号填满打印的空格。	* * # # . # #
$ $	将 $ 号打印在数字的左侧。	$ $ # # # . # #
* * $	用星号填满打头空格并在数字左侧加上 $ 号。	* * $ # # . # #
,	用逗号分隔每三位数字。	# # , # # # . # #
^^^^	指数格式 尾数部分被转换成规范形式，即小数点后面第一位小数不为 0 。	# . # # ^^^^

字串标识符	功能	示例
!	单个字符	!
\ 〈space〉 \	定义（2 + 空格数）个字符位置	\ \
&	可变长度的字串域	&

文字标识符	功能	示例

十、程序语句

1. 数据类型定义

语句/功能	示例
DEFINT〈字母范围〉 宣称此范围中的字母是整数型变量。	DEFINT I-N
DEFSNG〈字母范围〉 宣称此范围中的字母是单精度型变量。	DEFSNG A-H, O-P
DEFDBL〈字母范围〉 宣称此范围中的字母是双精度型变量。	DEFDBL X, Y, Z
DEFSTR〈字母范围〉 宣称此范围中的字母是字符串型变量。	DEFSTR A-C, Z

2. 赋值语句和内存分配语句

语句/功能	示例
DIM〈带下标的变量〉 分配数组的存储空间	DIM A(20), B(12, 2)
OPTION BASE n 宣称数组下标的最小值。补缺值为0，可以被修改为1。	OPTION BASE 1
ERASE〈数组名表〉 从程序中清除数组。	ERASE A, B
LET〈变量〉=〈表达式〉 表达式的值赋给变量。	LET SUM=A+B+C
REM〈注释内容〉 在程序中插入注释。	REM GRP IS CROSS PAY
SWAP〈变量〉, 〈变量〉 两个变量的值进行变换。	SWAP A, B

3. 执行顺序的控制

语句/功能	示例
END 终止程序运行，关闭所有文件，返回命令状态。	100 END
FOR〈V〉=〈X〉TO〈Y〉STEP〈Z〉 允许一系列语句重复执行。	FOR I=1 TO 100
GOSUB〈行号〉 转移到由〈行号〉开始的子程序。	GOSUB 100
GOTO〈行号〉 转移到指定行号的语句。	GOTO 400
NEXT〈变量〉 终止FOR循环。	NEXT I

ON〈表达式〉GOTO 行号1，……，行号K

先求表达式值。如果INT（〈表达式〉）的值是1～K等值中的某一个，则控制转到相应的这条行号语句去执行，否则继续执行下一条语句。

```
ON LI GOTO
10, 20, 30
```

ON〈表达式〉GOSUB 行号1，……，行号K

与ON……GOTO语句相同，但转往相应子程序。

```
ON L GOSUB
100, 200, 300
```

RETURN

子程序结束并返回。从调用这个子程序的GOSUB语句的下一条语句开始继续执行。

```
RETURN
```

STOP

结束程序运行，返回到命令状态。

```
STOP
```

4. 条件判断执行

语句/功能	*示例*
IF〈表达式〉THEN〈语句〉 ELSE〈语句〉 先求表达式的值，如果为真，则执行THEN语句，如果为假，则执行ELSE语句。	IF A=0 THEN A=1 ELSE A=0
WHILE〈表达式〉 〈循环语句〉 WEND 只要条件为真，则反复执行循环体内的语句序列。	WHILE A=0 PRINT "ZERO"WEND

5. 非磁盘输入/输出语句

语句/功能	*示例*
INPUT(；)〈"提示"；〉〈变量名表〉 程序运行时，从键盘输入数据。	INPUT "AGE" A
LINE INPUT〈；〉〈"提示"，〉〈字串变量〉 输入一行字串（直到255个字符）给字串变量，不需要分隔符。	LINE INPUT J$
DATA〈常量表〉 存储数值和字串常量，用READ语句将这些常量赋给变量。	DATA 34, 23.1, 45.0 DATA "HELLO","BYE"
PRINT〈表达式序列〉 向终端输出数据。	PRINT "HELLO" PRINT A$, Z, C
READ〈变量表〉 从DATA语句中，读入数据赋给指定的变量。	READ I, A, B READ A$, B$
RESTORE〈行号〉	RESTORE

重新设置读指针，让数据可重新读取。

LPRINT〈表达式序列〉　　LPRINT "HELLO"

在打印机上打印数据。

十一　字串函数

函数	操作	示例
ASC (X$)	产生字串变量中第一个字符的ASCII码值	ASC ("B")
CHR$(I)	给出ASCII码值等于I的那个字符。	CHR$(66) CHR$(N)
HEX$(I)	将一个数转换成十六进制。	HEX$(100)
INKEY$	从键盘上读入一个字符。	A$=INKEY$
INPUT$(X, Y)	从键盘或文件Y中读X个字符。	INPUT$(1, 1)
INSTR(I,X$, Y$)	返回字符串Y$在X$中第一次出现的位置，从X$的第I个位置开始找起。	INSTR(A$, ", ")
LEFT$(X$, I)	给出X$的右边I个字符	LEFT$(A$, 1) LEFT$(C$, 3)
LEN(X$)	返回字符串X$的长度。	LEN(A$)
MID$(X$,I,J)	返回X$的一个子串，从X$的第I个字符开始．取J个字符。	MID$(X$, 5, 10)
MID$(X$, I, J)=Y$	在字符串X$中，从第I个字符开始，用Y$字串来取代。J是将要被取代的字符的个数。	MID$(A$,1,2)= "Z"
OCT$(X)	将数X转换成八进制。	OCT$(24)
RIGHT$(X$,I)	给出字串X$的右边I个字符。	RIGHT$(X$, 8)
SPACE$(X)	给出X个空格组成字串。	SPACE$(20)
STR$(X)	将一个数值表达式转换成字串。	STR$(100)
STRING$(I, J)	产生一个长度为I，ASCII码为J的字符所组成的字串。	STRING$(20, 33)
STRING$(I,X$)	产生一个长度为I,由字串X$的第一个字符所组成的字串。	STRING$(20, "1")
VAL(X$)	将一个字串X$转换一个数值。	VAL ("3.14")

十二　算术函数

函数	操作	示例
ABS(X)	给出X的绝对值。	ABS(-1)
ATN (X)	给出X的反正切值（值为弧度）。	ATN (3)
CDBL (X)	把X转换为双精度数。	CDBL (A)
CINT (X)	用四舍五入法把X转换成整数。	CINT (46.6)

COS (X)	给出X的余弦（X为弧度）。	COS (A+B)
CSNG (X)	把X转换成单精度数。	CSNG(V)
EXP (X)	给出e^x的值，这里e≈2.718。	EXP (34.5)
FIX (X)	取X的整数部分。	FIX (23.2)
INT (X)	取不大于X的最大整数。	INT (−12.11)
LOG (X)	取X的自然对数lnX，X必须大于0。	LOG (45/7)
RND (X)	给出一个在〔0，1〕范围内的随机数。	RND (0)
SGN (X)	给出X的符号。 $SGN(X)=\begin{cases}1 & \text{当X>0时}\\0 & \text{当X=0时}\\-1 & \text{当X<0时}\end{cases}$	SGN (C/A)
SIN (X)	给出X的正弦（X为弧度）。	SIN (A*1.3)
SQR (X)	给出X的平方根$\sqrt{X}$，X不能是负数。	SQR (A*B)
TAN (X)	给出X的正切（X为弧度）。	TAN (X+Y+Z)

十三、特殊函数

函数	操作	示例
FRE (X)	给出BASIC-80没有使用的自由内存空间。	FRE (0)
INP (I)	给出从口I读入的字节。	INP (255)
LPOS (X)	给出打印机缓冲区内的打印头的现行位置。	LPOS (0)
NULL (X)	设置每行末端空字符的数量。	NULL (3)
OUT I, J	送字节J到口I中。	OUT 127, 255
PEEK (I)	从指定的内存地址中读一个字节。	PEEK (8192)
POKE I, J	送字节J到内存单元I中。	POKE (8192, 200)
POS (X)	给出现行光标位置。	POS (1)
SPC (I)	在终端上显示I个空格。	PRINT SPC (5)
TAB (I)	从第I＋1格位置起始显示成打印。	PRINT TAB (20)
VARPTR (X)	给出内存中变量的地址。	VARPTR (V)
WAIT I,J〔, K〕	口I的状态字先与K进行异或（XOR），再与J进行与（AND）运算，如结果不为0，则程序继续执行。	WAIT 21, 1
WIDTH I	设置显示行宽度。	WIDTH 80

WIDTH LPRINT 设置行打印机宽度。　　WIDTH LPRINT 132

I

十四　特殊功能

1　错误陷阱

语句/功能	示例
ON ERROR GOTO 〈行号〉 错误陷阱赋能，并定义错误陷阱程序的首行。	ON ERROR GOTO 100
RESUME 〈行号〉 在一个错误处理过程完成后继续执行程序。	RESUME RESUME NEXT RESUME 100
ERROR 〈整数表达式〉 模拟错误的发生，并允许用户定义错误代码。	ERROR 10
ERL 错误所在行的行号。	PRINT ERL
ERR 错误代码。	PRINT ERR

2．跟踪标志

语句/功能	示例
TRON 设置跟踪标志。	TRON
TROFF 撤消跟踪标志。	TROFF

3．覆盖处理

语句/功能	示例
CHAIN 〔MERGE〕 "〈文件名〉" 〔, 〔〈行号〉〕〔, ALL〕〔, DELETE 〈范围〉〕〕 调用程序，并从现行程序传递变量到调用程序。	CHAIN "PROG"
COMMON 〈变量表〉 传送变量到一个链接的程序中去。	COMMON A, B

十五、磁盘输入/输出语句

语句/功能	示例
CLOSE # 〔〈文件号〉〔, 〈文件号〉〕 关闭磁盘文件，如果没有提供参数，则关闭所有打开的文件。	CLOSE #6
FIELD # 〈文件号〉, 〈段宽度〉AS 〈字串变量〉 把随机缓冲区空间分配给〈字串变量〉，这里〈文件号〉是与这个随机缓冲区相连的那个文件的文件号，〈段宽度〉为给定的〈字串变量〉预留空间。	FIELD #1, 3 AS A$

GET #〈文件号〉[,〈记录号〉]　　GET#1, I

从随机文件〈文件号〉的〈记录号〉记录中传递数据到随机缓冲区。如果〈记录号〉省略，那么送下一个记录。

INPUT #〈文件号〉,〈变量表〉　　INPUT#3, A, B

从文件〈文件号〉中读数据，并且赋值给〈变量表〉中的元素。

KILL "〈文件名〉"　　KILL "A: GAME BAS"

删除一个磁盘文件。

LINE INPUT #〈文件号〉,〈字串变量〉　　LINE INPUT#1, A$

从文件〈文件号〉中读一整行，并且赋给〈字串变量〉。

LSET〈字串变量〉＝〈字串表达式〉　　LSET A$="HELLO"

以左对齐形式将数据装入随机文件缓冲区中。

OPEN〈方式〉,[#]〈文件号〉,〈"文件名"〉　　OPEN "O", 1, "GM.DAT"

打开一个磁盘文件。这里〈方式〉是文件操作方式，〈文件名〉是磁盘目录项。

PRINT #〈文件号〉,〈表达式序列〉　　PRINT #1, A$, B

把数据写入顺序磁盘文件。

PUT #〈文件号〉[,〈记录〉]　　PUT #2, 3

从随机文件缓冲区传送数据到随机文件〈文件号〉中，如果〈记录号〉省略，那么写入现行记录。

RSET〈字串变量〉＝〈字串表达式〉　　RSET B$="BYE"

以右对齐形式将数据装入随机文件缓冲区中。

WRITE #〈文件号〉,〈表达式序列〉　　WRITE #2, A, B$

把数据写入顺序磁盘文件，分隔符自动插入各项数据之间。

十六　磁盘输入/输出函数

函数	操作	示例
CVD (X$)	把8字节的字串转换成双精度数。	A#=CVD (A$)
CVI (X$)	把2字节的字串转换成一个整数。	I%=CVI (I$)
CVS (X$)	把4字节的字串转换成单精度数。	B=CVS (B$)
EOF (文件号)	如果文件结束，则返回真值(−1)。	IF EOF(1)
LOC (文件号)	对随机文件给出所读的那个记录号，对顺序文件给出访问的扇区号。	X=LOC(1)
MKD$ (Z#)	把双精度数转换成8字节的字串。	A$=MKD$(A#)
MKI$ (I%)	把一个整数转换成2字节的字串。	I$=MKI$(I%

MKS$ (B)	把一个单精度数转换成 4 字节的字串。	B$ =MKS$(B)

附　录

附录A　错误信息

在错误出现后，BASIC-80返回到命令状态，并打印OK（但是上溢和除数为0的情况发生时，不会使BABIC-80停止运行）。程序和变量值都保持完整，但是不能用CONT命令，而应该用GOTO命令来使程序继续运行。

错误信息的形式是：

直接语句：〈错误信息〉

间接语句：〈错误信息〉in nnnn

这里nnnn是出错行的行号。当一个错误出现在一个直接语句中时，没有行号印出。

如果出现没有错误代码的错误，那么MBASIC-80将显示信息"Unprintable error"（不可显示错误）。

下面按错误代码列出错误信息。

一、一般错误

1. NEXT WITHOUT FOR　有NEXT，而没有FOR

对于NEXT语句中的变量，并没有预先执行的FOR语句和它对应。

2. SYNTAX ERROR　语法错误

遇到不正确的字符序列（如括号不匹配、错误语句或命令、不正确的标点等）。

3. RETURN WITHOUT GOSUB　有RETURN，而没有GOSUB

在GOSUB执行之前就遇到了RETURN语句。

4. OUT OF DATA　数据不够读

欲执行一条READ语句，但程序中所有DATA语句都已读完。

5. ILLEGAL FUNCTION CALL　非法函数调用

传给算术函数或字串函数的参数超出规定的范围。出现非法函数调用的原因可能有：

1）负的数组下标（LET A(−1)=0）。

2）过大的数组下标（>32767）。

3）负数或0取对数。

4）负数开平方。

5）在乘方运算A↑B中，A是负数，或B不是整数。

6）在机器语言子程序的地址输入之前，调用USR函数。

7）调用MID$、LEFT$、RIGHT$、INP OUT WAIT PEEK POKE、TAB、SPC、STRING$、SPACE$ INSTR或者ON…GOTO时用了不合适的自变量。

6. OVERFLOW　上溢

运算的结果太大，不能用BASIC-80数字格式表示。如果出现下溢（即数字太小），则结果为零，程序继续执行而没有任何错误信息显示。

7. OUT OF MEMORY 内存不够

程序太大，变量太多，FOR循环太多，或者表达式太复杂。

8. UNDEFINED NUMBER 没有定义的行号

在GOTO、GOSUB、IF…THEN…ELSE或DELETE等语句中，写上了并不存在的行号。

9 SUBSCRIPT OUT OF RANGE 下标越界

试图使用超出定维范围的数组元素或者使用错误的下标个数。

10. DUPLICATE DEFINITION 重复定义

在一个数组定维后，又有一条定维语句给这个数组定维，错误经常发生在这样的场合：已经使用了数组定维的补缺形式（下标最大为10），而后又在这个程序中用DIM语句，对这个数组定维。

11. DIVISION BY ZERO 除数为零

在表达式中除数为0，或者负指数幂的底为0。在前一种情况下，机器提供负无穷大为结果；在后一种情况下，机器提供正无穷大为结果，而程序继续执行。

12 ILLEGAL DIRECT 非法的直接语句

在直接命令状态中执行非直接方式的语句。

13 TYPE MISMATCH 类型不匹配

字串变量赋了数值，或数值变量赋了字串值。应该以数值变量作为参数的函数中用了字串变量，或者应该以字串变量作为参数的函数中用了数值变量。

14 OUT OF STRING SPACE 字串存储空间不够

字串变量太多，使得BASIC-80运行时所有的内存空间都被占用。BASIC-80会动态分配字串空间，直到内存用完为止。

15. STRING TOO LONG 字串太长

企图建立一个长度大于256个字符的字串。

16 STRING FORMULA TOO COMPLEX 字串表达式太复杂

字串表达式太长或者太复杂。应把表达式分解成较短的表达式。

17. CAN'T CONTINUE 不能继续执行

企图对下列情况继续执行程序，会产生这种错误：

1) 由于错误引起的暂停。

2) 在程序运行暂停过程中，修改了程序。

3) 程序不存在。

18. UNDEFINED USER FUNCTION 调用了没有定义的用户函数

19. NO RESUME 没有RESUME语句

BASIC-80进入错误陷阱程序，但是程序在遇到RESUME语句之前就结束了。

20. RESUME WITHOUT ERROR 遇到RESUME而没有遇到错误

没有进入错误陷阱程序就遇到了RESUME语句。

21. UNPRINTABLE ERROR 不可显示的错误

错误无法描述，这通常是遇到一个没有定义错误代码的错误。

22. MISSING OPERAND 缺操作数

表达式求值时，遇到了没有操作数的操作符。

23. LINE BUFFER OVERFLOW 行缓冲区溢出

企图输入一个字符太多的行.

24 FOR WITHOUT NEXT 有FOR而无NEXT

程序中有FOR语句，但没有与之匹配的NEXT语句。

25 WHILE WITHOUT WEND 有WHILE而无WEND

程序中有WHILE语句，而无与之匹配的WEND语句.

26. WEND WITHOUT WHILE 有WEND而无WHILE

程序中有WEND语句，但没有与之匹配的WHILE语句。

二、有关磁盘操作的错误

50. FIELD OVERFLOW 定段溢出

企图在缓冲区中分配比指定的随机文件记录长度更多的字节。

51. INTERNAL ERROR 内部错误

BASIC-80内部功能紊乱.请把出现的错误情况和所有有关数据向ZENITH公司 汇报。

52 BAD FILE NUMBER 错误文件号

使用了未打开的文件号或文件超出了系统初始化时确定的文件号的范围。

53 FILE NOT FOUND 文件找不到

LOAD、KILL或OPEN语句调用的文件不存在。

54. BAD FILE MODE 错误文件方式

企图对随机文件进行PRINT或WRITE操作，或者打开一个已经打开的随机文件去做顺序文件操作，企图对顺序文件进行PUT或GET操作，企图LOAD一个随机文 件，或 企图对不是定义为I，O及R等方式的文件执行OPEN语句。

55 FILE ALREADY OPEN 文件已经打开

企图打开已经打开的文件，或删除一个打开着的文件。

57. DISK I/O ERROR 磁盘I/O错误

在磁盘I/O操作时出现错误，它是一个致命性的错误，操作系统无法恢复它。

58. FILE ALREADY EXISTS 文件已经存在

NAME语句给出的文件名与磁盘上已有的文件名相同。

61. DISK FULL 磁盘满

磁盘上所有的存储空间都被占用。

62 INPUT PAST END 越过文件末尾读入数据

在文件中的所有数据已全部输入后，或者对于一个空文件，执行INPUT语句。可使用EOF 函 数来检查文件的结束以避免这种错误。

63. BAD RECORD NUMBER 错误记录号

在PUT或GET语句中，记录号大于最大允许数（32768）或者等于0。

64 BAD FILE NAME 错误文件名

在LOAD SAVE KILL或OPEN语句中，使用了非法的文件名。

66 DIRECT STATEMENT IN FILE 在文件中有直接语句

在装入一个ASCII码形式的文件的过程中遇到了直接语句，装入过程被终止。

67. TOO MANY FILES 文件太多

当255个目录项都用完时，还企图用SAVE或OPEN语句去建立一个新文件。

三. 保留字

BASIC-80用来作为语句、命令、操作符等的词称为保留字。保留字不能用来做变量名和用户自定义函数名。保留字表如下。注意内部函数名也算保留字。

ABS	AND	ASC	ATN
AUTO	BASE	CALL	CHAIN
CINT	CDBL	CHR$	CLEAR
CLOSE	COMMON	CONT	COS
CSNG	CVD	CVI	CVS
DATA	DEF	DEFDBL	DEFINT
DEFSNG	DEFSTR	DEFUSR	DELETE
DIM	EDIT	ELSE	END
EOF	ERASE	ERL	ERR
ERROR	EXP	FIELD	EILES
FIX	FN	FOR	FRE
GET	GOSUB	GOTO	HEX$
IF	IMP	INKEY$	INP
INPUT	INSTR	INT	KILL
LEFT$	LEN	LET	LINE
LIST	LLIST	LOAD	LOC
LOF	LOG	LPOS	LPRINT
LSET	MERGE	MID$	MKD$
MKI$	MKS$	MOD	NAME
NEW	NEXT	NOT	NULL
OCT$	ON	OPEN	OPTION
OR	OUT	PEEK	POKE
POS	PRINT	PUT	RANDOMIZE
READ	REM	RENUM	RESET
RESTORE	RESUME	RETURN	RIGHT$
RND	RSET	RUN	SAVE
SGN	SIN	SPACE$	SPC
SQR	STEP	STOP	STR$
STRING$	SWAP	SYSTEM	TAB
TAN	THEN	TO	TROFF
TRON	USR	VAL	VARPTR
WAIT	WEND	WHILE	WIDTH
WRITE	XOR		

表 A-1 保留字表

附录B　ASCII码

一、　十进制　八进制　十六进制和ASCII码之间的转换表

I				II				III				IV			
DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	CED	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII
0.	000.	00.	NUL	32.	040.	20.	SPACE	64.	100.	40.	@	96	140.	60.	`
1.	001.	01.	SOH	33.	041.	21.	!	65.	101.	41.	A	97	141.	61.	a
2.	002.	02.	STX	34	042	22.	"	66.	102.	42.	B	98.	142	62.	b
3.	003.	03.	ETX	35.	043.	23.	#	67.	103.	43.	C	99.	143.	63.	c
4.	004.	04.	EOT	36.	044.	24	$	68.	104.	44.	D	100.	144	64.	d
5.	005.	05.	ENQ	37.	045.	25.	%	69.	105.	45.	E	101.	145	65.	e
6.	006.	06.	ACK	38.	046.	26.	&	70.	106.	46.	F	102.	146.	66.	f
7.	007.	07.	BEL	39.	047.	27.	'	71.	107.	47.	G	103.	147.	67.	g
8.	010.	08	BS	40.	050.	28.	(	72.	110.	48.	H	104.	150.	68.	h
9.	011.	09	HT	41.	051.	29.	)	73.	111.	49.	I	105.	151.	69	i
10.	012.	0A	LF	42.	052.	2A.	*	74.	112.	4A.	J	106.	152.	6A	j
11.	013.	0B	VT	43.	053.	2B	+	75.	113.	4B	K	107.	153	6B	k
12.	014.	0C.	FF	44.	054.	2C	,	76.	114.	4C.	L	108.	154	6C	l
13.	015.	0D.	CR	45.	055.	2D.	−	77.	115.	4D	M	109.	155	6D	m
14.	016.	0E	SO	46.	056.	2E.	PERIOD	78.	116.	4E.	N	110.	156	6E	n
15.	017.	0F	SI	47.	057.	2F.	/	79.	117.	4F	O	111.	157	6F	o
16.	020.	10.	DLE	48.	060.	30.	0	80.	120.	50.	P	112.	160.	70.	p
17.	021.	11.	DC1	49.	061.	31.	1	81.	121.	51.	Q	113.	161.	71.	q
18.	022.	12	DC2	50.	062.	32.	2	82.	122.	52.	R	114.	162.	72.	r
19.	023.	13.	DC3	51.	063.	33.	3	83.	123.	53.	S	115.	163.	73.	s
20.	024.	14.	DC4	52.	064.	34.	4	84.	124.	54.	T	116.	164.	74.	t
21.	025.	15.	NAK	53.	065.	35.	5	85.	125.	55.	U	117.	165.	75.	u
22.	026.	16.	SYN	54.	066.	36.	6	86.	126.	56.	V	118.	166.	76.	v
23.	027.	17	ETB	55.	067.	37.	7	87.	127.	57.	W	119	167.	77.	w
24.	030.	18.	CAN	56.	070.	38.	8	88.	130.	58.	X	120.	170.	78.	x
25.	031.	19.	EM	57.	071.	39.	9	89.	131.	59.	Y	121.	171.	79.	y
26.	032.	1A.	SUB	58.	072.	3A.	:	90.	132.	5A.	Z	122.	172.	7A.	z
27.	033.	1B	ESC	59.	073.	3B	;	91.	133.	5B	[	123.	173.	7B	{
28	034.	1C.	FS	60.	074.	3C	<	92.	134.	5C.	\	124.	174.	7C	|
29.	035.	1D	GS	61.	075.	3D	=	93.	135.	5D.	]	125.	175.	7D	}
30.	036.	1E	RS	62.	076.	3E.	>	94.	136.	5E.	△	126.	176.	7E	~
31.	037.	1F	US	63.	077.	3F.	?	95.	137.	5F.	—	127.	177.	7F	DELETE

二、控制字符的定义

NUL　空，走纸
SOH　引导孔开始，信息开始
STX　文本开始，地址结束
ETX　文本结束，信息结束
EOT　传送结束，关闭TWX机器
ENQ　询问，WRU
ACK　应答，RU
BEL　响铃
BS　退格
HT　水平TAB
LF　换行或空白
VT　垂直TAB
FF　换页
CR　回车
SO　移出（SHIFT OUT）
SI　移入（SHIFT IN）
DLE　数据链逸出
DC_1　设备控制1，读出器打开
DC_2　设备控制2，凿孔机打开
DC_3　设备控制3，读出器关闭
DC_4　设备控制4，凿孔机关闭
NAK　否定应答，错误
SYN　同步（SYNC）
ETB　传送快结束，载体逻辑终点
CAN　作废（CANCEL）
EM　载体终止
SUS　取代
ESC　逸出
FS　文件分隔符
GS　组分隔符
RS　记录分隔符
US　单元分隔符

请参考ASCII码的转换表（见附录B—1）。注意：

以上定义的且处在图表第一列中的任何打印控制字都可以通过CTRL键与在第三或第四列上属于同一行的字符结合（将字符放在CTRL的右边）而得到。例如：DLE就是CTRL-P或∧P，BEL是CTRL-G或∧G，等等。

附录C　BASIC-80的新功能*

1. 新保留字

BASIC-80采用的新的保留字有：CALL、CHAIN、COMMON、WHILE、WEND、WRITE、OPTION BASE、RANDOMIZE等。

2. 类型转换

利用四舍五入将浮点数转换为整型数（先前的MBASIC文本采用的是截尾方式求值）。不仅赋值语句是这样（例如：I%=2.5，结果是I%=3），而且函数和语句中的运算也是一样（例如：TAB（4.5）使光标跳到第五个位置，A(1.5)得出A(2)，X=11.5 MOD 4得到0）。

3　FOR/NEXT循环的计算

如果循环的初始值超过终止值（或如果STEP为负值，而初始值小于终止值），那么程序将跳过FOR/NEXT循环。参见第四章"程序语句"中关于FOR/NEXT循环的详细说明。

4. 被零除和溢出

被零除或者溢出时，不再产生致命错误。参见第二章"表达式"中的详细说明。

5. RND函数

RND函数有所改变。RND函数不带自变量与带正值自变量产生的结果相同。每次执行它时，RND函数产生同一序列随机数。要产生不同序列的随机数，必须用RANDOMIZE语句。参见第七章"函数"中的详细说明。

6　打印数值

单精度数和双精度数的打印规则有所改变。参见第四章"程序语句"中关于PRINT语句的详细说明。

7　字串空间的分配

字串空间是动态分配的，所以CLEAR语句不再专为字串设置内存空间。CLEAR语句中的前一个自变量用于设置字串空间的最高地址，后一个自变量用于设置栈空间。

8　无效输入

响应INPUT语句时，用了太多或太少的项，或用了错误类型的数据（如：数值弄成字串等），或仅用回车作输入，系统将输出"？Redo from start"（重新输入），变量并没有被赋值，直到输入一个可以接收的数据为止。

9　PRINT USING标识符

有两种新添的格式标识符用于PRINT USING。&符号用于表示可变长度的字符串，而格式串中的底划号表示后面的一个字符照样打印。

10. WIDTH语句

如果WIDTH语句指明的行宽是255，则BASIC-80使用"非限制"的行宽，也就是说，它不在语句中插入回车。WIDTH LPRINT语句用来设置打印机行宽。

---

* 与MBASIC相比而言。——译者注

11. EDIT控制字符

符号@和底划号不再作为编辑控制字符。

12. 变量名

变量名可多达40个字符，并能含有嵌入的保留字。然而，保留字必须用空格来定界。为继续保持与早期版本的BASIC相兼容，空格将自动地插入在保留字和邻接的变量名之间。如果插入空格使得该行的长度超过255个字节，则这行尾部的字符将被截去。

13. 保护的二进制格式

BASIC-80程序可以用带保护的二进制码形式进行存储，这种形式不能被显示或编辑。

附录D 程序设计的提示

随着程序设计经验的不断丰富，人们一定会开始关心程序的质量问题。如何节省计算机的两种主要资源即内存空间和执行时间是我们关心的主要问题。这节附录将有助于你在这方面所作的努力。

一 节省内存空间

若想节省内存空间，应按照下面的方法去做：

1. 多条语句放在一个程序行中

BASIC-80除跟踪每条程序行之外，还跟踪每个行号。如果把多条语句放在一行中，则内存在行号上的开销就较小。

2. 删去所有不需要的REM语句

当使用REM语句时，BASIC-80将用一个字节来存储关键字REM，其他注释内容以ASCII码进行存储。这样，大量的内存空间就会花在简单的注释上(当删除REM语句时，必须在程序可读性和内存空间的节省之间权衡)。

3 只有在不能使用GOTO语句时，才用子程序调用（GOSUB）

只有当从主程序里的几个不同地方需要调用相同的过程时，才必须使用GOSUB语句。如每次从同一个地方调用某个过程，则可以用GOTO语句。每次使用GOSUB语句将消耗内存空间（堆栈操作），而GOTO语句则不然。

4. 在表达式中应尽可能地少用括号

认真斟酌算术表达式以尽量少用括号。BASIC-80计算一个括号中的表达式，就要多消耗一些内存空间。BASIC-80还要设置临时存储单元来存放运算的中间结果，因而占用了更多的空间。

5. 在任何情况下都应尽量用整数型变量

这一点非常重要。因为一个整数型变量仅仅占用内存的两个字节，而一个单精度变量将占用4个字节，一个双精度变量将占用8个字节。

6. 经济地分配数组空间

注意仅分配给数组必要的内存空间。例如，如果你使用了11个元素的补缺形式的数组，但仅仅使用了其中4个元素，这样，就浪费了许多空间。所以，应该总是用定维语句来设置数组的大小，而不要采用补缺定维的方法（除非数组元素恰好是11个元素）。

7. 将大的程序分成较小的模块

BASIC-80允许使用CHAIN语句在程序之间进行连接和变量传递，可以很容易地将一个大的程序分开写成几个较小的模块，并在它们间传递变量。

8. 用DEF语句宣称变量类型

这将使你省去键入类型标志符的麻烦，并且每个变量都节省了一个字节的存储空间（单精度类型除外）。

9. 减少同时打开的数据文件的数目

每个数据文件需要一个缓冲区。让几个不同的文件使用同一个缓冲区，这个方法是非常有效的。你可以用文件号＃1打开第一个文件，根据需要进行存取，然后关闭这个文件，仍然利用文件号＃1打开第二个文件。虽然你不能同时对两个文件进行存取，但是仍然可以达到根据需要来存取两个文件的目的。

10. 在程序中减少变量和数组的数目

在程序中，当变量和数组已不再需要时，可以重新使用它们。当某一个变量选作FOR/NEXT循环的计数变量时，每一个独立的FOR/NEXT 循环都可以使用该变量计数。

二、节省运行时间

为了节省程序运行时间，应该按照下面的方法去做：

1. 先定义最常用的变量

在BASIC-80中，变量是按运行时遇到的次序先后放入各表的。当一个变量被引用时，要在各表中进行顺序检索。所以，如果一个变量靠近这个表的顶部，则存取时将花费较少时间。

2. 在FOR/NEXT循环中用整型循环变量

这点是非常重要的，它能显著节省时间。如果希望做一个实验，可以在FOR/NEXT循环中设置一个单精度循环计数变量，观察它的执行时间。然后，再换成整型循环计数变量，同样再观察它的执行时间(注意循环次数至少要10,000次)，你会注意到执行时间的显著差别。

3. 在算术表达式中用变量代替常量

BASIC-80中的数值是用十进制浮点数表示的。存取一个变量要比将一个常数转换成为浮点数花费的时间少。如果在一个程序中经常使用某个常数，那么必须把该常数赋给某个变量，用变量来代替。

提高程序质量的方法远不止这些。但是，如果能坚持这样做下去，那么，怎样写出好的程序，就有了门路。

附录E　汇编语言子程序

BASIC-80提供了两种方法从BASIC-80程序中调用汇编子程序。第一种方法是用USR函数，它的使用方式和BASIC-80的内部函数一样。第二种方法用CALL语句，其调用过程同Microsoft的FORTRAN、COBOL编译BASIC中的调用过程相同。

因为汇编语言子程序绕过了BASIC-80中的一些内部防护措施。因此，使得BASIC-80调用汇编语言子程序时，对错误的免疫功能减弱。所以写子程序时必须细心。

一、内存分配

当使用BASIC-80汇编语言子程序时，一个值得注意的问题是存储空间的分配。在汇编语言子程序装载以前，必须为它设置好内存空间。

在初始化时，应输入最高内存地址减去汇编语言子程序所需要的内存容量的差值。可用/M开关在初始化时设置内存的顶限(参见第一章“系统介绍和基础知识”中有关初始化过程的详细说明)。BASIC-80从有效内存空间的起始地址向上使用内存，因而，只有内存顶端的空间才能供用户子程序使用。

在调用汇编语言子程序以后，栈指针设置为八级（16个字节）。如果需要更多的栈空间，BASIC-80的当前栈被存储起来，并为汇编语言子程序设置新栈。然而，在从子程序返回以前，原先的栈内容必须恢复。

汇编语言子程序可以通过CP/M系统监控程序或用BASIC-80的POKE语句装入内存。汇编语言子程序也可以用MACRO-80汇编程序进行汇编，然后使用LINK-80装配程序进行装配（BASIC-80没有提供这些程序，它们必须另行配置）。

二，USR函数的调用

在使用USR函数以前，必须先在DEF USR语句中定义好USR子程序的入口地址。

DEF USR

（定义USR子程序入口地址）

形式：DEF USR〈数字〉＝〈表达式〉

DEF USR语句用来定义汇编语言的子程序的入口，最多不超过10个。

〈数字〉表示汇编语言子程序的号数。它可以是0～9之间的任何数字。如果〈数字〉被省略，它的补缺值为0。

〈表达式〉的值是汇编语言子程序的起始地址，这个地址约定为十进制的，除非有基数标志符加以说明——当它冠以&H时，表示十六进制，冠以&O或&时为八进制。

USR函数的使用格式是：

USR〔〈数字〉〕（自变量）

这里的〈数字〉是0～9，自变量是任何数字的或字串的表达式。〈数字〉指明被调用的是哪个用户子程序，它必须与在DEF USR语句中所定义的数字相符，如果〈数字〉被省略，则约定为USR0。在DEF USR语句中定义的地址就是子程序的起始地址。

当调用USR函数时，寄存器A中的值代表自变量的数据类型，它可能是：

A寄存器的值	自变量的类型
2	2个字节的整数（2的补码）
3	字串
4	单精度浮点数
8	双精度浮点数

表E-1 指明数据类型的寄存器值

如自变量是数值类型的，〔H，L〕寄存器对将指向存放自变量的浮点累加器（FAC）。FAC在内存中占八个字节，足够存放一个双精度数。

三、数值的存储格式

1. 整型数存储格式

一个整型自变量用两个字节存储，整型数是以2的补码形式进行存储的（在下面的讨论中，浮点累加器将简称为FAC)。一个整型变量将按如下方式存储在FAC中：

FAC-3——包含自变量的低八位（最低有效字节)。

FAC-2——包含自变量的高八位（最高有效字节)。

2. 单精度数的存储方式

一个单精度变量存储为4个字节。第一个字节将表示指数。指数是以加上128(八进制200)的形式存储的。这意味着200（八进制）表示指数为0，201（八进制）表示指数为1，177（八进制）表示指数为－1，等等。一个单精度的数值在FAC中的存储形式如下：

FAC-3——包含尾数部分的低八位。

FAC-2——包含尾数部分的中间八位。

FAC-1——包含尾数部分的最高七位。最高一位表示数值的符号（0表示正，1表示负)。

FAC-0——包含以加上128（八进制200）的形式存储的指数。

3. 双精度数的存储格式

双精度变量采用的存储格式与单精度数相同，仅仅是比单精度的数多用了4个字节来存放尾数。除了FAC-7到FAC-4中包含增加的4个字节的尾数外，双精度数在FAC中的存储方法与单精度数相同（FAC-7中含最低的八位，即最低有效位)。

四、字串的存储格式

如果自变量是字串，〔D，E〕寄存器对将指向称为“字串描述器”的三个字节。字串描述器的0号字节表示字串的长（0～255)，1、2字节分别表示字串在串存储空间里的起始地址的低八位和高八位。

注意：如果字串变量出现在程序文本中，则字串描述器会指向它在程序的位置。小心不要因此而改变原来的程序。为了避免预想不到的结果，可在程序文本中将字串加上一个空串。

例如：A$＝“BASIC-80“＋”“

这将迫使BASIC-80将该字串拷贝到字串空间中去，从而防止在子程序调用时，程序文本被修改。

数据类型的转换

通常，通过USR函数求得的值数据类型（整数、单精度数或双精度数）与传递给它的自变量的类型是相同的。然而，当调用MAKINT子程序时，将返回在〔H，L〕中的整数作为函数值，从而发生了数据类型的转换。

执行MAKINT时，从子程序返回过程如下：

```
MAKINT EQU  105H      ; CP/M的MAKINT地址
       PUSH H         ; 保存值以便返回
       LHLD MAKINT    ; 得到MAKINT子程序的地址
       XTHL           ; 在栈中存储返回地址，将原先的值返回到〔H，L〕中
       RET            ; 返回
```

同样，不管函数自变量的数据类型如何，执行下列程序后，都将强制变为一个整型值，它存放于〔H，L〕中。

```
FRCINT  EQU   103H     ; CP/M的FRCINT 的地址
        LXI   H        ; 得到子程序的连续地址
        PUSH  H        ; 放入栈中
        LHLD  FRCINT;  得到FRCINT的地址
        PCHL
```

五、调用语句

BASIC-80用户函数调用也可以通过CALL语句进行，这种调用过程与Microsoft的FORTRAN、COBOL和编译BASIC中的一样。

调用语句的一般格式：

CALL〈变量名〉〔(自变量表)〕

〈变量名〉被赋值为汇编语言子程序在内存中的起始地址。在执行CALL语句之前，这个地址就已被赋给〈变量名〉。〈变量名〉不可以是一个数组名。〈自变量表〉包含了传递给汇编语言子程序的自变量。

不带自变量的CALL语句产生一条简单的“CALL”指令。对应的子程序通过一个简单的“RET”返回（CALL和RET是8080指令码——详见8080手册）。

带自变量的子程序调用过程，可能较为复杂。对应于CALL语句自变量表中的每一个自变量名，都有一个参数传递给该子程序，真正传递的其实是自变量的低字节地址。所以，不管其类型如何，参数总是占用两个字节。

传递的方法取决于被传递参数的数目：

A. 如果参数的数目小于或等于3，则用寄存器传送，参数1将放入HL，参数2放入DE（如果有的话），参数3放入BC（如果有的话）。

B. 如果参数的数目大于3，它们将象下面这样传递：

1) 参数1放入HL。

2) 参数2放入DE。

3) 参数3到n放入内存中一个连续的局部数据块。BC将指向这个数据块的低位字节（即指向参数3的低位字节）。

注意：用这种方法，子程序为了寻找参数，必须要知道它们的个数。反过来，调用程序应该正确地传递相应个数的参数，因为对参数的数目和类型是不加检查的。

如果子程序需要三个以上的参数，并且要传输它们到一个局部数据块，则将有一个系统子程序$AT（在FORTRAN库FORLIB.REL中）完成该传输过程。

调用$AT时，HL指向该局部数据块，BC指向第三个参数，A包括了要传输的自变量的数目(即：自变量的总数－2)。在调用$AT之前，你须负责在子程序中存储第一、二个参数。

例如：如果一个子程序需要五个参数，应该使用下面的方法：

```
SUBR:  SHLD  P1   ; 存储参数1
       XCHG
       SHLD  P2   ; 存储参数2
       MVI   A,3  ; 剩下的参数数目放入A
       LXI        ; 指向局部数据块
```

```
        CALL    $AT ; 传输其他3个参数
          ⋮
        〔子程序体〕
          ⋮
        RET         ; 返回调用程序
P1:     DS  2       ; 参数1的存储空间
P2:     DS  2       ; 参数2的存储空间
P3:     DS  6       ; 参数3～5的存储空间
```

当在子程序中存取参数时，仅仅是在传输指向实际参数的指针。

程序员必须十分小心，以保证在调用程序中的自变量的个数、类型以及长度必须与子程序中需要的参数完全一致。

自变量传输子程序$AT开列如下：

```
00100                           ; 自变量传送程序
00200                           ; 〔B，C〕指向第三个参数
00300                           ; 〔H，L〕指向第三个参数的局部存储区
00400                           ; 〔A〕包含了参数的个数（总数－2）
00700       ENTRY   $AT
00800 $AT:  XCHG                ; 将〔H，L〕存于〔D，E〕中
00900       MOV  H，B
01000       MOV L，C            ;  〔H，L〕=参数指针
01100 AT1:  MOV C，M
01200       INX    H
01300       MOV B，M
01400       INX    H            ; 〔B，C〕=参数地址
01500       XCHG                ; 〔H，L〕指向局部存储区
01600       MOV M，C
01700       INC    H
01800       MOV M，B
01900       INX    H            ; 将参数存到局部存储区中
02000       XCHG                ; 因为要返回AT1
02100       DCR    A            ; 所有参数传输完毕？
02200       JNZ    AT1          ; 没有，转到AT1继续传输
02300       RET                 ; 已完，返回
```

六、中断

汇编语言子程序能对中断进行处理。所有的中断处理子程序都必须保护堆栈、寄存器A～L和PSW的内容。在子程序返回之前，中断应该重新赋能，因为中断一旦被响应，它便自动地屏蔽所有后面的中断。适当选择中断向量也是非常重要的，在CP/M BASIC-80中，所有中断向量都向用户开放。

附录F　随机I/O程序和顺序I/O程序举例

随机文件的一个最直观的实例就是计算机化的电话号码本。下面以两个简单程序说明这一技术。第一个程序“DIRECTORY”是接受所需的数据以建立随机文件和顺序目录文件。第二个程序“QUERY”检索目录文件的数据。

你要想完全了解这个随机文件的I/O方法，那么请首先查看目录文件包含了什么样的信息。目录文件有一个关键字，由姓和名组成。另外目录还有记录号，记录号用来作为索引，来指向随机文件中的各个入口。

当你运行“QUERY”程序时，必须提供人的姓名。如果名字是有效的（即它在目录里有一个项），那么将有一个记录号。它指出记录在随机文件中的实在位置，这样就能检索此人的电话号码。

这些程序本身效率并不是很高的，仅用来说明怎样使用随机文件和顺序文件。

这个例子没有说明一旦文件建立后，怎样在文件中增加数据。这样做是为了让例子简单。如果你想在文件中加入更多的名字，则必须改变程序或者重写建立文件的程序。

如上所述，这个建立的文件的程序假定文件原先并不存在而是第一次建立。如果这个程序要改成阅读原有程序的形式，那么就必须在程序中加入新的语句（在查找程序的50-80行就是阅读原文件的）。

如果你想这样做，应首先打开文件A:TABLE.EXT，并定义为输入，把文件全部读入数组，如NP$和SP。接着关闭文件，然后再重新打开这个文件并定义为输出，最后向这个文件写数据。

另外，这个例子效率不是很高。一个好的程序应该把目录放在A:RFILE.EXT的前面几个记录或后面几个记录中。另外目录应按字母排好序以便快速检索。

如果你把程序打入机子并运行它，那么就能更好地理解这些程序。

程序一：DIRECTORY

```
5 REM "DIRECTORY PROGRAM"
10 OPEN "O", 1, "A:TABLE. EXT"
20 OPEN "R", 2, "A: RFILE. EXT"
30 FIELD #2, 12 AS LN$, 9 AS SN$, 12 AS SR$, 12 AS CI$,
    10 AS SZ$, 2 AS CD$, 2 AS EX$, 2 AS PN$
40 REC=REC+1
50 LINE INPUT "LAST NAME? "; N1$
60 LINE INPUT "FIRST NAME? "; N2$
70 LINE INPUT "STREET ADDRESS? "; N3$
80 LINE INPUT "CITY? "; N4$
90 LINE INPUT "STATE ZIP? "; N5$
100 INPUT "PHONE NUMBER (XXX, XXX, XXXX) "; N%,
      N1%, N2%
110 LSET LN$=N1$:LSET SN$=N2$:LSET SR$=N3$:LSET
```

```
    CI$=N4$:LSET SZ$=N5$
120 LSET CD$=MKI$(N%) :LSET EX$ MKI$(N1%) : LSET
    PN$=MKI$(N2%)
130 KEY$=N1$+N2$
140 PRINT #1, KEY$:", "; REC
150 PUT #2, REC
160 LINE INPUT "MORE INPUT (Y OR NO)"; MI$: IF MI$=
    "Y" GOTO 40
170 CLOSE
180 END
```

行号	说明
10	打开一个输出目录文件，并取名为 "A:TABLE EXT"。
20	打开一个随机文件，并取名为 "A:RFILE.EXT"。
30	设置随机文件缓冲区空间以供变量输入。 LN$=姓　　SZ$=州名 SN$=名　　CD$=地区号码 SR$=街道地址　　EX$=电话局号 CI$=城市名　　PN$=电话号码的后四位
40	记录号加1
50-100	接收输入数据。
110	字串以左对齐形式装入随机缓冲区。
120	变整型数为字串，然后以左对齐形式装入随机缓冲区。
130	由姓和名组成关键字。
140	输入数据到目录文件。 KEY$=目录的关键字。 REC=随机文件记录号。
150	将随机缓冲区中的记录存入文件。
160	询问是否有更多的数据。
170	关闭所有文件。
180	程序结束，并回到BASIC命令状态。

程序二：OUERY

```
 5 REM "QUERY PROGRAM"
10 CLEAR 200
20 OPEN "I", 1, "A:TABLE. EXT"
30 OPEN "R", 2, "A:RFILE. EXT"
40 FIELD #2, 12 AS LN$, 9 AS SN$, 12 AS SR$, 12 AS CI$,
   10 AS SZ$, 2 AS CD$, 2 AS EX$, 2 AS PN$
50 IF EOF(1) THEN GOTO 90
60 CT=CT+1
```

```
70 INPUT #1, NP$(CT), SP(CT)
80 GOTO 50
90 INPUT "NAME (LAST, FIST)"; L$, F$
100 KEY$=L$+F$
110 FOR I%=1 TO CT
120 IF KEY$=NP$(I%) THEN GOTO 150
130 NEXT I%
140 PRINT "NO RECORD EXIST": GOTO 170
150 GET #2, SP(I%)
160 PRINT LN$, SN$, CVI(CD$); "-"; CVI(EX$); "-"; CVI
    (PN$)
170 INPUT "MORE QUERIES? (Y OR N)"; M$: IF M$="Y"
    GOTO 90
180 CLOSE
190 END
```

行号	说明
10	设置字串存储空间。
20	打开目录文件，并定义为输入方式。
30	打开随机文件。
40	设置随机文件缓冲区空间。
50	检测文件结束标志。
60	目录记录号加1。
70	从输入文件读入数据赋给字串变量。
80	返回检测EOF。
90	输入你需要找电话号码的人的姓名。
100	由姓和名构成关键字。
110	设置循环以寻找记录。
120	输入关键字与目录关键字比较。
130	如果初次比较不匹配，进行下一次。
140	如果比完所有的关键字仍然没找到，则打印信息。
150	如果找到，则将相应的记录装入随机缓冲区。
160	把需要的记录转换成整型数后打印出来。
170	检测更多的查询要求。
180	关闭所有的文件。
190	结束程序，并回到BASIC命令状态。

索　引*

A

B

---

*本索引中，英、中文条目均按首字母编排（中文条目用其汉语拼音），每个字母所属条目的编排顺序是：1.英文条目，①命令，②语句，③函数，④特殊名词，⑤错误信息；2.中文条目，按内容归类。——译者注

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

参加本手册编译工作的有：贺健华（第一、二章），林锡来（第三、四章），张世和（第五、六、七章），魏开惠（第七、八、九章），朱逸芬（第十章），曹彩萍、石洪梅（第十一章 附录），汪明霓（索引）。

全文由汪明霓校对。

muLISP/muSTAR-80

人工智能开发系统参考手册

(CP/M 版本)

引　言

LISP 计算机语言基于1960年约翰·麦卡锡 (John McCarthy) 题为“符号表达式的递归函数及其机器计算”的文章。其第一次实现是在60年代初期。在美国麻省理工学院，它一直是人工智能界的“机器语言”。该语言及其许多派生语言继续统治着诸如机器人、自然语言翻译、定理验证、医学诊断、博奕游戏及程序检验等许多领域。LISP 之所以成为一种机器智能语言，有下列原因：

1. LISP 是一种应用的递归语言，这使它成为描述复杂数学概念的理想的形式方法。

2. LISP 的基本数据结构是二叉树，这种抽象对象可以做到与大多数 AI (人工智能) 问题中的实际数据同构(即成为一一对应的模型)。这样，原始问题的特征就可通过对 LISP 数据结构的变换来进行研究。

3. 当计算机编程用于模拟智能时，它必须能对任意难点询问作出回答。一般程序设计语言的静态存储分配模式使它们很难处理这种无终止的问题。在 LISP 中，通过数据存储资源的动态分配及再循环(自动进行无用单元收集)，使这些困难得到缓解。

4. 对智能人-机通信，需要高度的交互环境。由于 LISP 函数定义可很容易地当作数据来处理，这就加快了系统的开发，并使 LISP 成为一种理想的交互式语言。

由 SOFT WAREHOUSE 开发和提供的 muLISP-79 使 LISP 可用于迅速增长的微型机用户，它是 LISP 语言用于微型计算机的第一个有效的程序产品，已经在各种应用中广泛使用。

为适应大部分 AI 软件系统日益增长的对内存及速度的需求，促进了muLISP-80的开发。muLISP-80使用伪码编译和解释程序，使生成的代码密度增加三倍，执行速度提高20%。由于编译和反编译是自动发生的，这个过程对于用户是不可见的。因此，为了提高效率，并没有损害 muLISP 的交互特性 (这对于大多数 AI 应用是非常重要的)。

最后，在 muLISP 中增加了 muSTAR-80AI 开发系统，这是面向显示的常驻编辑程序和调试工具，它使得系统的功能更加完美。

第一章　介绍muLISP-80

muLISP-80人工智能开发系统是用于微型计算机的一个高级软件包，可用于范围很广的各种人工智能（AI）研究。要把muLISP-80用作大型AI软件的开发工具，需要一定的研究水平和耐心。

本章提供关于该系统装入和使用的必要资料，其余各章将详细介绍muLISP数据结构、内存管理和原始定义函数。各章均力求尽量清晰、准确。学习muLISP的最好方法是在学习本手册的同时探讨该系统的使用。但本书不是LISP程序设计语言的教程，如果在使用过程中发现对LISP尚无足够了解，可参阅本手册最后所列有关该语言的参考书目。

第一节　主要特性

muLISP-80的主要特性如下：

1. 有80多个函数以机器语言定义，以求获得最高效率。这些函数提供了一组数据结构基元，包括整套选择函数、构造函数、修改函数、识别函数和比较函数。

2. 无穷精度整数算法，可以用2～36进制的任何一种数制表示。有一组完整的数值原语支持。

3. 两遍扫描的密集式的无用单元收集程序，对所有数据空间实行自动的和动态的存储管理。无用单元收集一般只需不到一秒钟便可完成。

4. 可自动执行数据空间边界的动态再分配，以最有效地使用全部可用的内存资源。

5. 程序控制结构包括扩展的COND，多出口LOOP，以及强有力的函数体求值机制。这些特性允许程序以精巧的纯LISP方式书写，同时又仍然具有循环式程序的效率。

6. LAMBDA定义的函数可做成按值调用（CBV）或按名调用（CBN）。此外，函数可定义为伸展函数或非伸展函数。

7. 除muLISP交互环境外，程序调试可用面向显示的常驻编辑程序和跟踪程序包，使调试简便化。

8. muLISP与宿主计算机磁盘操作系统完全结合在一起，字符输入/输出可以从控制台转换到ASCII文本文件中，且muLISP内存映象文件可以装入和保存。

9. muLISP所占内存相对来说比较小，计算机内存剩余部分可用于用户数据结构。最小系统可在小至20KB内存的计算机中运行。

10. 通过使用变量浅约束、按地址分类的数据结构和闭指针域等技术，使得程序的执行速度非常快。

11. 函数定义自动地被编译为净化码或D代码，当恢复函数的定义时，则自动执行相反的反编译过程。这种编译使产生的代码在存储器中的密度比链接表增加三倍，执行速度约提高20%。

12. 有若干输入/输出控制变量，用于处理诸如大小写转换、控制台行编辑方式以及引号内字符串的打印。

13. 提供了方便地连接到用户定义机器语言子程序的手段。

第二节 系统内容

muLISP 系统 CP/M 版本，包含下列各项：

1. muLISP/muSTAR-80参考手册。
2. muLISP-80主系统磁盘，包含：

① muLISP-80伪码编译程序和解释程序，作为可执行的命令文件MULISP.COM提供。

② muSTAR-80 AI 开发系统和常驻 LISP 编辑程序，作为系统文件 MUSTAR.SYS 提供。

③ 库文件UTILITY.LIB，包含各种有用的muLISP实用函数。

④ 函数跟踪调试程序包，作为库文件TRACE.LIB提供。

⑤ Master Mind游戏程序，作为库文件METAMIND.LIB提供。

⑥ 动物游戏程序，作为库文件ANIMAL.LIB提供。

⑦ 由麻省理工学院约瑟夫·魏森保 (Joseph Weizenbaum) 写的原始ELIZA或博士程序，作为库文件 DOCTOR.LIB、SCRIPT.LIB和 CONTALK.LIB 提供。

⑧ 梵塔问题 (Tower of Hanoi) 解答作为库文件HANOI.LIB提供。

⑨ 应用程序继续例程 CONTINUE.COM。

第三节 主磁盘副本

在得到 muLISP 系统盘后，尽快制备一份拷贝作为工作盘，以防丢失、损坏或不小心擦除主磁盘。拷贝后，主磁盘放在安全、阴凉处，不要和工作盘放在一起。在拷贝盘上最好写上版本号和日期，以及muLISP系统的顺序号。

如果使用的 CP/M 系统有磁盘拷贝实用程序，可用它复制 muLISP 主磁盘。对多驱动器的 CP/M 系统还有一种办法制作拷贝（但较慢），可执行下列步骤：

1. 开计算机电源，引导CP/M系统
2. 将经过格式化的空盘插入驱动器B：
3. 打入CP/M命令PIP，并按〈RETURN〉
4. 在出现“*”提示符后，将muLISP主磁盘插入驱动器A：
5. 打入命令B：=A：*.* 并按〈RETURN〉
6. 系统在拷贝每一个文件至B：盘时，显示文件名
7. 拷贝结束后，放上版权说明

第四节 基本交互式作业周期

在制备主磁盘副本后，开始启动执行 muLISP。首先，按正常方式启动计算机磁盘操作系统，然后打入下述操作系统命令，并按回车键：

A>MULISP

在经过几秒钟装入时间后，系统回答下列信息：

muLISP-80 2.xx (mm/dd/yy)

Copyright (C) 1981 MICROSOFT

Licensed from The SOFT WAREHOUSE

其中，2.xx为相应版本号，mm/dd/yy为日期。

muLISP以美元符作为提示符，指出准备从控制台接收字符输入。在用户从控制台打入完整的表达式并按回车键后，muLISP对该表达式求值，从新行开始打印结果值。该交互式作业周期可以无限制重复，直至执行SYSTEM函数，或打入〈CTRL-C〉。这时，结束muLISP，控制返回主操作系统。

muLISP使用主操作系统的行输入例行程序，因此具有系统的行编辑功能。可用〈BACKSPACE〉、〈DELETE〉、〈RUB〉或〈CTRL-H〉键删除字符。通常，整行删除可用〈CTRL-X〉。一般来说，只能对当前行进行编辑。一旦打入回车后，便没有办法再修改输入行。但运行中的程序可以被中断。

〈CTRL-P〉可用作为触发开关，使所有输出到控制台的字符同时也送到打印机。用这个方法可以产生muLISP运行过程的硬拷贝输出。在程序控制下，可用LPRINTER控制变量，将muLISP输出送到打印机（见第五章第十四节）。

可以用〈CTRL-S〉作触发开关，“冻结”muLISP到控制台输出，然后重新启动。这样可以在显示文本看完之前，不让它滚动离开屏幕。

第五节　执行驱动循环

缺省的muLISP驱动循环执行Eval-LISP函数。首先，显示提示符“$”，表明系统等待控制台输入，然后可打入LISP表达式，并跟以回车。也可以打入多行表达式，因为在一个表达式中，所有括号配对之后才被求值。在表达式送入后，由READ函数读入，用EVAL函数求值，其结果由PRINT函数打印出来。下面是说明基本驱动循环的muLISP对话样本：

```
$ DOG
DOG
$ (PLUS 5 -2)
3
$ (EQUAL DOG CAT)
NIL
$ (MEMBER DOG (QUOTE (CAT COW DOG PIG)))
T
```

定义一个特别适于某种应用的执行驱动循环常常是很有用的。这可以通过对DRIVER函数重定义来实现。例如，在库文件UTILITY.LIB中给出一个Eval-quote-LISP驱动函数。如果发生错误或中断，程序控制返回到用户定义的驱动函数，而不是返回到缺省驱动循环。

第六节　muLISP程序设计

下列muLISP对话描述了如何定义和使用LAMBDA定义的函数：

```
$ (PUTD (QUOTE FACTORIAL) (QUOTE (LAMBDA (N)
      (COND
        ((ZEROP N) 1)
        (T (TIMES N (FACTORIAL (DIFFERENCE N 1)))) ) )))
FACTORIAL
$ (FACTORIAL 5)
120
```

这个阶乘函数的定义是按原始LISP（如1962年麦卡锡（McCarthy）的LISP1.5程序员手册中所述）的方式定义的。虽然它未能充分使用muLISP的功能，但它还是完全可以接受的定义。

研究一下这本手册以及muLISP库文件就会发现，muLISP-80包含了大量的对LISP1.5的向上兼容的扩充。这些扩充大部分是有用的、已定义好的结果，这在原始LISP是没有的。这极大地提高了可读性和执行速度，同时减少了用于函数定义的存储要求。下面是一些主要的扩充：

1. 当一个名由muLISP读入或生成时，其值自动地设置为其本身。这种新名的自身用引称为自动引用（auto-quoting），这使得EVAL-LISP执行驱动循环用于输入时减少了使用QUOTE函数的必要性。

2. COND函数已经被广义化，所有跟在谓词后的表达式都被依次求值。由这个扩展COND返回的值是最后那个表达式的值。

3. 函数体或lambda表达式的求值算法包含隐式COND，这就避免在函数定义内部显式使用COND。例如，将下述FACTORIAL的定义与上面给定的进行比较：

```
(PUTD FACTORIAL (QUOTE (LAMBDA (N)
      ((ZEROP N) 1)
      (TIMES N (FACTORIAL (DIFFERENCE N 1))) )))
```

原始LISP在这些PROG程序控制结构之类的辅助成分普遍使用之前很久，曾是一种样本应用和结构化的语言。muLISP有两个特性，使它完全不需要这种非结构控制功能：

① 功能很强的多出口LOOP函数，允许非递归循环的程序设计。这是没有使用完全非结构GO功能而实现的。

② 在函数的形式变元表中，超出调用该函数时使用的实际变元数的变元必须为NIL。这些超出的变元可用作该函数内部的局部变量，因此不需要用PROG建立这种局部变量。

由于这些原因，muLISP-80没有原始定义的PROG、GO或RETURN结构。在muLISP函数体内部的程序控制主要由表达该函数定义的连接表结构来控制，而不用这些显式控制结构。隐式程序控制使函数定义外观不杂乱，意义更明显。

下例是FACTORIAL函数定义，这个例子说明如何避免使用显式控制结构，写出更加结构化的LISP程序：

```
(PUTD FACTORIAL (QUOTE (LAMBDA (N)
```

```
(PROG (M)
      (SETQ M 1)
A     (COND
            ((ZEROP N) (RETURN M)) )
      (SETQ M (TIMES M N))
      (SETQ N (DIFFERENCE N 1))
      (GO A) ) )))
(PUTD FACTORIAL (QUOTE (LAMBDA (N M)
      (SETQ M 1)
      (LOOP
            ((ZEROP N) M)
            (SETQ M (TIMES M N))
            (SETQ N (DIFFERENCE N 1)) ) )))
```

第一个是常规的非递归定义（注意，这只是一个例子，不是可以工作的muLISP函数定义），第二个是等价的muLISP-80定义，也是非递归的。你可以决定，对你来说哪一个定义更加精巧，更加结构化。注意到muLISP定义只需要29个节点，而传统版本需要38个节点，对于在极有限的地址空间运行的LISP系统来说，这个比率是不无意义的。

第七节　中断程序执行

在程序执行期间，用户可以启动软件中断，暂停程序执行。这对于停止一个脱离控制的或不终止的程序，将控制返回操作控制台是必要的。中断可按〈ESC〉键启动，控制台则显示下列任选信息：

INTERRUPT；Continue:〈RETURN〉

Executive:〈ESC〉 Restart:〈DELETE〉 System:〈CTRL-C〉

用户可打入适当的任选字符，选择下列任选项之一：

1. continue任选使程序从中断点继续执行，该任选可按〈RETURN〉键选择。

2. executive任选返回控制至当前的执行驱动循环。所有变量约束、函数定义或特性值均被保存。该任选可按〈ESC〉键选择。对没有〈ESC〉键的终端，可打〈CTRL-[〉（即在按下control键同时，按左方括号键），它与〈ESC〉键等价。

3. restart任选从工作盘重新启动muLISP，所有变量约束、非原始函数以及特性值均被破坏。该任选可按〈DELETE〉、〈RUBOUT〉或〈BACKSPACE〉键选择。打〈CTRL-H〉键等价于按〈BACKSPACE〉键。

4. system任选终结muLISP，控制返回主操作系统，该任选由按〈CTRL-C〉选择。

第八节　读库文件

muLISP库文件是ASCII文本文件，用于永久存放 muLISP 字符数据。这些数据也可以在每次要用到时从控制台打入。库文件可以使用muSTAR（见第六章）生成，也可以使用适合于你的计算机操作系统的文本编辑程序产生。

用RDS函数（即ReaD选择）可将muLISP 字符输入从控制台重新引导至库文件中。muLISP主磁盘上的库文件或使用 muSTAR 生成的库文件可从执行驱动循环中读入，用RDS命令，其格式如下：

(RDS⟨name⟩⟨type⟩⟨drive⟩)

其中，⟨name⟩为库文件名，⟨type⟩为文件类型，⟨drive⟩为驱动器名，是任选变元，其缺省值为当前联机磁盘。

注意，文件名和文件型之间用空格而不用句号分隔开，驱动器名后面不需要加冒号。

假如你想运行动物游戏程序 ANIMAL.LIB，可用下述命令从联机磁盘上读入该库文件：

$ (RDS ANIMAL LIB)

在读入 ANIMAL.LIB 之后，该程序提供一个如何进行这个动物游戏的说明。muLISP主磁盘上的其他库文件也可以用类似方法读入运行中的muLISP系统。

如果用外部编辑程序写自己的库文件，通常希望该文件最后一行为：

(RDS)

使muLISP输入源回到控制台。或者，控制不返回到控制台，而是使用RDS函数将输入源转换到另一文件的方法，自动地“链接”装入库文件。RDS 使用的细节，见第五章第十三节。

第九节　环境 SYS 文件

muLISP函数SAVE用于保存当前的环境，以备后面恢复用。所谓“环境”，由所有当前活动的muLISP数据结构，包括原子值、特性值和函数定义组成。环境作为SYS类型的磁盘文件保存。例如，下列命令将在当前联机磁盘上产生名为WHALE.SYS的SYS 文件：

$ (SAVE (QUOTE WHALE))

该环境可在以后任何时刻用LOAD函数恢复。如果现在WHALE.SYS文件在驱动器B 上，则可用下列muLISP命令将前面保存的该SYS文件装入：

$ (LOAD (QUOTE WHALE) (QUOTE B))

SYS文件也可以和muLISP一起从操作系统级装入。例如，下列操作系统命令在装入muLISP之后，从驱动器B装入SYS文件WHALE.SYS。

A>MULISP B:WHALE

当使用后一种方法装入时，如果指定的 SYS 文件不存在，则在 muLISP 标题信息之后，将出现下列警告信息：

SYS file NOT found

上述两种装入SYS文件的方法恢复的环境与保存时完全相同。但当SYS文件被装入时，各种数据空间根据计算机当前内存大小重新分配。这就是说，当前内存大小不必与SYS文件建立时完全一样。

SYS文件可用在许多方面。例如，在进行交互对话期间，中间结果可作备份保存，以防止计算机发生故障时丢失，然后继续进行交互对话，或者把交互开发的一组函数定义保存下来。

程序开发时，通常是先建立muLISP源文件，然后用RDS命令读入。但是，程序已经完善之后，每次读同一个程序就显得很乏味，如果生成包含该源程序的SYS文件，用户就可以快速方便地装入该应用程序。

第十节　临时退出muLISP

退出muLISP，可用SYSTEM命令，格式如下：

$ (SYSTEM)

SYSTEM命令返回控制至主操作系统，使标准操作系统命令可以随意使用。在这种情况下，STAT命令具有特殊价值，该命令可以指出磁盘上剩余的自由空间数量。

只要muLISP程序的内存空间不被运行中的CP/M暂驻程序(如STAT或PIP)占用，可在发SYSTEM命令之后重新启动muLIPS。如果文件CONTINUE.COM驻留在当前联机磁盘上，则只需简单地发一条CP/M命令：

CONTINUE

因为CONTINUE.COM文件长度为零，这就导致跳转到CP/M应用程序（例如MULISP.COM）的标准起始单元。要执行CONTINUE实用程序，必须具备下列条件：

1. 被重新启动的程序必须设计成从100H单元开始重新进入（对Heath版本的CP/M为5300H)。它必须存放一个标志，可以通过检查这个标志，知道该程序是第一次进入或者仅是重启动。

2. 热启动时对内存高地址端的复盖不得破坏该程序或其数据，在热启动期间，CP/M处理暂驻命令的部分（约2KB)，被读入内存高地址端，正好放在CP/M常驻部分的下面。

注：当发出muLISP SYSTEM命令时，在退出之前，所有数据空间均被移入内存低地址端。因此可以满足上述第2个条件，允许muLISP系统重启动。但是，如用〈CTRL-C〉退出muLISP，且muLISP堆栈尺寸小于被CP/M命令处理程序复盖的内存空间，则重新恢复muLISP环境是不可能的。因此，最好是用SYSTEM命令退出muLISP，而不要用〈CTRL-C〉。

第十一节　警告信息

在muLISP-80中仅在一种情况下不能满意地恢复，以至产生错误陷阱。在四个数据空间中全部内存耗尽时可导致这样的错误陷阱，见第三章第四节。

发生不太严重的问题时，在控制台显示警告信息。在原始函数中发生错误，则返回NIL值。用户程序负责识别错误并采取相应动作。可能的警告信息有下面三种，产生的原因将在后

面相应的章节中详细叙述：

ZERO Divide Error

End-of-File Read

NO Disk Space

第二章　原始数据结构

muLISP有三种不同类型的基本构件，统称为数据对象。一个数据对象可是muLISP名、数或者结点。利用识别函数，可以确定任一数据对象的类型。给定类型的所有数据对象，由固定数目的指针单元组成。一个指针单元可以指向其他数据对象或指向专用结构，如字符串或二进制数向量。因此，该组数据对象可以想象为相互连接的指针网络，称为指针空间。

三种数据类型均至少有car单元和cdr单元。在muLISP中，car和cdr单元仅指向同一指针空间内部的其他对象。因此，仅按照car和cdr单元指针，不会有跑到这个闭指针空间外面去的危险。这种封闭性的好处是 muLISP 求值机理更简单，更有逻辑性。例如，这样循着指针单元移动的函数就不再需要作耗费时间的运行时类型校验。

第一节　名

名的内部表示如下：

值	特　性	函　数	打　印　名

名是由四个指针单元组成的可识别数据对象。名是唯一存放的，也就是说，在该系统中两个名不能具有同一个打印名。四个指针单元的作用分述如下：

1．car或值单元包含一个指针，指向求值函数识别时该名的当前值。当建立一个名时，其值单元被初置为指向该名本身。名的这种自身引用称为自动引用，这就使得在函数定义中常常可以不需要显式使用QUOTE函数。赋值函数改变名的值单元，当该函数被调用时，函数的形式变元被临时重新赋值，在该函数出口时恢复其原来的值。

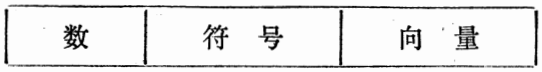
2．cdr 或特性表单元包含一个指针，指向该名的特性表。这个表由特性表函数和标志函数使用和修改。特性表中的标志可以被当作该表的原子元素（即名或数）加以识别。但是，该表的特性是非原子元素（即结点）。特性元素的car指向特性指示符，cdr指向特性值。当建立一个名时，其特性表被设置为NIL，指出没有标志或特性出现。

3．函数单元包含一个指针，指向该名当前的定义。这个函数定义可以是机器码例行程序或LAMBDA表达式的D代码表示。当进行函数调用时，muLISP 求值机制用该函数单元查找该名的定义。对函数单元的访问，只限于函数定义原语，它可用于恢复和修改函数定义。当建立一个名时，其函数单元指向一个未定义的函数陷阱。

4．打印名单元包含一个指针，指向该名的打印名串。这个字串是由用于打印该名的ASCII字符组成的，并以零字节结束。字串可为任意长度。对打印名单元和字符串的访问，只限于I/O 和子原子函数。当系统读入或者产生一个打印名串时，首先要进行校验，看看是否已经存在带有同样打印名的名。如果已存在，则使用现存名；如果不存在，则用新打印名建立一个新名。一经建立，一个名的打印名串就不能修改。

第二节　数

数的内部表示如下：



数是由三个指针单元组成的可识别的数据对象。数不是唯一存放的，因此，在系统中可以出现带有相同向量的数。三个指针单元的作用分述如下：

1．数的car单元包含一个指针，指向该数本身。因此，对数不必加引号，它们求值的结果是本身。自然，该单元内容可以使用赋值函数改变，但是没有理由要这样做，也不推荐这样做。

2．数的cdr或符号单元当该数为正或零时指向NIL，否则指向T。在建立一个数时，建立这个单元值。

3．向量单元包含一个指针，指向该数向量的字节计数字节。数向量是内存中用于存放二进制无符号muLISP数值的一个连续字节序列。向量的字节计数字节存放在内存高地址，且前面的低地址是组成该向量的字节（从最低到最高位字节）。字节计数器的大小限制了数的值为256 ^ 254-1，大于10 ^ 600。

第三节　结　点

结点的内部表示如下：

在muLISP中二叉树是主要的数据对象。内部实现是作为单元对的网络，称为结点。每个结点由car单元和cdr单元组成。如前所述，结点的单元可以只指向其他真实的muLISP数据对象：名、数或结点。

结点常称为点对，服从于点标志法（dot notation），用于在打印中表示。表达式

(X.Y)

代表一个结点，其car单元指向对象X，cdr单元指向对象Y。点标志法完全可以表示任何LISP数据结构。

把数据看作是元素的连接表，而不作为多层嵌套的二叉树结构，常常更为方便。由表 (X1 X2 X3 ··· Xn) 表示的结构可以使用点标志法表示为：

(X1 .(X2 .(X3 .(···.(Xn.NIL)...))))

因此，一个表由通过结点的cdr单元连接的结点链组成。最后一个cdr单元使用名 NIL 以终结该表。NIL还用于表示空表，因此（ ）被READ函数读作NIL，并且被PRIN1和PRINT函数打印作NIL。

函数READ接受由点标志法和表标志法表示的数据结构。但是，当打印二叉树结构时，函数 PRIN1 和PRINT最大限度地使用表标志法。

第三章　内存管理

LISP使用动态的、不可见的内存管理，使程序员不必关心对给定的问题分配足够的内存，大大提高了LISP的能力。对于大多数AI应用来说，由于问题通常不能事先确定其大小，因此，要在程序运行之前进行内存分配即便不是不可能，也是很困难的。

在muLISP-80中，内存管理分三个阶段完成，现分述如下。

第一节　初始数据空间分区

在muLISP初始化阶段，算出该系统可用内存的数量，然后按下表给出的比例，分成四个不同的数据空间：

比　例	空　间	内　容
4:32	原子空间	名和数指针单元
3:32	向量空间	打印名串和数向量
23:32	指针空间	D代码和结点
2:32	栈空间	系统控制／值栈

为建立运行程序所需的数据对象，空间由上述四个空间中取出。新名和数的指针单元所用的空间从原子空间取得。同时，相关的打印名串或数向量所用的空间从向量空间取得。指针空间是最大的空间，用于存储D代码（即编译函数定义产生的净化码）和结点。组合的控制栈和值栈放在栈空间。上面的比例是按照大多数应用程序对空间相对使用的大概情况给出的。

第二节　无用单元收集

堆栈的运行特性使栈空间管理自动、连续进行。但是，其余三个空间的管理要求显式地重复使用不再需要的数据对象。在LISP程序执行期间，新的数据对象不断建立，而另一些数据对象当它们不再被活动的数据结构引用时，则隐含地丢弃。当建立数据对象过程中用完了某空间内的全部可用资源时，则执行无用单元收集，把被丢弃的数据结构腾出的空间重新收集起来，使过程能继续下去。

在muLISP-80中，当原子空间、向量空间或结点空间资源耗尽时，均执行无用空间收集。收集过程的第一遍扫描标识出那些可从系统名的值单元和特性表单元，以及从变量堆栈中存取的数据结构。在第二遍扫描时，将已被标记的或活动的数据对象压缩到各自的数据空间的一端，剩余空间可用于新的数据对象。

尽管无用空间收集是自动进行的，但对于用户也并非是完全不可见的，因为它使程序执行产生周期性的暂停，所需时间的精确数量取决于当前在系统中的 muLISP 数据量，通常收集过程大约不到一秒钟。一般来说用户不大注意，但在使用 muLISP 设计实时系统时应该考虑。

第三节　数据空间边界重分配

如果在无用单元收集后，某数据空间中仍无足够自由存储空间供程序继续运行，则所有空间的分区将重新分配，给已耗尽的空间以更多的内存。这样，muLISP就可以满足不同应用程序对于各种数据空间的不同要求。

第四节　内存不足的陷阱

通常，为了连续不断地满足muLISP程序的要求，可自动调用无用单元收集和数据空间动态重分配，给每个空间提供足够的内存。但是，如果存储数据对象对内存的需求量耗尽所有可用资源，则将发生内存不足的陷阱。由于在muLISP中所有其他问题发生时，都可进行满意的恢复，因此，这是导致错误陷阱的唯一情况。这时，控制台显示如下信息：

ALL Spaces Exhausted

Executive：〈ESC〉 Restart：〈RUB〉，〈DEL〉 System：〈CTRL-C〉

用户可打入相应的任选字符，选择上述的其中一种任选。Executive任选是对于错误陷阱的通常回答，它将控制返回到LISP执行驱动循环，而不改变函数定义、特性值等。Restart任选破坏所有的非原始muLISP函数、变量值和特性值，从起始状态重新启动muLISP。System任选终结muLISP，控制返回CP/M。

第五节　系统失效

当系统不得不用很多的时间去作无用单元收集和数据空间重分配，而收益却非常小时，则发生系统失效现象。系统失效的征兆是对给定任务执行时间大为增加。解决的办法是增大计算机内存容量，或减少程序和数据对内存的需求量。

第四章　muLISP元语言

LISP语言的应用特性使它成为用于精确描述自然语言和计算机语言的一种理想的形式语言或元语言 (meta-Language)。事实上，LISP可用作其本身的元语言。LISP 的这种使用方式可追溯到约翰·麦卡锡对该语言的最初研究及迄今为止继续不断的实践。

当LISP被用作元语言时，它照例以更自然的高级语法书写，称为元LISP。在借用奥连 (Allen) 的“LISP剖析”(1978) 概念的同时，muLISP在句法上作了增强，使之更清楚地反映了基础muLISP扩充的求值能力。

第一节　元语法

下表使用巴科斯范式 (BNF) 定义了muLISP元语言。巴科斯范式首次描述在1963年的ALGOL60报告中，并在本手册附录A中作了概括。

〈定义〉　：：=〈函数型〉〈标识符〉〔〈变量表〉〕:=〈体〉；
〈函数型〉：：=CBV | CBN
〈变量表〉：：=〈变量〉，…，〈变量〉
〈体〉　　：：=〈子句〉，…，〈子句〉
〈子句〉　：：=〈形式〉 | 〈条件〉
〈条件〉　：：=〈形式〉→〈体〉；
〈形式〉　：：=〈常数〉 | 〈变量〉 | 〈应用〉 | 〈赋值〉
〈赋值〉　：：=〈变量〉←〈形式〉
〈应用〉　：：=〈标识符〉〔〈形式〉，…，〈形式〉〕
〈变量〉　：：=〈标识符〉
〈常数〉　：：=NIL | TRUE | LAMBDA | NLAMBDA | 〈数〉
〈标识符〉　是任意的muLISP名
〈数〉　　是任意的muLISP数

本语法将用于第五章描述原始muLISP函数。但是，为了提高可读性，对形式语法规则有时也要作补充。例如，逻辑运算符“AND”和“OR”，以及算术运算符“+”、“*”等，按常规的中缀形式书写

第二节　元语义

1. 元LISP常数NIL代表相应的 muLISP 名，该名表示空表（ ）和真值为假。常数TRUE代表muLISP名T，表示真值为真。

2. 〈函数型〉指明了在调用该函数时所用的变元求值模式。如果函数型为按值调用(CBV)，则在函数调用时，变元在传递给该函数前求值。如果函数型为按名调用 (CBN)，则函数从该调用中接收其变元而不求值。

3. 形式求值根据所涉型式的特殊类型。常数的值就是该常数，变量的值是该变量名的值单元内容。与赋值右边有关的形式首先被求值，然后将该变量值置为该结果。函数应用进行如下：

① 如果函数是CBV，则变元从左至右依次求值。

② 函数的形式变元被实际变元替换，形式变元的求值与否应适当决定，保存旧值。没有对应的实际变元的附加形式变元置为NIL。

③ 然后，应用该函数如下，保存结果。

④ 形式变元值恢复到其原始值。

⑤ 送回应用结果。

4. 函数定义体或条件语句体包含零个或多个子句，用逗号分隔，用分号结束。在函数应用中，它的子句被依次求值如下：

① 如果子句是一形式，则求值方法如上所述。

② 如果子句是条件的，则其谓词形式被求值。如果求值为NIL，则对当前子句表中的下一个子句进行正常求值。否则，条件内的体取代原来的体，作为当前子句表，且继续对后面的子句求值。

上述过程一直继续下去，直至到达子句表结束，在此时最后一个子句的值作为该应用值返回。

第五章　原始定义函数

本章详细描述已作为机器语言子程序实现的全部函数。除了用户可实际访问的函数外，还定义了一些辅助函数，这些辅助函数将单独介绍，以使用户直接访问的函数定义简单、明了。下面的定义中列出了可访问的函数。

在定义每个函数的作用和值时，应尽可能使用第四章所述的muLISP元语言。仅当必须引入不可分解的原语概念时，才牵扯到英语文本。定义后面的解释对函数做什么，一般怎样使用作了非形式的描述。

第一节　选择函数

选择函数用于从给定的二叉树中选择所需的子树。这就给出了从LISP原有的数据结构中提取信息的方法。使用这组函数，可达到给定的树的任意子树或终端结点。函数CAR和CDR分别返回树的分支car和cdr，连续应用这两个函数足以遍历任何树。从下面可以看出，其余函数只不过是CAR和CDR的组合。用机器语言定义主要是为了效率和方便。

1. CBV CAR〔X〕:=

   被X的car单元指向的结构;

解释：一个表达式的CAR的确切解释，取决于该表达式是否是原子：如果不是原子，是否可被当作表或二叉树。如果X是原子，则CAR〔X〕返回X的当前值。如果X是表，则CAR〔X〕返回该表的第一个元素。最后，如果X是二叉树，则CAR〔X〕返回该树的左分支，即car分支。

2. CBV CDR〔X〕:=

   被X的cdr单元指向的结构;

解释：注意到与上面类似的解释可适用于一个表达式的CDR。如果X是原子则，CDR〔X〕返回X的特性表。如果X是表，则CDR〔X〕返回该表的尾部，即除第一个元素外的全部。如果X是二叉树，则CDR〔X〕返回该树的右分支，即cdr分支。

3. CBV CAAR〔X〕:=
   CAR〔CAR〔X〕);

4. CBV CADR〔X〕:=
   CAR〔CDR〔X〕);

解释：这个函数返回第二个表元素。

5. CBV CDAR〔X〕:=
   CDR〔CAR〔X〕);

6. CBV CDDR〔X〕:=
   CDR〔CDR〔X〕);

7. CBV CAAAR〔X〕:=
   CAR〔CAR〔CAR〔X〕)));

8. CBV CAADR〔X〕:=
CAR〔CAR〔CDR〔X〕〕〕;
9. CBV CADAR〔X〕:=
CAR〔CDR〔CAR〔X〕〕〕;
10. CBV CADDR〔X〕:=
CAR〔CDR〔CDR〔X〕〕〕;

解释：这个函数返回第三个表元素。

11. CBV CDAAR〔X〕:=
CDR〔CAR〔CAR〔X〕〕〕;
12. CBV CDADR〔X〕:=
CDR〔CAR〔CDR〔X〕〕〕;
13. CBV CDDAR〔X〕:=
CDR〔CDR〔CAR〔X〕〕〕;
14. CBV CDDDR〔X〕:=
CDR〔CDR〔CDR〔X〕〕〕;

第二节 构造函数

构造函数用于生成解决特定问题所需的数据结构。在 LISP 中，这些结构作为树或连接表实现。它们可被设计来反映或模仿任何实际问题的数据结构。这组的主要成员 CONS 函数建立新结点。该结点所需存储区从内存的指针空间中获得。如果前面的构造过程已把指针空间用尽，则自动执行无用单元收集，以回收不再需要的数据结构使用的空间。

1. CBV CONS〔X, Y〕:=

建立一结点或单元对，其car单元指向X，cdr 单元指向Y。

解释：CONS 的确切解释取决于所建立的数据结构如何表达。如果该数据结构被当作表，则CONS〔X, Y〕返回为表，表的第一个元素为X，表的尾部为 Y。如果该结构是二叉树，则返回为树，树的左单元即car分支为X，树的右单元即 cdr 分支为Y。注意，没有任何办法可改变结构X或 Y。

2. CBN LIST〔X_1, X_2, …, Xn〕:=
如果n=0 → NIL;
CONS〔EVAL〔X_1〕, LIST〔X_2, X_3, …, Xn〕〕;

解释：这个按名调用函数取任意数目的变元，返回求值结果表。

3. CBV REVERSE〔X, Y〕:=
ATOM〔X〕→Y;
REVERSE〔CDR〔X〕, CONS〔CAR〔X〕, Y〕〕;

解释：当给出表X时，这个函数按相反顺序返回X 的元素。通常 REVERSE 调用仅带一个变元。但如果给出第二个变元，则后面附加表Y。

4. CBV OBLIST〔 〕:=

系统中当前活动名表；

解释：对象表，对 muLISP 更确切来说称为名表，是当前系统中所有名的列表。名列表顺序按它们被读入或被生成的顺序，最近的名在前，原来的名在最后。

第三节　修改函数

修改函数实际上更改了 muLISP 数据结构中的指针 因此，使用修改函数主要在于它们的效果，而不在于其返回值。用修改函数可以非常有效地修改现有结构，这样就不必花费代价整个重新组成新结构。但是，由于它们很容易产生不希望的副作用，如循环表，因此，只有有经验的 LISP 程序员才应使用这些函数。修改函数的使用实例，见 muSTAR 编辑程序的源程序清单。

1. CBV RPLACA〔X, Y〕:=
   X的car单元←Y,
   X;

解释：

① 用Y代换表X中第一个元素，

② 用Y代换点对X的左元素，

③ 用Y代换原子X的值。

2. CBV RPLACD〔X, Y〕:=
   X的cdr单元←Y,
   X;

解释：

① 用Y代换表X的尾部，

② 用Y代换点对X的右元素，

③ 用Y代换原子X的特性表。

3. CBV NCONC〔X, Y〕:=
   ATOM〔X〕→Y;,
   ATOM〔CDR〔X〕〕→ RPLACD〔X, Y〕;,
   NCONC〔CDR〔X〕, Y〕,
   X;

解释：连接（不重新构造）表Y至表X的末尾，结果表与用 APPEND 产生的表相同 但是实际上 NCONC 修改第一个表，把该表的最后一个 cdr 单元改为指向第二个表。因此，如果X和Y指向同一个表，则结果导致循环表。如果试图打印这个表，则打印将会无休止地进行下去。

第四节　识别函数

识别函数用于标识数据结构，所有识别函数均取一个变元，且返回值或者为T，或者为NIL，

1. CBV NAME〔X〕:=

```
        如果X是名→TRUE;,
        NIL;
```

解释：这个函数识别名。

```
2. CBV NUMBERP (X) :=
        如果X是整数→TRUE;,
        NIL.
```

解释：这个函数识别数。

```
3. CBV ATOM (X) :=
        NAME (X) OR NUMBERP (X);
```

解释：这个函数识别原子，即非结点。

```
4. CBV NULL (X) :=
        EQ (X,NIL);
```

解释：这个函数识别空表。

```
5. CBV PLUSP (X) :=
        GREATERP (X, 0);
```

解释：这个函数识别正数。

```
6. CBV MINUSP (X) :=
        LESSP (X, 0);
```

解释：这个函数识别负数。

```
7. CBV ZEROP (X) :=
        EQ (X, 0);
```

解释：这个函数识别零。

```
8. CBV EVEN (X) :=
        ZEROP (REMAINDER (X, 2));
```

解释：这个函数识别偶数。

第五节　比较函数

比较函数用于比较数据结构，它们要求两个变元，返回值或者为T，或者为NIL。

```
1. CBV EQ (X, Y):=
        NUMBERP (X) AND NUMBERP (Y) →X=Y;,
    如果X和Y指向同一对象→TRUE;,
    NIL;
```

解释：通常EQ测试用于原子（即名和数）的相等比较。对于数以外的对象，EQ测试检查X和Y是否指向内存同一单元。如第二章所述，在muLISP中，名是唯一存放的，因此EQ能高效率地判断名是否相同。但数不是唯一存放的，故对于数值变元，EQ实际上是比较数向量。

```
2. CBV EQUAL (X, Y) :=
        ATOM (X)→EQ (X,Y);,
```

```
ATOM〔Y〕→NIL;,
EQUAL〔CAR〔X〕, CAR〔Y〕〕→
    EQUAL〔CDR〔X〕, CDR〔Y〕〕;,
NIL;
```

解释：函数 EQUAL 用于两个对象的相等比较，与EQ测试提供的全同比较不一样。如果结构X和Y具有同形的树结构且具有相等的原子终端结点，则认为这两个结构相等。或者更粗略地说，如果打印出来一样，则X和Y相等。

3.
```
CBV MEMBER〔X, Y〕:=
    ATOM〔Y〕→NIL;,
    EQUAL〔X, CAR〔Y〕〕→ TRUE;,
    MEMBER〔X, CDR〔Y〕〕
```

解释：如果表达式X等于 (EQUAL) 表Y的任一成员，则 MEMBER〔X, Y〕返回T，否则返回NIL。

4.
```
CBV GREATERP〔X, Y〕:=
    NUMBERP〔X〕AND NUMBERP〔Y〕→
        X>Y;,
    NIL;
```

解释：数X和Y的简单大于比较。注意，如果任一个变元不是整数，则返回NIL。

5.
```
CBV LESSP〔X, Y〕:=
    NUMBERP〔X〕AND NUMBERP〔Y〕→
        X<Y;,
    NIL;
```

解释：数X和Y的简单小于比较。注意，如果任一个变元不是整数，则返回NIL。

6.
```
CBV ORDERP〔X, Y〕:=
    NUMBERP〔X〕AND NUMBERP〔Y〕→LESSP〔X, Y〕;,
    如果对象X地址小于对象Y地址→TRUE;,
    NIL;
```

解释：这个函数对系统名按照其引入顺序进行一般的排序。因此，如果名X在名Y之前引入（即在对象表中，X出现在Y的右边），则 ORDERP〔X, Y〕返回 TRUE;否则返回NIL。当与非原子变元一起使用时，这个函数的结果没有意义。

第六节　逻辑函数

逻辑函数允许真值的布尔组合，如识别函数和比较函数返回的就是真值。这里和 mu-LISP 其他地方一样，任何非NIL值均被认为是逻辑真。

1.
```
CBV NOT〔X〕:=
    EQ〔X, NIL〕;
```

解释：逻辑“非”（NOT）函数当且仅当其变元为 NIL 时返回T。因此，它与函数NULL完全等价。

2. CBN AND〔X_1, X_2, …, Xn〕:=

如果 n=0→TRUE;,

NOT〔EVAL〔X_1〕〕→NIL;,

AND〔X_2, X_3, …, Xn〕;

解释：逻辑“与”（AND）函数当且仅当其每一变元求值为非 NIL 时返回T。注意，AND是CBN 函数，其变元被顺序求值，直至有一个求值为 NIL，或全部求值为非 NIL 为止。因此，并非总是必须对全部变元求值。

3. CBN OR〔X_1, X_2, …, Xn〕:=

如果n=0→NIL;,

EVAL〔X_1〕→TRUE;,

OR〔X_2, X_3, …, Xn〕;

解释：逻辑“或”（OR）函数当其任一变元求值为非 NIL 时返回 T。变元被依次求值，如果任一变元求得值为非NIL时，则返回T，其余变元不再求值。

第七节 赋值函数

赋值函数通常用于对程序变量赋值。例如，它们允许修改函数的形式变元值，而不必作递归函数调用。因而在某些情况下，可以大大提高函数的执行速度。如果被赋值的变量不是一个局部变量（即不是当前正在执行的函数的形式变元），即使在该函数出口后，赋值仍然有效。如前面修改函数所述，这种现象称为函数的副作用。

SETQ 函数的使用对于传统语言如 BASIC、PASCAL 的程序员来说应该是比较熟悉的，在这些语言中，赋值语句用得很普遍。虽然赋值函数功能很强，但是如果不加选择地使用，将导致非结构化的程序及产生有害的副作用，使纠错非常困难。初学 LISP 的程序员应尽量避免使用这些函数，因为它们使该语言的精巧、适用的特性变得模糊不清。

1. CBV SET〔X, Y〕:=

RPLACA〔X, Y〕,

Y;

解释：将名X的值置为Y，且返回Y。注意，当X不是名时，该函数也被定义，但这种情况下最好不用。

2. CBN SETQ〔X, Y〕:=

SET〔X, EVAL〔Y〕〕;

解释：将X本身的值置为Y值，并返回该值。这个函数比 SET 更常用，因为通常需要对变量赋值，而不是对变量的值赋值。

SET和 SETQ 对它们的变元作用之间是有区别的。如下例所示，如果 DOG 的值是 BARK，则 SETQ〔DOG, '(ABC)〕将 DOG 值从 BARK 变为 (ABC)。可是，如果 DOG 的值是 BARK，则SET〔DOG, '(ABC)〕将 BARK 的值变为 (ABC)，而 DOG 的值仍为 BARK。

3. CBN POP〔X〕:=

POP1〔X, EVAL〔X〕〕;

```
CBV POP1 〔X, Y〕 :=
        SET 〔X, CDR 〔Y〕〕;,
        CAR 〔Y〕;
```

解释：如果X是表名，则 POP〔X〕返回该表的 car 而置X为该表的cdr，这个操作是大家熟悉的在机器语言中广泛使用的 POP 栈操作的 LISP 模拟。

```
4. CBN PUSH 〔X, Y〕 :=
        SET 〔Y, CONS 〔EVAL 〔X〕, EVAL 〔Y〕〕〕;
```

解释：如果Y是表名，且X是表达式，则 PUSH〔X, Y〕将X CONS 置于表Y上，并更新Y，使之指向这个扩大的表，这个操作等价于将信息推至栈上的操作。

第八节　特性函数

特性函数提供了建立全程特性与名关系的好方法。名的特性表用于存放这些特性和指示符标记，与指示符相关的特性值可在以后任何时刻用 GET 函数检索。与下节所述的标志函数一起使用，可以自然方便的形式建立特别灵活有效的数据库。

```
1. CBV ASSOC 〔X, Y〕 :=
        ATOM 〔Y〕→Y;,
        ATOM 〔CAR 〔Y〕〕→ASSOC 〔X, CDR 〔Y〕〕;,
        EQUAL 〔CAAR 〔Y〕, X〕→CAR 〔Y〕;,
        ASSOC 〔X, CDR 〔Y〕〕;
```

解释：这个函数执行关联表Y的线性检索，查找 car 等于X的非原子元素。如果找到，返回整个元素；否则返回NIL。

```
2. CBV GET 〔X, Y〕 :=
        X←ASSOC 〔Y, CDR 〔X〕〕,
        ATOM 〔X〕→NIL;,
        CDR 〔X〕;
```

解释：GET〔X, Y〕返回与名X在指示符Y下相关的特性值，如果该指示符未找到，则返回NIL。

```
3. CBV PUT 〔X, Y, Z〕 :=
        NULL 〔GET 〔X, Y〕〕 →
            RPLACD 〔X, CONS 〔CONS 〔Y, Z〕, CDR 〔X〕〕〕,
            Z;,
        RPLACD 〔ASSOC 〔Y, CDR 〔X〕〕 . Z〕.
            Z.,
```

解释：PUT 〔X, Y, Z〕将特性值Z放在名X的特性表的指示符Y下，原来的特性值不管是什么都被破坏。

```
4. CBV REMPROP 〔X, Y〕 :=
        ATOM 〔CDR 〔X〕〕→CDR 〔X〕;,
        EQUAL 〔CAADR 〔X〕, Y〕→
```

```
            Y←CDADR〔X〕,
            RPLACD〔X, CDDR〔X〕〕, Y;,
        REMPROP〔CDR〔X〕, Y〕;
```

解释：REMPROP〔X, Y〕从名X特性表中将与指示符Y相关的特性值删去，并返回该特性值。

第九节　标志函数

象特性函数一样，标志函数也用名的特性表存放信息。但是，该名仅仅被标志为或者有某特殊属性，或者没有。

```
1. CBV FLAGP〔X, Y〕:=
        MEMBER〔Y, CDR〔X〕〕;
```

解释：当且仅当属性Y是X的特性表元素时，该谓词返回T。注意，如果Y是用PUT函数放入特性表中的指示符，则它仅能被GET函数而不能被FLAGP函数识别。

```
2. CBV FLAG〔X, Y〕:=
        FLAGP〔X, Y〕→Y;
        RPLACD〔X, CONS〔Y, CDR〔X〕〕〕,
        Y;
```

解释：FLAG〔X, Y〕用属性Y标志名X，将Y作为X特性表的第一个元素。

```
3. CBV REMFLAG〔X, Y〕:=
        ATOM〔CDR〔X〕〕→NIL;,
        EQUAL〔Y, CADR〔X〕〕→
            RPLACD〔X, CDDR〔X〕〕,
            Y;,
        REMFLAG〔CDR〔X〕, Y〕;
```

解释：该函数从名X特性表中取消属性Y，如未找到该属性，则返回NIL。

第十节　定义函数

函数定义函数是访问名的函数定义单元的唯一手段。当函数用PUTD定义时，该定义被伪编译为特别密集的形式，称为D代码或净化码。这种编译使代码密度增加到三倍，执行速度约提高20%（与muLISP-79相比）。对含有指向非NIL原子的cdr单元的S表达式，编译时该原子被NIL代换。通常建议带引号的非原子常量不直接包含在函数定义中，而是将常量赋值给函数定义中用于取代该常量的名。当用GETD检索函数定义时，自动进行反编译，即将D代码反编为连接表的相反过程。因此，D代码的使用对用户是不可见的，LISP的交互特性没有因为高效和紧凑而受到损害。

```
1. CBV GETD〔X〕:=
        NOT〔NAME〔X〕〕OR UNDEFINED〔X〕→NIL;,
        SUBR〔X〕OR FSUBR〔X〕→
```

机器语言子程序的内存地址;,
定义X函数的D代码等价的S表达式;

解释：这个函数用于取函数定义以作进一步处理。如果取的是机器语言子程序，则返回其物理存储器地址；否则返回D代码等价的连接表，SUBR和FSUBR定义见本章第十五节。

2. CBV PUTD〔X, Y〕:=
NOT〔NAME〔X〕〕→NIL;,
NUMBERP〔Y〕→
X的函数单元←Y给定的地址;,
X的函数单元←Y的等价D代码.
Y;

解释：如果Y是数，PUTD〔X, Y〕将X的函数单元置为与数Y相等的内存地址（模64K）。否则，定义单元被置为Y的等价D代码。用PUTD连接机器语言子程序的过程在附录C中说明。

3. CBV MOVD〔X, Y〕:=
NOT〔NAME〔X〕〕OR NOT〔NAME〔Y〕〕→NIL;,
Y的函数单元←X的函数单元.
GETD〔Y〕;

解释：置Y的函数单元与X的函数单元指向内存相同地址单元。在使用MOVD就足够的情况下，应使用MOVD而不用GETD和PUTD。

第十一节　子原子函数

子原子函数之所以如此命名，是因为它可访问名的打印名串或数的数向量。这就使得可以临时拆开原子的打印名，对产生的字符表进行运算，最后重新装配该表，形成新名。LENGTH函数除了具有子原子能力外，还确定S表达式的顶层长度。

1. CBV PACK〔X〕:=
ATOM〔X〕→" ";,
NAME〔CAR〔X〕〕→
连接CAR〔X〕打印名与PACK〔CDR〔X〕〕
返回结果名;,
NUMBERP〔CAR〔X〕〕→
连接数值CAR〔X〕的打印名串与PACK〔CDR〔X〕〕
返回结果名;,
PACK〔CDR〔X〕〕;

解释：这个函数在有些LISP方言中称为COMPRESS，它返回一个名，其打印名由表X中各原子的打印名压缩而成。当前RADIX基数用于确定数的打印名串。注意，PACK总是返回一个名，即使它只包含数字。PACK可用于写GETSYM(生成符号)函数.

2. CBV UNPACK〔X〕:=

```
NAME〔X〕→
    名表，其打印名对应于X的打印名中的字符；
NUMBERP〔X〕→
    名表，其打印名是数，对应于以当前数制表示的
    X的数字；，
NIL；
```

解释：这个函数在有些LISP方言中称为EXPLODE，它返回一个名表，这些名的单字符打印名对应于X的打印名中连续的字符。当前基数用于数，且数字被转换为带有单个数值打印名的名字。

3. CBV LENGTH〔X〕：=

```
NAME〔X〕→
    X的打印名中的字符数；，
NUMBERP〔X〕→
    X等价的打印名中的数字数；，
ATOM〔CDR〔X〕〕→1；，
1+LENGTH〔CDR〔X〕〕；
```

解释：这个函数的作用相当于三个函数合在一起。返回值可直观地从其变元的数据类型得到：

① 如果X是名，返回打印X所需的实际字符数。在计算长度时计及PRIN1的当前值。PRIN1的作用在打印函数一节描述。

② 如果X是数，返回打印X所需的实际字符数。在计算长度时，要考虑当前数制的基数，如果前面有负号或前零，也应计及。

③ 如果X是非原子，返回表中顶层结点数。muLISP中，一个cdr是原子，则它用作表的结束符。

4. CBV ASCII〔X〕：=

```
NAME〔X〕→
    返回X的打印名第一个字符的等价整数；，
GREATERP〔X，-1〕AND LESSP〔X，256〕→
    返回单个打印名字符等价于X的名；，
NIL；
```

解释：如果X是名，ASCII（X）返回用于表示X打印名中首字符的ASCII代码整数值。如果X是小于256的非负整数，ASCII（X）返回打印名为ASCII字符X的muLISP名。这个函数对于生成ASCII控制字符特别有用。

第十二节　数值函数

数值函数对于数值不超过2^2032-1，或精度约611位十进制数字的数进行高精度的整数算术运算。如果非数值变元被传递到任何数值函数，或者发生溢出时，函数返回值NIL。对于函数QUOTIENT、REMAINDER或DIVIDE，被零除导致在控制台显示下列错误信

息：

ZERO DIVIDE ERROR

且函数返回值NIL。

```
1. CBV MINUS〔X〕:=
       NUMBERP〔X〕→-X;
       NIL;
2. CBV PLUS〔X, Y〕:=
       NUMBERP〔X〕AND NUMBERP〔Y〕→
           X+Y;
       NIL;
3. CBV DIFFERENCE〔X, Y〕:=
       NUMBERP〔X〕AND NUMBERP〔Y〕→
           X-Y;
       NIL;
4. CBV TIMES〔X, Y〕:=
       NUMBERP〔X〕AND NUMBERP〔Y〕→
           X*Y;
       NIL;
5. CBV QUOTIENT〔X, Y〕:=
       NUMBERP〔X〕AND NUMBERP〔Y〕→
           X/Y (关于零截断);
       NIL;
```

解释：截断的整数商。

```
6  CBV REMAINDER〔X, Y〕:=
       DIFFERENCE〔X, TIMES〔Y, QUOTIENT〕X, Y)));
7. CBV DIVIDE〔X, Y〕:=
       NUMBERP〔X〕AND NUMBERP〔Y〕→
           CONS〔QUOTIENT〔X, Y〕, REMAINDER〔X, Y〕);
       NIL;
```

第十三节　阅读函数和控制变量

阅读函数对muLISP程序提供字符输入。该函数从当前输入源读字符。输入源可以是控制台，也可以是磁盘或其他辅存设备上的正文文件。当前输入源可通过使用函数RDS和控制变量RDS进行控制。

```
1. CBV RDS〔X, Y, Z〕:=          (读选择)
       NULL〔X〕→
           RDS←NULL;,
       NAME〔X〕AND NAME〔Y〕→
```

```
            NULL〔Z〕→
                名为X、Y的文件存在于当前登录的磁盘→
                    打开X、Y作输入,
                    RDS←X;,
                RDS←NIL;,
            NAME〔Z〕→
                名为X、Y的文件存在于磁盘Z→
                    打开Z上的X、Y作输入,
                    RDS←X;,
                RDS←NIL;,
            RDS←NIL;,
        RDS←NIL;
```

解释：读选择函数用于选择输入源文件。如果找到所选择的文件，则将该文件打开用于输入，且变量RDS值置为该文件名。这样，该文件就变成当前输入源。如果RDS调用不带变元，或变元无效，或者文件未找到，则变量RDS值置为NIL，使控制台作为当前输入源。缺省DRIVER函数置RDS为NIL，使控制台在系统初启时，以及发生中断或错误陷阱后，作为当前输入源。

2. CBV RATOM〔 〕:= (读原子)

从当前输入源读一个记号并返回相应的名或数。所谓“记号”，是用分隔符或间断符限定的字符串。见本节末尾注解中关于在记号内部包含注释及引号字串的详细说明。与分隔符不同，间断符由RATOM作为单字符LISP名返回。RATOM分隔符有：空格，回车，换行，制表标记 (CTRL-I)。RATOM间断符有：! $ & ' ( ) * + , - . / @ : ; < = > ? 〔 \ 〕 ^ _ ` { | } ~

```
3. CBV READ〔 〕:=
        READ0〔RATM〔 〕〕;                    (读表达式)
   CBV READ0〔X〕:=
        EQ〔X, '( 〕→
            READLIST〔RATM〔 〕〕;
        EQ〔X, '〔 〕→
            READBRACKET〔READLIST〔RATM〔 〕〕, RATM〔 〕〕;,
        EQ〔X, ') 〕OR EQ〔X,'.〕OR EQ〔X,'〕〕→
            READ〔 〕;,
        X;
   CBV READBRACKET〔X、Y〕:=
        EQ〔Y, ') 〕OR EQ〔Y, '〕〕→X;,
        CONS〔X, READBRACKET〔READLIST〔Y〕, RATM〔 〕〕〕;
   CBV READLIST〔X〕:=
        TERMINATOR〔X〕→NIL;,
        EQ〔X, '.〕→
```

```
        READDOT〔READ〔 〕, RATM〔 〕)),
    CONS〔READ〔 〕X〕, READLIST〔RATM〔 〕〕)),
CBV READDOT〔X, Y〕:=
    TERMINATOR〔Y〕→X,,
    CONS〔X, READLIST〔Y〕〕,,
CBV TERMINATOR〔X〕:=
    EQ〔X, ']〕→TRUE,,
    EQ〔X, '[〕→
        未读"]"字符送回输入源供下一个RATM读,
        TRUE,,
    NIL,
CBV RATM〔 〕:=
```

从当前输入源读一个记号，返回相应的名或数。这个函数与RATOM相同，但RATM分隔符为：空格，逗号，回车，换行和制表标记；RATM间断符有（）·⌣〕。见注解中关于注释及引号字串的详细说明。

解释：READ函数从当前输入源读入一个完整的符号表达式，同时构成等价的LISP数据结构。使用表表示法、点表示法，或两者的组合构成表达式，都是可接受的输入方法。方括号用作超级圆括号。因此，右方括号封闭所有的左圆括号（直至表达式开始），或者配对的左方括号。多余的右圆括号、右方括号和点忽略不计。

4. CBV READCH〔 〕:= （读字符）

从当前输入源读一个字符，并返回对应的LISP原子。如果字符是小于当前基数的十进制数字，则返回一个数。

注：

① 如果当前输入源是磁盘文件，试图读入时超出文件末尾（EOF），则在控制台显示下列警告信息.

End-of-File Read

且控制变量RDS置为NIL，使控制台作为当前输入源。

② 注释可在输入源正文的任何地方出现，只要它们用配对的百分号"%"定界。函数RATOM和READ完全忽略注释文字。但是，READCH读并返回百分号，如同读任何其他字符一样。

③ 特殊字符如百分号、双引号、分隔符和间断符，可以使用引号串的方法，作为名或名的一部分读入，这种串以双引号定界。在串内可包含双引号，这时对每个所需的双引号要使用两个连续的双引号。

④ 为使编程方便，当一个名被READ RATOM或READCH读入时，变量RATOM值被置为该新名。

阅读函数控制变量给用户提供了几种输入任选，扩充这些函数的灵活性。这些控制变量作为标志或触发开关使用，可以开或关。除了ECHO之外，阅读函数控制变量与阅读函数同名。这样做主要是为了节省宝贵的内存空间。每个变量通常与其同名函数一起使用。

1. RDS：通常，当前输入源控制是通过使用RDS函数，如前所述。但是，当文件被

打开，成为现行文件后，将控制变量RDS值置为NIL，可使控制返回控制台而不必关闭输入文件。此后对RDS赋非NIL值时，控制返回原先打开的磁盘文件中，读被挂起的点。

2. READ：变量READ控制对输入字母的小写到大写转换。通常READ值为非NIL，小写字母与其对应的大写字母是有区别的。但是，如果READ值为NIL，则所有小写字母在读入时均转换为对应的大写字母。在任何情况下，已在系统中的所有小写字母仍保持为小写。

3. READCH：当控制台作为当前输入源时，控制台输入方式由变量READCH控制。通常，变量READCH值为非NIL，控制台输入为行编辑方式。在这种方式下，当所有字符均已从当前行读入时，调用操作系统行编辑例程，作进一步输入。在打入回车前，系统的正常编辑过程如输入回送、退格、行删除、用CTRL-P作打印机输出开关等等均有效。如果变量READCH是NIL，则取消缓冲和输入回送。这种原始输入方式用于对单字符命令作立即响应，如muSTAR编辑程序所述。

4. ECHO：如果磁盘文件是当前输入源，且控制变量ECHO值为非NIL，则从输文件读入的字符被回送到当前输出接收器，通常是控制台。注意，因为注释也被回送，注释中的英文可方便地显示，而不必作实际处理。ECHO的其他作用见打印函数控制变量。

第十四节　打印函数和控制变量

muLISP打印函数将字符输出引导至当前输出接收器，接收器可以是控制台或磁盘文件，由WRS函数和变量确定。

1. CBV WRS (X, Y, Z) =　　(写选择)

```
NOT NULL (WRS)→
    写出当前打开的磁盘文件上最后一个记录并关闭该文件。
    WRS←NIL,
    WRS (X, Y, Z);,
NULL (X)→
    WRS← NIL;,
NAME (X) AND NAME (Y) →
    NULL (Z) →
        在当前登录磁盘上如果名为X.Y的文件存在，删除任何现存
        的名为X.BAK的文件，然后将X.Y文件重命名为X.BA-
        K，为X.Y登记新目录项。
        WRS← X;,
    NAME (Z) →
        在磁盘Z上如果名为X.Y的文件存在，删除任何现存的名为
        X.BAK的文件，然后将X.Y文件重命名为X.BAK，为
        X.Y登记新目录项，
        WRS← X;,
```

```
        WRS← NIL;,
    WRS← NIL.
```

解释：写选择函数用于选择及以后关闭输出接收磁盘文件。如果当前输出接收器是磁盘文件，且调用WRS时，则关闭该文件。其次，如果一文件名、文件型和驱动器（任选）作为变元提供给WRS，则具有该名的现存文件被改名为同名的.BAK文件，这就提供了一级文件后援功能。最后，在适当的驱动器上建立一个给定文件名和文件型的新文件，作为输出文件，变量WRS置为该文件名。该文件就变成新的当前输出接收器。缺省的DRIVER函数将WRS置为NIL，使系统初启时，以及发生中断或错误陷阱后，以控制台作为当前输出接收器。

2.
```
CBV PRINT (X) :=
    PRIN1 (X),
    TERPRI ( ),
    X;
```

解释：打印表达式X，终结最后一行，并返回X。

3.
```
CBV PRIN1 (X) :=
    NAME (X) →
        将X表示的打印名输出至当前输出接收器;,
    NUMBERP(X)→
        将以当前数制表示X的数字输出至当前输出接收器，如果 MINUSP
        (X) 则前面加 "—" 号;,
    PRIN1 ("("),
    PRINLIST(X),
    X;
CBV PRINLIST (X) :=
    PRIN1 (CAR (X)),
    NULL (CDR (X)) →PRIN1 (")");,
    PRIN1 (" "),
    ATOM (CDR (X)) →
        PRIN1 (".")
        PRIN1 (CDR (X)),
        PRIN1 (")");,
    PRINLIST (CDR (X));
```

解释：函数PRIN1将对象X的标准表表示输出至当前输出接收器，X是返回值。

4.
```
CBV TERPRI (X) :=
    ZEROP (X) →NIL;,
    输出一回车换行至当前输出接收器,
    PLUSP (X) AND LESSP (X, 256)→
        TERPRI (X-1);,
    NIL;
```

解释：如果X为非负数，TERPRI〔X〕将行数为X的新行输出至当前输出接收器；否则输出一个新行。

5. CBV SPACES〔X〕:=

```
PLUSP〔X〕AND LESSP〔X, 256〕→
    PRIN1〔" "〕,
    SPACES〔X-1〕;,
当前光标位置;
```

解释：如果X为非负数，SPACES〔X〕将X个空格输出至当前输出接收器；否则无空格输出。在任何情况下，返回结果光标位置。

6. CBV LINELENGTH〔X〕:=

```
GREATERP〔X, 11〕AND LESSP〔X, 256〕→
    置最大输出行长度为X,
    返回原先的行长度;,
返回当前的行长度;
```

解释：如果X是11和256之间的数，LINELENGTH〔X〕将每行的字符输出最大数置为X，该函数返回原先的行长度。如果X不是数，或者超出允许取值范围，则返回当前行长度。缺省的行长度是72。

7. CBV RADIX〔X〕:=

```
GREATERP〔X, 1〕AND LESSP〔X, 37〕→
    置数制基数为X.
    返回原先的基数;,
返回当前的基数;
```

解释：如果X是1～37之间的数，RADIX〔X〕设置表达输入输出数的数制基数，则该函数返回原先的基数。如果X不是数，或者超出允许取值范围，则返回当前数制基数。缺省的基数是10。

注：

① 如果当前输出接收器是磁盘文件，且无足够磁盘空间，则控制台显示下列信息：

No Disk Space

并且将控制变量WRS置为NIL，使控制台成为当前输出接收器。

② 在WRS函数建立新的磁盘文件前，进行操作系统复位。这就使得可在muLISP作业期中间安全地变换磁盘，无需退回到操作系统。但在变换磁盘时必须遵照下列注意事项：被取出的磁盘不得包含当前输入源文件或当前输出接收文件。例如，在出现上述磁盘空间错误信息时，可在该驱动器中放入空磁盘，重启磁盘写操作。

③ 如果主操作系统支持打印机，且muLISP处在编辑方式下，打入CTRL-P将使系统控制台上显示的所有muLISP输出同时在打印机上打印出来。

打印函数控制变量作用类似于对应的阅读函数控制变量。它们作为触发开关控制当前输出接收器，大小写转换，打印引号字串以及将输出字符回送到控制台。

WRS：通常，当前输出接收器控制是通过使用WRS函数，如上所述。但是，在使用WRS打开文件之后，可用把WRS值设置为NIL的方法，将输出引导至控制台，而不

关闭磁盘文件。此后，对WRS赋值为非NIL时，将输出重新引导至磁盘文件，并将数据添加到该文件末尾。

2. PRINT：变量PRINT控制输出字母从大写到小写的转换。通常，PRINT为非NIL，所有字符均按存放形式打印。但如果PRINT为NIL，则所有大写字母在打印时均转换为小写字母。这种转换丝毫不影响任何名的字符串的内部存储形式。

3. PRIN1：如果PRIN1为NIL，则包含分隔符或间断符的名必要时将用双引号打印，以便允许该名在以后以相同的名重新读回。用带引号的字串打印这种名主要使用在LISP编辑程序一类的应用中。但通常，PRIN1为非NIL，名的打印名串按原样输出。

4. ECHO：如果磁盘文件是当前输出接收器，且控制变量ECHO值为非NIL，则被输出到该文件的字符也回送到控制台。

5. LPRINTER：如果控制变量LPRINTER为非NIL，所有到当前输出接收器（通常是控制台）的输出，也被送到行打印机。注意，使用系统控制台打入的文本这种输入并不回送到打印机，但是结果输出将送到打印机。因此，通常用于产生muLISP作业期间硬拷贝清单的方法是使用CTRL-P任选。如注③中所述。LPRINTER主要用于函数定义内部，在程序控制下将输出引导至打印机。

6. LINELENGTH：如果LINELENGTH是NIL，则正常的muLISP自动行结束特性无效。由LINELENGTH函数设置的行长度被忽略。这个特性对于不需要自动行结束的应用，如屏幕编辑，是很有用的。

第十五节　求值函数

求值函数用于表达式求值和程序控制。在muLISP中，求值函数体算法功能增强，使程序控制隐含在函数体本身的结构中。如第一章第六节所述。这使函数定义清晰、简短，且易于解释。

muLISP函数定义由表达该定义的连接表描述。表的第一个元素确定函数类型，它应为名LAMBDA或NLAMBDA。LAMBDA指明该函数是按值调用函数（CBV）。当调用CBV函数时，首先对变元求值，且仅将结果值传递给该函数。用NLAMBDA定义的函数是按名调用函数（CBN）。CBN函数以非求值方式接收其变元，变元形式如同调用该函数时给出的一样。在LISP中，按值调用函数使用远比按名调用函数普遍。因此，初学的程序员暂不必关心按名调用函数的使用，可至熟悉了该语言其他特性之后再说。

函数定义的第二个元素应为名或名表，用于定义该函数的形式变元。如果形式变元为非NIL的名，则该函数被认为是非伸展函数，非伸展函数把它的变元作为一个表来接收并把该表约束到那个形式变元。因此，非伸展函数可有任意数目的变元。但如果第二个元素是原子的表，则这些变元将被交给这个伸展函数，并分别约束到组成该表的每个形式变元。注意，一个函数是伸展函数或非伸展函数，与其是CBV或CBN无关。

该表的其余元素组成定义的函数体。函数体是当函数被调用时依次执行的一系列任务（tasks）。函数的返回值是被执行的最后一个任务的值，给定任务如何执行取决于该任务的结构。

① 如果该任务是一个原子，则该任务的值就是该原子的值。

② 如果该任务的car是一个原子，则认为car是应用于组成该任务cdr的变元表的一个函数名。这些变元在应用于CBV函数之前求值。

③ 如果该任务car的car是一个原子，则认为该任务的car是谓词（predicate），其求值方法如②所述。如果该谓词值是NIL，则该任务值亦为NIL。但如果该谓词值为非NIL，则不再看原来的函数体，而用该任务的cdr作为新函数体进行求值。

④ 否则在继续对顶层函数体求值之前，将该任务作为函数体进行递归求值。这就允许函数体中的条件分支在以后重新组合。

这种求值模式功能很强，但对于非递归程序控制结构不提供任何功能。这种迭代能力可很容易地加到上述算法中。含在LOOP控制结构内部的函数体将按上述方法求值，但在最后一个任务执行完后，在函数体开始处重新求值，而不返回，直至一个谓词值为非NIL，如前面第③种情形所规定的。该循环结构的值是跟在非NIL谓词后函数体的值。注意，在一个循环内任何位置可出现任意数目的谓词。LOOP的实现既保留了muLISP的基本求值方法，又极大地提高了该语言的性能。

下列辅函数是识别函数，用于定义求值和函数定义函数：

```
CBV UNDEFINED〔X〕:=                    (未定义函数识别函数)
    NULL〔GETD〔X〕〕;
CBV CBVP〔X〕:=                         (按值调用识别函数)
    SUBR〔X〕OR EXPR〔X〕;
CBV CBNP〔X〕:=                         (按名调用识别函数)
    FSUBR〔X〕OR FEXPR〔X〕;
CBV SUBR〔X〕:=                         (CBV子程序识别函数)
    X是CBV机器语言子程序→TRUE,,
    NIL;
CBV FSUBR〔X〕:=                        (CBN子程序识别函数)
    X是CBN机器语言子程序→TRUE,,
    NIL;
CBV EXPR〔X〕:=          (CBV LAMBDA定义函数识别函数)
    EQ〔CAR〔X〕,'LAMBDA〕;
CBV FEXPR〔X〕:=         (CBN LAMBDA定义函数识别函数)
  EQ〔CAR〔X〕,'NLAMBDA〕;
1. CBN QUOTE〔X〕:=
       X;
```

解释：这个函数抑制其变元求值，返回对象X本身。

```
2. CBV EVAL〔X〕:=
       ATOM〔X〕→CAR〔X〕;,
       NAME〔CAR〔X〕〕→
           UNDEFINED〔CAR〔X〕〕→
               EQ〔CAR〔X〕 EVAL〔CAR〔X〕〕〕→
                   EVLIS〔X〕;,
```

```
                EVAL〔CONS〔EVAL〔CAR〔X〕〕, CDR〔X〕〕〕;,
            CBVP〔GETD〔CAR〔X〕〕〕→
                APPLY〔CAR〔X〕, EVLIS〔CDR〔X〕〕〕;,
            CBNP〔GETD〔CAR〔X〕〕〕→
                APPLY〔CAR〔X〕, CDR〔X〕〕;,
            EVLIS〔X〕;,
        EXPR〔CAR〔X〕〕→
            APPLY〔CAR〔X〕, EVLIS〔CDR〔X〕〕〕;,
        FEXPR〔CAR〔X〕〕→
            APPLY〔CAR〔X〕, CDR〔X〕〕;,
        EVLIS〔X〕;
```

解释：如果X是原子，EVAL〔X〕返回X值单元内容。否则，如果X的car是CBV函数，则X的cdr每个元素被求值，且该函数作用于该结果。如果X的car是CBN函数，则该函数直接作用于X的cdr。最后，如果X的car不是一个函数，则对X每个元素依次求值，将结果表返回。

```
CBV EVLIS〔X〕:=
    ATOM〔X〕→ NIL;,
    CONS〔EVAL〔CAR〔X〕〕, EVLIS〔CDR〔X〕〕〕;
```

解释：这个函数对表中每个元素求值，返回结果表。

3.
```
CBV APPLY〔X, Y〕:=
    NAME〔X〕→
        UNDEFINED〔X〕→
            EQ〔X, EVAL〔X〕〕→NIL;,
            APPLY〔EVAL〔X〕, Y〕;,
        SUBR〔GETD〔X〕〕→
            ATOM〔Y〕→
                X〔NIL, NIL, NIL〕;,
            ATOM〔CDR〔Y〕〕→
                X〔CAR〔Y〕, NIL, NIL〕;,
            ATOM〔CDDR〔Y〕〕→
                X〔CAR〔X〕, CADR〔X〕, NIL〕;,
            X〔CAR〔Y〕, CADR〔Y〕, CADDR〔Y〕〕;,
        FSUBR〔GETD〔X〕〕→
            X〔Y〕.
        EXPR〔GETD〔X〕〕OR FEXPR〔GETD〔X〕〕→
            BIND〔CADR〔GETD〔X〕〕, Y〕,
            Y ← EVALBODY〔NIL, CDDR〔GETD〔X〕〕〕,
            UNBIND〔CADR〔GETD〔X〕〕〕,
            Y;,
```

```
            NIL;,
        EXPR〔X〕OR FEXPR〔X〕→
            BIND〔CADR〔X〕, Y〕,
            Y ← EVALBODY〔NIL, CDDR〔X〕〕,
            UNBIND〔CADR〔X〕〕,
            Y;,
        NIL.
```

解释 APPLY〔X, Y〕将函数X应用于变元表Y。如果X是机器语言例行程序，则控制传送给该例程。如果X是LAMBDA定义函数，则X的形式变元临时约束到这些实际变元，该函数体被求值，恢复形式变元的原始值，返回函数体所求值。

```
CBV EVALBODY〔X, Y〕:=
    ATOM〔Y〕→ X;,
    ATOM〔CAR〔Y〕〕OR ATOM〔CAAR〔Y〕〕→
        EVALBODY〔EVAL〔CAR〔Y〕〕, CDR〔Y〕〕;,
    ATOM〔CAAAR〔Y〕〕→
        X ← EVAL〔CAAR〔Y〕〕,
        NOT〔X〕→
            EVALBODY〔NIL, CDR〔Y〕〕;,
        EVALBODY〔X, CDAR〔Y〕〕;,
    EVALBODY〔EVALBODY〔X, CAR〔Y〕〕, CDR〔Y〕〕;
```

解释：这个函数对函数体Y求值，如本章序言所述。返回被求值的最后一个表达式的值。

```
CBV BIND〔X, Y〕:=
    ATOM〔X〕→
        ATOM〔Y〕→ NIL;,
        PUSH〔EVAL〔CAR〔X〕〕, ARGSTACK〕,
        SET〔CAR〔X〕, NIL〕,
        BIND〔CDR〔X〕, Y〕;,
    ATOM〔X〕→ NIL;,
    PUSH〔EVAL〔CAR〔X〕〕, ARGSTACK〕,
    SET〔CAR〔X〕, CAR〔Y〕〕,
    BIND〔CDR〔X〕, CDR〔Y〕〕;
```

解释：这个函数在名为ARGSTACK的变元堆栈中保存组成形式变元表X的各原子的初始值。同时，这些形式变元值置为表Y中给出的相对应的实际变元元素。

```
CBV UNBIND〔X〕:=
    ATOM〔X〕→ NIL;,
    UNBIND〔CDR〔X〕〕,
    SET〔CAR〔X〕, POP〔ARGSTACK〕〕;
```

解释：这个函数恢复存放于ARGSTACK的形式变元表X中原子初始值。

```
4   CBN COND〔X1, X2, …, Xn〕:=
        EVALCOND〔LIST〔X1, X2, …, Xn〕);
    CBV EVALCOND〔X, Y〕:=
        ATOM〔X〕→ NIL;,
        Y ← EVAL〔CAAR〔X〕),
        NOT〔Y〕→ EVALCOND〔CDR〔X〕);,
        EVALBODY〔Y, CDAR〔X〕);
```

解释：COND函数依次对X1, X2, …, Xn的car求值，直至碰到一个非NIL值，或全部求值为NIL。在前一种情况下，该变元的cdr被作为函数体求值（详见APPLY解释）；在后一种情况下，COND返回NIL。这个扩充的COND函数功能很强，是原始LISP 1.5 COND函数向上兼容的扩充。

```
5   CBN LOOP〔X1, X2, …, Xn〕:=
        EVALLOOP〔LIST〔X1, X2, …, Xn〕,
              LIST〔X1, X2, …, Xn〕);
    CBV EVALLOOP〔X, Y, Z〕:=
        ATOM〔Y〕→
              EVALLOOP〔X, X〕;,
        ATOM〔CAR〔Y〕〕OR ATOM〔CAAR〔Y〕〕→
              EVAL〔CAR〔Y〕〕,
              EVALLOOP〔X, CDR〔Y〕〕;,
        ATOM〔CAAAR〔Y〕〕→
              Z ← EVAL〔CAAR〔Y〕〕,
              NOT〔Z〕→EVALLOOP〔X, CDR〔Y〕〕;,
              EVALBODY〔Z, CDAR〔Y〕〕;,
        EVALBODY〔NIL, CAR〔Y〕〕,
        EVALLOOP〔X, CDR〔Y〕〕;
```

解释：LOOP结构对其变元求值方式与函数体中子句求值方式相同。但如果所有变元均被求值，而任何条件未满足，则重新从第一个变元开始求值。

```
6.  CBN PROG1〔X1, X2, …, Xn〕:=
        EVALPROG1〔EVAL〔X1〕, LIST〔X2, X3, …, Xn〕);
    CBV EVALPROG1〔X, Y〕:=
        ATOM〔Y〕→X;,
        EVAL〔CAR〔Y〕〕,
        EVALPROG1〔X, CDR〔Y〕〕;
```

解释：这个函数依次对X1, X2, …, Xn求值，返回X1的求值结果。

```
7.  CBV DRIVER〔 〕:=                    (Eval muLISP 驱动函数)
        RDS ← NIL,
        WRS ← NIL,
        ECHO ← NIL,
```

```
        READCH ← `READCH,
        TERPRI [ ],
        PRIN1 [ ` "$" ],
        PRINT [EVAL [READ [ ]]];
```

解释：这是缺省的Eval-LISP驱动函数，是以机器语言形式定义的原始函数。muLISP执行驱动循环对函数DRIVER重复求值，因此，DRIVER可随意重新定义以满足用户需要。下面是Eval-quote-LISP驱动函数的样本实例：

```
    CBV DRIVER [ ] :=                (Eval-quote muLISP 驱动函数)
        RDS ← NIL,
        WRS ← NIL,
        ECHO ← NIL,
        READCH ← `READCH,
        TERPRI [ ],
        PRIN1 [` ">" ],
        PRINT [APPLY [READ [ ], READ [ ]]];
```

第十六节　内存管理函数

内存管理函数用于强制进行无用单元收集，并返回当前可用数据空间数量。因为在muLISP中内存管理是完全自动的，所以若不是为了取得它的值，通常没有必要显式使用该函数。见第三章关于内存管理系统的讨论。

1. CBV RECLAIM [ ] :=

    执行无用单元收集

    自由空间数量以字节表示；

解释：这个函数强制执行无用单元收集，返回原子空间、向量空间和指针空间的自由空间总量，以字节表示。注意，对于每个muLISP指针单元，需要2个字节。因此，结点需要4个字节，数需要6个字节，名需要8个字节。此外，名和数还分别需要存放打印名和数向量的存储空间。

2. RECLAIM：是进行无用单元收集时，控制打印统计信息的变量。如果RECLAIM值是NIL，则在无用单元收集开始和结束时，控制台显示如下信息：

```
GC:   nn  aaaa  vvvv  pppp  ssss  tttt
GC:   nn  aaaa  vvvv  pppp  ssss  tttt
```

nn，aaaa，vvvv，pppp，ssss，tttt表示十六进制数。其中，nn是两位数字，给出从前一次数据空间边界重分配以来，无用单元收集次数。aaaa、vvvv、pppp、ssss是四位数字，分别给出在原子、向量、指针和栈空间中剩余的可用空间，以字节为单位。tttt是四位数字，给出可用空间总量（即aaaa、vvvv、pppp和ssss之和）。

显示的第一行表示收集前的可用空间，第二行表示收集后的可用空间。如果有第三行显示，则表示进行了内存重分配，并给出重分配后的可用空间。

这些统计信息是与具体实现有关的，一般muLISP用户不用关心。但是，这些信息对于

确定由于哪个数据空间引起系统失效问题是有用的。

第十七节　环境函数

环境函数用于保存和装入muLISP环境。在保存系统之前，所有数据空间均被自动地压缩到一个内存区中，使结果SYS文件占用内存最小。SYS文件可装入至与产生该SYS文件的系统内存容量不同的计算机系统中。

```
1.  CBV SAVE〔X, Y〕:=
        NOT NULL〔WRS〕→
            写出当前输出接收器中最后一个记录，并关闭该文件，
            WRS ← NIL,
            SAVE〔X, Y〕:,
        NAME〔X〕AND NAME〔Y〕→
            NULL〔Y〕→
                紧致所有当前数据结构.
                将当前环境的内存映象作为名 X 的SYS 文件保存在当前磁盘
                  中.
                TRUE:,
                紧致所有当前数据结构.
                将当前环境的内存映象作为名X的SYS文件保存在Y磁盘中.
            TRUE:
        NIL:
```

解释：这个函数将当前muLISP环境保存在磁盘文件中。因为在保存前，当前活动数据结构已被压缩，所以SYS文件大小与该系统中这些数据结构数成比例。

```
2.  CBV LOAD〔X,Y〕:=
        NAME〔X〕AND NAME〔Y〕→
            NULL〔Y〕→
                从当前磁盘装入名为X的内存映象 SYS 文件，
                RDS←NIL.
                直接返回至执行驱动循环;.
            从Y磁盘装入名为X的内存映象SYS文件，
            RDS←NIL.
            直接返回至执行驱动循环;
        NIL:
```

解释：这个函数恢复SAVE保存的muLISP环境。如果SYS文件已成功装入，则LOAD不返回值，而直接转移到新的执行驱动循环。

```
3.  CBV MEMORY〔X,Y〕:=
        GREATERP〔X,-1〕AND LESSP〔X,65536〕→
            GREATERP〔X,-1〕AND LESSP〔X,256〕→
```

```
                以Y取代内存单元X内容,
                返回X原来内容;,
            返回内存单元X内容;,
        NIL;
```

解释：这个函数允许对计算机的任何内存单元进行测试或改变。如未给出第二个变元，或第二个变元超出单字节数的范围，则只返回该单元值。注意：MEMORY函数应该慎用，因为对muLISP解释程序和主操作系统未加任何保护，可能被破坏。

4. CBV SYSTEM〔 〕:=

```
        将所有数据结构压缩到内存低端,
        返回至操作系统;
```

解释：当执行这个函数时，所有数据被压缩到内存低端，控制返回操作系统。数据压缩允许重新进入muLISP时，系统环境与调用 SYSTEM 函数时的环境完全相同。对于CP/M系统下的muLISP版本，重进入地址是100H（十六进制）。

第十八节　图形和专用函数

对于运行在 ADIOS 操作系统下的 Apple Ⅱ 计算机和运行在 CP/M 操作系统下带有 Microsoft公司的 Soft Card的 Apple 计算机，均有muLISP的专用版本。它们除了具有本手册中所述的全部功能外，还有下列原始定义函数，这些函数充分利用了计算机专用硬件的优点，特别是，以类似于 BASIC 的方式支持低分辨率图形、游戏控制板和扬声器。

1. CBV TEXT〔 〕:=

```
        置屏幕为满屏正文方式 (40列×24行)
        光标移至屏幕最后一行,
        NIL;
```

解释：这个函数用于在以图形方式使用屏幕编辑后，返回到通常的文本方式。如BASIC版本的TEXT命令一样，不清除屏幕。

2. CBV GRAPHICS〔X,Y〕:=

```
        NULL〔X〕→
            0<=Y<16→
                置屏幕为混合方式 (40×40),
                    低分辨率图形,
                以Y颜色号填充屏幕,
                光标移至4行正文窗口,
                NIL;,
            置屏幕为混合方式 (40×40), 低分辨率图形,
            清屏幕,
            光标移至4行正文窗口,
            NIL;,
        0<=Y<16→
```

```
            置屏幕为满屏方式 (40×48), 低分辨率图形,
            以Y颜色号填充屏幕.
            NIL;,
        置屏幕为满屏方式 (40×48), 低分辨率图形,
        清屏幕,
        NIL;,
```

解释：GRAPHICS函数用于建立屏幕，供PLOT函数的低分辨率图形操作使用。如果第一个变元是NIL，在屏幕底部留出4行窗口，以文本方式使用，方便于编程；否则建立全屏幕方式的图形。第二个变元用于控制清除屏幕所用的颜色。如果不是有效的颜色号，则假定为黑。

注：如该函数调用不带变元，则其作用与 APPLESOFT BASIC GR 命令相同。

3. CBV PLOT〔X,Y,Z〕:=

```
0<=Z<16→
  0<=X<40→
    0<=Y<48→
      置图形色点号Z于座标〔X,Y〕,
      TRUE;,
    0<=CAR〔Y〕<48 AND 0<=CADR〔Y〕<48→
      绘颜色号Z的垂直线于X座标X,
        Y座标CAR〔Y〕和CADR〔Y〕之间,
      TRUE;
    NIL;,
  0<=CAR〔X〕<40 AND 0<=CADR〔X〕<40→
    0<=Y<48→
      绘颜色号Z的水平线于Y座标Y,
        X座标CAR〔X〕和CADR〔X〕之间,
      TRUE;
    0<=CAR〔Y〕<48 AND 0<=CADR〔Y〕<48→
      绘颜色号Z的矩形. 在X座标CAR〔X〕
        和CADR〔X〕之间. Y座标CAR〔Y〕
        和CADR〔Y〕之间,
      TRUE;,
    NIL;,
  NIL;,
0<=X<40 AND 0<=Y<48→
    返回座标为〔X,Y〕的图形点的颜色号;,
NIL;
```

解释：muLISP 将 APPLESOFT BASIC 中由 COLOR、PLOT、HLIN、VLIN和SCRN命令提供的低分辨率图形能力方便地组合到一个 PLOT 函数之中。此外，

还可容易地绘制任意尺寸、位置和颜色的矩形。

PLOT 函数的前二个变元指定40×48的图形屏幕的X和Y座标。如果这些变元为正整数，则指定屏幕上的一个点。但如果变元之一或二个变元都是由 2 个正整数组成的表，则指定对应的座标值域。这个值域用于绘图，并提供绘制水平线、垂直线和矩形的手段。

PLOT 函数任选的第三个变元用于指定Apple Ⅱ 颜色。如果未给出第三个变元，或给出的第三个变元不是有效的颜色号，则 PLOT 函数只返回由前二个变元（即〔X,Y〕）指定的点的当前颜色号，这与BASIC中SCRN命令等价。但如果给出颜色号，则该函数在所给座标处“绘制”相应颜色的图形点。

注：下面是可用颜色及相应颜色号列表：

0 黑	4 深绿	8 褐	12浅绿
1 绛	5 灰 1	9 橙	13黄
2 深蓝	6 中蓝	10灰 2	14蓝绿
3 紫红	7 浅蓝	11桃红	15白

例：下列函数用游戏板确定X、Y座标，在屏幕上连续绘制出给定颜色的点。

```
DEFUN PLOTTER (LAMBDA (COLOR)
    (LOOP
        (PLOT (QUOTIENT (PADDLE 0) 7)
              (QUOTIENT (PADDLE 1) 6)
              COLOR) ) ))
```

4. CBV PADDLE〔X〕:=

```
    0<=X<4→
        返回游戏控制板号X的当前值 (0～255之间的数);,
    NIL;
```

解释：这个函数用于自动检测游戏控制板的当前位置，其返回值与 Apple BASIC 的 PDL 函数返回值相同。在 APPLESOFT 手册中读不同的控制板太快的问题已经解决，因此无需人为延迟，便可以得到精确的读入。

5. CBV BEEP〔X,Y〕:=

```
    0<=X<256 AND 0<=Y<256→
        建立音调X音程Y的Apple扬声器声调,
        TRUE;,
    NIL;
```

解释：BEEP函数的变元应为 0～255之间的数，第一个变元指定音调，0 为最高音调，255为最低音调。第二个变元指定音程，0 为最短，255 为最长（约 1 秒）。这个函数只图产生音响效果，而不想匹配音乐的音调。

例：下列循环产生一个连续的音响，其音调和音程由游戏控制板控制：

```
(LOOP (BEEP (PADDLE 0) (PADDLE 1)))
```

Apple Ⅱ Soft Card用户请注意：muLISP的标准 CP/M版本与Soft Card版本之间有一些微小的区别。

本手册第五章第十四节给出LINE LENGTH 的缺省值为72，但对于使用标准40列监

控器的系统，该缺省值为39；对80列的系统，行长度的初值为79（行长度比实际屏幕长度小1，可以避免在屏幕最右列位置显示字符之后，由Apple监控器自动输出回车和换行，造成屏幕上出现不希望的空行）。

由于Apple计算机有“←”键，用于取代“BACK SPACE”键，在本手册第一章第七节中给出的任选可用信息变成：

INTERRUPT；continue：〈RETURN〉

Executive：〈ESC〉 Restart：←

System：〈CTRL-C〉

自然，如果你的系统只支持大写字符，则所有小写字符在输出时均转换为大写字符。

因为标准的Apple键盘没有办法产生左方括号(即“〔”)，CP/M的Soft Card版本自动地将〈CTRL-K〉转换为左方括号，这对于打入超级左圆括号（见第五章第十三节）是很有用的。但是，这也就意味着〈CTRL-K〉不能再用于muLISP程序，如muSTAR中。因此，muSTAR的Apple Ⅱ版本使用〈SHIFT CTRL-N〉取代muSTAR手册中出现的〈CTRL-K〉。

类似地，Soft Card CP/M将〈CTRL-B〉转换为后斜杠字符（即”＼”），因此，用〈CTRL-I〉取代在muSTAR手册中所有出现的〈CTRL-B〉。

如Apple Ⅱ参考手册第五章所述，该计算机具有大量的专用内存单元，用于存取内部I/O硬件。在必要时，可用muLISP的MEMORY函数读或触发这些单元。

因为Soft Card的内存地址转换对这些I/O功能的地址单元进行修改，因此，Apple Ⅱ手册给出的I/O功能地址是无效的。下表列出一些专用内存单元，以十进制和十六进制形式给出其原来的6502地址及Z80地址。

I/O功能	6502地址		Z80地址	
	十六进制	十进制	十六进制	十进制
键盘数据输入	$C000	49152	$E000	57344
清键盘选通	$C010	49168	$E010	57360
盒带输出	$C020	49184	$E020	57376
扬声器开关	$C030	49200	$E030	57392
实用程序选通	$C040	49216	$E040	57408
置图形方式	$C050	49232	$E050	57424
置文本方式	$C051	49233	$E051	57425
非混合文本和图形	$C052	49234	$E052	57426
混合文本和图形	$C053	49235	$E053	57427
显示主页	$C054	49236	$E054	57428
显示次页	$C055	49237	$E055	57429
显示低分辨率图形	$C056	49238	$E056	57430
显示高分辨率图形	$C057	49239	$E057	57431
报警器输出#0 OFF	$C058	49240	$E058	57432
报警器输出#0 ON	$C059	49241	$E059	57433
报警器输出#1 OFF	$C05A	49242	$E05A	57434

报警器输出#1	ON	$C05B	49243	$E05B 57435
报警器输出#2	OFF	$C05C	49244	$E05C 57436
报警器输出#2	ON	$C05D	49245	$E05D 57437
报警器输出#3	OFF	$C05E	49246	$E05E 57438
报警器输出#3	ON	$C05F	49247	$E05F 57439
盒带输入		$C060	49248	$E060 57440
按钮输入#1		$C061	49249	$E061 57441
按钮输入#2		$C062	49250	$E062 57442
按钮输入#3		$C063	49251	$E063 57443
游戏控制#1		$C064	49252	$E064 57444
游戏控制#2		$C065	49253	$E065 57445
游戏控制#3		$C066	49254	$E066 57446
游戏控制#4		$C067	49255	$E067 57447

第六章　muSTAR辅助程序

muLISP源文件可用下列两种方法之一产生：外部编辑程序或常驻LISP编辑程序。随同计算机磁盘操作系统提供的文本编辑程序是外部编辑程序的一个例子。用外部编辑程序进行程序开发就是对源文件反复进行编辑、装入和测试。

使用外部编辑程序的主要优点是一般用户对它比较熟，且注释较容易加到文本中。虽然外部编辑程序是相当满意的一种程序开发方法，但在某种程度上说，它比较慢，不方便，且不利于创造性工作，甚至对函数定义作微小的改变，也要重复对磁盘进行存取。

在muLISP-80中引入muSTAR人工智能开发系统就方便地解决了这些问题。muSTAR大大减少了程序开发时间，并充分利用了LISP的交互特性。利用muSTAR提供的常驻编辑程序和跟踪功能，函数定义的创建、测试及排错全部可在系统内部进行。这就增强了编程手段、并避免了繁琐的编辑—排错—重编辑的周期。程序开发工作快得多、也有趣得多。

当要保存以上述方式编好的程序时，muSTAR产生格式打印的muLISP源文件，包括全部变量值、函数定义和已建立的特性值。由于这个源文件可不用muSTAR，而装入单独的muLISP系统中，因此，全部内存空间均可用于运行应用程序。

本章后面部分将介绍muSTAR辅助程序的使用：如何进入，如何充分使用各种任选变量，如何使用文本编辑命令。文本编辑命令包括光标控制、文本显示以及文本的插入、删除。第三节描述muSTAR如何工作，这对于想对它进行扩充或修改的用户是有价值的。

muSTAR编辑程序可在任何高数据传输速率（波特率）的现代CRT终端上使用。muSTAR使用的唯一一个非标准特性是：光标上移一行和光标移至初始位置（即屏幕左上角）的字符。muSTAR提供的版本假定在Lear Siegler ADM-3A，Soroc IQ-120和标准Apple Ⅱ终端上运行。如果用其他终端，必须写用户化函数执行光标移动功能。如何进行，见本章第三节详细叙述。

第一节　命令总清单

muSTAR是以muLISP的SYS文件形式提供的。因此，可按第一章第八节所述的方法装入。最容易的办法是使用下列CP/M级命令：

```
A>MULISP MUSTAR
```

该命令首先装入muLISP，显示注册提示信息，随后装入muSTAR系统，调用执行驱动函数。驱动函数将显示muSTAR用户可用的任选命令：

```
O P T I O N S
F  EDIT FUNCTION
V  EDIT VARIABLE
P  EDIT PROPERTY
E  EVAL LISP
```

Q EVAL-QUOTE LISP
T TRACE FUNCTION
U UNTRACE FUNCTION
R READ FILE
W WRITE FILE
D SELECT DRIVE
X EXIT TO DOS

ENTER CHOICE:

muSTAR 中可用的功能分为四类：如上表中列出，前三个任选调用面向显示的编辑程序，接着四个用于程序调试，再下面三个用于磁盘文件存储，最后一个X任选退出muLISP muSTAR，控制返回主操作系统。

当这个命令单显示出来时，打入任一有效任选字符将启动相应的功能。系统只对有效任选字符响应。对非期望的选择，在提示用户输入时打回车便可中止。

当已经记住了上述命令单，不再需要用它时，可将变量MENU置为NIL，禁止其输出。

下面详述各种任选功能：

1. 编辑功能

① F EDIT FUNCTION

这个任选用于建立和（或）编辑muLISP函数的LAMBDA定义。当启动该任选时，显示下列提示：

FUNCTION NAME (S):

然后可打入多个函数名，所有函数名应能放在一行内。打回车后，系统以格式打印形式显示函数定义，用缩进去的格式表示出函数定义结构。光标定位在文本开始处。然后可用编辑程序修改文本，如下节所述。一旦编辑完成后，如果用户需要，就可用修改过的定义，重新定义这些函数。

② V EDIT VARIABLE

这个任选用于建立和（或）编辑程序的变量值。当启动该任选时，显示如下提示：

VARIABLE NAME (S):

可打入多个变量名，所有变量名应能放在同一行内。打回车后，系统显示这些变量和它们的格式打印形式的值，光标定位在文本开始处。然后可用编辑程序修改正文，以及对变量重新赋值。

③ P EDIT PROPERTY

这个任选用于建立和（或）编辑名的特性值。当启动该任选时，首先显示下列提示：

NAME:

只允许对一个名作特性编辑；多余的名忽略不计。在打回车后，系统将显示下列提示：

INDICATOR:

同样，只能指定一个特性指示符。打回车后，系统显示变量名、指示符和当前特性值。编辑后，可对修改过的表达式重新求值。

2. 调试功能

① E EVAL LISP

这个任选用于启动一个eval-LISP驱动循环。其提示字串为“*”，用于区别缺省的muLISP提示字串“$”。这个eval-LISP循环一直继续到一个表达式求值为名EXIT。

② Q EVAL-QUOTE LISP

这个任选用于启动一个eval-quote-LISP驱动函数，它主要供LISP用户作程序调试用，其提示字串为“#”。eval-quote-LISP循环一直继续到执行EXIT（）函数为止。

③ T TRACE FUNCTION

这个任选用于跟踪LAMBDA或NLAMBDA定义的函数，作调试用。当启动该任选时，显示下列提示：

FUNCTION NAME (S):

可打入多个函数名，所有函数名应能放在同一行内，然后打回车。当调用被跟踪的函数时，显示函数名和实际变元。当函数返回时，显示函数名和返回值。缩进去的格式用于表示函数调用的嵌套。

④ U UNTRACE FUNCTION

这个函数用于去除跟踪，即把被跟踪函数定义恢复为原始定义。显示的提示信息与跟踪函数相同，请求用户输入一系列函数来去除跟踪。

3. 磁盘 I/O 功能

① R READ FILE

这个任选用于读muLISP源文件。当启动该任选时，显示下列提示：

FILE NAME:

打入一个名和回车后，试图打开给定名的LIB型文件，用于输入，并将该文件读入系统中。如果文件没有找到，则显示如下信息：

FILE NOT FOUND

并重新提示要求文件名。

由muSTAR或外部编辑程序产生的文件均可用此法装入。对于外部建立的文件，文件中最后一条语句应该是调用RDS函数，使控制返回muSTAR。函数DRIVER不应被源文件修改。

② W WRITE FILE

这个任选用于生成muLISP源文件。当启动该任选时，显示如下提示：

FILE NAME:

打入一个名和回车后，将建立给定名的LIB型新文件。一系列函数定义和（或）变量值以及特性，以格式打印的形式写入该文件。被写入的函数是在该文件名特性表中指示符FUNCTIONS下的那些函数。被写入的变量和（或）特性值是在该文件名特性表中指示符VARIABLE下的那些名的变量或特性值。这些特性值本身也写入到该文件中，这样可被自动读回，将来作muSTAR编辑用。注意，当开发阶段完成后，muSTAR产生的文件可由单独的muLISP系统读入。

③ D SELECT DISK

这个任选用于选择后继磁盘文件存取所用的磁盘驱动器。在用该任选改变驱动器之前，使用当前缺省磁盘。

第二节　文本编辑

muSTAR 编辑程序是专门为方便对muLISP函数的编辑而设计的。它是面向屏幕的LISP编辑程序，该编辑程序连续显示组成函数定义文本的“窗口”或“画面”。因此，用户可以不断地作修改，并且看见结果。

因为 muSTAR 完全是用 muLISP 写的，用户可以按自己的意愿扩充该系统。基本上每个 ASCII 控制字符都是一个函数，可随意重定义以执行所需功能，在本章第三节将详细叙述。

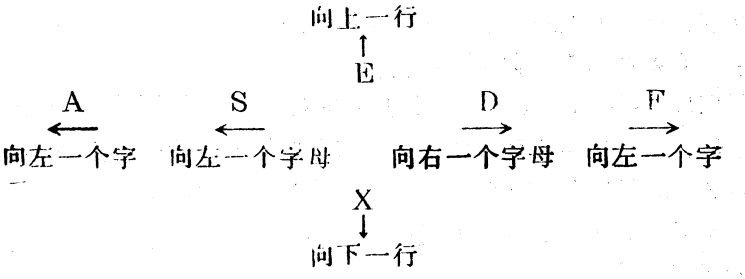
在 muSTAR 中使用的可能与用户程序名冲突的函数和变量名以“$”结束。为避免这种矛盾，建议用户程序中不要使用“$”结尾的名。

使用 muSTAR 编辑程序的一般规则可描述为下列两个步骤：首先，将光标移到要编辑的位置；然后，在该点作文本的插入、删除或修改。学会使用这个系统的最好办法是对一些简单函数的定义作编辑的试验。

2.1　光标控制

光标控制命令用于将光标定位于文本中打算进行修改的位置。光标主要通过左手敲入的键盘命令控制。在大部分终端上，控制字符输入的办法是在按控制键（通常标为〈CTRL〉）的同时打入所需字符。

基本光标控制以加号方式定向，因此在标准键盘上，键的位置相应于光标移动的方向。如下图所示：



上图中，箭头方向和长度指出由相关命令引起的光标移动方向及移动量。

〈CTRL-D〉将光标右移一个字符。〈CTRL-S〉将光标左移一个字符（相当于〈CTRL-H〉或 backspace键）。〈CTRL-F〉将光标右移一个字。〈CTRL-A〉将光标左移一个字。这两条命令都把光标停在一个字的开始位置。

在 muSTAR 中，字（WORD）定义为文本中的 muLISP 名或定界符，如括号或句号。注意，对上述四条命令，屏幕上的文本应看成是一长串正文。因此，当到达当前行末尾时，光标将自动前进到下一行。相反，当光标退到一行开始时，将自动移到前一行末尾。

〈CTRL-E〉将光标上移一行，〈CTRL-X〉将光标下移一行。用这两条命令将光标移到新行后，光标自动进到该行的下一个字。打入换行或〈CTRL-J〉将光标移到下一行开始。

用〈CTRL-Q〉命令，将光标移到一行开始，打〈CTRL-B〉，将光标向相反方向移动到一行的结尾（最后一个字符后面）。Apple Ⅱ SoftCard 用户使用〈CTRL-I〉代

替〈CTRL-B〉。

回车，即〈CTRL-M〉，在非插入方式下使用时，将光标移至下一行开始。如果光标已在正文的底部，则插入一新行。插入方式下回车键的作用，见2.3。

光标控制命令只能移动光标，将其放在屏幕上文本字符位置。要将光标移至超出当前行结束处或超出正文最后一行的位置，然后插入新的文本，则使用空格键或回车键，而不用光标控制字符。

2.2 显示控制

在屏幕上显示长的定义或多个定义时，在一个时间里只能看见一部分文本。可以想象把文本写在一个长卷上，CRT屏幕是文本的窗口，只能看到长卷的一部分。因为CRT这个窗口是不能移动的，所以muSTAR将文本卷上或卷下，文本的这种向上或向下的移动称为"滚动"。

要看文本的下一行，可打〈CTRL-Z〉将文本向上卷一行，这相当于窗口下移一行。要看文本的前一行，可打〈CTRL-W〉将文本向下卷一行，屏幕滚动时，光标停留在文本同一位置上，随文本移上移下。但当光标到达屏幕顶部或底部时，则停留在那里。光标永远不离开屏幕。

〈CTRL-C〉用于将窗口下移一个屏幕(即在24行屏幕上显示下面18行)。〈CTRL-R〉相反，将窗口上移一个屏幕。如果在窗口移动方向没有文本时，显示控制命令无效。

2.3 送入文本

送入文本的方式有两种.〈CTRL-V〉是触发开关，用于在两种方式之间转换。

开始，触发开关OFF，编辑程序在非插入方式(non-INSERT mode)，打入的文字复盖原来的文字，这是开始送入文本的最简单办法。改正错误的文字时只需简单地在其上打入正确的文字。空格杠用于"擦除"字符。

当打入〈CTRL-V〉时，muSTAR进入插入方式(INSERT mode)。打入的字符和空格插入到该行其余字符的前面。插入字符右边的那些字符被向右推。例如，在正文"CONS BETA"的"BETA"前面插入"ALPHA"，变为"CONS ALPHA BETA"。其过程在控制台上显示如下，底线表示光标位置：

```
(CONS BETA)
(CONS ABETA)
(CONS ALBETA)
(CONS ALPBETA)
(CONS ALPHBETA)
(CONS ALPHABETA)
(CONS ALPHA BETA)
```

当muSTAR在插入文本方式时，回车(即〈CTRL-M〉导致文本中插入新行，光标停在新行结束处。非插入方式下回车的作用前面已经提及(见2.1)。打入〈CTRL-N〉在文本中插入新空行，但光标停在该行上。

〈CTRL-P〉是转义控制字符，用于将定界符转义作为名而不作为定界符。为使用空格、圆括号、句号、方括号作为名，可在这些定界符前面打〈CTRL-P〉。

2.4 删除文本

删除文本的方法有四种：

（1）在非插入方式，只需在欲删除的文字上打入空格或新字符。

（2）为删除光标所在字符，打〈CTRL-G〉。

（3）为删除光标右边的字，打〈CTRL-T〉。

（4）打〈CTRL-Y〉删除当前行 。

注意：因为打T时很容易不小心碰到Y，所以删除字时要特别小心，以免删除整行。尽量少用〈CTRL-Y〉也是一个办法，因为重复使用〈CTRL-T〉命令也可达到删除一行的效果，这个方法还可使用户将某些字重新安排在新的一行中。

2.5 查找名

下列命令具有查找文本中 muLISP 名的功能。〈CTRL-O〉在屏幕首行显示下列提示.

FIND NAME?

然后用户可打入一个名，以回车结束。则光标移到文本中该名下一次出现的地方。该命令只对实际在屏幕上的文本进行查找，如果在屏幕上未找到指定的名，则从屏幕顶端重新开始查找。如果仍未找到，则光标停在原位置右边前进一个字的位置。

为查找与前一次〈CTRL-O〉给出的相同的名的下一次出现，打〈CTRL-L〉。

2.6 退出编辑程序

当变量、函数或特性的编辑完成后，打入〈CTRL-K〉(对Apple Ⅱ SoftCard用户使用〈SHIFT CTRL-N〉)，控制台屏幕显示下列内容：

```
        OPTIONS
    E  EVALUATE  TEXT
    A  ABANDON  TEXT
    C  CONTINUE  EDIT
ENTER  CHOICE:
```

通常选择“E”任选，对编辑好的文本求值。“A”任选废弃已编辑文本，不求值。如果不小心打了〈CTRL-K〉，可用“C”任选，返回编辑环境。

2.7 命令一览表

控制字符	功 能
A	光标左移一个字
B*	光标移至当前行结束
C	文本向上卷一屏
D	光标右移一个字符
E	光标上移一行
F	光标右移一个字
G	删除当前字符
H	光标左移一个字符
I	未使用
J	光标下移一行，跟在前导空格后（换行）
K**	退出编辑程序

L　重复前一个查找命令
M　光标下移一行，至一行开始处（回车）
N　插入一新行，不移动光标
O　找一个名
P　转义字符，用于送入定界符作名
Q　移至当前行开始
R　文本向下卷一屏
S　光标左移一个字符
T　删除光标右边一个字
U　未使用
V　转换插入方式开关
W　文本向下卷一行
X　光标下移一行
Y　删除当前行
Z　文本向上卷一行

* Apple Ⅱ SoftCard用户使用〈CTRL-I〉代替〈CTRL-B〉
* * Apple Ⅱ SoftCard用户使用〈SHIFT CTRL-N〉代替〈CTRL-K〉

第三节　muSTAR用户化

3.1　控制台用户化

不同的终端使用不同的控制字符来移动光标。muSTAR提供的版本在Lear Siegler ADM-3A，Soroc IQ-120或仿制它们的终端上可直接运行，不用作任何改动。对Apple Ⅱ计算机，已为标准的Apple终端作了用户化。其他终端用户必须写一个简单的muLISP库文件（LIB文件），包含光标上移一行和光标移至初始位置（屏幕左上角）的函数。

muSTAR函数UP-LIN$用于将光标移至当前行开始位置和上移给定行数。如果没有给出变元，则上移一行。HOME$将光标移至屏幕左上角位置，不清除屏面。PRINT函数在HOME中是必须的，它用于将光标位置计数器复位。下面是这些函数的缺省定义。

```
(DEFUN UP-LIN$ (LAMBDA (NUM)
    ((ZEROP NUM))
    (PRIN1 CR)
    (LOOP
        (PRIN1 UPLINE)
        (SETQ NUM (SUB1 NUM))
        ((NOT (PLUSP NUM))) ) ))
(DEFUN HOME$ (LAMBDA ()
    (PRINT HOME) (PRIN1 HOME) ))
(SETQ PAG-LEN$ 24)      %Number of lines on CRT screen%
(SETQ LIN-LEN$ 79)      %Number of columns on CRT -1%
```

```
(SETQ HOME CTRL-^)     %Cursor home char, CTRL-^=1E hex%
(SETQ UPLINE CTRL-K)   %Upline char, CTRL-K=0B hex%
(SETQ CR CTRL-M)       %Carriage return, CTRL-M=0D hex%
(SETQ BACK CTRL-H)     %Backspace char, CTRL-H=08 hex%
(RDS)
```

为编程方便，每个 ASCII 字符都指定了相应的 muLISP 名。例如在上面的赋值中，CTRL-H求值为一个名，其ASCII打印名为08。

变量 PAG-LEN$ 和 LINLEN$ 用于调整终端的页和行长度（注意，行长度必须设置为比实际终端上的列数小 1，以防在大多数终端上当屏幕最右列的字符被打印时，产生自动回车换行输出）。

为了对你的终端做用户化，首先用外部编辑程序，建立一个 LIB 型文件，可用上述定义作为模型。注意，在该文件末尾必须包含RDS函数调用，使控制台作为当前输入源。在装入MUSTAR.SYS之后，选择“R”任选，读入你的用户化文件。然后，将经过用户化的muSTAR版本作为SYS文件保存，并使用新名，与未改制版本相区别。为完成这一工作，可选择eval-LISP “E” 任选，然后作下列命令：

```
(SAVE MYSTAR)
```

建立名为MYSTAR.SYS的磁盘文件。此后，启动muSTAR只需装入 MYSTAR，如本章第一节所述。

3.2 muSTAR执行

执行是简单的循环，它显示任选项目总清单，然后调用函数，执行所需任选。在每个任选字符的特性中，在指示符EXECUTIVE下，是LAMBDA体，当相应的任选被选择时，对该体求值。函数QUERY $用于显示提示符及输入回答。它返回由用户送入的一系列表达式。

如果选择编辑程序任选，这个表传递给S表达式一文本翻译程序，这里，表达式转换为内部文本数据结构，然后调用编辑程序，进行格式打印及进行文本编辑。编辑完成后，如果用户想对文本求值，则调用文本-S表达式翻译程序，翻译文本并求出结果。如果括号不配对，或选择CONTINUE任选，则重新调用编辑程序，继续编辑。

T (TRACE) 和U (UNTRACE) 任选将用户送入的名表分别传递给 TRACE 和 UNTRACE函数。对一个函数进行跟踪是通过对它重新定义，调用EVTRACE函数。因此，当被跟踪函数被调用时，它调用EVTRACE函数，打印出它的变元，并对它的原始函数求值，将结果值打印出来。

W (WRITE FILE) 任选打开驱动器 * DRIVE * 上的文件，用于输出。首先，它打印出为读入该文件剩余部分所需的基本函数（即DEFUN、SETQQ、PUTQQ 和 FLAGQQ），然后打印出该文件中的函数和变量的表。最后输出格式打印的函数定义、名值和特性值。如果要保存的函数已被跟踪，则保存的是函数的原始定义而不是被跟踪版本。

3.3 文本数据结构

对于有经验的muLISP程序员来说，要在编辑程序中加一些功能以适合自己的要求，相对来说是比较简单的。可作的改动包括修改控制键的作用和利用特殊的 CRT 功能。第一步是大概熟悉文本如何存放和修改，然后，对系统中用于管理文本的现有构件进行组合，以

达到预期效果。

任何时候，屏幕上现有文本都是作为约束到全程变量 * TEXT * 上的表来存放的。除 * TEXT * 的第一个元素（总是NIL）外，该表每一个元素都代表一行正文。全程变量 PAG-LEN $ 设置为CRT屏幕允许的最大行数，其缺省值为 24。因此，表 * TEXT * 的长度永不大于PAG-LEN $。

正文每一行（即 * TEXT * 元素）是一个表，说明该行的正文。一行的第一个元素总是非负整数，给出该行前导空格的数目。表中其余元素称为记号，记号或者代表 muLISP 定界符，或者组成被表示的正文的名。有四种可能的定界符，如下面双引号内所示：

"(" ")" " " " "

这四种定界符在表中作为名存放，反之，代表名的记号在该表上作为子表出现。通常，子表是单元素表，其元素是我们所说的名。仅当光标在一个名的内部时（即光标定位于首字符之后），才使用UNPACK $ 函数将名拆成组成该名的字符。当光标离开该名时，这些字符用PACK $ 函数重新装配成新名。

在任何给定时刻，全程变量 * ROW * 是 * TEXT * 的子表，该子表的第二个元素（即 CADR）是光标当前所在的正文行。这就是 * TEXT * 第一个元素是空行NIL的原因。无论如何，当前行上面有一行，对于使用RPLACD从 * TEXT * 中删除当前行是方便的。

光标在当前行中的现位置是由全程变量 * COL * 确定的。它总是文本当前行的子表。由于和 * ROW * 同样的原因，子表开始于当前记号左边一个记号处。注意，光标永不在一行的前导空格左边。因此，* COL * 可以总在当前记号前面。

被卷出屏幕顶或底部的文本行作为表分别存放在 * PRE-TEXT * 和 * POST-TEXT * 变量下。这两个表最好看成是堆栈，与当前"窗口"紧邻的文本行是栈顶元素。因而，组成 * PRE-TEXT * 的行是以相反顺序存放的。一行正文仅当屏幕上没有足够地方存放时，才能被推至 * PRE-TEXT * 或 * POST-TEXT * 上。

3.4 文本源语

当运行文本编辑程序时，控制台输入是以原始输入方式，如第五章第十三节所述。一个输入字符可以是定界符，正常的可打印字符或控制字符。定界符或可打印字符按照上面为存放文本而定义的结构，作为当前文本行的一部分。如果控制变量 * INSERT * 为非 NIL，该字符被加到文本中，而不是复盖旧文本。

当muSTAR初始装入时，每个ASCII控制字符均被赋予一个别名作为值。对称地，这些别名也被赋予一个控制字符作为值。例如，下面的赋值用于设置 control A 的值（表示为"^A"）为muLISP名CTRL-A，反之亦然：

```
(SETQQ "^A" CTRL-A)
(SETQQ CTRL-A "^A")
```

其余控制字符被赋予类似的别名，当打入控制字符时，对相应的别名定义求值。因此，函数CTRL-A将光标和文本指针左移一个记号。这些函数每一个均已在前面（见本章第二节2.7）作了描述。修改这些别名函数的定义是作 muSTAR 用户化最简单的办法。在 muSTAR源文件中，现在有一组完整的原语，其每一个的目的，可从函数名明显看出。

附　录

附录A　巴科斯范式

巴科斯范式（BNF）提供了对语言语法进行形式定义的标准规则。在BNF等式内前后由“〈”和“〉”括住的字符串表示由该字符串命名的对象类别。没有这样定界的字符串代表该串本身。注意这种规定与常规应用的区别，在常规应用中带引号的串表示其本身，无引号的串是对象的名。

BNF等式定义一组从属于特定种类的语言语法的语法对象。符号“::=”用作“is a”（是）的缩写，“|”用作“or”（或）的缩写。

数的BNF定义可描述如下：

〈数〉::=〈数字〉|〈数字〉〈数〉

〈数字〉::=0|1|2|3|4|5|6|7|8|9

如上例所示，在BNF等式中经常使用递归定义。还要注意，类名并置表示在这些类内部对象的对应连接。

关于巴科斯范式的定义，请参阅：

ALGOL 60 Report, Revised Report on Algorith Language ALGOL 60", Communications of the ACM, Volume 6, January 1963.

附录B　执行机器语言子程序

某些特殊的muLISP应用可能需要写机器语言子程序。例如，用户可能希望muLISP增加图形能力，或者为提高效率，将特别关键的函数递交编译。

muLISP用编址类型来确定函数类型。这就要求所有机器语言子程序开始于内存低地址。从0103H（十六进制）单元开始设置的空转移表，用于从低址内存到用户定义例程的转跳。对4条JMP指令（操作码为C3H）有足够的内存空间，这些转移指令可被修改为转跳至用户定义例程所在的地址。在不同系统中，内存零页面未使用区可能还有一些空间用于转跳。

因为muLISP使用DOS（磁盘操作系统）下面全部可用内存，所以，用户定义的例程最好放在DOS上面的“被保护”内存区。当然，这就要求生成比你的计算机系统通常可以支持的稍小一些的DOS。另一个方法是将0005H单元的JMP指令改变到比当前地址小的地址上，使muLISP认为它运行在较小的DOS下。当然，在这个新地址上必须放一条JMP指令，转移至DOS。这样就可以将DOS下面的一些内存空出来。

所有用户定义的子程序都是CBV，并且是最多有三个变元的伸展函数。如果要求三个变元以上，可将这些变元作为一个表来传送。变元地址传递到机器语言例程，放在寄存器对中。第一个变元在HL寄存器，第二个变元在DE寄存器，第三个变元在BC寄存器。

为将控制返回muLISP，可用RET指令终结所有例程。函数返回值将是HL寄存器对

指向的数据结构，它必须是真正的muLISP数据对象的地址。即使不需要返回特殊值，HL也将被设置为某值，如NIL。NIL值可由函数NULL的机器语言定义反汇编来确定。

连接机器语言例程通过使用muLISP函数PUTD完成。例如，下列表达式求值将定义函数FOO为例行程序，该例程开始于0103H单元：

```
(PUTD (QUOTE FOO) 259)
```

可用GETD函数找到原始定义的muLISP子程序开始地址。知道了这些地址，用户定义的子程序可直接调用原始子程序。

附录C　程序清单

```
%File: UTILITY.LIB        08/11/80        The Soft Warehouse%
(PROG1""
  (PUTD DEFUN (QUOTE (NLAMBDA (FUNC DEF)
    (PUTD FUNC DEF)
   FUNC) )))

%Function APPEND returns a list consisting of the elements of LST1
appended to LST2.%

(DEFUN APPEND (LAMBDA (LST1 LST2)
  ((NULL LST1) LST2)
  (CONS (CAR LST1) (APPEED (CDR LST1) LST2)) ))

%Function COPY returns a copy of its argument.%

(DEFUN COPY (LAMBDA (EXPN)
  ( (ATOM EXPN) EXPN)
  (CONS (COPY (CAR EXPN)) (COPY (CDR EXPN))) ))

%Function UNION returns the union of LSF1 and LST2.%

(DEFUN UNION (LAMBDA (LST1 LST2)
  ((NULL LST1) LST2)
  ((MEMBER (CAR LST1) LST2)
    (UNION (CDR LST1) LST2) )
  (CONS (CAR LST1) (UNION (CDR LST1) LST2)) ))

%Function INTERSECTION returns the intersection of LST1 and
LST2.%
```

```
(DEFUN INTERSECTION (LAMBDA (LST1 LST2)
((NULL LST1) NIL)
  ((MEMBER (CAR LST1) LST2)
    (CONS (CAR LST1) (INTERSECTION (CDR LST1) LST2)))
(INTERSECTION (CDR LST1 LST2) ))
```

%Function SUBSET is a comparator returning T iff LST1 is a subset of LST2.%

```
(DEFUN SUBSET (LAMBDA (LST1 LST2)
  ((NULL LST1))
  ((MEMBER (CAR LST1) LST2)
    (SUBSET (CDR LST1) LST2)) ))
```

%Function SUPERREVERSE returns a list of the elements of LST1 reversed at all levels.%

```
(DEFUN SUPERREVERSE (LAMBDA (LST1 LST2)
  ( (NULL LST1) LST2)
  ((ATOM (CAR LST1))
    (SUPERREVERSE (CDR LST1) (CONS (CAR LST1) LST2)))
  (SUPERREVERSE (CDR  LST1) (CONS (SUPERREVERSE (CAR-
LST1) ) LST2)) ))
```

%Function REMBER is a constructor returning a list in which all occurrences of ATM has been removed from LST.%

```
(DEFUN REMBER (LAMBDA (ATM LST)
  ((NULL LST) NIL)
  ((EQ ATM (CAR LST))
    (REMBER ATM (CDR LST)) )
  (CONS (CAR LST) (REMBER ATM (CDR LST))) ))
```

%Function SUBST is a constructor returning the expression resulting from replacing all occurrences of OLD by NEW in EXPN.%

```
(DEFUN SUBST (LAMBDA (OLD NEW EXPN)
  ((EQUAL OLD EXPN) NEW)
  ((ATOM EXPN) EXPN)
  (CONS (SUBST OLD NEW (CAR EXPN)) (SUBST OLD NEW (CDR-
```

```
 EXPN))) ))
```

%Function NTH is a selector which returns the result of removing the first NUM elements from the list LST.%

```
(DEFUN NTH (LAMBDA (LST NUM)
  ((NOT (PLUSP NUM))
   LST)
  (LOOP
    (SETQ LST (CDR LST))
    (SETQ NUM (SUB1 NUM))
    ((ZEROP NUM)
     LST) ) ))
```

%Function GENSYM is a constructor which returns a new name of the form Gxxxx where xxxx is a number incremented each time GENSYM is called.%

```
(SETQ GENSYM 0)

(DEFUN GENSYM (LAMBDA (NUM LST)
  (SETQ NUM (DIFFERENCE 4 (LENGTH GENSYM)))
  (LOOP
    ((ZEROP NUM))
    (PUSH 0 LST)
    (SETQ NUM (SUB1 NUM)) )
  (RPOG1
    (PACK (NCONC (CONS (QUOTE G) LST) (LIST GENSYM)))
    (SETQ GENSYM (ADD1 GENSYM)) ) ))
```

%Function MAX returns the greater of two numbers.%

```
(DEFUN MAX (LAMBDA (M N)
  ((GREATERP M N ) M)
 N))

(DEFUN ADD1 (LAMBDA (NUM)
  (PLUS NUM 1) ))
```

```
(DEFUN SUB1 (LAMBDA (NUM)
  (DIFFERENCE NUM 1) ))
```

%Function DEPTH returns the maximum depth of an expression.%

```
(DEFUN DEPTH (LAMBDA (EXPN)
  ((ATOM EXPN) 0)
  (ADD1 (MAX (DEPTH (CAR EXPN)) (DEPTH (CDR EXPN)))) ))
```

%Function ABS returns the absolute value of NUM.%

```
(DEFUN ABS (LAMBDA (NUM)
  ((MINUSP NUM)
    (MINUS NUM) )
 NUM))
```

%Function FACTORIAL returns NUM factorial.%

```
(DEFUN FACTORIAL (LAMBDA (NUM
   ANS)
  ((NOT (GREATERP NUM -1)) NIL)
  (SETQ ANS 1)
  (LOOP
    ((EQ NUM 0) ANS)
    (SETQ ANS (TIMES NUM ANS))
    (SETQ NUM (SUB1 NUM)) ) ))
```

%Function POWER returns NUM1 raised to the NUM2 power. NUM3 is a local or temporary variable for the function POWER.%

```
(DEFUN POWER (LAMBDA (NUM1 NUM2
   NUM3)
  (SETQ NUM3 1 )
  (LOOP
    (SETQ NUM2 (DIVIDE NUM2 2 ))
    ( ((EQ (CDR NUM2) 1)
       (SETQ NUM3 (TIMES NUM1 NUM3)) ) )
    (SETQ NUM2 (CAR NUM2))
    ((ZEROP NUM2) NUM3)
```

```
    (SETQ NUM1 (TIMES NUM1 NUM1)) ) ))

%Function GCD returns the Greatest Common Divisor of NUM1 and
NUM2.%

(DEFUN GCD (LAMBDA (NUM1 NUM2
  NUM3)
  (LOOP
    ((ZEROP NUM2) NUM1)
    (SETQ NUM3 NUM2)
    (SETQ NUM2 (REMAINDER NUM1 NUM2))
    (SETQ NUM1 NUM3) ) ))

%The following are examples of Mapping Functions equivalent to the
definitions found in LISP tutorials.%

(DEFUN MAPC (LAMBDA (LST FUN)
  (LOOP
    ((NULL LST) NIL)
    (FUN (POP LST)) ) ))

(DEFUN MAPCAR (LAMBDA (LST FUN)
  ((NULL LST) NIL)
  (CONS (FUN (CAR LST)) (MAPCAR (CDR LST) FUN)) ))

(DEFUN MAPLIST (LAMBDA (LST FUN)
  ((NULL LST) NIL)
  (CONS (FUN LST) (MAPLIST (CDR LST) FUN)) ))

(RDS)              %DELETE THIS LINE IF YOU WANT AN EVA-
LQUOTE DRIVER%

%Eunction DRIVER is originally defined in machine language to be an
EVAL-LISP executive driver. However, it may be redefined as desi-
red The following is an EVAL-QUOTE driver, which must be used to
load the remainder of the functions in this file.%

(DEFUN DRIVER (LAMBDA (RDS WRS)
  (LOOP
```

```
    (TERPRI)
    (PRINI (QUOTE"> "))
    (PRINT (APPLY (READ) (READ) (TERPRI))) ) ))

(DRIVER)

%File: MUSTAR LIB        01/04/82        The soft Warehouse%

(LOOP (PRINI*) (EVAL (READ)) ((NULL RDS)) )

%******** UTILITY ROUTINES ********%

(PUTD DEFUN (QUOTE (NLAMBDA (FUN$ EXP$)
  ((EQUAL (GETD FUN$) EXP$))
  ( ((NULL (GETD FUN$)))
    (PRINI "REDEFINED ")
    (PRINT FUN$) )
  (PUTD FUN$ EXP$)
 PUN$))) )

(DEFUN SETQQ (NLAMBDA (NAM$ EXP$)
  (SET NAM$ EXP$)
 NAM$ ))

(DEFUN PUTQQ (NLAMBDA (NAM$ ATM$ EXP$)
  (PUT NAM$ ATM$ EXP$)
 NAM$ ))

(DEFUN FLAGQQ (NLAMBDA (NAMS ATMS)
  (FLAG NAM$ ATM$)
 NAM$))

(DEFUN PACK$ (LAMBDA (TOKEN)
  ((NULL (CDR TOKEN)) TOKEN)
  ((INT-PCK TOKEN))
  (CONS (PACK TOKEN)) ))

(DEFUN INT-PCK (LAMBDA (LST$ NUM EXP$)
  ( ((EQ (CAR LST$)  (QUOTE -))
```

```
      (POP LST$)
      (SETQ EXP$ T) ) )
  (SETQ NUM 0)
  (LOOP
    ((NULL LST$)
      ((NOT EXP$)
        (LIST NUM) )
      (LIST (MINUS NUM)) )
    ((NOT (NUMBERP (CAR LST$))) NIL)
    (SETQ NUM (PLUS (TIMES (RADIX) NUM) (POP LST$))) ) ))

(DEFUN UNPACK$ (LAMBDA (TOKEN)
  ((OR
       (CDR TOKEN)
       (EQ (LENGTH (CAR TOKEN)) 1) ) TOKEN)
  (DIG-NUM (UNPACK (CAR TOKEN))) ))

(DEFUN DIG-NUM (LAMBDA (LST$)
  ((NULL LST$) NIL)
  ((ASSOC (CAR LST$) DIG-NUM)
    (CONS (CDR (ASSOC (CAR LST$) DIG-NUM)) (DIG-NUM (CDR-
     LST$))) )
  (CONS (CAR LSTS) (DIG-NUM (CDR LST$))) ))

(SETQQ DIG-NUM (("0". 0) ("1". 1) ("2". 2) ("3". 3) ("4". 4) ("5".5)
  ("6". 6) ("7". 7) ("8". 8) ("9". 9) ))

(DEFUN RDC$ (LAMBDA (LST$ EXP$)
  (LOOP
    ((EQ (CDR LST$) EXP$)
     LSTS )
    (POP LST$) ) ))

(DEFUN CHOP$ (LAMBDA (LST$)
  (LOOP
    ((NULL (CDDR LST$))
     PROG1 (CADR LST$) (PRLACD LST$ NIL)) )
    (POP LST$) ) ))
```

```
(DEFUN SPLIT$ (LAMBDA (LST$ NUM)
  (LOOP
    ((ATOM LST$) NIL)
    (SETQ NUM (SUB1 NUM))
    ((ZEROP NUM)
      (PROG1 (CDR LST$)) RPLACD LST$ NIL)) )
    (POP LST$) ) ))

(DEFUN MENU$ (LAMBDA (LST$
   READCH READ)
  ( ((NOT MENU))
    (TERPRI)
    (SPACES (QUOTIENT (DIFFERENCE LIN-LEN$ 14) 2))
    (PRTSEN$ (QUOTE (O P T I O N S)) 2)
    (MAPC LST$ (QUOTE (LAMBDA (LINE)
      (SPACES (QUOTIENT (DIFFERENCE LIN-LEN$ 18) 2))
      (PRIN1 (POP LINE))
      (SPACES 2)
      (PRTSEN$ LINE)
      (TERPRI) ))) )
  (TERPRI)
  (SPACES 4)
  (PRTSEN$ (QUOTE (ENTER "CHOICE:")))
  (LOOP
    ((ASSOC (READCH) LST$)
     PRINT RATOM) ) ) ))

(DEFUN PRTSEN$ (LAMBDA (LST$ NUM)
  (MAPC LST$ (QUOTE (LAMBDA (ATM$)
    (PRINI ATM$)
    (SPACES ) )))
  ((NULL NUM))
  (TERPRI NUM) ))

(DEFUN QUERY$ (LAMBDA (ATM$)
  (TERPRI)
  (SPACES (DIFFERENCE 16 (LENGTH ATM$)))
  (PRINI ATM$) (PRIN1 :) (SPACES 1)
```

```
  (RD-LIN$) ))

(DEFUN RD-LIN$ (LAMBDA (
   ATM$ LINE)
  (LOOP
    (SETQ ATM$ (RD-WRD$))
    ( ((NULL ATM$))
      (PUSH ATM$ LINE) )
    ((EQ RATOM CR)
      (REVERSE LINE)) ) ))

(DEFUN RD-WRD$ (LAMBDA (
   ATM$)
  (LOOP
    ((OR(FLAGP(READCH) (QUOTE DEL-CHAR)) (EQ RATOM  CR))
      ((NULL ATM$) NIL)
      (PACK (REVERSE ATM$)) )
    (PUSH RATOM ATM$) ) ))

(DEFUN UP-LIN$ (LAMBDA (NUM)
  ((ZEROP NUM))
  (PRIN1 CR)
  (LOOP
    (PRIN1 UPLINE)
    (SETQ NUM (SUB1 NUM))
    ((NOT (PLUSP NUM))) ) ))

(DEFUN BACKUP$ (LAMBDA (NUM)
  (BCK-SPACE$ NUM)
  (SPACES NUM)
  (BCK-SPACE$ NUM) ))

(DEFUN BCK-SPACE$ (LAMBDA (NUM)
  (LOOP
    ((NOT (PLUSP NUM)))
    (PRINI BACK)
    (SETQ NUM (SUB1 NUM)) ) ))
```

```
(DEFUN HOME$ (LAMBDA ( )
  (PRINT HOME)
  (PRIN1 HOME) ))

(DEFUN SPACE$ (LAMBDA (CHARS)
  (EQ CHAR$ " ") ))

(DEFUN PRIN2 (LAMBDA (EXP$ PRINI)
  (PRIN1 EXP$) ))

(DEFUN APPEND (LAMBDA (LST$ EXP$)
  ((NULL LST$) EXP$)
  (CONS (CAR LST$) (APPEND (CDR LST$) EXP$)) ))

(DEFUN ADD1 (LAMBDA (NUM)
  (PLUS NUM 1) ))

(DEFUN SUB1 (LAMBDA (NUM)
  (DIFFERENCE NUM 1) ))

(DEFUN MAPC (LAMBDA (LST$ FUN$)
  (LOOP
    ((NULL LST$) NIL)
    (FUN$ (POP LST$)) ) ))
```

******** EDITOR EXECUTIVE ********

```
(DEFUN DRIVER (LAMBDA ()          %MUSTAR MAIN DRIVER
                                    FUNCTION%
  (SETQ RDS)
  (SETQ WRS)
  (SETQ ECHO)
  (SETQ *DRIVE*)
  (SETQ PRIN1 T)
  (SETQ READCH T)
  (TERPRI 6)
  (SPACES (QUOTIENT (DIFFERENCE LIN-LEN$ 32) 2))
  (PRINT "***muSTAR AIDS, Version 1.3***")
```

```
 (LOOP
  LINELENGTH (ADD1 LIN-LENS))
   (TERPRI 3)
   (APPLY (GET (MENU$ MENU$) (QUOTE EXECUTIVE ))) ) ))

(SETQ  LIN-LEN$ (DIFFERENCE (LINELENGTH) 1))   %SCREEN-
                                              WIDTH VARIABLE%
(SETQ PAG-LEN$ 24)          %SCREEN HIGHT VARIABLE%
(SETQ MENU T)               %DISPLAY MENU OPTION%

(SETQQ MENU$ (              %MAIN OPTION MENU%
  (F EDIT FUNCTION)
  (V EDIT VARIABLE)
  (P EDIT PROPERTY)
  (E EVAL LISP)
  (Q EVAL-QUOTE LISP)
  (T TRACE FUNCTION)
  (U UNTRACE FUNCTION)
  (R READ FILE)
  (W WRITE FILE)
  (D SELECT DRIVE)
  (X EXIT TO DOS) ))

(PUTQQ F EXECUTIVE (LAMBDA (  %EDIT FUNCTION OPTION%
   LST$)
  (SETQ LST$ (QUERY$ "FUNCTION NAME (S)"))
  ((NULL LST$))
  (MAPC LST$ (QUOTE (LAMBDA (ATM$)
    ((TRACED ATM$)
      (UNTRACE (LIST ATM$))
      (FLAG ATM$ (QUOTE TRACED)) ) )))
  (EDIT-TXT (DEF-TO-TXT LST$))
  (MAPC LST$ (QUOTE (LAMBDA (ATM$)
    ((FLAGP ATM$ (QUOTE TRACED))
      (TRACE (LIST ATM$))
      (REMFLAG ATM$ (QUOTE TRACED)) ) ))) ))

(PUTQQ V EXECUTIVE (LAMBDA (  %EDIT VARIABLE OPTION%
   LST$)
```

```
  (SETQ LST$ (QUERY$ "VARIABLE NAME(S)"))
  ((NULL LST$))
  (EDIT-TXT (SET-TO-TXT LST$)) ))

(PUTQQ P EXECUTIVE (LAMBDA ( %EDIT PROPERTY OPTION%
   NAM$ INDICATOR)
  (SETQ NAM$ (QUERY$ (QUOTE NAM$)))
  ((NULL NAM$))
  (SETQ INDICATOR (QUERY$ (QUOTE INDICATOR)))
  ((NULL INDICATOR))
  (EDIT-TXT (PUT-TO-TXT (CAR NAM$) (CAR INDICATOR))) ))

(PUTQQ E EXECUTIVE (LAMBDA ( )    %EVAL LOOP DRIVER-
   OPTION%
  (LINELENGTH LIN-LEN$)
  (LOOP
   (TERPRI)
   (PRIN1 "*")
   ((EQ (PRINT (EVAL (READ))) EXIT)) ) ))

(PUTQQ Q EXECUTIVE (LAMBDA ( )    %EVAL-QUOTE LOOP-
   OPTION%
  (LINELENGTH LIN-LEN$)
  (LOOP
   (TERPRI)
   (PRIN1 "#")
   ((EQ (PRINT (APPLY (READ) (READ))) EXIT)) ) ))

(DEFUN EXIT (LAMBDA ()
 EXIT ))

(PUTQQ T EXECUTIVE (LAMBDA ( )    %FUNCTION TRACE-
   OPTION%
  (TRACE (QUERY$ "FUNCTION NAME(S)" )) ))

(PUTQQ U EXECUTIVE (LAMBDA ( )    %FUNCTION UNTRACE-
   OPTION%
  (UNTRACE (QUERY$ "FUNCTION NAME(S)" )) ))
(PUTQQ R EXECUTIVE (LAMBDA (      %READ FILE OPTION%
```

```
    NAM$ ECHO)
  (LOOP
    (SETQ NAM$ (QUERY$ "FILE NAME" ))
    ((NULL NAM$))
    (SETQ NAM$ (CAR NAM$))
    ((RDS NAM$ (QUOTE LIB) *DRIVE*))
    (TERPRI)
    (PRINT "FILE NOT FOUND") )
  ((NULL (NAM$))
  (LOOP
    (EVAL (READ))
    ((NULL RDS)) ) ))

(PUTQQ W EXECUTIVE (LAMBDA ()     %WRITE FILE OPTION%
  (W-EXEC) ))
(DEFUN W-EXEC (LAMBDA (
    (NAM$ ECHO)
  (SETQ NAM$ (QUERY$ "FILE NAME"))
  ((NULL NAM$))
  (SETQ NAM$ (CAR NAM$))
  (WRS NAM$ (QUOTE LIB) *DRIVE*)
(PRIN2 (LIST (QUOTE PUTD) (QUOTE DEFUN) (LIST
  (QUOTE QUOTE) (GETD DEFUN) )))
(TERPRI)
(PRIN2 (LIST (QUOTE DEFUN) (QUOTE SETQQ) (GETD SETQQ)))
(TERPRI)
(PRIN2 (LIST (QUOTE DEFUN) (QUOTE PUTQQ) (GETD PUTQQ)))
(TERPRI)
(PRIN2 (LIST (QUOTE DEFUN) (QUOTE FLAGQQ)(GETD FLAGQQ))
(TERPRI 3)
(PRT-TXT (PUT-TO-TXT NAM$ (QUOTE FUNCTIONS)))
(TERPRI)
(PRT-TXT (PUT-TO-TXT NAM$ (QUOTE VARIABLES)))
(TERPRI)
(MAPC(GET NAM$(QUOTE FUNCTIONS))(OUOTE(LAMBDA(ATM$)
  (TERPRI)
  ((TRACED ATM$)
    (UNTRACE (LIST ATM$))
```

```
    (PRT-TXT (DEF-TO-TXT (LIST ATM$)))
    (TRACE (LIST ATM$)) )
  (PRT-TXT (DEF-TO-TXT (LIST ATM$))) )))
(TERPRI)
(MAPC (GET  NAM$  (QUOTE  VARIABLES)) (QUOTE(LAMBDA-
    (ATM$)
  (TERPRI)
  ( ((EQ ATM$ (EVAL ATM$)))
    (PRT-TXT (SET-TO-TXT (LIST ATM$))) )
  (MAPC (CDR ATM$) (QUOTE (LAMBDA (EXP$)
    (TERPRI)
    ((ATOM EXP$)
      (PRIN2  (LIST (QUOTE FLAGQQ) ATM$ EXP$))
      (TERPRI) )
    (TRT-TXT (PUT-TO-TXT ATM$ (CAR EXP$))) ))) )))
  (TERPRI)
  (PRINT "(RDS)" )
  (WRS) ))

(PUTQQ D EXECUTIVE (LAMBDA (    %CHANGE DRIVE OPTION%
   CHAR$)
  (LOOP
    (SETQ CHAR$ (QUERY$ "DRIVE LETTER"))
    ((NULL CHAR$))
    (SETQ CHAR$ (CAR CHAR$))
    ((EQ (LENGTH CHAR$) 1)) )
  ((NULL CHAR$))
  (SETQ *DRIVE* CHAR$) ))

(PUTQQ X EXECUTIVE (LAMBDA ( )   %EXIT  TO  OPERATING-
   SYSTEM%
  (SYSTEM) ))


********** TEXT EDITING FUNCTIONS **********

(DEFUN EDIT-TXT (LAMBDA (*TEXT*
   *PRE-TEXT*  *POST-TEXT*  *ROW*  *COL*  *INSERT*
     *STRING* CHAR$ READCH )
```

```
  (SETQ *PRE-TEXT*)
  (SETQ *POST-TEXT* (SPLIT$ *TEXT* PAG-LEN$))
  (LOOP
    (SETQ *ROW* *TEXT*)
    (SETQ *COL* (CADR *ROW*))
    (DISP-TXT *TEXT* *ROW* *COL*)
    (LOOP
      (SETQ CHAR$ (READCH))
      ((EQ (EVAL CHAR$)(QUOTE CTRL-K))) %EXIT EDIT CHAR%
      ( ((FLAGP CHAR$ (QUOTE DEL-CHAR))
          (DEL-CHAR CHAR$) )
        ((OR (FLAGP CHAR$ (QUOTE PRT-CHAR)) (NUMBERP-
         CHAR$))
          (PRT-CHAR CHAR$) )
        ((NULL (GETD (EVAL CHAR$))))
        (APPLY (EVAL CHAR$) (LIST *INSERT*)) ) )
    (CTRL-F)
    (TERPRI (LENGTH *ROW*))
    (SETQ CHAR$ (MENU$ (QUOTE (
      (E EVALUATE TEXT)
      (A ABANDON TEXT)
      (C CONTINUE EDIT) ))))
    ((EQ CHAR$ (QUOTE A)))
    ((AND
        (EQ CHAR$ (QUOTE E))
        (EVAL-TEXT (CONS NIL (APPEND
          (REVERSE *PRE-TEXT*)
          (APPEND (CDR *TEXT*) *POST-TEXT*) ))) )) ))

(DEFUN PRT-CHAR (LAMBDA (CHAR$    %PRINTABLE CHARS%
   TOKEN)
  ( ((LESSP (ADD1 (SPACES)) LIN-LEN$))
    (CTRL-N) (CTRL-J) )
  (PRIN1 (COND
    ((GET CHAR$ (QUOTE ALIAS)))
    (CHAR$) ))
  ((NULL (CDR *COL*))
    (INS-PRT CHAR$) )
  ((NOT *INSERT*)
```

```
    (SETQ TOKEN (DEL-CHR))
    (INS-PRT CHAR$) )
  (INS-PRT CHAR$)
  ((LESSP (PRT-RST-LIN *ROW* *COL* 0) LIN-LEN$))
  (CTRL-N) ))

(MAPC (QUOTE (
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 a b c d e f g h i j k l m n o p q r s t u v w x y z
 ! # $ ' * & + - / : ; < = > ? @ \ ^ _ ` { | } ~ "%" """"))
    (QUOTE (LAMBDA (CHAR$) (FLAG CHARS (QUOTE PRT-
     CHAR)))) )

(DEFUN DEL-CHAR (LAMBDA (CHARS    %DELIMITER CHAR$%
   TOKEN)
  ( ((LESSR (ADD1 (SPACES)) LIN-LEN$))
    (CTRL-N) (CTRL-J) )
  (PRIN1 CHAR$)
  ((NULL (CDR *COL*))
    (INS-DEL CHAR$) )
  (NOT *INSERT*)
    (SETQ TOKEN (DEL-CHR))
    (INS-DEL CHAR$) )
  (INS-DEL CHAR$)
  ((LESSP (PRT-RST-LIN *ROW* *COL* 0) LIN-LEN$))
  (CTRL-N) ))

(MAPC (QUOTE (" " "," "(" ")" "." "(" ")" ))
  (QUOTE (LAMBDA (CHAR$) (FLAG CHAR$ (QUOTE-
                                        DEL-CHAR) ))) )

(DEFUN CTRL-P (LAMBDA (      %ESCAPE CHAR%
   CHAR$)
  (SETQ CHAR$ (READCH))
  (PRT-CHAR CHAR$) ))

(PUTQQ "(" ALIAS {)
(PUTQQ ")" ALIAS })
(PUTQQ " " ALIAS —)
```

```
(DEFUN CTRL-V (LAMBDA ()     %TOGGLE *INSERT* %
  (SETQ *INSERT* (NOT *INSERT*)) ))

(DEFUN CTRL-D (LAMBDA (      %ADVANCE CHAR%
   CHAR$ TOKEN)
  (SETQ TOKEN (NXT-RHT-TOK))
  ((NULL TOKEN)              %IF AT END OF CURRENT LINE%
    ((NULL (CDDR *ROW*)))    %AND NOT END OF TEXT, %
    (CTRL-M) )               %THEN NEW LINE.%
  ((ATOM TOKEN)              %IF TOKEN IS A DELIMITER%
    (PRIN1 TOKEN)            %PRINT TOKEN AND%
    (MOV-RHT-TOK) )          %ADVANCE COL.%
  (SETQ TOKEN (UNPACK$ TOKEN))
  (SETQ CHAR$ (CONS (POP TOKEN)))
  (PRT-TOK CHAR$)
  ((ATOM (CAR *COL*))
    ((NULL TOKEN)
      (POP *COL*) )
    (RPLACD *COL* (CONS CHAR$ (CDR *COL*)))
    (POP *COL*)
    (RPLACA (CDR *COL*) TOKEN) )
  (RPLACA *COL* (APPEND (CAR *COL*) CHAR$))
  ((NULL TOKEN)
    (RPLACD *COL* (CDDR *COL*)))
  (RPLACA (CDR *COL*) TOKEN) ))
(DEFUN CTRL-F (LAMBDA (      %ADVANCE TOKEN%
   TOKEN)
  (SETQ TOKEN (MOV-RHT-TOK))
  ((NULL TOKEN)
    (LOOP
     NULL (CDDR *ROW*)))
      (CTRL-J)
      ((CDR *COL*)) ) )
  (PRT-TOK TOKEN)
  (LOOP
    ((NOT (SPACE$ (NXT-RHT-TOK))))
    (MOV-RHT-TOK)
    (SPACES 1) ) ))
```

```
(DEFUN CTRL-B (LAMBDA (      %MOVE TO END LINE%
  TOKEN)
 (LOOP
  (SETQ TOKEN (MOV-RHT-TOK))
  ((NULL TOKEN))
  (PRT-TOK TOKEN) ) ))

(DEFUN CTRL-S (LAMBDA (      %RETREAT CHAR%
  CHAR$ TOKEN)
 (SETQ TOKEN (CAR *COL*))
 ((ATOM TOKEN)
  ((MOV-LFT-TOK)
   (BCK-SPACE$ 1) )
  ((NULL (CAR *ROW*)))
  (CTRL-E)
  (CTRL-B) )
 (BCK-SPACE$ 1)
 (SETQ TOKEN (UNPACK$ TOKEN))
 ((NULL (CDR TOKEN))
  ((OR (NULL (CDR *COL*)) (ATOM (CADR *COL*)))
   (SETQ *COL* (RDC$ (CADR *ROW*) *COL*)) )
  (RPLACA (CDR *COL*) (PACK$ (APPEND TOKEN (CADR-
   *COL*))))
  (SETQ *COL* (RDC$ (CADR *ROW*) *COL*))
  (RPLACD *COL* (CDDR *COL*)) )
 (SETQ CHAR$ (CONS (CHOP$ TOKEN)))
 (RPLACA *COL* TOKEN)
 ((OR (NULL (CDR *COL*)) (ATOM (CADR *COL*)))
  (RPLACD *COL* (CONS CHAR$ (CDR *COL*))) )
 (RPLACA (CDR *COL*) (NCONC CHAR$ (CADR *COL*))) ))

(MOVD (QUOTE CTRL-S) (QUOTE CTRL-H))   %RETREAT CHAR%

(DEFUN CTRL-A (LAMBDA (      %RETREAT TOKEN%
   TOKEN)
 (LOOP
  ((NUMBERP (CAR *COL*))
   (SETQ TOKEN NIL) )
  (SETQ TOKEN (MOV-LFT-TOK))
```

```
    ((NOT (SPACE$ TOKEN)))
    (PRIN1 BACK) )
  ((NULL TOKEN)
    ((LOOP
      ((NULL (CAR *ROW*)))
      (CTRL-E)
      ((CDR *COL*)
        (CTRL-B) ) ) )
  (BCK-SPACE$ (TOK-PRT-LEN TOKEN)) ))

(DEFUN CTRL-Q (LAMBDA ( )        %MOVE TO BEGIN LINE%
  (PRIN1 CR)
  (SETQ *COL* (CADR *ROW*))
  (SPACES (CAR *COL*)) ))

(DEFUN CTRL-E (LAMBDA (       %MOVE UP LINE%
   NUM)
  ((NULL (CAR *ROW*))
    ((NULL *PRE-TEXT*))
    (CTRL-W)
    (CTRL-E))
  (SETQ NUM (SPACES))
  (UP-LIN$ 1)
  (SPL-TOK *COL*)
  (SETQ *COL* (CAR *ROW*))
  (SETQ *ROW* (RDC$ *TEXT* *ROW*))
  (SPACES (CAR *COL*))
 LOOP
    ((NULL (CDR *COL*)))
    ((NOT (LESSP (SPACES) NUM)))
    (CTRL-F) ) ))

(DEFUN CTRL-X (LAMBDA (        %MOVE DOWN LINE%
   NUM)
  ((NULL (CDDR *ROW*))
    ((NULL *POST-TEXT*))
    (CTRL-Z)
    (CTRL-X) )
  (SETQ NUM (SPACES))
```

```
  (CTRL-J)
  (LOOP
    ((NULL (CDR *COL)))
    ((NOT (LESSP (SPACES) NUM)))
    (CTRL-F) ) ))

(DEFUN CTRL-W (LAMBDA ( )        %SCROLL DOWN%
  ((NULL *PRE-TEXT*))
  ( ((NULL (CDDR *ROW*))
       (CTRL-E) ) )
  (RPLACD *TEXT* (CONS(POP *PRE-TEXT*)(CDR *TEXT*)))
  (PUSH (CHOP$ *TEXT*) *POST-TEXT*)
  ( ((NULL (CAR *ROW*))
    (POP *ROW*) ) )
  (DISP-TXT *TEXT* *ROW* *COL) ))

(DEFUN CTRL-R (LAMBDA (                    %SCROLL DOWN
      NUM)                                   PAGE%
  ((NULL *PRE-TEXT*))
  (SETQ NUM (DIFFERENCE PAG-LEN$ 6))
  (LOOP
    (PUSH (CHOP$ *TEXT*) *POST-TEXT*)
    (RPLACD *TEXT*(CONS(POP *PRE-TEXT*)(CDR*TEXT*)))
    ((NULL *PRE-TEXT*))
    (SETQ NUM (SUB1 NUM))
    ((ZEROP NUM)))
  (SETQ *ROW* *TEXT*)
  (SETQ *COL* (CADR*ROW*))
  (DISP-TXT *TEXT* *ROW* *COL*) ))

(DEFUN CTRL-Z (LAMBDA( )              %SCROLL UP%
  ((NULL *POST-TEXT*))
  ( ((NULL (CAR *ROW*) )
       (CTRL-X) ) )
  ( ((EQ *ROW* (CDR*TEXT*))
      (SETQ *ROW* *TEXT*) ) )
  (HOMES)
  (TERPRI (SUB1 PAG-LEN$))
  (PRT-ROW (CAR *POST-TEXT*))
```

```
  (TERPRI)
  (PUSH (CADR *TEXT*) *PRE-TEXT*)
  (RPLACD *TEXT* (CDDR *TEXT*) )
  (NCONC *TEXT* (CONS (POP *POST-TEXT*)))
  (MOVE-CUR *TEXT* *ROW* *COL*) ))

(DEFUN CTRL-C (LAMBDA (                    %SCROLL UP PAGE%
      NUM)
 ((NULL *POST-TEXT*))
  (SETQ NUM (DIFFERENCE PAG-LEN$ 6))
  (LOOP
    (PUSH (CADR *TEXT*) *PRE-TEXT*)
    (RPLACD *TEXT* (CDDR *TEXT*))
    (NCONC *TEXT* (CONS (POP *POST-TEXT*)))
   ((NULL *POST-TEXT*))
    (SETQ NUM (SUB1 NUM))
    ((ZEROP NUM)) )
  (SETQ *ROW* *TEXT*)
  (SETQ *COL* (CADR*RCW*))
  (DISP-TXT *TEXT* *ROW* *COL*) ))

(DEFUN CTRL-N (LAMBDA (                    %INSERT NEW LINE%
      NUM)
  ( ((NULL (CDR *COL*)))
    ((ATOM (CADR *COL*) ) )
    (RPLACA (CDR* COL*) (PACK$ (CADR *COL*))) )
  (RPLACD (CDR *ROW*) (CONS
    (CONS 0 (CDR *COL*)) (CDDR *ROW*)))
  (SPACES (ROW-PRT-LEN (CADDR*ROW*)))
  (RPLACD *COL* NIL)
  (RPLACA (CADDR *ROW*) (IND-NXT (CADR*ROW*)))
  ( ((GREATERP (LENGTH *TEXT*) PAG-LEN$)
    (PUSH (CHOP$ *TEXT*) *POST-TEXT*)
    (SETQ NUM (ROW-PRT-LEN (CAR *POST-TEXT*))))
    (SETQ NUM 0) )
  (ROLL-DWN-ROW (CDR *ROW*) NUM)
  (MOVE-CUR *TEXT* *ROW* *COL*) ))

(DEFUN CTRL-J (LAMBDA ( )                  %NEWLINE TAB%
```

```
  ((NULL (CDDR *ROW*))
    ((NULL *POST-TEXT*)
    (CTRL-M)
    (RPLACA *COL* (IND-NXT (CAR *ROW*)))
    (SPACES (CAR *COL*)))
  (CTRL-Z)
  (CTRL-J))
(SPL-TOK *COL*)
(POP *ROW*)
(TERPRI)
(SETQ *COL* (CADR *ROW*))
((NULL (CDR *COL*))
  (RPLACA *COL* (IND-NXT (CAR *ROW*)))
  (SPACES (CAR *COL*)) )
(SPACES (CAR *COL*)) ))

(DEFUN CTRL-M (LAMBDA (*INSERT*)          %CARRIAGE
                                            RETURN%

 ((EVAL *INSERT*)
    (CTRL-N)
    (CTRL-M) )
 ((NULL (CDDR *ROW*))
  ((NULL *POST-TEXT*)
    (SPL-TOK *COL*)
    (POP *ROW*)
    (TERPRI)
    (SETQ *COL* (CONS 0))
    (RPLACD *ROW* (CONS *COL*))
   ((GREATERP (LENGTH *TEXT*) PAG-LEN$)
      (PUSH (CADR *TEXT*) *PRE-TEXT*)
      (RPLACD *TEXT* (CDDR *TEXT*)) ))
  (CTRL-Z
  (CTRL-M) )
(SPL-TOK *COL*)
(TERPRI)
(POP *ROW*)
(SETQ *COL* (CADR *ROW*))
(LOOP
 ((ZEROP (CAR *COL*)))
```

```
  (RPLACA *COL* (SUB1 (CAR *COL*)))
  (RPLACD *COL* (CONS (QUOTE" ") (CDR *COL*))) ) ))

(DEFUN CTRL-G (LAMBDA (                          %DELETE CHAR%
   TOKEN)
 (SETQ TOKEN (DEL-CHR))
  ((NULL TOKEN))
  (PRT-RST-LIN *ROW* *COL* 1) ))

 (DEFUN CTRL-T (LAMBDA (                         %DELETE TOKEN%
    TOKEN NUM)
     (SETQ TOKEN (DEL-TOK))
    ((NOT TOKEN))
     (SETQ NUM (TOK-PRT-LEN TOKEN))
     ( ((ATOM (CAR *COL*))
        ((SPACE$ (NXT-RHT-TOK))
         (SETQ NUM (ADD1 NUM))
         (DEL-CHR) ) ) )
  (PRT-RST-LIN *ROW* *COL* NUM) ))

 (DEFUN CTRL-Y (LAMBDA ()                        %DELETE LINE%
  ((NULL (CDDR *ROW*))
    (PRIN1 CR)
    (SPACES (ROW-PRT-LEN (CADR *ROW*)))
    (PRIN1 CR)
    (SETQ *COL* (CONS 0))
    (RPLACA (CDR *ROW*) *COL*) )
((AND (NULL *POST-TEXT*) *PRE-TEXT*)
    (RPLACD *TEXT* (CONS (POP *PRE-TEXT*) (CDR *TEXT*)))
    (RPLACD *ROW* (CDDR *ROW*))
    (SETQ *COL* (CADR *ROW*))
    (DISP-TXT *TEXT* *ROW* *COL*) )
( ((NULL *POST-TEXT*))
  (NCONC *TEXT* (CONS (POP *POST-TEXT*))) )
(ROLL-UP-ROW (CDR *ROW*))
(RPLACD *ROW* (CDDR *ROW*))
(SETQ *COL* (CADR *ROW*))
(SPACES (CAR *COL*)) ))
```

```
(DEFUN CTRL-O (LAMBDA (                          %FIND TOKEN%
      READCH)
  (HOME$)
  (SPACES (ROW-PRT-LEN (CADR *TEXT*)))
  (PRIN1 CR)
  (PRIN1 "FIND NAME? " )
  (SETQ READCH T)
  (SETQ *STRING* (CAR (RD-LIN$)))
  (HOME$)
  (SPACES LIN-LEN$)
  (HOME$)
  (PRT-ROW (CADR *TEXT*))
  ((NULL *STRINC*)
   (MOVE-CUR *TEXT* *ROW* *COL*))
  (CTRL-L) ))

(DEFUN CTRL-L (LAMBDA ( )                        %SEARCH AGAIN%
  ((NULL *STRING*))
  (MOV-RHT-TOK)
  (SRCH-TXT *STRING* *ROW* *COL*)
  (MOVE-CUR *TEXT* *ROW* *COL*) ))

(DEFUN SRCH-TXT (LAMBDA (TOKEN ROW COL)
   (LOOP
    ((SETQ COL (SRCH-ROW COL))
      (SETQ *COL* COL)
      (SETQ *ROW* ROW))
     (POP ROW)
     ((NULL (CDR ROW))
      (SETQ ROW *TEXT*)
      (LOOP
        (SETQ COL (CADR ROW))
        ((SETQ COL (SRCH-ROW COL))
          (SETQ *COL* COL)
          (SETQ *ROW* ROW))
        (POP ROW) ) )
      (SETQ COL (CADR ROW)) ) ) )

(DEFUN SRCH-ROW (LAMBDA (COL)
```

```
  (LOOP
    ((NULL (CDR COL)) NIL)
    ((EQ TOKEN (CAADR COL)) COL)
    (POP COL)
    ((EQ COL*COL*) COL) ) ))


********* CURSOR CONTROL PRIMITIVES *********%

(DEFUN INS-PRT (LAMBDA (CHAR$)    %INSERT NORMAL CHAR
  ((ATOM (CAR *COL*))
    (RPLACD *COL* (CONS (CONS CHAR$) (CDR *COL*)))
    (POP *COL*))
  (RPLACA*COL*(NCONC (UNPACK$ (CAR *COL*))(CONS CHA-
   R$))) ))

(DEFUN INS-DEL (LAMBDA (CHAR$)      %INSERT DELIMITER%
  ((AND (NUMBERP (CAR *COL*)) (SPACE$ CHAR$) )
    (RPLACA *COL* (ADD1 (CAR *COL*)))
  (RPLACD *COL* (CONS CHAR$ (CDR *COL*)))
  (SPL-TOK *COL*)
  (POP *COL*) ))

(DEFUN DEL-CHR (LAMBDA (                    %DELETE CHAR%
   CHAR$ TOKEN)
  (SETQ TOKEN (NXT-RHT-TOK))
((ATOM TOKEN)
   (DEL-TOK) )
(SETQ TOKEN (UNPACK$ TOKEN))
(SETQ CHAR$ (CONS (POP TOKEN)))
((NULL TOKEN)
   (RPLACD *COL* (CDDR*COL*))
    CHAR$)
(RPLACA (CDR *COL*) TOKEN)
 CHAR$))

(DEFUN DEL-TOK (LAMBDA (                    %DELETE TOKEN%
  TOKEN)
```

```
  (SETQ TOKEN (NXT-RHT-TOK))
  ((NULL TOKEN) NIL)
  (RPLACD *COL* (CDDR *COL*))
  ((OR (ATOM (CAR *COL*)) (ATOM (CADR *COL*)))
   TOKEN)
  (RPLACA *COL* (UNPACK$ (CAR *COL*)))
  (RPLACA (CDR *COL*) (UNPACKS (CADR *COL*)))
   TOKEN))
(DEFUN MOV-RHT-TOK (LAMBDA (        %MOVE RIGHT TOKEN%
   (TOKEN))
  (SETQ TOKEN (NXT-RHT-TOK))
  ((NULL TOKEN) NIL)
  ((AND (NUMBERP (CAR *COL*)) (SPACE$ TOKEN))
    (RPLACA *COL* (ADD1 (CAR *COL*)))
    (RPLACD *COL* (CDDR *COL*))
     TOKEN)
  ((SPL-TOK *COL*)
    (POP *COL*)
   TOKEN)
 TOEN) )

(DEFUN NXT-RHT-TOK (LAMBDA ()       %NEXT RIGHT TOKEN%
   ((NULL (CDR *COL*)) NIL)
   (CADR *COL*) ))

(DEFUN MOV-LFT-TOK (LAMBDA (        %MOVE LEFT TOKEN%
   TOKEN)
  (SETQ TOKEN (CAR *COL*))
  ((NUMBERP TOKEN)
    ((ZEROP TOKEN)
     NIL)
    (RPLACA *COL* (SUB1 TOKEN))
    (RPLACD *COL (CONS " " (CDR *COL*)))
    " " )
    (SPL-TOK *COL*)
    (SETQ *COL* (RDC$ (CADR*ROW*) *COL*))
   TOKEN))

(DEFUN SPL-TOK (LAMBDA (COL             %SPLICE TOKEN%
```

```
    LST$)
    ((ATOM (CAR COL))
     ((NOT (NUMBERP (CAR COL))))
     (POP LST$)
     (LOOP
       ((OR (NULL LST$)  (NOT (SPACE$ (POP LST$)))))
       (RPLACA COL (ADD1 (CAR COL)))
       (RPLACD COL LST$) ) )
 ((OR (NULL (CDR COL))  (ATOM (CADR COL) ) )
  (RPLACA COL (PACK$ (CAR COL)))
   T )
    (RPLACA COL (PACK$ (APPEND (CAR COL)  (CADR  COL ))))
    (RPLACD COL (CDDR COL))
   NIL ))

  (DEFUN IND-NXT (LAMBDA (COL
   NUM)
  (SETQ NUM (POP COL) )
  (LOOP
   ((NULL COL)
    ((MINUSP NUM) 0)
     NUM)
  ( ((EQ (CAR COL)  " (" )
     (SETQ NUM (PLUS NUM 2)) )
  ((EQ (CAR COL ") " )
     (SETQ NUM (DIFFERENCE NUM 2)) ) )
  (POP COL) ) ))

(RDS MUSTAR NUM)


%* * * * * * * * * * * *  TEXT PRINTER  * * * * * * * * * * * * * * * %

(DEFUN ROLL-UP-ROW (LAMBDA (LINE
    NUM LENGTH)
  (SETQ LENGTH (SUB1 (LENGTH LINE)))
  (SETQ NUM (REPL-ROW (CAR LINE) 0))
  (POP LINE)
  (LOOP
```

```
    ((NULL LINE))
    (SETQ NUM (REPL-ROW (POP LINE) NUM))
    (TERPRI) )
   (SPACES NUM)
   (UP-LIN$ LENGTH) ))

(DEFUN ROLL-DWN-ROW (LAMBDA (ROW NUM)
    (TERPRI (SUB1 (LENGTH ROW)))
    (SETQ ROW (REVERSE ROW))
    (LOOP
      (SETQ NUM (REPL-ROW (POP ROW) NUM) )
      ((NULL ROW) )
      (UP-LIN$) )
    (PRIN1 CR) ))

(DEFUN REPL-ROW (LAMBDA (COL LENGTH
     NUM)
    (PRIN1 CR)
    (PRT-ROW COL)
    (SETQ NUM (SPACES) )
    (SPACES (DIFFERENCE LENGTH NUM) )
     NUM) )

(DEFUN DISP-TXT (LAMBDA (TEXT ROW COL) %DISPLAY TEXT%
  (TERPRI 47)                          %CLEAR SCREEN%
  (HOME$)                              %MOVE TO HOME$%
  (PRT-TXT TEXT)                       %PRINT TEXT%
  (MOVE-CUR TEXT ROW COL) ))           %RESTORE CURSOR%


(DEFUN PRT-TXT (LAMBDA (TEXT)          %PRINT TEXT%
  (MAPC (CDR TEXT)  (QUOTE (LAMBDA (COL)
    (PRT-ROW COL)
    (TERPRI) ))) ))
(DEFUN PRT-ROW (LAMBDA (COL)  %PRINTA ROW OF TEXT%
    (SPACES (CAR COL) )
    (MAPC (CDR COL)  (QUOTE PRT-TOK)) ))

(DEFUN MOVE-CUR (LAMBDA (TEXT ROW COL)
  (HOME$)
```

```
  (LOOP                                          %MOVE TO ROW%
    ((EQ ROW TEXT) )
    (TERPRI)
    (POP TEXT) )
  (SETQ ROW (CADR ROW) )
  (SPACES (CAR ROW) )
  (LOOP                                          %MOVE TO COL%
  ((EQ ROW COL) )
  (POP ROW)
  (PRT-TOK (CAR ROW)) ) ))

DEFUN PRT-RST-LIN (LAMBDA (ROW COL NUM
   NUM0 )
(SETQ NUM0 (SPACES) )
(MAPC (CDR COL) (QUOTE PRT-TOK) )
(SETQ NUM (DIFFERENCE (SPACES NUM) NUM0) )
((LESSP NUM NUM0)
    (SETQ NUM0 (SPACES))
    (BCK-SPACE$ NUM)
    (ADD1 NUM0) )
(SETQ NUM0 (SPACES) )
(PRIN1 CR)
(SETQ ROW (CADR ROW) )
(SPACES (CAR ROW) )
(LOOP
  ((EQ ROW COL)
      (ADD1 NUM0) )
  (POP ROW)
   (PRT-TOK (CAR ROW)) ) ))

(DEFUN PRT-TOK (LAMBDA (TOKEN)
 ((ATOM TOKEN)
   (PRIN1 TOKEN) )
 ((NULL (CDR TOKEN) )
   ((NULL (CDR TOKEN) )
   ((NULL WRS)
     (PRIN1 (COND
       ((GET (CAR TOKEN) (QUOTE ALIAS)))
       ((CAR TOKEN)) )) )
```

```
    (PRIN2 (CAR TOKEN)) )
   ((NULL WRS)
    (MAPC TOKEN (QUOTE PRIN1)) )
   (MAPC TOKEN (QUOTE PRIN2)) ))

(DEFUN ROW-PRT-LEN (LAMBDA (COL
   NUM)
  (SETQ NUM (POP COL) )
  (LOOP
    ((NULL COL) NUM)
    (SETQ NUM (PLUS NUM (TOK-PRT-LEN (POP COL)))) ) ))

(DEFUN TOK-PRT-LEN (LAMBDA (TOKEN)
  ((ATOM TOKEN) 1)
  ((NULL (CDR TOKEN))
    (LENGTH (CAR TOKEN)) )
  (LENGTH TOKEN) ))
```

*** * * * * * * S-EXPRESSIONT TOTEXT TRANSLATOR * * * * * * %**

```
(DEFUN DEF TO-TXT (LAMBDA (VAR    %TRANSLATE DEFINITI
   TXT)
  (SETQ TXT (CONS) )
  (LOOP
    (NCONC TXT (CDR
      (EXP-TO-TXT (GETD (CAR VAR) )  (LIST (QUOTE DEFUN)
      (POP VAR))) ))
    ((NULL VAR) TXT)
    (NCONC TXT (CONS (CONS 0 ))) ) ))
(DEFUN SET-TO-TXT (LAMBDA (VAR    %TRANSLATE VALUE%
   TXT)
  (SETQ TXT (CONS))
  (LOOP
    (NCONC TXT (CDR
      (EXP-TO-TXT (EVAL (CAR  VAR)) (LIST (QUOTE  SETQQ)
      (POP VAR))) ))
    ((NULL VAR) TXT)
```

```
    (NCONC TXT (CONS (CONS 0))) ) ))

(DEFUN PUT-TO-TXT (LAMBDA(VAR ATM) %TRANSLATE PROP-
    ERTY%
  (EXP-TO-TXT (GET VAR ATM)  (LIST (QUOTE PUTQQ) VAR A-
    TM)) ))

(DEFUN EXP-TO-TXT (LAMBDA (EXP LST$
    *TEXT*  *LINE*  *LENGTH* TAB INDENT)
  (SETQ TAB 0)
(SETQ INDENT 1)
  ( ((LESSP LIN-LEN$ 60))
    (SETQ INDENT 2) )
  (NEW-LIN TAB)
  (PUSH " (" *LINE*)
  (LOOP
    ((NULL LST$) )
    (EXP-TO-LIN (POP LST$) )
    (PUSH " " *LINE*) )
  (TSK-TO-TXT EXP TAB)
  (PUSH " (" *LINE*)
  (NEW-LIN TAB)
  (REVERSE *TEXT*) ))

(DEFUN TSK-TO-TXT (LAMBDA(TSK TAB)%TRANSLATE TASK1%
  (SETQ TAB (PLUS TAB INDENT) )
  ((ATOM TSK)
    (EXP-TO-LIN TSK) )
  ((ATOM (CAR TSK) )
     ((MEMBER (CAR TSK) (QUOTE (LAMBDA NLAMBDA) ) )
          (PUSH "(" *LINE*)
          (EXP-TO-LIN (POP TSK) )
          (PUSH " " *LINE*)
          (EXP-TO-LIN (POP TSK) )
          (BDY-TO-TXT TSK TAB)
          (PUSH " (" *LINE*) )
((MEMBER (CAR TSK)  (QUOTE (LOOP COND PROGN PROG1 AND-
    OR)))
  (PUSH "(" *LINE*)
  (EXP-TO-LIN (POP TSK)
```

```
    (BDY-TO-TXT TSK TAB)
    (PUSH ") " *LINE*) )
 (EXP-TO-LIN TSK) )
((ATOM (CAAR TSK))
 (PUSH " (" *LINE*)
 (TSK-TO-TXT (POP TSK) TAB)
 ((AND TSK (ATOM (CAR TSK) ) (NULL (CDR TSK) ) )
    (PUSH " " *LINE*)
    (EXP-TO-LIN (CAR TSK) )
    (PUSH ") " *LINE*) )
  (BDY-TO-TXT TSK TAB)
  (PUSH ") " *LINE*) )
 (PUSH " (" *LINE*)
 (PUSH " " *LINE*)
 (BDY-TO-TXT1 TSK TAB)
 (PUSH ") " *LINE*) ))

(DEFUN BDY-TO-TXT (LAMBDA (BDY TAB  %TRANSLATE BODY%
  ((NULL BDY) )
  (NEW-LIN TAB)
  (BDY-TO-TXT1 BDY TAB) ))

(DEFUN BDY-TO-TXT1 (LAMBDA (BDY TAB)
  (LOOP
    (TSK-TO-TXT (POP BDY) TAB)
    ((NULL BDY)
      (PUSH " " *LINE*) )
    (NEW-LIN TAB) ) ))

(DEFUN EXP-TO-LIN (LAMBDA (EXP)
((ATOM EXP)
  (PUSH (CONS EXP) *LINE*) )
    (PUSH " (" *LINE*)
    (LOOP
      (EXP-TO-LIN (POP EXP) )
      ((ATOM EXP)
        ((NULL EXP) )
        (PUSH " " *LINE*)
        (PUSH "." *LINE*)
```

```
      (PUSH " " *LINE*)
      (PUSH (CONS EXP) *LINE*) )
   (PUSH " " *LINE*) )
  (PUSH " (" *LINE*) ))

(DEFUN NEW-LIN (LAMBDA (TAB)
  (SETQ *LENGTH *TAB)
  (SETQ *LINE* (REVERSE *LINE*) )
  (SETQ TAB (PLUS (CAR *LINE*) (TIMES 2 INDENT)))
  (CUT-LIN *LINE* (CAR *LINE*) )
  (SETQ *LINE* (LIST *LENGTH*)) ))

(DEFUN CUT-LIN (LAMBDA (LINE LENGTH)
  (PUSH *LINE* *TEXT*)
  ((NULL *LINE*))
  (SETQ LENGTH (PLUS LENGTH (UNIT-LEN (CDR LINE))))
  (LOOP
   ((NOT (LESSP LENGTH LIN-LEN$))
    (SETQ *LINE* (CONS TAB (CDDR LINE)))
    (RPLACD LINE NIL)
    (CUT-LIN *LINE* (CAR *LINE*)) )
  (SETQ LINE (NXT-UNIT (CDR LINE)))
  ((NULL LINE) )
  (SETQ LENGTH (PLUS LENGTH (ADD1 (UNIT-LEN (CDDR LIN-
   E))))) ) ))

(DEFUN UNIT-LEN (LAMBDA (LINE
   PRIN1)
  ((OR (NULL LINE) (SPACE$ (CAR LINE))) 0)
  ((ATOM (CAR LINE) )
   (ADD1 (UNIT-LEM (CDR LINE))) )
  ( ((NULL WRS)
    (SETQ PRIN1 T) ) )
  (PLUS (LENGTH (CAAR LINE)) (UNIT-LEN (CDR LINE))) ))

(DEFUN NXT-UNIT (LAMBDA (COL)
   (LOOP
     ((NULL (CDR COL)) NIL)
     ((SPACE$ (CADR COL) ) COL)
```

```
      (SETQ COL (CDR COL)) ) ))


*******TEXT TO S-EXPRESSION TRANSLATOR********%

(DEFUN EVAL-TEXT (LAMBDA (ROW
   COL TOKEN LST$ ERROR)
  (LOOP
    (SETQ TOKEN (NXT-TOK) )
    ((NULL TOKEN)
      (MAPC (REVERSE LST$)  (QUOTE EVAL)) T )
    ( ((EQ TOKEN "] " ))
      (PUSH (TXT-TO-SEX TOKEN) LST$) )
    ((EVAL ERROR) NIL) ) ))

(DEFUN TXT-TO-SEX (LAMBDA (TOKEN
   LST$)
   ((NULL TOKEN)
    (SETQ ERROR T) )
   ((ATOM TOKEN)
    ((EQ TOKEN " (" )
      (TXT-TO-LST (NXT-TOK)) )
    ( (EQ TOKEN "[" )
      (SETQ LST$ (TXT-TO-LST (NXT-TOK)))
      ((EVAL ERROR))
      (SETQ TOKEN (NXT-TOK) )
      ((OR (EQ TOKEN ") " )  (EQ TOKEN "]" ))
       LST$)
    (SETQ ERROR T) )
  (SETQ ERROR T) )
(CAR TOKEN) ))

(DEFUN TXT-TO-LST (LAMBDA (TOKEN
   LST$ TOK)
(LOOP
    ((EQ TOKEN ") " )
      (REVERSE LST$) )
    ((EQ TOKEN "]" )
      (PUSH "]" COL)
```

```
    (REVERSE LST$) )
   (PUSH (TXT-TO-SEX TOKEN) LST$)
   ((EVAL ERROR))
   (SETQ TOKEN (NXT-TOK))
   ((EQ TOKEN "." )
    (SETQ TOKEN (TXT-TO-SEX (NXT-TOK)))
    ((EVAL ERROR))
    (SETQ TOK (NXT-TOK))
    ((EQ TOK ")" )
      (REVERSE LST$ TOKEN) )
    ((EQ TOK "]" )
      (PUSH ")" COL)
      (REVERSE LST$ TOKEN) )
     (SETQ ERROR T) ) ) ))

(DEFUN NXT-TOK (LAMBDA (
      TOKEN)
   (LOOP
    ((NULL COL)
    (POP ROW)
    ((NULL ROW) NIL)
    (SETQ COL (CDAR ROW))
    (NXT-TOK) )
   (SETQ TOKEN (POP COL))
   ((NOT (OR (SPACE$ TOKEN) (EQ TOKEN "," )))
     TOKEN) ) ))


********* TRACE DEBUGGING PACKAGE ********* %

(DEFUN TRACED (LAMBDA (FUN$)
  (EQ (CAR (CADDR (GETD FUN$))) (QUOTE EVTRACE)) ))

(DEFUN TRACE (LAMBDA (LST$)
  (SETQ INDENT 0)
  (MAPC LST$ (QUOTE (LAMBDA (FUN$ EXP$ FUN#)
   ((TRACED FUN$))
   (SETQ EXP$ (GETD FUN$))
   (SETQ FUN# (PACK (LIST FUN$ #)))
```

```
    (MOVD FUN$ FUN#)
    ((MEMBER (CAR EXP$) (QUOTE (LAMBDA NLAMBDA)))
     (PUTD FUN$ (LIST (CAR EXP$) (CADR EXP$)
      (LIST EVTRACE FUN$ (CADR EXP$) FUN#) )) )
    (PRTSEN$ (CONS FUN$ (QUOTE (UNDEFINED FUNCTION)))1)-
    ))) ))

(DEFUN UNTRACE (LAMBDA (LST$)
  (MAPC LST$ (QUOTE (LAMBDA (FUN$ FUN#)
    (SETQ FUN# (PACK (LIST FUN$ #)))
    ((GETD FUN#)
      (MOVD FUN#FUN$)
      (MOVD NIL FUN#))))) ))

(DEFUN EVTRACE (NLAMBDA (FUN$ LST#FUN#)
  (PRTARGS FUN$ LST#)
  (PRTRSLT FUN$ (APPLY FUN# (MAKARGS LST#))) ))

(DEFUN PRTARGS (LAMBDA (FUN$ LST#)
  (SPACES INDENT)
  (SETQ INDENT (PLUS INDENT 1))
  (PRIN1 FUN$)
  (PRIN1 "(" )
  ((NULL LST#)
    (PRINT ")" ) )
  (LOOP
    ((ATOM LST#)
     (SETQ LSU# (EVAL LST#))
     (LOOP
         (PRIN1 (POP LST#))
         ((ATOM LST#))
         (PRIN1 ", " ) ) )
  (PRIN1 (EVAL (POP LST#)))
  ((NULL LST#) )
  (PRIN1 "," ) )
(PRINT ")" ) ))

(DEFUN PRTRSLT (LAMBDA (FUN$ EXP$)
  (SETQ INDENT (DIFFERENCE INDENY 1))
```

```
  (SPACES INDENT)
  (PRIN1 FUNS)
  (PRIN1 "=" )
 -(PRINT EXP$)
   EXP$))

(DEFUN MAKARGS (LAMBDA (LST#)
   ((NULL LST#) NIL)
   ((ATOM LST#)
     (EVAL LST#) )
   (CONS (EVAL (POP LST#)) (MAKARGS LST#)) ))
```

%* * * * * * CONTROL CHARACTER ASSIGNMENTS * * * * * *%

```
(SETQ CTRL-A (ASCII 1))
(SETQ CTRL-B (ASCII 2))
(SETQ CTRL-C (ASCII 3))
(SETQ CTRL-D (ASCII 4))
(SETQ CTRL-E (ASCII 5))
(SETQ CTRL-F (ASCII 6))
(SETQ CTRL-G (ASCII 7))
(SETQ CTRL-H (ASCII 8))
(SETQ CTRL-I (ASCII 9))
(SETQ CTRL-J (ASCII10))
(SETQ CTRL-K (ASCII11))
(SETQ CTRL-L (ASCII12))
(SETQ CTRL-M (ASCII 13))
(SETQ CTRL-N (ASCII 14))
(SETQ CTRL-O (ASCII 15))
(SETQ CTRL-P (ASCII 16))
(SETQ CTRL-Q (ASCII 17))
(SETQ CTRL-R (ASCII 18))
(SETQ CTRL-S (ASCII 19))
(SETQ CTRL-T (ASCII 20))
(SETQ CTRL-U (ASCII 21))
(SETQ CTRL-V (ASCII 22))
(SETQ CTRL-W (ASCII 23))
(SETQ CTRL-X (ASCII 24))
```

```
(SETQ CTRL-Y (ASCII 25))
(SETQ CTRL-Z (ASCII26))
(SETQ CTRL-^ (ASCII 30)

(SET CTRL-A (QUOTE CTRL-A))
(SET CTRL-B (QUOTE CTRL-B))
(SET CTRL-C (QUOTE CTRL-C))
(SET CTRL-D (QUOTE CTRL-D))
(SET CTRL-E (QUOTE CTRL-E))
(SET CTRL-F (QUOTE CTRL-F))
(SET CTRL-G (QUOTE CTRL-G))
(SET CTRL-H (QUOTE CTRL-H))
(SET CTRL-I (QUOTE  CTRL-I))
(SET CTRL-J (QUOTE CTRL-J))
(SET CTRL-K (QUOTE CTRL-K))
(SET CTRL-L (QUOTE CTRL-L))
(SEF CTRL-M (QUOTE CTRL-M))
(SET CTRL-N (QUOTE CTRL-N))
(SET CTRL-O (QUOTE CTRL-O))
(SET CTRL-P (QUOTE CTRL-P))
(SET CTRL-Q (QUOTE CTRL-Q))
(SET CTRL-R (QUOTE CTRL-R))
(SET CTRL-S (QUOTE CTRL-S))
(SET CTRL-T (QUOTE CTRL-T))
(SET CTRL-U (QUOTE CTRL-U))
(SET CTRL-V (QUOTE CTRL-V))
(SET CTRL-W (QUOTE CTRL-W))
(SET CTRL-X (QUOTE CTRL-X))
(SET CTRL-Y (QUOTE CTRL-Y))
(SET CTRL-Z (QUOTE CTRL-Z))
(SET CTRL-^ (QUOTE CTRL-^))

(SETQ HOME CTRL-^)          %HOME CURSOR CHARACTER%
(SETQ BACK CTRL-H)          %BACKSPACE CHARACTER%
(SETQ UPLINE CTRL-K)        %UPLINE CHARACTER%
(SETQ CR CTRL-M)            %CARRIAGE RETURN CHAR%

(ROS)
```

注，以上程序中，因排版关系而转行者，行末有连字符“-”。

附录D LISP和AI参考书目

1. Allen, J.R., Anatomy of LISP,
McGraw-Hill Book Company, New York, NY, 1978.

2. Berkeley, E.C., and Bobrow, D.G., (eds),
The Proqramming Language LISP: Its Operation and Applications,
The M.I.T.Press, Cambridge, MA, 1964.

3. BYTE: The Small Systems Journal, LISP Issue, Vol 4, No 8,
BYTE Publications Inc., Peterborough, NH, August 1979.

4. Charniak, E., etal, Artificicial Intelligence Programming,
Lawrence Erlbaum Associates, 365 Broadway, Hillsdale, NJ, 1980.

5. Henderson, P., Function Programming: Application and Implee-
mentation, Prentice-Hall, Englewood Cliffs, NJ, 1980.

6. Friedman, D.P., The Little LISPer,
Science Research Associates Inc.,
London, England, 1974.

7. McCarthy, J., Recursive Functions of Symbolic Expressions
and Their Computation by Machine,
Comm.ACM, Pages 184-195, April 1960.

8. McCarthy, J., et al., LISP 1.5 Programmer's Manual,
The M.I.T.Press, Cambridge, MA, 1963.

9. Maurer, W.D., A Programmer's Introduction to LISP,
American Elsevier, New York, NY, 1973.

10. Siklossy, L., Let's Talk LISP,
Prentice-Hall, Englewood Cliffs, NU, 1976.

11. Weissman, C., LISP 1.5 Primer,
Dickenson Publishing Co., Belmont, CA, 1968.

12. Winston, P.H., Artificial Intelligence,
Addison-Wesley publishing Co., Readinq, MA, 1977.

13. Winston, P.H.&Hom, B.K.P., LISP,
Addison-Wesley Publishing Co., Reading, MA, 1981.

编译校对：刘运基　宋柔
责任编辑：谢雪峰、王宏成、慕雪琴、吴雪莹、王晓东