微型计算机软件资料汇编

第四册

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（第四册）

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：8403

内 容 提 要

《微型计算机软件资料汇编》第四册包括《PL/I-80 应用指南和语言手册》和《SuperSoft C 编译程序使用手册》两篇。

PL/I-80是以ANS PL/I标准化委员会X3J1定义的子集G语言为基础，在CP/M 和MP/M 操作系统下使用的应用程序设计软件包。本书对PL/I-80语言作了详细描述，并提供了一些程序实例，说明该语言在应用程序设计中的应用。

SuperSoft C 编译程序基本上接受全功能C语言，但有少量的例外。它是自编译、优化、两遍扫描的编译程序，可在相对小的计算机系统上进行系统级和应用级程序设计。本书对SuperSoft C 编译程序的使用及其标准库函数作了较详细描述，对于它与Unix C 之间的区别也作了说明。

本书可作为用户使用这些语言的参考手册，也可作为应用软件人员学习程序设计语言的参考资料。

微型计算机软件资料汇编

第　四　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

PL/I-80应用指南和语言手册

SuperSoft C 编译程序使用手册

PL/I-80 应用指南和语言手册

第一章 PL/I-80应用指南

第一节 绪 言

PL/I-80是为在Digital Research CP/M和多道程序设计MP/M操作系统下进行应用程序设计的一个完整的软件程序包。它是以ANS PL/I标准化委员会X3J1定义的新的子集G语言为基准的。这个子集包含了整个PL/I的必要的应用程序设计结构，而去掉了不常用或冗余形式，使之更便于进行程序设计，同时简化了编译工作。

PL/I-80象所有程序设计语言（及大多数自然语言）一样，通过研究实例很容易学会。本章主要目的是介绍编译、连接和程序执行的技巧以及语言的一些有用功能，并且有详细的实例程序，说明输入/输出处理、科学计算、商业应用及表处理。

学习PL/I-80的最好方法是读有关教科书，研究实例程序，必要时查阅参考手册，在弄懂一个样本程序的操作后，可以修改这个程序以提高它的操作能力并增加你的语言经验。

PL/I-80系统软盘不包括CP/M操作系统，所以首先应制备PL/I-80程序的拷贝作日常使用，并在前两个系统磁道上生成CP/M系统。将新建立的软盘装到驱动器A，引导CP/M，打入DIR命令可以找到下列几类文件：

COM CP/M命令文件或组合程序（PLI.COM是其中之一）
DAT 缺省数据文件类型
IRL 索引浮动码（PLILIB.IRL是库文件）
OVL PL/I-80编译程序复盖（PLI0.PLI1和PLI2）
PLI PL/I-80源程序（即OPTIMIST.PLI）
PRL 页浮动目标码（用于MP/M分区）
PRN 打印机磁盘文件（磁盘上的程序清单）
REL 浮动目标码（如用户开发的程序）

含有可打印字符的文件只有“PLI”源程序及“PRN”打印机列表文件。PL/I-80系统盘里包含了不同的程序，包括本手册中的实例以及其他更复杂的程序。开始，先试着运行一下已经编译和连接到PL/I-80运行库的程序。命令格式为：

```
OPTIMIST
```

这时，OPTIMIST程序被装入并响应：

```
What's up?
```

可以打下面这个句子来回答：

None of these programs make Sense.（须用一句号结束输入，接着打回车）。从OPTIMIST得到响应后，还可以再打入一些句子，然后打control-c停止OPTIMIST。

OPTIMIST是一个PL/I程序，它以源码方式存于PL/I-80系统软盘中。显示这个程序可以打入命令：TYPE OPTIMIST.PLI

作为一个例子，按下面的步骤对一个OPTIMIST程序进行完整的编译和测试。注意，虽然OPTIMIST程序可在任意存储容量下运行，但PL/I-80编译程序运行至少需要48K的

CP/M系统。PLI.COM 和覆盖文件一定要在缺省磁盘上，否则当启动编译程序时将出现错误信息“NOFILE:PLI0.OVL”。可打入

PLI OPTIMIST

来编译OPTIMIST程序。

编译程序将用三个步骤处理程序，称为“扫描”。其信息标记是：

NO ERROR(S) IN PASS 1
NO ERROR(S) IN PASS 2
END COMPILATION

如果检查目录表，便会找到OPTIMIST.REL文件。它包含了由PL/I-80编译程序为OPTIMIST程序生成的浮动机器码。如果需要，可用列表任选重新编译，这样在编译时可以查看程序。要达到上述目的，只要打入：

PLI OPTIMIST SL

编译程序和以前一样进行，但这次在最后一遍扫描中要产生程序清单。

编译产生的浮动机器码不是直接可执行的，所以必须将REL文件与PL/I-80运行子程序库连接，这可通过打入“LINK OPTIMIST”来实现。LINK-80程序产生一个OPTIMIST.COM文件，它替换原来软盘上的同名文件，新的OPTIMIST程序可以用同样的方式操作。

第二节 PL/I-80系统操作

通常，PL/I-80编译程序读入在CP/M或MP/M下用标准的编辑程序（ED）建立的程序文件。程序先由PL/I-80编译程序处理，再用LINK-80连接，然后进行测试。例如一个简单的工资单程序的编译测试，如图2-1所示。编译程序通过进行前两遍扫描并列出含有错误的每一行，左边带行号，还有一个简单的错误信息，并在错误行下面位置上有一个“?”。无论在什么情况下，都可在控制台上打入一个回车来终止编译过程。这个功能对于错误诊断数过多或想在继续执行前作一改正是很有用的。程序的行号列在左边，后面跟着字符 a～z，表示每一行嵌套层。主程序层为“a”，每个嵌套的BEGIN由前进一个字符表示，而每个嵌套的PRODURE层由前进两个字符表示，其后是每行相对的机器码地址，它是用四个十六进制数表示的。这个地址对于确定为每个语句生成的机器码的数量和为各程序行生成的相对机器码地址是很有用的。源语句打印在相对机器码后面。

在启动编译程序命令行时给出的SL参数称为“编译程序开关”，用于启动列表任选。编译程序开关表在下面列出。每一命令字符跟在命令行给出的符号“$”后，在美元符（$）后面最多可有七个命令字符。没有指定参数时，缺省值表示编译不产生清单，所有错误信息送到控制台。

B　内部子例程跟踪，给出被PL/I程序调用的库函数

D　磁盘文件打印。将列表文件送到磁盘，文件型为PRN

I　交叉列出源码和机器码。将编译产生的机器码以伪汇编语言形式表示

L　列出源程序。产生带有行号和机器码地址单元（由I开关自动设置）的源程序清单

```
PL/I-80 V1.0, COMPILATION OF:WAGE
L : List Source Program
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0, COMPILATION OF:WAGE
   1 a 0000 payroll:
   2 a 0006      procedure options(main);
   3 a 0006
   4 c 0006      declare
   5 c 0006         name (100) character(30) varying,
   6 c 0006         hours (100) fixed decimal(5,1),
   7 c 0006         wage (100) fixed decimal(5,2),
   8 c 0006         done bit(1),
   9 c 0006         next fixed;
  10 c 0006
  11 c 0006      declare
  12 c 0006         (grosspay, withhold, netpay) fixed decimal(7,2);
  13 c 0006
  14 c 0006      /* read initial values */
  15 c 0006      done='0'b;
  16 c 000B         do next = 1 to 100 while(^done);
  17 c 0023         put list('Type ''employee'', hours, wage:');
  18 c 003A         get list(name(next),hours(next),wage(next));
  19 c 00A0         done = (name(next) = 'END');
  20 c 00C8         end;
  21 c 00C8
  22 c 00C8      /* all names have been read, write the report */
  23 c 00C8      put list('Adjust Paper to Top of Page, Type return');
  24 c 00DF      get skip(2);
  25 c 00F0
  26 c 00F0         do next = 1 to 100 while(name(next)^= 'END');
  27 c 011F         grosspay = hours(next) * wage(next);
  28 c 0157         withhold= grosspay * .15;
  29 c 0177         netpay = grosspay-withhold;
  30 c 0192         put skip(2) list('$',netpay,'for',name(next));
  31 c 01EA         end;
  32 c 01EA
  33 a 01EA      end payroll;
CODE  SIZE=01ED
DATA  AREA=0ED2
```

图2-1 工资程序清单

N　显示嵌套层。作第一遍扫描跟踪，表示出DO、PROC和BEGIN语句与和其相应的END语句的匹配关系

P　页方式打印。每60行插入换页符，并将列表送打印机

S　显示符号表。表示出程序变量名，以及它们的指派、缺省和扩展属性

PL/I-80允许每个过程分别编译，每次编译产生一个“REL”文件。只有一个过程可包含“options(main)”,这个过程就是模块的主程序,而所有其他子程序要有通常的PL/I过程标题。

PLILIB. IRL文件含有子程序，可以由PL/I80程序调出。如上一节所示，要使浮动机器码与PL/I运行库子程序相连接，可打入：

```
link wage
```

生成一个组合程序。如果是在MP/M系统下操作，命令Link wage〔op〕也生成一个组合程序，但在这种情况下，机器码是页浮动格式，在一个MP/M分区中运行。在第一种情况下，LINK-80产生一个可执行的“wage.com”文件，供在CP/M下执行或在MP/M下的一个绝对段中执行。在第二种情况下,LINK-80产生一个名为“wage. prl”的文件。除机器码文件外，LINK-80还产生一个符号表文件,称为“wage.sym”,它可在SID或ZSID下装入用于纠错。

图2-2列出了一个简单工资程序的LINK-80输出。按照惯例,从PL/I-80库中提出的子程序前面要加“？”符号，以避免与用户定义的符号名相冲突。用两个“/”括起来的是EXTERNAL变量，后面有‘*’号的是未定义的符号。注意，LINK-80采用Microsoft连接编辑格式，这是为了与其他语言处理程序相兼容。但这种格式将外部名的长度限制在6个字符，所以尽管内部变量名可以长达31个字符，但要保证所有外部定义名在前6个位置上是唯一的。

```
A>link wage
LINK V0.4
PAYROL  0100  /SYSIN/ 1F19 /SYSPRI/ 1F3E
ABSOLUTE   0000
CODE SIZE   1DCF (0100-1ECE)
DATA SIZE   10C5 (1F94-3058)
COMMON SIZE 00C5 (1ECF-1F93)
USE FACTOR  4E
```

图 2-2　工资程序的简单连接编辑

采用缺省时，LINK-80不从库中列出“？”符号，如果想列出这些符号，只须打入：

```
link wage〔q〕
```

要执行程序，可打入COM或PRL文件名，如图2-3所示。程序执行并提示控制台输入，如后面I/O节所述，从控制台输入是带有CP/M及MP/M的整行编辑的自由字段。在程序完成回到操作台命令级时，显示信息：

```
End of Execution
```

如在PL/I程序里不显式截获各种运行错误,这些错误会使程序执行终止,并显示下列信息：

error-condition (code),file-option, auxiliary-message

Traceback: aaaa bbbb cccc dddd # eeee ffff gggg hhhh

其中“error-condition”是下列标准PL/I条件之一：

ERROR FIXED OVERFLOW OVERFLOW UNDERFLOW ZERODIVIDE END OF FILE UNDEFINED FILE

“code”是一个错误子代码，它指明错误的起因。当错误产生于I/O操作时，打印出“file-option”，并有如下形式：internal=external

其中“internal”是引用了含有出错文件的内部程序名，而“external”是与该文件有关的外部设备或文件名。当上述信息不足以指明错误时，就打印出“auxiliary-message”，最后“traceback”段列出多达8个内部堆栈元素，以便帮助标识产生错误的程序语句。如果堆栈中超出8个元素，则把最上面的4个元素置于“#”的左边，最下面的4个元素置于右边。在上面所示的形式中，元素aaaa对应于堆栈顶部，而hhhh对应于堆栈底部。主程序错误语句由hhhh值确定，除非错误语句已用一个字符或十进制临时值填充栈底。

图2-4表示一个工资程序的执行，它给出了一个诊断形式的例子。在这个例子中，第一行控制台输入已正确打入，但第二行用文件结束标志（control-z）结束控制台输入，对标

```
A>wage
Type 'employee',hours,wage:'Sidney Abercrombie', 35,6.70
Type 'employee',hours,wage:'Yolanda Carlsbad',  42,7.10
Type 'employee',hours,wage:'Ebenizer Eggbert'   30,5.50
Type 'employee',hours,wage:'Hortense Gravelpaugh',40,6.50
Type 'employee',hours,wage:'Franklin Fairweather',10,15.00
Type 'employee',hours,wage:'Tilly Krabnatz',32,4.10
Type 'employee',hours,wage:'Ricardo Millywatz', 45,7.20
Type 'employee',hours,wage:'Adolpho Quagmire', 60,4.30
Type 'employee',hours,wage:'Pratney Willowander',43,5.50
Type 'employee',hours,wage:'Manny Yuppgander',  40,3.25
Type 'employee',hours,wage:'END',0,0
Adjust Paper to Top of Page,Type return
$    199.33 for Sidney Abercrombie
$    253.47 for Yolanda Carlsbad
$    140.25 for Ebenizer Eggbert
$    221.00 for Hortense Gravelpaugh
$    127.50 for Franklin Fairweather
$    111.52 for Tilly Krabnatz
$    275.40 for Ricardo Millywatz
$    219.30 for Adolpho Quagmire
$    201.03 for Pratney Willowander
$    110.50 for Manny Yuppgander
End of Execution
```

图 2-3 工资程序的执行

准的控制台输入文件SYSIN，产生了END OF FILE条件。在本例中，连接到SYSIN的外部设备是操作员控制台，由CON表示。

```
A>wage
Type 'employee',hours,wage:'Sally Switzwigg',23,3.10
Type 'employee',hours,wage:^Z
END OF FILE(1),File:SYSIN=CON
Traceback: 0930 08DB 0146 3300 # 1F07 049A 8082 0146
End of Execution
```

图 2-4　工资程序的错误追踪

Traceback表示出最低的堆栈单元为0146（十六进制），对应于错误的主程序语句。回过来再看图2-2，PAYROL程序地址在左上角表示为0100，它是CP/M下通常用的起始单元；差值0146－0100＝0046是产生错误的相对位置；图 2-1 清单表明地址0046落在18行旁边列出的（003A～00A0）代码地址之间，因此错误就发生在这行里。

第三节　PL/I-80程序设计特点

在研究PL/I-80程序设计的细节之前，有必要讨论一下它的程序设计特点。PL/I是一个“自由格式”语言，也就是说编程序时可不考虑列的位置及特殊的行格式，每行长度可达120字符（由回车结束），且可顺次逻辑地连接到下一行。编译程序从第一行读源程序到最后一行，而不管行的界限。这种自由的表示法，要求程序员遵守习惯的编程规则，以使别的程序员可容易地阅读及了解这些程序。专业程序员都知道，仅有一个产生正确输出的子程序是不够的，程序还必须在格式上一致，并可分成容易理解的逻辑段。一个程序是从它的结构及其功能评价的。

下面给出的规则说明了一组编程约定，本手册中全部实例都将使用这些约定。

首先，要注意编写PL/I程序既可用大写也可用小写。在内部，PL/I 编译程序将所有字符串引号以外的字符变成大写。通常我们更喜欢整个程序用小写，因为这样可减少程序密度及增加可读性。其次，整个PL/I用缩入空格来开始各种说明和语句。为了简化，PL/I编译程序将制表标记（Control-I字符）扩展到每四列一个，然而我们知道，CP/M实用程序如ED，将标记扩展到 8 的倍数列的位置上，所以在编辑和显示操作时，行变得较宽些。但要注意，当扩展的行长度超过120列时，将发出TRUNC（截断）错误信息。外部分程序级的程序语句起始于第一列位置。由一个DO，BEGIN或PROCEDURE组开始的每个后继分程序级开始于一个新的缩入空格层，可以是四个空格也可以是一个制表标记。同一组内的语句在相同的缩入层给出，过程名和标号单独在一行中。一个IF语句后面应直接跟着条件和关键字THEN，下一条语句缩进去放在下一行。当IF语句有相关的ELSE时，它起始于与IF相同的级，跟在ELSE后面的语句缩进去置于下一行。最后，说明语句格式是将关键字DECLARE单独放在一行，接着在下一行缩进去，写出要说明的元素。应该避免复杂属性的因子分解，因为它减少了程序的可读性。为使段落清楚，可加入空行（即，只含有回车的行），空行常常用于划分逻辑上独立的程序段。许多较长的 PL/I 关键字有缩写形式

(如DCL等于DECLARE)。因为缩写形式的用法不一致会造成程序不易懂，所以在一个程序设计方案里或用全称，或用缩写形式，而不能两种一起用。

一般说来，可将大程序分成一些逻辑组或“模块”，每个模块完成一个特定的基本功能。这些模块用PL/I的子例程表示，这些子例程，或者是局部的或者是外部定义的。局部子例程成为同一个主程序或子程序的一部分，而外部子例程是单独编译并用LINK-80连接的。局部定义的子例程放在程序末尾以便使程序开始部分只包含说明和调用局部子例程的顶层语

```
PL/I-80 V1.0,COMPILATION OF:TEST
L : List Source Program
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0,COMPILATION OF:TEST
   1 a 0000          test:
   2 a 0006               proc options(main);
   3 c 0006               dcl
   4 c 0006                  (a,b,c) float binary;
   5 c 0006                  put list ('Type Three Numbers:');
   6 c 001D                  get list (a,b,c);
   7 c 0056                  put list ('The Largest Value is',
   8 c 007B                      max3(a,b,c));
   9 c 007B
  10 c 007B               max3:
  11 c 007B                  proc(x,y,z) returns (float binary);
  12 e 007B                  dcl
  13 e 008B                  (x,y,z,max) float binary;
  14 e 008B                  /* compute the largest of x,y,and z */
  15 e 008B                  if x > y then
  16 e 0099                  if x > z then
  17 e 00A7                  max=x;
  18 e 00B5                  else
  19 e 00B5                  max=z;
  20 e 00C3                  else
  21 e 00C3                  if y > z then
  22 e 00D1                  max=y;
  23 e 00DF                  else
  24 e 00DF                  max=z;
  25 e 00EA                  return(max);
  26 c 00F3                  end max3;
  27 a 00F3               end test;
CODE  SIZE=00F6
DATA  AREA=0044
```

图 3-1　典型规则说明

句。通常规定，顶层语句或局部定义的子例程长度不能超过两页。在初学PL/I-80编程时，可能常常用到带有多个局部定义子例程的主程序，本书中这种例子很多。当应用程序较大时，最好的方法是将程序划分成独立的模块，以便对每个单独的程序段逐段进行编译和连接，这样可减少总的程序编制时间。

在程序中加注释可使程序一目了然，但不要在源文件中到处都加注释，因为这样有损于整体结构。另外，要注意一致性：最好是把注释放在子程序或逻辑语句组前面，在分析程序时，会发现这些注释及格式安排得很好的程序可提供了解程序操作所需要的信息。图3-1中的程序说明了本节中讲的一些规则。

第四节　PL/I-80输入/输出规则

本节我们对PL/I-80 I/O系统作详细讨论，为后面出现的例子提供必要的基础。如果你认为这部分讲得太细，可先看GET和PUT语句，其中讲到了最简单的I/O功能，再看一下后面的实例程序，然后再回来重读这些细节，可以加深理解。

PL/I-80提供一个独立于设备的I/O系统，此系统是PL/I-80程序与CP/M和MP/M文件系统的接口，接口的参数是由OPEN语句及通过GET，PUT，READ，WRITE语句的缺省方式提供的。

4.1 OPEN语句

OPEN语句是任选的，如果没有显式的OPEN，文件用GET、PUT、READ或WRITE存取时，自动执行打开功能。如果不想用缺省文件属性，就必须在文件存取之前显式地打开该文件。OPEN语句的格式为：

```
OPEN
    FILE (f)
    STREAM  RECORD
    PRINT
    INPUT  OUTPUT  UPDATE
    SEQUENTIAL  DIRECT
    KEYED
    ENV (B(i))  ENV (F(i))  ENV (F(i),B(j))
    LINESIZE(i)
    PAGESIZE(i)
    TITLE(c)
```

其属性可以用任意的次序列出。f表示文件常数或变量值，而且必须在OPEN语句里命名。其他所有属性是任选的，并且取下面给出的缺省值。值i及j表示定点二进制(FIXED BINARY)表达式，而c表示一个字符表达式。在同一行中示出的属性是矛盾的，如不包括这些属性，在有多个属性的行中的第一个属性就成为缺省值。最后四个属性取如下缺省值：

```
ENV (B(128))
LINESIZE(80)
PAGESIZE (60)
```

TITLE ('f.DAT')

STREAM文件含有变长ASCII数据，而RECORD文件一般包含纯二进制数据，一个ASCII数据文件的行由插入的回车换行序列来定义。注意，当文件用ED程序建立时，每个回车后包含换行。用PL/I-80建立的文件可以含有一系列换行而前面没有回车。在这种情况下，当遇到换行时就断定是行的末尾。PRINT属性只适用于STREAM文件，而通常假定最终数据显示在行式打印机上。

在OPEN语句出现时要求INPUT文件已经存在，而OUTPUT文件如果存在则先删除，并在OPEN语句中建立。UPDATA文件不能有STREAM属性，并且可被读和写。如果UPDATA文件不存在，则建立该文件。

SEQUENTIAL文件是从头至尾读写的，而DIRECT文件则可随机存取。DIRECT文件自动接受RECORD属性。

KEYED文件可通过使用键值来存取，并自动接受RECORD属性。在PL/I-80中，KEYED文件是简单的定长记录文件，这里的键是被存取记录的相对记录位置（根据定长记录大小计算）。

ENV（环境）属性定义定长或变长文件及内部缓冲容量。ENV(B(i))使得I/O系统设置缓冲存储器为i字节，其中i内部补足到128字节的倍数。在这种情况下，假设文件是变长记录的，因而不能有KEYED属性。

ENV(F(i))定义一个记录长度为i字节的定长记录文件，它在内部补足到128字节的倍数。为了遵守PL/I标准，也需要将定长记录文件定义为KEYED。这时，缺省的缓冲区容量是i字节补足到128字节的倍数。

ENV(F(i), B(j))定义一个含有i字节定长记录及j字节缓冲区容量的文件（i,j按上述方法补足）。注意，可以指定定长记录的长度大于缓冲区容量，另外，要包含KEYED属性以便与标准PL/I保持兼容性。

如果指定KEYED属性，则记录长度须用ENV(f(i))或ENV(F(i), B(j))形式给出。而且，PL/I-80要求所有的UPDATE文件用DIRECT属性说明，以便可以查找单个记录。在用缺省值后，要加入下面的属性：

```
          SEQUENTIAL  ---→  RECORD
             UPDATE*  ---→  RECORD
              KEYED** ---→  RECORD
DIRECT ---→ KEYED**   ---→  RECORD
              PRINT   ---→  STREAM
                      ---→  OUTPUT
```

* 在PL/I-80中，UPDATE也必须是DIRECT

** 在PL/I-80中，KEYED必须有ENV(F(i))或ENV(F(i), B(j))

这就是说，将RECORD属性加到SEQUENTIAL、UPDATE和KEYED文件中，而将STREAM属性加到PRINT文件中。PRINT文件也将被自动地给予OUTPUT属性。KEYED属性加到DIRECT文件中（依次加RECORD属性）。

OPEN语句本身不能含有不相容的属性，也不能通过缺省或蕴含的方法获得不相容的属性。

这即意味着，如果要读一个含有ASCII字符的文件，必须将它定义为STREAM文件，否则它必须是RECORD文件。一般来说，这都是将要遇到的。如果要进行随机存取，可将文件定义为DIRECT并用ENV定义记录长度。如果要读键值，则可将文件定义为KEYED，并去掉DIRECT属性。

LINESIZE任选项仅仅适用于STREAM文件，它定义了输入、输出行的最大长度。PAGESIZE任选项仅适用于STREAM OUTPUT文件，它定义了页的长度。

TITLE(c) 任选项允许一个内部文件名与一个外部设备或CP/M文件之间建立程序上的连接，若没有指明时，外部文件名取文件引用值，文件型为“DAT”。否则对字符串c求值以产生设备名：

```
$CON    系统控制台
$LST    系统列表设备
$RDR    系统阅读设备
$PUN    系统穿孔设备
```

或磁盘文件名：

```
d: x.y  Disk d, File x.y
```

其中“d:”是任选的设备名，而x和y分别表示文件名和文件类型。注意，x和y可以是$1或$2，如果指定了$1，则从命令行得到第一个缺省名并将它填入到$1位置上。同理，$2从第二个缺省名得到，并填入到它所在的位置上。文件名x不能是空白，而且x，y或d也不能含有“?”符号。物理I/O设备$CON、$RDR、$PUN和$LST只能作为STREAM文件打开，$RDR必须有INPUT属性，而$PUN和$LST必须有OUTPUT属性。

注意，当一个OPEN语句引用一个已经打开的文件时，该语句被忽略。

CLOSE语句的格式是：

```
CLOSE FILE(f);
```

其中f是文件变量或文件常数。所有打开的文件在程序结尾或执行STOP语句时自动关闭。

用STREAM属性打开的文件可通过GET或PUT语句存取，而由RECORD属性打开的文件要通过READ和WRITE存取，但后面要讲到一个例外情况。

4.2 PUT LIST语句

PUT LIST语句的格式是：

```
PUT
    FILE(f)
    SKIP  SKIP(i)
    PAGE
    LIST(d)
```

这里所有元素都是可任选的（但至少要指定一个）。PUT LIST任选项可按任何次序给出，但如指定了LIST任选项，则它必须出现在最后。在上面所示的格式中，f是一个文件变量或常数，i是一个整数表达式。LIST任选项包括一个数据表，由d指出并在后面描述。

PUT语句将数据或控制字符写入到由FILE(f)给出的文件，或所有PL/I-80程序中隐含说明的标准控制台文件SYSPRINT，如果这个文件以前没有打开，则当PUT语句执行时自动打开。SYSPRINT文件被隐式打开为：

OPEN FILE (SYSPRINT) PRINT ENV(B(128)) TITLE ('$CON');

SKIP任选项可取下面形式的一种：

SKIP　SKIP(i)

这里i是FIXED表达式。第一种形式使得一个回车换行序列插入到输出文件中，将输出文件的列位置重置为1。SKIP(i)形式将单个回车插入到输出流里，后面跟着i个换行字符。注意，SKIP(0)将列位置移到1（即在CRT显示时，光标位置定在行的左边），不换行。

PAGE任选项自动产生SKIP(0)，并将一个换页字符置入输出流中。在PUT语句里列出的PAGE和SKIP任选项的次序是无关紧要的：如果已指明，则首先执行PAGE任选，再执行SKIP任选。

在LIST任选中给出的数据表d的一般格式是：

LIST (d_1, d_2, …, d_n)

其中d_i可以是一个简单常数，也可以是标量表达式，或者是迭代组。迭代组的格式是：

(e_1, e_2, …, e_m DO 迭代)

其中e_1到e_m本身是常数、标量表达式或迭代组。表的“迭代”部分取与PL/I DO组标题相同的形式，并控制所写的每组嵌套次数。迭代组与下面表示的PL/I-80 DO组有相同作用：

```
DO 迭代;
PUT LIST(e1, e2, …, em) ;
END;
```

数据表的每个元素被求值，并根据通常的PL/I-80转换规则将它转换为串常数。如果数据项的值是一个串，而输出文件没有PRINT属性，则在串的前后放置引号，并将每个内含的单引号变成两个单引号。如果数据项是一个位串，则在输出值的末尾附加字符“b”。用这种方式写到磁盘文件上的值适于以后用GET LIST语句输入。

在将数据项转换为串值时，要对当前列位置与行尺寸比较，以保证数据项放在当前行，否则就自动发出SKIP，并将数据项写到下一逻辑行上。如果该项不是一行中的第一项，则前面写入空格以分隔每个数据项（当写入多个数据项时，这个空格包含在数据项格式里）。

下面列出PUT语句的例子，并对它们的作用作简短说明：

```
put skip;
```

在文件SYSPRINT里（通常是控制台）移到新行。

```
put list ('Type Name:');
```

将一（字符）串写到标准输出文件SYSPRINT，产生

```
Type name:
```

或

```
'Type name:
```

如果PRINT属性存在，就产生第一种形式。

```
put file(f) skip(3) page list (a,b,c);
```

将一个换页符写到由f指定的文件中，跟着一个回车和三个换行字符。而三个变量a,b,c则

转换成变长字符串并送到文件 f 中。

4.3 GET LIST语句

与PUT 语句类似，GET 语句用于读有STREAM 属性的文件。GET 语句的格式为：

```
GET
  FILE(f)
  SKIP SKIP(i)
  LIST(d)
```

其中FILE，SKIP及LIST任选遵守上面列出的PUT语句的规则。如果不含有 FILE(f) 任选，则存取标准的输入文件SYSIN，并自动执行OPEN语句：

```
OPEN FILE(SYSIN) STREAM ENV(b(128)) TITLE('$CON');
```

文件 f 必须有STREAM INPUT属性。SKIP 任选可使输入流到行的末尾，而SKIP(i)语句使得跳过后面的 i 个换行字符。在LIST任选中给出的数据项必须是标量变量或迭代组，象PUT语句给出的一样，而且必须是合法的赋值语句目标。

当通过GET语句访问控制台时，PL/I-80 I/O系统询问控制台并等待输入。在发出回车之前（在第80字符后自动发出回车），操作员可用CP/M行编辑功能打入80个字符。在这种情况下，回车返回到左边，后面跟着一个换行。这种缓冲行（包括换行)用于后继GET语句输入。

PL/I-80把由GET语句读入的外部数据取为一个字符序列，或作为由引号括住的位串或字符串。每个数据项由一个或多个空格及一个任选的逗号分开。用这种方式读入数据项须能转换成目标项的数据类型。例如，如果在GET语句中指定一个十进制数，则输入值必须包含一个合法的十进制数。

注意，如果一个数据项为空（即遇到一对单引号，中间可能插入空格),则目标数据项的值不改变。当重读及有选择地改变控制台上显示的数据时，这种PL/I 的特殊性能是非常有用的。

每个输入行末尾的回车起定界符（空格或逗号）的作用。且串常数不能超过一个行边界，事实上，当遇到一个行的末尾时就自动结束（因此，当在控制台打入串数据时仅前面的引号是必要的)。可在控制台上打入通常的CP/M文件结束字符（control-z)，但它必须是该行的第一个字符。

4.4 PUT EDIT语句

PUT EDIT语句与上面讲的PUT LIST 语句相似，只是要将数据按格式项表的描述写到输出行的特定字段。PUT EDIT语句格式是：

```
PUT
  FILE(f)
  PAGE
  SKIP SKIP(i)
  EDIT(d)(f1)
```

其中数据表指出了写入由格式表f1定义的固定字段里的数值，所写的数据项表由 d 说明，遵守与PUT LIST一样的规则。在表f1里给出一个或多个格式项，由逗号分开，并可在括号

里分组。任何格式项都可以用一个不超过254的正的常数值打头，它确定了格式项或格式项组的应用次数。数据表中的每个元素都要与格式项配对。详细说明见第二章及GET EDIT语句。格式项是：

A　写下一个字母数字字段。用转换后的字符长度作为字段宽度

A(n)　与A格式相似，只是字段宽度为n，右截断或右边填充空格

B　把位串值写到输出，字段宽度由数据项的精度决定

B(n)　与B相似，只是字段宽度由常数n给出，右截断或右边填充空格

B1　与B格式等价

B1(n)　与B1格式等价

B2　与B等价，只是数字用基数为4的表示法（0,1,2,3）

B2(n)　等价于B(n)，只是打印出4进制数字

B3　等价于B，只是输出用基数8表示法（0～7）

B3(n)　等价于B(n)，只是打印出八进制数字

B4　等价于B，只是写到字段上的是十六进制数字（0～9，A～F）

B4(n)　等价于B(n)，只是将打印出十六进制数字

COLUMN(n)　在写下一个数据项之前，移动至n列位置

E(n)　用科学表示法将数据项写到n个字符的字段中，最大精度允许在字段宽度内（n必须至少是6）

E(n.m)　将数据项写到n个字符的字段中，精度为m个小数位，用科学表示法写数，小数点左边有一个数字

F(n)　在n个数字的字段里写上数字值，有小数部分。在打印前数值被舍入

F(n.m)　在n个数字的字段里写上数字值，有m个小数位。打印前，在m+1的小数位上的值被舍入

LINE(n)　在写下一个数据项之前，移动至输出行n

PAGE　对打印文件执行跳页

R(fmt)　指明一个间接格式。在PL/I-80里，如果出现R格式，它必须是f1里唯一的格式项

SKIP　在写下一个数据项之前跳到下一个输出行

SKIP(n)　在打印下一个数据项之前在输出中跳n行

TAB(n)　在输出行中移到第n个标记位，其中制表标记定义在8列的倍数上

X(n)　在写下一个数据项之前，在输出流中插入n个空格字符

和PUT LIST语句不同，数据字段写到行的末尾时，不预先测试。如果写不下整个字段，就把能适合当前行的部分送到输出，写一个回车换行并把该字段其余部分写到后面行上。在整个数据表写完之后，格式表末尾出现的COLUMN、LINE、PAGE、SKIP、TAB及X格式项不起作用。合法的PUT EDIT语句如下所示：

```
put file(f) edit ('Next', Value)(a,f(4));
put edit ((a(i) do i=q to r)) (page, 40(3e(10, 2), x(3)));
put edit(u,v,w) (r(fmt2));
```

4.5 GET EDIT语句

GET EDIT语句与GET LIST语句相似，不同的是数据要从输入流中特定的字段读取。GET LIST语句适合控制台输入，而GET EDIT常用于读取已经由其他程序写好的数据。GET EDIT语句的格式是：

```
GET
    FILE(f)
    SKIP  SKIP(i)
    EDIT(d)(f1)
```

其中FILE和SKIP任选与GET LIST语句中的相同，EDIT 任选指出一个接收数据的目标变量表，这些数据又与GET LIST的数据描述相匹配。EDIT任选后面跟着格式表，这个表包括由下面定义的格式项序列：

A　读相邻的字母数字字段，一直读到下一个回车、换行或文件结束（不是标准PL/I)为止

A(n)　读下面n个字符作为一个字母数字字段

B(n)　读下面n个字符并解释为位串。该字段必须都是空格或者是包含右边或左边用空格填充的1和0的序列

B_1(n)　按与B相同的方式解释

B_2(n)　与B_1(n)相似，但序列必须包含从0、1、2、3中选择的数字

B_3(n)　与B_1(n)相似，但序列必须包含从0～7的数字

B_4(n)　与B_1(n)相似，但序列必须含有0～9及A～F中的数

COLUMN(n)　移动至输入中n列位置，可以要求读超过行末尾的数

E(n)　读下面n个字段作为数字值，前面和后面可带有空格。数值必须是适当形式的常数，但可以是一个有符号或无符号的整数形式，带十进制小数的数或一个用科学记数法表示的数

E(n,m)等价于E(n)，在输入时忽略标度因子m

F (n)　等价于上面的E (n)

F(n,m)等价于E(n)，只是在字段里如果没有小数点，则假设十进制小数点在最小有效数字左边m个位置上

LINE(n)　在读下一个字段之前移动至输入行n

R(fmt)指明一个间接格式。在PL/I-80中，如果出现R格式项，它必须是f_1中的唯一格式项

SKIP　在读其他数据项之前，空过当前输入行及以下的n-1行

X(n)　在读下一个字段之前，在输入流里移动n个字符

在A(n)、B(n)、B_1(n)、B_2(n)、B_3(n)、E(n)、E(n,m)、F(n)及F(n,m)格式中忽略回车换行序列；当遇到回车换行时，就读下一个输入行以得到该字段的其余字符。每个格式项可由一个重复计数打头，项组可用括号括起来用逗号分隔，前面冠以重复计数。重复计数r必须是一个正的常数值，不得超过254，并等于同样格式写r次。在处理GET EDIT语句时，PL/I-80的I/O系统记住下一个被读数据项以及输入处理中使用的下一个格式项。当读每个数据项时，选择表中下一个后继格式项，如果存在重复计数，则还要重复每项，直到读完整个数据表。表中剩余的格式项不处理；当数据表用完时，剩余的控制格式

项 (COLUMN、LINE、SKIP及X) 无效。在读所有数据项之前，如果格式项表已用完，则重新从格式表头开始。下面是一些合法的GET EDIT语句实例：

```
get edit(hours, pay) (f(4), f(5,2));
get file(employee) (hours, pay) (r(fmt1));
```

4.6 FORMAT语句

FORMAT语句允许在不同的GET和PUT EDIT语句中间共享一个格式项表。其格式为：

```
fmtname:
    FORMAT(f1)
```

其中f_1表示一个格式项表，如GET和PUT EDIT语句中所示。在GET和PUT格式表中用R格式引用该格式项表。还要注意，PL/I-80限制间接格式的使用。如果它出现在GET或PUT EDIT中，它必须是表中唯一格式项。合法的FORMAT语句如下所示：

```
fmt1: format(5(x(3),4(b1(2),x(1),f(4)),skip),skip(2));
fmt2: format(skip(3),e(10.2),f(8,3),2(x(4),b4(4)));
```

4.7 WRITE语句

WRITE语句主要用于从存储器到外部文件传送数据，而不转换成字符形式。WRITE的基本格式如下所示

```
WRITE
    FILE(f)
    FROM(x)
```

其中FILE和FROM两项必须存在，但次序可以是任意的，f是文件引用，x是标量或连接的集合数据类型。文件f自动打开为：

```
OPEN FILE(f) OUTPUT SEQUENTIAL
     TITLE('f.DAT') ENV(b(128));
```

如果已经打开，f的文件属性决不能与缺省属性相矛盾。例如允许一个KEYED文件（因为这仅蕴含固定长度记录），但不允许DIRECT。

如果文件f以前已用KEYED属性打开，则每个记录长度是固定的，并由OPEN语句中给出的ENV(F(i))任选决定。否则假如文件含有变长记录，则每个记录长度由x集合数据长度决定。若已给出一个记录长度为i的KEYED文件，则从x中写i的最大值个字节到每个记录。如果x的长度小于i，则用零来填充记录。如果f不是KEYED，则记录长度就是x的长度。

WRITE语句的另一个格式为：

```
WRITE
    FILE (f)
    FROM (x)
    KEYFROM (k)
```

其中各元素的次序可以是任意的。如果文件f还没有打开，则在文件存取前产生缺省语句：

```
OPEN FILE(f) OUTPUT DIRECT ENV(f(128));
```

注意，DIRECT属性蕴含一个KEYED文件（后者又蕴含一个RECORD文件）。文件可

以已经打开，具有OUTPUT、INPUT或UPDATE属性.但必须有DIRECT属性。前面讲到在OUTPUT情况下，如果文件存在，则被删除，并建立一个新文件。如果文件被记为INPUT，则文件必须已经存在。如果UPDATE文件存在，则被打开用于存取，如果它不存在就要建立。

当有KEYFROM任选时，通过关键字k存取每个记录，关键字k在PL/I-80里是一个固定表达式，它按照每个记录的固定长度提供被写记录的记录号。最低的键值是k＝0，而最大的键值要根据ENV(f(j)) 属性中的记录长度而定：如果j是固定记录长度，而j′是舍入后的记录长度，则最大键数乘以j′不得超过磁盘的容量。

PL/I-80支持WRITE语句的一种特殊形式，用于处理由回车换行序列限定的变长STREAM数据。如果给出了一个有STREAM OUTPUT属性的文件f及一个可变字符串v，则语句：

```
WRITE FILE (f) FROM(v);
```

将v的字符写到STREAM文件f中，包括任何嵌入的控制字符。格式：

```
WRITE FROM(v);
```

将串值v写到标准输出设备上，等价于：

```
WRITE FILE (SYSPRINT) FROM(v);
```

为了简化控制字符的处理，PL/I-80允许控制字符插到串常数中。一般地，字符“^”在一个串常数中表示后面有一个控制字符。若在串中出现两个“^”字符，则变为一个“^”字符。开头的“^”的作用是屏蔽其后字符的高4位，将它们置成“0”。这样在串常量中的“^m”序列就转换为回车。嵌入的控制字符在后面的例子中示出。

综上所述，令f是一个文件，x是一个纯量或连接的集合数据形式，v是一个可变字符串，k是一个定点二进制表达式。下面的形式表示了在不同情况下所需要的文件属性：

```
write file (f) from (x);
  SEQUENTIAL OUTPUT (可有 KEYED) RECORD
write file (f) from (x) keyfrom (k);
  DIRECT OUTPUT or DIRECT UPDATE
write file(f) from(v);
  STREAM OUTPUT
write from(v);
  STREAM OUTPUT (自动地为 SYSPRINT)
```

4.8 READ语句

READ（除一例外情况外）用于读定长或变长RECORD文件，而不从字符形式转换，即将数据从外部文件传送给存储器中的数据元素，这里假设外部文件包含二进制数据。解释被传送数据的含义是程序员的职责。基本READ语句的格式是：

```
READ
    FILE(f)
    INTO(x);
```

其中f是文件引用，x是连接集合数或纯量数据型（如结构、数组或简单变量）。FILE和INTO元素都必须存在，但次序可任意。如果文件f还没有打开，则它被自动打开为：

OPEN FILE(f) INPUT SEQUENTIAL TITLE('f.DAT')ENV (b(128));

类似WRITE语句的情况，如f已打开，则其属性决不能与上面表示的那些属性相矛盾。

如果文件已用KEYED属性打开，则假设每个记录都是定长的，长度在ENV(f(i))属性中定义。否则，假设记录长度是可变的，根据INTO元素中指定的目标数据x的长度而定。当给出一个KEYED文件时，如果记录长度i大于x的长度，则在该记录中剩余的字节都被忽略。如果记录长度小于x的长度，则仅将i个字节读入x中。如果文件不是KEYED，则读入字节数正好是x的长度。

当文件用下面的格式顺序读时，可有选择地提取一个特定文件的键：

```
READ
    FILE(f)
    INTO(x)
    KEYTO(k)
```

其中元素次序可任意指定。这个格式的作用与前面讲的READ语句相同，只是记录的键值存放在由k表示的FIXED BINARY变量中。但要注意，为了读键和数据，文件必须是KEYED。这样，对于这种形式的READ，自动打开语句是：

OPEN FILE(f) INPUT KEYED TITLE('f.DAT') ENV(f(128));

如果以前已被打开，f的属性决不能与这些缺省属性相矛盾。特别要注意，KEYED必须存在，而不允许DIRECT，因为KEYTO任选仅仅提取键，而不指定所读记录的键。这个READ语句的格式经常用于先顺序读文件，确定键值，然后直接存取、读、写或修改指定的文件中的记录。

READ语句的第三种格式指定读的记录的键：

```
READ
    FILE(f)
    INTO(x)
    KEY(k)
```

其中各元素的次序可以是任意的。如果文件还没打开，则执行下面的缺省OPEN：

OPEN FILE(f) INPUT DIRECT ENV(f(128)) TITLE('f.DAT');

如果文件已被打开，打开属性不能与这些缺省属性相矛盾，但文件可以已由UPDATE属性打开。注意，DIRECT属性也蕴含这个文件是KEYED的。

这个READ语句的作用是直接存取带有键值k的记录。因为文件是KEYED的，所以记录长度必须是固定的，如同ENV(f(i))属性中定义的一样，而且数据传送遵循上面讲的定长记录的规则。

在PL/I-80里，允许READ语句的一个特殊格式来处理变长的STREAM INPUT文件：

```
    READ FILE(f) INTO(v);
```

和

```
    READ INTO(v);
```

这里v是可变字符串，而f是一个ASCII码文件（或字符设备），记录用回车换行序列定界。如果没指明FILE(f)，则假设为标准输入文件SYSIN。如果f没打开，则由下面语句打开：

OPEN FILE(f) PRINT TITLE('f.DAT') ENV (b(128));

这个语句的作用是从文件读数据至达到 v 的最大长度或读到换行字符为止。v 的长度值置为被处理的字符数，包括控制字符，这里具体地包括回车和换行符。如果标准的 SYSIN 文件连接到控制台上，则在自动回车或换行发出之前最多可读80字符。

总之，如果 f 是一个文件，x 是一个纯量或集合数据引用，v 是可变字符串，k 是定点二进制键，则下面的形式表示了所需要的文件属性：

```
read file(f) into(x);
    SEQUENTIAL INPUT (可有 KEYED) RECORD
read file(f) into(x) keyto(k);
    SEQUENTIAL INPUT KEYED RECORD
read file(f) into(x) key(k);
    DIRECT INPUT 或 DIRECT UPDATE
read file(f) into(v);
    STREAM INPUT
read into(v);
    STREAM INPUT (自动为 SYSIN)
```

第五节　标号常数、变量和参量

在PL/I-80中有很多可检测数据结束条件的好方法。检测数据结束在第六节“异常处理”中讨论。在讨论异常处理之前，我们需要有标号处理的基本知识，因为在处理异常条件时常常涉及标号语句。也可以先看后面的一些例子然后再回过来看这部分。

现代的程序设计提倡避免标号语句和GO TO语句，由于用这样的语句编出来的程序并非结构化的，别的程序员很难看懂甚至不可读。当程序加大时，对程序设计者也是如此。PL/I-80提供一组容易理解的控制结构，这组结构采用带有REPEAT和WRITE任选项的迭代DO组的形式，这样就不必使用一般程序设计方案中的标号语句。

但必要时也可用标号语句。例如，在一个程序异常处理过程中，发生严重错误诸如错误打入输入数据行，这时，只须将控制转移到外部分程序标号，进行程序恢复即可。在这种情况下，程序流程比用标志、测试和返回语句的系统容易理解得多。

通常，当可直接使用PL/I-80程序控制结构时，应避免标号语句和GO TO语句，其仅限于在异常处理和局部计算中使用。

程序标号，象其他PL/I-80数据类型一样，分为两个主要类型：标号常数和标号变量。标号常数是那些在源程序中以文字形式出现并在程序执行过程中不变的量。标号变量没有初始值，它必须通过直接赋值语句或通过在子程序调用中的实在参数对形式参数的赋值，赋予一个标号常数值，标号常数最简单的形式如下所示：

```
lab: put skip list('Bad Input Try Again');
```

在这种情况下，“lab”有与put语句起始地址相对应的标号常数值。

一个标号常数也可以包含简单的正的或负的文字下标，对应一个 n路分支的目标。下面的程序段表示了一个具体例子。

```
get list(x);
go to q(x);
q(-1):
        y=f1(x);
        go to endq;
q(0):
        y=f2(x);
        go to endq;
q(2):;
q(3):
        y=f3(x);
endq:
put skip list('f(x)=',y);
```

上例中，在程序内部定义了四个标号常数q(－1)、q(0)、q(2)和q(3)。标号常数向量：

```
q(－1, 3) label constant
```

被自动定义以保存这些标号常数的值。必须保证程序控制不转向没有相应标号常数值的下标。如在上述情况中，若i低于－1、等于1或高于3，则转移到q(i)产生一个未定义的值。

标号常数可被局部引用，也可非局部引用，当GO TO语句的目标只在它出现的PROCEDURE或BEGIN分程序内时采用局部引用标号常数。如果出现在给标号变量赋值的右边，作为子程序的一个实在参数或作为内层嵌套的PROCEDURE或BEGIN分程序中的GO TO语句的目标，这时标号常数是非局部引用。虽然在局部引用和非局部引用的标号常数之间无功能上的区别，但非局部引用标号的处理需要增加空间和时间开销。由于这个原因，PL/I-80假设下标标号常数仅可局部引用；如果控制从当前作用域外转向一个下标标号常数，其结果是未定义的。

下面所示的非功能程序段提供了一个实例：

```
main:
    proc options(main);
    p1:
        proc;
        go to lab1;
        go to lab2;
        p2:
            proc;
            go to lab2;
            end p2;
        lab1:;
        lab2:;
        end p1;
        end main;
```

```
PL/I-80 V1.0, COMPILATION OF: GOTO
L: List Source Program
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/-80 V1.0, COMPILATION OF: GOTO
   1 a 0000 main:
   2 a 0006      proc options(main);
   3 c 0006      dcl
   4 c 000D           i fixed,
   5 c 000D           (x, y, z(3)) label;
   6 c 000D      x=labl;
   7 c 0013      y=x;
   8 c 0019
   9 c 0019      go to labl;
  10 c 001c      go to x;
  11 c 0020      go to y;
  12 c 0024
  13 c 0024      call p(lab2);
  14 c 0030
  15 c 0030           do i=1 to 3;
  16 c 0042           z(i)=c(i);
  17 c 005E           end;
  18 c 005E
  19 c 005E      i=2;
  20 c 0064      go to z(i);
  21 c 0073      go to c(i);
  22 c 0082
  23 c 0082      c(1):;
  24 c 0082      c(2):;
  25 c 0082      c(3):;
  26 c 0089
  27 c 0089      labl:;
  28 c 0090      lab2:;
  29 c 0090
  30 c 0090      p:
  31 c 0090           proc(g);
  32 e 0090           dcl
  33 e 009A                g label;
  34 e 009A           go to g;
  35 c 00A2           end p;
  36 a 00A2      end main;
CODE SIZE=00A5
DATA AREA=001A
```

图 5-1　标号变量和常量的说明

标号常数“lab1”只在过程p1中局部引用，而“lab2”是p1中的局部引用和p2中的非局部引用的目标。

标号变量的取值通过一个显式的赋值语句，或当子程序被调用时，通过隐含的赋值来实现。与其他PL/I-80变量相似，标号变量必须加以说明，并可以是下标变量。

图5-1的程序表示了不同的标号常数和变量。在这个程序中标号常数是c(1)、c(2)、c(3)、lab1和lab2，它们由程序中的文字值定义。标号变量是x、y、z和g，由第5行和第33行上的说明来定义。执行开始时，标号变量值未定义，变量x首先被赋予常数值lab1，然后标号变量y（间接地）通过第7行的赋值接收常数值lab1。因此从第9行开始的3个GO TO语句功能上等价：每次执行GO TO，就将控制转移到第27行上标号lab1后面的空语句上。

第13行上的子程序调用表示了变量赋值的一种形式。lab2是一个实在参数，它被送到过程p中，并赋值给形式标号变量g。在这个特定程序中，子程序调用将程序控制直接转向标号为“lab1”的语句。

第15行开始的DO组将变量标号向量z预置为相应的c的常数标号向量值，注意因为c是标号常数的向量，反向的赋值c(i)=z(i)在此程序中是无效的；由于预置初值，第20行开始的两个GOTO语句实际上作用相同。

第六节　异常处理

程序设计语言的一个重要功能是它们有截获运行时错误状态的能力，以便能采取程序定义的动作去处理错误。例如：出现一个异常情况，当输入数据从交互控制台读入时，操作员不小心打入了与输入变量不一致的数值。在正常情况下，由运行时系统提出“转换”异常，如果没有任何程序定义的动作，则程序执行结束。然而在一个生产环境，这种中止可能在数据录入后的几小时内发生，致使大量的工作被浪费。

因此，PL/I-80以ON　REVERT和SIGNAL语句形式组成一组用于处理异常情况的操作。ON语句定义了遇到一个异常情况时的动作，REVERT语句使ON语句无效，SIGNAL语句允许程序产生各种不同的条件。

有9种主要异常类型。它们是：ERROR、FIXEDOVERFLOW、OVERFLOW、UNDERFLOW、ZERODIVIDE、ENDFILE、UNDEFINEDFILE、KEY和ENDPAGE。前5种包括所有运算错误情况和可能在I/O建立和处理过程中以及在不同类型数据转换中产生的其他错误情况。后4种适用于正在由运行时I/O系统处理的特定文件。每种条件有一个相关的子代码，它提供了发生异常的根源方面的信息。

下面首先用ENDFILE条件作为例子，讨论ON、REVERT和SIGNAL不同的语句格式，然后详细叙述每一种条件。

6.1　ON语句

当运行时系统或SIGNAL语句产生特定条件时，ON语句用于在程序中截获这个条件。ON语句的格式是：

ON　条件　on语句体；

这里“条件”是上面所述的异常类型之一，“on语句体”是当条件产生时执行的PL/I-80语句或语句组。为了避免多义性，这个语句必须是简单语句（不是条件语句），或是BEGIN-

END组，这个组本身可包含除RETURN外任何有效的PL/I-80语句（当然，在BEGIN-END组中，过程定义内部是允许RETURN的）。

在语句的末尾或 BEGIN-END 组末尾控制从ON语句返回。换句话说，控制可以转向ON语句体外部的非局部标号。如果条件已经建立，另一个有相同条件的ON语句的执行会将原先的条件保存在堆栈中并设置新的条件。当执行REVERT语句或含有ON语句的分程序结束时，已在栈内的条件重新恢复。

6.2 REVERT语句

REVERT语句的执行使当前活动的命名条件无效，并恢复以前进入堆栈的条件。REVERT语句的格式是：

```
REVERT  条件;
```

其中“条件”是上面讨论的类型之一。对于在一个过程内设置的任何ON条件，当该过程出口时，自动执行REUERT语句。例如，下面的程序段示出了在DO组中使用一对ON和REVERT语句：

```
do while ('1'b);
on endfile(sysin)
   eofile='1'b;
   ...
revert endfile(sysin);
  end;
```

在上例中，ON语句在每次迭代开始时执行，激活ENDFILE条件。这个组的末尾的REVERT语句使得在这个分程序开始建立的条件无效。然而，ON和REVERT语句需要一些简单的运行时处理，因此将上面的程序段写成：

```
on endfile(sysin)
   eofile='1'b;
   do while ('1'b);
   ...
   end;
```

更为有效。

注意，在程序执行中，任意给定点上最多可有16个ON条件可以是活动的或入栈的。如果超过堆栈容量，则出现信息：Condition Stack Overflow，同时程序结束。

图6-1中表示的程序说明了过程出口的自动REVERT语句。过程“p”在第8行被调用。DO组索引以及标号常数“exit”一起作为实在参数，过程p内部的ON语句在每次调用时执行。如没有自动REVERT语句，当索引i达到17时，条件堆栈将发生溢出。有三种方法可以退出p：第一，如果操作员打入文件的结束符号（control-z跟着一个回车），则执行启动的ON条件，并通过标号变量“lab”将控制转向标号为“exit”的语句。由于这个GOTO从p环境外部获得控制，所以ON条件自动恢复。

第二个可能的出口跟在第21行的测试后面，如果操作员打入一个值等于索引值，则执行第22行的GOTO语句，同时将控制转向非局部标号“exit”，恢复到初始条件。

最后，如果到了过程p的结束，则控制可正常返回。在这种情况下，仍执行自动的RE-

VERT，同时使第17行的ON条件无效。这样已启动的ON条件总是被禁止，而不管程序控制是否离开P环境。

```
PL/I-80 V1.0, COMPILATION OF:REVERT
L: List Source Program
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0, COMPILATION OF: REVERT
   1 a 0000 revert:
   2 a 0006      proc options(main);
   3 c 0006      dcl
   4 c 000D          i fixed,
   5 c 000D          sysin file,
   6 c 000D
   7 c 000D      do i=1 to 10000;
   8 c 001F      call p(i, exit);
   9 c 003C      exit:
  10 c 003C      end;
  11 c 003C
  12 c 003C      p:
  13 c 003C          proc(index, lab);
  14 e 003C          dcl
  15 e 004C              (t, index) fixed,
  16 e 004C              lab label;
  17 e 004C          on endfile(sysin)
  18 f 0054             go to lab;
  19 e 005F          put skip list(index,':')
  20 e 008A          get list(t);
  21 e 00A2          if t=index then
  22 e 00B6              go to lab;
  23 c 00C2          end p;
  24 a 00C2      end revert;
CODE SIZE=00C5
DATA AREA=0011
```

图 6-1　说明REVERT处理的程序

6.3 SIGNAL语句

对应于特殊ON语句的ON语句体可通过执行SIGNAL语句激活，SIGNAL 语句的格式为：　　SIGNAL　条件;

该语句的作用相当于条件已在外部启动。如果堆栈中有ON条件，则执行栈顶的ON条件；如果没有活动的ON条件，则执行缺省的系统动作，下面的程序段说明了 SIGNAL 语句的特殊用法。

```
on endfile (sysin)
  stop;
do while ('1'b);
get list (buff);
if buff='END' then
    signal endfile (sysin);
put skip list (buff);
end;
```

在这个例子中，每当从SYSIN文件中读“END”值时就执行SIGNAL语句。ON条件设置在程序的开始，这样可在文件实际结束或读到“END”值时接受控制。

6.4 ERROR异常

ERROR条件是所有PL/I-80异常条件中范围最广的条件，包括了系统定义的和程序员定义的条件，通过它的子代码来表示。ERROR条件形式是：

```
ON ERROR on语句体;
  SIGNAL ERROR;
  REVERT ERROR;
ON ERROR (整数表达式) on语句体;
  SIGNAL ERROR (整数表达式);
  REVERT ERROR (整数表达式);
```

在前三种情况下，假设ERROR子代码为零，而后边三种情况包括从0到255范围内的特定的子代码。形式：

```
ON ERROR on语句体
ON ERROR (0) on语句体
```

截获任何错误状态，不管设置什么子代码。

例如：形式 ON ERROR (3)…；

仅截获由子代码3伴随的错误条件。一般来说，具有0～127子代码的ERROR条件被认为是严重错误，因此这些条件的on语句体不得返回，而必须执行一个GOTO转向非局部标号，子代码在128～255范围内被认为是无害的，并可在完成一些局部的动作后返回。

ERROR条件的子代码分成四组：

（a）0～63　保留作PL/I-80用

（b）64～127　程序员定义

（c）128～191　保留作PL/I-80用

（d）192～255　程序员定义

（a）组中目前已指定的子代码是：

ERROR(1)——数据转换。在赋值、计算或输入处理过程中数据类型不一致。

ERROR(2)——I/O堆栈溢出。

ERROR(3)——超越函数的自变量超出值域。

ERROR(4)——I/O矛盾。一个打开文件的属性与一个特指的GET、PUT、READ或WRITE要求的属性不匹配。

ERROR(5)——格式堆栈溢出，嵌套的格式求值超过32级。

ERROR(6)——无效格式项，数据项与格式项不一致或遇到不可识别的格式项。

ERROR(7)——自由空间用尽，在动态存储区没有可用的空间。

下面的程序段提供了ERROR条件用法的简单例子：

```
on error (1)
    begin;
    put skip list ('Invalid Input:');
    go to retry;
    end;
retry:
    get list (x);
```

GET 语句从SYSIN文件中读入一个变量x。如果操作员在输入操作时打入非法数据，运行系统就会给出ERROR(1)的信号。在这种情况下，on 语句体得到控制并将错误 信息写到控制台上，同时从“retry”标号重新开始执行。

SIGNAL语句可与on语句配合使用标志终止或非终止条件。语句：

```
signal error (64)
```

产生ERROR (64) 条件。如果有一个ON ERROR (64) 是活动的，则相应的on语句体接受控制，否则程序以一个错误信息结束。语句：

```
signal error (255)
```

完成相似的工作，只是如果ERROR (255) 条件不活动则程序不结束。注意ON ERROR或ON ERROR(0)将截取任何0～255范围内的子代码。然而可用下面讨论的ONCODE功能提取特殊的错误子代码。

6.5 FIXEDOVERFLOW, OVERFLOW, UNDERFLOW 和 ZERO DIVIDE

算术运算异常条件是：

FIXEDOVERFLOW或FIXEDOVERFLOW(i)

OVERFLOW或OVERFLOW(i)

UNDERFLOW或UNDERFLOW(i)

ZERODIVIDE或ZERODIVIDE(i)

其中 i 表示任选的整数表达式。与ERROR功能相似.ON, REVERT和SIGNAL语句可以指定任何条件。另外，如果没有整数表达式，则假设为零值。子代码值为零的ON 语句截获值为0～255范围内的任何子代码。在列出ERROR功能时，子代码分为系统定义的 和用 户定义的值。但要注意，我们认为所有运算错误缺省为终止条件，即如对一个算术运算异常设置on条件，则on语句体必须包含一个到全程标号的转移，否则程序从on单元返回时结 束。

目前已定义的系统子代码如下：

FIXEDOVERFLOW——十进制加、乘或存储

OVERFLOW(1)——浮点压缩

UNDERFLOW(1)——浮点压缩

ZERODIVIDE(1)——十进制除

ZERODIVIDE(2)　浮点除

ZERODIVIDE(3)——整数除

6.6 ENDFILE UNDEFINEDFILE, KEY和ENDPAGE

在与特定文件存取有关的I/O处理过程中会出现一些异常条件。这些条件表示为：

ENDFILE（文件引用）

UNDEFINEDFILE（文件引用）

KEY（文件引用）

ENDPAGE（文件引用）

其中“文件引用”表示一个文件值表达式，它产生的文件不要求为打开的文件。

当从STREAM文件中读到文件结束符（Control-z）或在以SEQUENTIAL方式处理的RECORD文件中遇到物理文件结束时，都将出现ENDFILE条件。带键的DIRECT READ超出文件结束时也产生ENDFILE条件。同样，RECORD或STREAM OUTPUT操作如果超出磁盘容量时，也将指示为ENDFILE。

对文件进行INPU或OUTPUT存取时，如果指示的盘上不存在文件，则出现UNDEFINEDFILE条件。如果一个物理设备（$CON, $LST, $RDR, $PUN）被作为KEYED或UPDATE文件存取，也出现这种条件。

当程序想存取一个超出磁盘容量的键值时，就出现KEY条件。

在当前行的值达到指定文件的PAGESIZE时，PRINT文件就会出现ENDPAGE条件。当前行从零开始，每送到文件中一个换行，当前行加1。如果文件开始是以PAGESIZE(0)打开，则不出现ENDPAGE条件（PL/I-80以零PAGESIZE打开缺省为控制台输入SYSIN）。每当通过ON单元中PUT语句的PAGE任选给输出文件送一个换页，或通过系统缺省自动地插入一个换页时，当前行重置为1。无论哪一种情况，如果在SKIP任选执行时产生ENDPAGE条件则SKIP结束。

必须注意，如果一个ON单元截获ENDPAGE条件，但不执行带有PAGE任选的PUT语句，则当前行不置“1”。其结果是，因为当前行连续无界，所以不给ENDPAGE信号，直到执行带有PAGE任选的PUT语句为止。实际上，当前行要计数到32767，然后再从1开始。由于行计数总大于零，所以对于PAGESIZE为零的文件永不会出现ENDPAGE条件。

如上所述，如果没有活动的ENDPAGE ON单元，缺省系统动作是将换页符插入到输出文件中。但对ENDFILE UNDEFINEDFILE和KEY，缺省系统动作是终止有错误信息的程序执行。

如果一个ON单元对ENDFILE、UNDEFINEDFILE或KEY接受控制，并返回到发信号的点，则当前I/O操作结束，并且控制转向导致该条件的OPEN、GET、PUT、READ或WRITE后面的语句。

6.7 ONCODE, ONFILE, ONKEY, PAGENO和LINENO

PL/I-80提供一些内部构造函数，辅助异常处理。具体地说，在所有PL/I-80程序的范围中有五种函数说明：

```
dcl
    oncode entry returns (fixed),
```

```
onfile entry returns (char(31) varying),
onkey entry returns (fixed),
pageno entry (file) returns (fixed),
lineno entry (file) returns (fixed);
```

ONCODE函数返回最近标志的子代码，如果没有出现错误条件就返回零。例如，这个函数可用于确定在ON单元被激活后错误发生的确切根源：

```
on error
    begin;
    dcl code fixed;
    code = oncode();
    if code = 1 then
        do;
        put list('Bad Input:');
        go to retry;
        end;
    put list('Error#', code);
    end;
retry:
```

ONFILE函数返回产生错误条件的I/O操作中所含的最后一个内部文件名的串值。在转换错误情况下，ONFILE函数产生当时活动的文件名。如果在所标志的条件中不包含文件，则ONFILE函数返回长度为零的串。下面给出一个ONFILE函数的实例：

```
on error(1)
    begin;
    put list ('Bad Data:', onfile());
    go to retry;
    end;
retry:
```

ONKEY函数返回产生ONKEY条件的I/O操作所含的最后一个键的值，而且仅在活动单元的on语句体中有效。ONKEY的用法的实例：

```
on key (newfile)
    put skip list ('bad key', onkey());
```

最后两个函数，PAGENO和LINENO，返回参数中给出PRINT文件的当前页号和当前行号，注意当ENDPAGE条件由除SIGNAL以外的语句发出时，行数比该文件页尺寸大1。

第七节　表处理的应用

在每个PL/I-80程序中都含有动态存储管理子程序。当程序装入到内存中执行时，所做的第一个动作是所将有剩余自由存储区作为一个连接表结构。程序可通过ALLOCATE

语句动态分配这块存储区，然后用FREE语句释放。所有存储管理都在后台使用 PL/I-80 库中的子程序进行。以这种方式分配的内存段可通过表逻辑地连接到另一个内存段。表元素是PL/I结构，它包括信息字段和一个或多个指针变量，对其他表元素进行存取。

表处理的动态性质常常用于当程序管理的数据元素的数目和结构变化较大的时候。这节的程序说明了在两种情况下表处理的用法，在这两种情况下，数据分配不容易预先确定。下面详细讨论每个程序以提供一些PL/I-80表处理的具体实例。

7.1 词表的管理

在详细讨论之前，我们要弄懂一些关于表处理的技巧问题。首先，有基变量正好是一个适合于内存区，但不具有直接分配给它的存储区的属性单元。有基变量属性单元用指针变量以程序的方法置于一个特定内存区，完成这个工作可有两种方法，根据有基变量说明的形式而定。如果在说明中不包括隐含的基数，则对于有基变量的任何引用必须是指针受限的。如果说明中有隐含的基数，则引用可以包括一个指针受限符，也可以简单地用在说明中给出的隐含的指针作为基数。这种情况可用一个实例说明。假定有下列说明：

```
dcl
  i fixed,
  mat (0: :5) fixed
   (p, q) pointer
  x fixed based,
  y fixed based (p),
  z fixed based (f( ));
```

两个变量 i 和mat不是有基变量，所以这些数据项分配有存储区。同样，两个指针变量 p 和 q已经分配了存储单元。然而 3 个变量 x 、y 和 z 被说明为有基变量，其实际内存地址在程序执行时决定。变量 x 没有隐含的基数，所以对 x 的所有引用必须用指针受限符。如：

```
p->x= 5 和 q->x= 6 ;
```

在第一种情况下，在由 p 给出的内存单元中定点（两字节）变量被赋值为 5，而第二个语句将值 6 存入由q给出的内存单元中。另一方面变量 y 有一个隐含的基数，可以用也可不用 指针受限符加以引用。

引用y= 5 ; 等价于p->y= 5 ; 所以y= 5 ; 和q->y= 6 ; 与上面对 x 的两种赋值有相同的效果。

变量 z 与 y 一样，有一个隐含基数。在这种情况下，这个基数是一个没有自变量的指针值函数的调用。例如：函数 f 的格式取为：

```
f:
    proc returns (ptr);
    return (addr (mat(i)));
    end f;
```

对 z 合法的引用是：

```
p->z= 5 ; 和  z= 6 ;
```

第一种形式与上面表示的那些相似，其中单元是指针变量 p 指出的。而第二种形式是：

```
f ( )->z= 6 ;
```

的缩写形式。在这种情况下，对函数 f 求值以产生有基变量 z 的存储区地址。用这种形式有两个好处。首先，指针值表达式可以是复型的，不限于一个简单的指针变量。第二，函数 f 的代码只在内存中出现一次，而不是在每次变量引用中重复出现，这样可节省大量的程序空间。另外，必须注意，隐含基数必须在对该有基变量说明的作用域内。下面的非功能程序段描述了这个概念。

```
main:
    proc options(main);
    dcl
        x based(p),
        y based(q),
        p ptr;
        begin;
        dcl
          (p, q) ptr;
        x=5;
        y=10;
        end;
    dcl
        q ptr;
    end main;
```

在BEGIN分程序内的 p 和q的说明没有起作用。因为有基变量 x 和 y 引用 p 和 q 必须在相同的或包含的作用域内。

下面我们来讨论表处理的例子。句子变换程序称为 REV，在图 7-1 中示出。这个程序可分成三个主要部分。第一部分，从第 3 行到第14行，完成读句子和以反向的次序写回到控制台的功能。为了简化整个程序结构，读功能由一个独立的子程序完成，称为 READ，它起始于第16行。同样，程序输出由第三部分中称为 WRITE 的子程序完成，它从第34行开始，每个输入句子由一系列词组成，最长 30 个字符，足以容纳floccinaucinihilipilification，它是英语中最长的词。为了简化输入处理，在结束句子的句号之前，REV 需要一个空格。当操作员打入一个空句子时，程序结束。图 7-2 表示了控制台与 REV 程序的对话。

```
PL/I-80 V1.0, COMPILATION OF:REV
   1 a 0000 reverse:
   2 a 0006     proc options (main);
   3 c 0006     dcl
   4 c 0006       sentence ptr,
   5 c 0006       1 wordnode based (sentence),
   6 c 0006         2 word char (30) varying,
   7 c 0006         2 next ptr;
   8 c 0006
```

```
 9 c 0006        do while ('1'b);
10 c 0006        call read ( );
11 c 0009        if sentence = null then
12 c 0011           stop;
13 c 0014        call write ( );
14 c 001A        end;
15 c 001A
16 c 001A     read:
17 c 001A        proc;
18 e 001A        dcl
19 e 001A           newword char (30) varying,
20 e 001A           newnode ptr;
21 e 001A        sentence = null;
22 e 0020        put skip list ('what''s up?');
23 e 003C           do while ('1'b);
24 e 003C           get list (newword);
25 e 0056           if newword = '.' then
26 e 0064              return;
27 e 0065           allocate wordnode set (newnode);
28 e 006E           newnode->next = sentence;
29 e 007D           sentence      = newnode;
30 e 0083           word          = newword;
31 e 0091           end;
32 c 0091        end read;
33 c 0091
34 c 0091     write:
35 c 0091        proc;
36 e 0091        dcl
37 e 0091           p ptr;
38 e 0091        put skip list ('Actually, ');
39 e 00AD           do while (sentence ^= null);
40 e 00B5           put list (word);
41 e 00CA           p = sentence;
42 e 00D0           sentence = next;
43 e 00DE           free p->wordnode;
44 e 00E7           end;
45 e 00E7        put list ('.');
46 e 00FE        put skip;
47 c 0110        end write;
48 c 0110
49 a 0110        end reverse;
```

图 7-1　REV句子变换程序清单

```
A>b:rev
What's up? Now is the time for all good parties .
Actually, parties good all for time the is Now
What's up? The rain in Spain falls mainly in the plain .
Actually, plain the in mainly falls Spain in rain The .
What's up? a man a plan a canal panama .
Actually, panama canal a plan a man a .
What's up?
End of Execution
```

图 7-2 与REV程序对话

REV程序一般操作如下。将每个词存入用ALLOCATE语句建立的单独的内存区中，每个ALLOCATE语句获得足以容纳第5行上的“wordnode”结构的唯一的一段自由存储区，每个单元数量为32字节(可以通过检查编译开关$产生的符号表来核实)。wordnode元素通过每个单元的“next”字段连接，表的开始由“sentence”指针变量的值给出。

在PL/I-80中，指针变量实际是两字节16位字，它包含变量地址。例如，语句：

```
allocate wrodnote set (newonde)
```

找到一块33字节存储域来容纳wordnode数据项，并将指针变量newnode值置为这个存储域的地址（wordnode是33字节，因为变长的串“word”需要一个字节存放当前长度，30字节存放串本身，后面跟着两个字节的指针值)。例如，给出输入句子：

FLICK YOUR BIC

执行三次ALLOCATE语句（对表中的每个字执行一次)。为了说明方便，假设这三个存储区分配地址为1000、2000和3000，REV程序通过READ过程内的主循环读入该句子。

REV开始将句子指针初置为空地址，即地址0000。进入第23行DO组时，sentence的值表现为：

SENTENCE: |0000|

第一个词FLICK由24行上的GET语句读入，因为这个值不是一个句号，分配第一个33个字节区域以便存放该词。当构造一个句子时，sentence变量的指针值被置于“next”字段中，同时输入词被存入“word”字段。则最近读入的词成为表的新的表头。在处理完FLICK之后，表变为：

SENTENCE: |1000|

1000: |FLICK|

|0000|

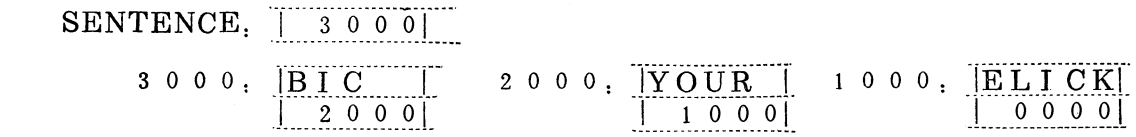
然后程序通过循环继续进行。这时读入词YOUR并分配第二个33字节的区域。最新分配的区域成为新的表头，产生的指针结构为：

SENTENCE: |2000|

2000: |YOUR |

|1000|

然后处理最后一个词BIC，分配最后的33字节区域并置于表的头部，产生节点序列：



由于表的构成是按次序的，实际上没有必要对表作转换处理。事实上，WRITE函数只是对表进行简单查找，从sentence指针开始将在39行到44行中的循环中遇到的每个词打印出来。打印完后，立即将分配给它的33字节区域用第43行上的FREE语句释放。重要的是，要注意在释放当前元素之前，先将next移入指针变量sentence。因为在它被释放后不能保证存储仍然是完整的。

在这种情况中，表结构的主要优点是句子可以任意长，它只受现存储器的尺寸限制。缺点是句子由254字符串表示，大大增加了存储消耗。

第八节　递归在 PL/I-80 中的使用

递归处理是简化编程中部分自定义问题的常用语言工具。我们将考察这样的三个问题，其前两个说明基本概念，最后给出一个在实际问题中常用的递归举例。

暂先引入递归处理的技巧，递归过程是在返回第一级调用前直接地或间接地重新进入该过程的一种隐含调用。在PL/I中，所有的递归过程必须有RECURSIVE属性，以便在每级递归时能对局部数据域进行适当的保护及恢复。在PL/I-80中，递归过程有两个限制。其一，所有的过程参数是'按值调用'的，这意味着不能用对形式参数的赋值来从递归过程中返回值。代之，程序可以返回一个函数值或赋值给全程变量。为保持与全集PL/I的兼容，在PL/I-80递归过程中，形式参数不能用在赋值语句的左端。其二，PL/I-80不允许在递归过程中含有BEGIN分程序，但允许有嵌套过程及DO组。递归过程的严格系统的阐述在下面的举例中给出。

8.1　阶乘的计算

不描述阶乘的计算便不能很好地介绍递归。阶乘计算自始至终所用的技巧很容易使用"迭代"及递归定义；它是一个很好的举例。阶乘函数的迭代定义是：

$$k!=(k)(k-1)(k-2)\cdots(2)(1)$$

其中$k!$是非负整数k的阶乘函数。注意，由于：

$$(k-1)!=(k-1)(k-2)\cdots(2)(1)$$

我们可以由阶乘函数本身的定义，使用递归关系：

$$k!=k(k-1)!$$

这里我们定义：

$$0!=1$$

用迭代或递归计算阶乘函数将得出如下的值：

$$0!=1$$

$1!=1$

$2!=(2)(1)=2$

$3!=(3)(2)(1)=6$

$4!=(4)(3)(2)(1)=24$

$5!=(5)(4)(3)(2)(1)=120$

图8-1给出了用迭代计算阶乘函数值的程序IFACT的程序清单。变量FACT是一个定点二进制数据项，它累积阶乘的值直到最大值32767。从IFACT的输出在图8-2中给出。所给的阶乘函数的值到$7!=5040$。在该点，FACT变量溢出产生不正确的结果。回顾一下，由于其开销将对执行时间有较大影响，故在PL/I-80中对于二进制计算不发FIXED-OVERFLOW信号。

```
PL/I-80 V1.0,COMPILATION OF:IFACT
L: List Source Program
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0,COMPILATION OF:IFACT
    1  a  0000  f:
    2  a  0006       proc options (main);
    3  c  0006       dcl
    4  c  0006           (i,n,fact) fixed;
    5  c  0006           do i = 0 by 1;
    6  c  000C           fact = 1;
    7  c  0012               do n= i to 1 by -1;
    8  c  0021               fact = n * fact;
    9  c  0039               end;
   10  c  0039           put edit ('factorial(',i,')=',fact)
   11  c  0081                   (skip,a,f(2),a,f(7));
   12  c  0081           end;
   13  a  0081           end f;
CODE  SIZE=0081
DATA  AREA=0021
```

图 8-1 IFACT 程序清单

图8-3给出了阶乘函数的等价递归求值。比较一下，REPEAT形式的DO组用控制测试，在这种情形下，FACT是一个标志作RECURSIVE的过程，它是在第6行最高层的PUT语句中被调用的，在第14行的RETURN语句中隐含了递归调用。注意，当输入值为零时FACT立即返回。所有其他情况则要求有一个或多个FACT的递归求值来产生结果。例如，$3!$产生计算序列：

```
fact(3)=3 * fact(2)
            fact(2)=2 * fact(1)
                        fact(1)=1 * fact(0)
                                    fact(0)=1
```

```
A>b:ifact
factorial( 0)=       1
factorial( 1)=       1
factorial( 2)=       2
factorial( 3)=       6
factorial( 4)=      24
factorial( 5)=     120
factorial( 6)=     720
factorial( 7)=    5040
factorial( 8)=  -25216
factorial( 9)=  -30336
factorial(10)=   24320
factorial(11)=    5376
factorial(12)=   -1024
factorial(13)=  -13312
factorial(14)=   10240
factorial(15)=   22528
factorial(16)=  -32768
factorial(17)=  -32768
factorial(18)=       0
factorial(19)=       0
factorial(20)=       0
factorial(21)=       0
```

图 8-2 IFACT 程序的输出

$$=1*1$$
$$=2*1*1$$
$$=3*2*1*1$$

得出最终值6。递归阶乘计算的输出值列在图8-4中，再次注意，由于计算的精度，在5040后，数值溢出。

我们可以用这种机会检查修改数据项类型和精度时输出值的变化。图 8-5 给出了阶乘的递归求值，其中十进制值用了最大精度。由程序所产生的最大值在图8-6中是：

factorial(17)=355,687,428,096,000

在该点，PL/I-80发出FIXEDOVERFLOW信号表示十进制计算已溢出最大值15位值。与其类似，在图8-7中给出了用浮点二进制数据项计算的阶乘函数，该程序的输出列在图8-8中。虽然该函数可以计算比17!更大的值，但函数的有效数位将从右边截掉至约等于$7^1/_2$个十进制的数位。当 PL/I-80发出 OVERFLOW 条件信号时，浮点二进制计算终止。该条件是在指数值不能以浮点法表示时产生的。

```
PL/I-80 V1.0, COMPILATION OF: FACT
L: List Source program
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0, COMPILATION OF: FACT
   1  a  0000 f:
   2  a  0006      proc options(main);
   3  c  0006      dcl
   4  c  0006          i fixed;
   5  c  0006          do i = 0 repeat(i+1);
   6  c  000C          put skip list('factorial(',i,')=',fact(i));
   7  c  0056          end;
   8  c  0056      stop;
   9  c  0056
  10  c  0056      fact;
  11  c  0056          procedure(i) returns(fixed) recursiye;
  12  e  0056          dcl i fixed;
  13  e  0070          if i = 0 then return(1);
  14  e  0080          return (i * fact(i-1));
  15  c  0099          end fact;
  16  a  0099      end f;
CODE SIZE = 0099
DATA AREA = 0018
```

图 8-3 定点二进制阶乘程序清单

```
A>b:fact
factorial(    0)=      1
factorial(    1)=      1
factorial(    2)=      2
factorial(    3)=      6
factorial(    4)=     24
factorial(    5)=    120
factorial(    6)=    720
factorial(    7)=   5040
factorial(    8)= -25216
factorial(    9)= -30336
factorial(   10)=  24320
factorial(   11)=   5376
factorial(   12)=  -1024
factorial(   13)= -13312
factorial(   14)=  10240
factorial(   15)=  22528
```

```
factorial(      16)=    -32768
factorial(      17)=    -32768
factorial(      18)=         0
factorial(      19)=         0
factorial(      20)=         0
factorial(      21)=         0
factorial(      22)=         0
```

图 8-4 FACT 程序的输出

```
PL/I-80 V1.0, COMPILATION OF: DFACT
L: List Source program
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0, COMPILATION OF:DFACT
   1  a  0000 f:
   2  a  0006      proc options(main);
   3  c  0006      dcl
   4  c  0006          i fixed;
   5  c  0006          do i = 0 repeat(i+1);
   6  c  000C          put skip list('Factorial(',i,')=',fact(i));
   7  c  0058          end;
   8  c  0058      stop;
   9  c  0058
  10  c  0058      fact:
  11  c  0058          proc (i)
  12  c  0058               returns(fixed dec(15,0)) recursive;
  13  e  0058      dcl
  14  e  0072               i fixed;
  15  e  0072      dcl
  16  e  0072               f fixed dec(15,0);
  17  e  0072               if i = 0 then
  18  e  007B                   return (1);
  19  e  0089               return (decimal (i,15) * fact(i-1));
  20  c  00A5          end fact;
  21  a  00A5      end f;
CODE SIZE = 00A5
DATA AREA = 0028
```

图 8-5 十进制阶乘程序清单

```
A>b:dfact
Factorial(         0)=                    1
Factorial(         1)=                    1
Factorial(         2)=                    2
Factorial(         3)=                    6
Factorial(         4)=                   24
Factorial(         5)=                  120
Factorial(         6)=                  720
Factorial(         7)=                 5040
Factorial(         8)=                40320
Factorial(         9)=               362880
Factorial(        10)=              3628800
Factorial(        11)=             39916800
Factorial(        12)=            479001600
Factorial(        13)=           6227020800
Factorial(        14)=          87178291200
Factorial(        15)=        1307674368000
Factorial(        16)=       20922789888000
Factorial(        17)=      355687428096000
Factorial(        18)=
FIXED OVERFLOW (1)
Traceback: 0007 019F 0018 0000 # 2809 6874 0355 0141
End of Execution
```

图 8-6 DFACT 程序的输出

```
PL/I-80 V1.0,COMPILATION OF: FFACT
L: List Source Program
   NO ERROR (S) IN PASS 1
   NO ERROR (S) IN PASS 2
PL/I-80 V1.0, COMPILATION OF: FFACT
   1  a  0000 f:
   2  a  0006      proc options (main);
   3  c  0006       dcl
   4  c  0006            i fixed;
   5  c  0006            do i = 0 repeat(i+1);
   6  c  000C            put skip list('factorial(',i,')=',fact(i));
   7  c  0056            end
   8  c  0056       stop;
   9  c  0056
  10  c  0056      fact:
  11  c  0056          procedure(i) returns (float) recursive;
  12  e  0056          dcl i fixed;
```

```
 13  e  0070          if i = 0 then
 14  e  0079              return(1);
 15  e  0085          return (i * fact (i-1));
 16  c  00A1          end fact;
 17  a  00A1      end f;
CODE SIZE =00A1
DATA AREA =001C
```

图 8-7 浮点二进制阶乘程序清单

```
A>b:ffact
factorial(         0)=  1.000000E+00
factorial(         1)=  1.000000E+00
factorial(         2)=  2.000000E+00
factorial(         3)=  0.600000E+01
factorial(         4)=  2.400000E+01
factorial(         5)=  1.200000E+02
factorial(         6)=  0.720000E+03
factorial(         7)=  0.504000E+04
factorial(         8)=  4.032000E+04
factorial(         9)=  3.628799E+05
factorial(        10)=  3.628799E+06
factorial(        11)=  3.991679E+07
factorial(        12)=  4.790015E+08
factorial(        13)=  0.622702E+10
factorial(        14)=  0.871782E+11
factorial(        15)=  1.307674E+12
factorial(        16)=  2.092278E+13
factorial(        17)=  3.556874E+14
factorial(        18)=  0.640237E+16
factorial(        19)=  1.216450E+17
factorial(        20)=  2.432901E+18
factorial(        21)=  0.510909E+20
factorial(        22)=  1.124000E+21
factorial(        23)=  2.585201E+22
factorial(        24)=  0.620448E+24
factorial(        25)=  1.551121E+25
factorial(        26)=  4.032914E+26
factorial(        27)=  1.088387E+28
factorial(        28)=  3.048383E+29
factorial(        29)=  0.884176E+31
factorial(        30)=  2.652528E+32
factorial(        31)=  0.822283E+34
```

```
factorial(        32)=  2.631308E+35
factorial(        33)=  0.868331E+37
factorial(        34)=
OVERFLOW (1)
Traceback: 0046 13BF 13E6 019B # 8608 0B15 FB51 0141
End of Execution
```

图 8-8 FFACT 程序的输出

```
PL/I-80 V1.0, COMPILATION OF: ACK
      L: List Source Program
         NO ERROR (S) IN PASS 1
         NO ERROR (S) IN PASS 2
PL/I-80 V1.0, COMPILATION OF: ACK
   1  a  0000 ack:
   2  a  0006      procedure options(main,stack(2000));
   3  c  0006      dcl
   4  c  0006         (m,maxm,n,maxn) fixed;
   5  c  0006      put skip list('Type max m,n:');
   6  c  0022      get list(maxm,maxn);
   7  c  0046      put skip
   8  c  0095         list('       ',(decimal(n,4) do n=0 to maxn));
   9  c  0095         do m = 0 to maxm;
  10  c  00AE         put skip list(decimal (m,4),':');
  11  c  00DA            do n = 0 to maxn;
  12  c  00F3            put list(decimal(ackermann(m,n),4));
  13  c  0126            end;
  14  c  0126         end;
  15  c  0126      stop;
  16  c  0129
  17  c  0129       ackermann:
  18  c  0129          procedure(m,n) returns(fixed) recursive;
  19  e  0129          dcl (m,n) fixed;
  20  e  0153          if m = 0 then
  21  e  015C             return(n+1);
  22  e  0164          if n = 0 then
  23  e  016D             return(ackermann(m-1,1));
  24  e  017E          return(ackermann(m-1,ackermann(m, n-1)));
  25  c  019F          end ackermann;
  26  e  019F       end ack;
CODE SIZE = 019F
DATA AREA = 0048
```

图 8-9 Ackermann 程序清单

8.2 Ackermann 函数的计算

PL/I-80运行系统有一个存储器堆栈，其中存放了子程序返回地址和一些临时结果。在通常情况下，分配给堆栈的存储域对于非递归过程处理及简单的递归过程求值是足够大的。下面用从数论导出的一个函数给出更综合性的递归举例。Ackermann函数记为A(m,n)，其递归定义如：

```
A(m,n)=
        if m=0  then  n+1, otherwise
        if n=0  then  A(m-1,1), otherwise
            A(m-1,A(m,n-1))
```

我们用 Ackermann 函数阐明多重递归。它使用的栈深超出了PL/I-80提供的缺省值。图8-9所示的程序，读入了两个最大的m和n值计算 Ackermann函数的值。程序的对话列在图8-10中，应注意，虽然 Ackermann 函数返回一个定点二进制值，但在8、10和12行中用了 DECIMAL 内部构造函数来控制 PUT LIST 输出转换域的大小。

```
A>B:ACK
Type max m,n: 4,6
          0    1    2    3    4    5    6
    0 :   1    2    3    4    5    6    7
    1 :   2    3    4    5    6    7    8
    2 :   3    5    7    9   11   13   15
    3 :   5   13   29   61  125  253  509
    4 :  13
```

图 8-10 Ackermann 迭代程序会话

该例的重点是在 2 行用了 STACK 任选项来增加分配给运行堆栈的存储域的大小。STACK 任选项仅在有 MAIN任选时才是合法的，在这种情况下，将堆栈的大小从缺省的512字节增加到2000字节。由于编译不能计算递归的深度，STACK 任选项的值必须根据经验确定。当递归期间堆栈溢出时，便发出信息：

Free Space Overwrite

随之，程序终止，表明所分配的堆栈域太小。

8.3 算术表达式的求值程序

我们每天都在实际使用的递归之一是在编程语言的翻译和执行中。这种用法主要是由于这样一种事实，即语言通常是递归定义的。类似于PL/I-80这样的分程序结构语言，如DO组、BEGIN 和 PROCEDURE 分程序是可以自身嵌套的，所以这种结构就能容易地用递归处理。另一个例子也是本小节的目的，即算术表达式的求值法。表达式的一种简单形式可递归定义如下：

表达式是一个简单数字

表达式是由+、-、*或/分隔并由括号括住的表达式对

由这种定义，数值3.6是一个表达式，这是由于它是一个简单数字，进而

(3.6+6.4)

也是一表达式，它是由＋号分隔的两个简单数组成的表达式对，并括在括号中，由上结果，

(1.2＊(3.6＋6.4))

是一个合法的表达式。它也是由两个合法的表达式1.2和(3.6＋6.4) 组成并由＊号分隔且括在括号中。而序列

3.6＋6.4

(1.2＋3.6＋6.4)

则是非法表达式，因为第一个不在括号中，而第二个括号中并不是一对表达式。上述给出的表达式的定义稍有限制，但一旦有了基本的基础，便可容易地扩充到包含所讲的 PL/I-80 的复杂表达式。

```
PL/I-80 V1.0, COMPILATION OF:EXPR1
L:List Source Program
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0,COMPILATION OF:EXPR1
   1  a  0000  expression:
   2  a  0006       proc options (main);
   3  c  0006       dcl
   4  c  000D           sysin file,
   5  c  000D           value float,
   6  c  000D           token char(10) var;
   7  c  000D
   8  c  000D       on endfile(sysin)
   9  d  0015           stop;
  10  d  001B
  11  c  001B       on error(1)
  12  c  0022           /* conversion or signal */
  13  d  0022           begin;
  14  e  0025           put skip list('Invalid Input at',token);
  15  e  004A           get skip;
  16  e  005B           go to restart;
  17  d  005E           end;
  18  d  005E
  19  c  005E       restart:
  20  c  0065           do while('1'b);
  21  c  0065           put skip(3) list('Type expression:');
  22  c  0081           value = exp( );
  23  c  008A           put skip list('Valueis :',value);
  24  c  00B4           end;
  25  c  00B4
  26  c  00B4       gnt:
  27  c  00B4           proc;
  28  e  00B4           get list(token);
  29  c  00CF           end gnt;
```

```
30  c  00CF
31  c  00CF     exp:
32  c  00CF         proc returns(float binary) recursive;
33  e  00CF         dcl x float binary;
34  e  00D8         call gnt( );
35  e  00DB         if token = '('then
36  e  00E9         do;
37  e  00E9                 x = exp ( );
38  e  00F2                 call gnt ( );
39  e  00F5                 if token = '+' then
40  e  0103                     x=x+exp( );
41  e  0115                 else
42  e  0115                 if token = '-' then
43  e  0123                     x=x-exp( );
44  e  0135                 else
45  e  0135                 if token = '*' then
46  e  0143                     x=x * exp( );
47  e  0155                 else
48  e  0155                 if token = '/'then
49  e  0163                     x=x/exp( );
50  e  0175                 else
51  e  0175                 signal error(1);
52  e  017C                 call gnt( );
53  e  017F                 if token ^= ')' then
54  e  018D                     signal error(1);
55  e  0197                 end;
56  e  0197             else
57  e  0197                 x=token;
58  e  01A6                 return(x);
59  c  01B2                 end exp;
60  a  01B2     end expression;
CODE SIZE = 01B2
DATA AREA = 0046
```

图 8-11　表达式计算程序清单

表达式计算程序如图8-11所示。其基本处理在20到24行，从控制台读入表达式并将计值结果打印给操作员。与控制台的对话如图8-12所示，其中操作员已输入了几个正确的和不正确的表达式。

表达式分析的核心是递归过程EXP，它根据如上给出的递归定义处理输入的表达式。过程EXP将表达式分解为逐一的递归项，GNT过程（Get Next Token）从输入行中读入下一元素（或记号），它是左括号、右括号、数字或某一算术操作符。由于GNT采用GET LIST，故这些记号要求必须用空格或行结束符分隔。EXP在34行开始调用GNT，然后，GNT将下一个输入的记号放在称作token的CHAR(10)变量中，如果所读的第一

```
A>b:expr1

Type expression:(4+5.2)

Value is: 0.920000E+01

Type expression: 4.5e-1

Value is: 4.499999E-01

Type expression:(4&5)

Invalid Input at &

Type expression:((3+4)-(10/8))

Value is: 0.575000E+01

Type expression:(3*4)

Value is: 1.200000E+01

Type expression: ^Z

End of Execution
```

图 8-12　表达式迭代计算程序

项是数字，那么在EXP中的测试系统将控制送到57行，通过对x的赋值自动地将token中的字符值转换为浮点值。该转换后的值从EXP返回到22行，存入“value”，作为表达式的结果。如果表达式是非无效的，EXP在35行扫描前导左括号并进入36行的DO组，立即对第一个表达式求值，而不管其如何复杂，并在37行存入变量x，而后再检查token的值是+、-、*或/。例如，假定在45行遇到的是*操作符，46行的语句递归地调用EXP过程计算表达式右边的值，返回值则是该值与先前所求得的左边的值的乘积。52行开始检查右括号的成对情况，并在58行将产生的结果作为EXP的值而返回。

异常条件的处理有三个地方，由8行的中断续元处理输入文件的文件结束条件，它执行

一条 STOP 语句。可能发生错误的第二个点是在行57 从字符到浮点赋值的转换中，如发生该条件，则行11的中断续元接受控制，显示错误的 token 值并用 GET SKIP 语句清掉该行的所有数据。然后程序的控制在 "restart" 标号处重新开始，并提示操作员输入另一个表达式。

当遇到非法操作符或不成对的表达式时，由程序本身产生一个异常条件。如果操作符并非+、-、*或/，那么执行51语句，行11的中断续元发出信号，由此导致一个错误信息并转到 "restart"，当前的递归级被取消，程序重新开始。类似地，缺少右括号将在行53引起error(1)中断续元，报告一个错误并重开始执行程序。

这里的一个主要问题是每个输入记号之间要有空格隔开，有些不方便。图8-13给出了将先前具有 GNT 过程的表达式处理程序用自由域扫描所取代的另一表达式处理程序。由于在程序重开始时必须将当前输入的剩余字符丢弃(行20)，错误的恢复也作了改变。图8-14给出了对表达式分析器作了改进后的控制台对话。

在本例中，有足够的扩充余地。首先可以在基本的算术函数上添加更多的操作符，如想添加操作符的优先级及要求加括号，除此之外，可添加变量名及赋值语句。事实上只须再做一点工作，便可将程序变为一个 Basic 解释程序。

PL/I-80 V1.0, COMPILATION OF:EXPR2

```
 1  a 0000 expression:
 2  a 0006     proc options(main);
 3  a 0006
 4  c 0006     % replace
 5  c 000D           true by'1'b;
 6  c 000D
 7  c 000D     dcl
 8  c 000D           sysin file,
 9  c 000D           value float,
10  c 000D           (token char(10), line char (80)) varyinq
11  c 000D              static initial(' ');
12  c 000D
13  c 000D     on endfile(sysin)
14  d 0015           stop;
15  d 001B
16  c 001B     on error(1)
17  c 0022           /* conversion or signal */
18  d 0022           begin;
19  e 0025           put skip list('Invalid Input at', token);
20  e 004A           token=' '; line=' ';
21  e 0054           go to restart;
22  d 0057           end;
23  d 0057
24  c 0057     restart:
25  c 005E
```

```
26  c 005E      do while('1'b);
27  c 005E      put skip(3) list ('Type expression:');
28  c 007A      value=exp( );
29  c 0083      put edit('Value is:', value)  (skip, a, f(10, 4));
30  c 00AE    end;
31  c 00AE
32  c 00AE  gnt:
33  c 00AE      proc;
34  e 00AE    dcl
35  e 00AE    i fixed;
36  e 00AE
37  e 00AE        line=substr(line, length(token)+1);
38  e 00C8           do while(true);
39  e 00C8           if line=' 'then
40  e 00D6              get edit(line) (a);
41  e 00F3           i=verify(line, ' ');
42  e 0108           if i=0 then
43  e 0111             line=' ';
44  e 0119             else
45  e 0119             do;
46  e 0119             line=substr(line, i);
47  e 012F            i=verify(line, '0123456789.');
48  e 0144            if i=0 then
49  e 014D              token=line;
50  e 015B            else
51  e 015B            if i=1 then
52  e 0164              token=substr(line, 1, 1);
53  e 017A             else
54  e 017A                 token=substr(line, 1, i-1);
55  e 0193            return;
56  e 0197            end;
57  e 0197            end;
58  c 0197        end gnt;
59  c 0197
60  c 0197    exp:
61  c 0197        proc returns(float binary) recursive;
62  e 0197        dcl x float binary;
63  e 01A0        call gnt( );
64  e 01A3        if token='(' then
65  e 01B1           do;
66  e 01B1           x  =exp( );
67  e 01BA           call gnt( );
68  e 01BD           if token='+' then
```

```
69  e 01CB              x=x+exp( );
70  e 01DD            else
71  e 01DD            if token='-' then
72  e 01EB               x=x-exp( );
73  e 01FD            else
74  e 01FL            if token='*' then
75  e 020B               x=x*exp( );
76  e 021D            else
77  e 021D            if token='/' then
78  e 022B               x=x/exp( );
79  e 023D            else
80  e 023D            signal error(1);
81  e 0244            call gnt( );
82  e 0247            if token^='(' then
83  e 0255               signal error(1);
84  e 025F            end;
85  e 025F         else
86  e 025F            x=token;
87  e 026E         return(x);
88  c 027A         end exp;
89  c 027A
90  a 027A      end expression;
CODE SIZE=027A
DATA AREA=00B5
```

图 8-13　扩充表达式计算程序

```
A>b:expr2
Type expression:(2*14.5)
Value is:     29.0000
Type expression: ((2*3)/ (4.3-1.5))
Value is:      2.1429
Type expression:zot
Invalid Input at z
Type expression: ((2*3
          )    -5)
Value is:      1.0000
Type expression:  (2 n5)
Invalid Input at n
Type expression: ^Z
End of Execution
```

图 8-14　扩充计算程序控制台会话

第九节　分段编译和连接

到现在为止讨论的所有程序都是一个独立的单元，其中有许多程序包括有嵌套的局部过程。如前所提到的一个很有用的方法是将较大的程序分为单独的模块，其后再与另外的程序及PL/I-80子例程库连接在一起。用这种方法分别地编译和连接程序有二条理由。第一，大的程序需要较长的编译，事实上，其标号表不能超过可用内存区域；较小的程序段可以独立地研制、组合、调试，对于整个的目标需要的总编译时间也少。第二，能较快地确定特定的对应用编程很有用的子程序，可以建立自己所需要的子程序库，有选择地将它们连接在一起成为所要的程序。本节提供了一些对程序和数据段进行连接所需的基本信息，并给出了分段编译和连接的完整的举例。

9.1　数据和程序说明

在PL/I-80中，项的说明中含有EXTERNAL属性时，其数据可以共享。例如，说明：

```
dcl x(10) fixed binary external;
```

定义了一个命名为x的变量，占有10个定点二进制单元（20个相邻字节），可由对它进行了同样说明的其他模块进行存取。与其类似，

```
dcl
    1 s,
     2 y(10)   bit(8),
     2 z       char(9) var;
```

定义了一个命名为s的具有20个字节的数据域，它可由其他模块存取。关于外部数据的说明有几项基本规则：

(a) 外部数据项自动地具有STATIC属性。

(b) EXTERNAL数据项在所有对它进行了说明的分程序中是可存取的，并在这个范围内取代内部数据规则。

(c) 由于连接编辑格式要求截掉第七个以后的字符，故EXTERNAL数据项的前6个字符必须是唯一的。

(d) 所有的EXTERNAL数据域必须在它们所出现的所有模块中说明有同样的长度。

(e) 由于问号（？）字符在PL/I-80中用作库命名的前缀，因此要避免在变量名中使用它。

(f) 一个由几个模块引用的EXTERNAL数据项最多只能由一个模块对其初始化。

与第五节中的标号说明相类似，入口常数和入口变量是标识过程名及描述它们的参数值的数据项，入口常数对应于在程序中定义的过程（内部过程）或在连接时定义的过程（外部过程）。在程序的执行中，入口变量可由直接赋值语句或在子程序调用时实在参数对形式参数的隐含赋值而获得入口常数值。过程可以通过入口常数调用，也可间接地通过调用一个持有过程常数值的入口变量调用。与标号变量相类似，入口变量可以带下标。图9-1所示的程序给出了入口常数和入口变量的举例。

```
PL/I-80 V1.0, COMPILATION OF:CALL
L:List Source Program
  NO ERROR(S)IN PASS 1
  NO ERROR(S)IN PASS 2
PL/I-80 V1.0, COMPILATION OF:CALL
    1  a 0000  call:
    2  a 0006      proc options(main);
    3  c 0006      dcl
    4  c 0006          f (3) entry (float) returns (float) variable,
    5  c 0006          g entry(float) returns (float);
    6  c 0006      dcl
    7  c 0006          i fixed, x float;
    8  c 0006
    9  c 0006      f (1) =sin;
   10  c 000c      f (2) =g;
   11  c 0015      f (3) =h;
   12  c 001E
   13  c 001E          do i=1 to 3;
   14  c 0030          put skip list ('Type x');
   15  c 004C          get list (x);
   16  c 0067          put list ('f(', i,')=',f(i) (x));
   17  c 00BD          end;
   18  c 00BD    stop;
   19  c 00C0
   20  c 00C0    h:
   21  c 00C0        proc(x) returns (float);
   22  e 00C0        dcl x float;
   23  e 00C7        return(2*x+1);
   24  c 00DB        end h;
   25  a 00DB    end call;
CODE SIZE=00DB
DATA AREA=0023
```

图 9-1 ENTRY变量和常数的使用

程序含有四个入口常数：标号为“call”的主程序；行5说明的外部过程“g”；“sin”函数，它是PL/I库的组成部分；从行20开始的内部过程“h”。一个入口变量，在行4的说明中称作“f”，它含有三个元素。从行9开始，分别对各元素初始化为常数sin、g和h，DO组给控制台提示出送到每一函数的值（行16的中间），每个函数正好由

f(i) (x)

调用一次，其中第一对括号定义下标，第二个括号中是实在变元。应注意，入口常数和入口变量的说明类似于文件常数和文件变量，所有说明为ENTRY类型的形式参数自动地假定为入口变量，对其他的所有情形，除了说明有VARIABLE关键字外，入口是常数。对于外部

过程说明则应用上述规则 (b)、(c) 和 (e)。另外,对于每一个形式参数必须与实际过程的说明相匹配，要保证RETURNS属性正好与函数子过程返回的形式相匹配。

9.2 分段编译举例

下面给出一个分段编译和连接编辑的完整的举例。特别是图9-2的程序与图9-3的程序一起形成了一个模块，在控制台交互处理而产生一个联立方程的解。考虑下面的具有三个未知数的一组方程：

$$a - b + c = 2$$
$$a + b - c = 0$$
$$2a - b = 0$$

值 a = 1、b = 2、c = 3 是该方程组的解；因为

$$1 - 2 + 3 = 2$$
$$1 + 2 - 3 = 0$$
$$2 * 1 - 2 = 0$$

图9-2列出控制台读入系数及方程组的解向量输入的交互处理。而图9-3则给出了分段子程序 "invert" 的编译，它在解方程组中用来完成矩阵转置，这两个过程的主要区别在过程语句首："inv" 过程是主程序，由OPTIONS (MAIN) 定义，而 "invert" 程序是一个子程序，它由主程序调用。参考图9-2，从行 16 开始的说明定义了外部入口常数 "invert"，它是在主程序的行46调用的，invert子程序的参数在行18说明为 "maxrow" 乘 "maxcol" 的浮点数矩阵，其中maxrow和maxcol 实际上是由行 7 到 8 所给出的字间常数。invert 子程序另定义有二个fixed(6)参数，但值不返回。

图9-3所示为invert过程，它有三个形式参数，称作 a、r 和 c，它在行 2 中定义并在

```
PL/I-80 V1.0, COMPILATION OF:INVERT1
L:List Source Program
% include 'matsize.lib';
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0, COMPILATION OF:INVERT1
   1  a 0000  inv:
   2  a 0006      procedure options (main);
   3  c 0006      % replace
   4  c 0006          true  by'1'b,
   5  c 0006          false by'0'b;
  6+c 0006      % replace
  7+c 0006          maxrow by 26,
  8+c 0006          maxcol  by 40;
   9  c 0006      dcl
  10  c 0006        mat (maxrow, maxcol) float (24);
  11  c 0006      dcl
  12  c 0006        (i, j, n, m)  fixed (6);
  13  c 0006      dcl
```

```
14  c 0006          var char (26) static initial
15  c 0006          ('abcdefghijklmnopqrstuvwxyz');
16  c 0006      dcl
17  c 0006          invert entry
18  c 0006              ( (maxrow, maxcol) float (24), fixed (6), fixed (6));
19  c 0006
20  c 0006      put list('Solution of Simultaneous Equations');
21  c 001D          do while (true);
22  c 001D          put skip (2) list('Type rows, columns: ');
23  c 0039          get list (n);
24  c 0052          if n=0 then
25  c 0059              stop;
26  c 005C
27  c 005C      get list (m);
28  c 0075      if n>maxrow | m>maxcol then
29  c 0087       put skip list ('Matrix is Too Large');
30  c 00A6      else
31  c 00A6          do;
32  c 00A6          put skip list ('Type Matrix of Coefficients');
33  c 00C2          put skip;
34  c 00D3              do i=1 to n;
35  c 00E8              put list('Row', i, ':');
36  c 0119              get list((mat(i, j) do j=1 to n));
37  c 0173              end;
38  c 0173
39  c 0173          put skip list('Type Solution Vectors');
40  c 018F          put skip;
41  c 01A0              do j=n+1 to m;
42  c 01B7              put list('Variable', substr (var, j-n, 1), ':');
43  c 01F3              get list((mat(i, j) do i=1 to n));
44  c 024D              end;
45  c 024D
46  c 024D          call invert (mat,n,m);
47  c 0253          put skip(2) list('Solutions:');
48  c 026F              do i=1 to n;
49  c 0284              put skip list (substr(var, i, 1), '=');
50  c 02B4              put edit((mat(i,j) do j=1 to m-n))
51  c 0314                  (f(8, 2));
52  c 0314              end;
53  c 0314
54  c 0314          put skip(2) list('Inverse Matrix is');
55  c 0330              do i=1 to n;
56  c 0345              put skip edit
```

```
 57  c 03AF                ((mat(i,j) do j=m-n+1 to m))
 58  c 03AF                (x(3), 6f(8,2),skip);
 59  c 03AF             end;
 60  c 03AF          end;
 61  c 03AF    end;
 62  a 03AF end inv;
CODE SIZE=03AF
DATA AREA=1120
```

图 9-2　矩阵求逆主程序清单

行 7 和 8 中说明。要注意maxrow和maxcol的实际字面值（分别对应于行和列的最大值）取自一个内含的文件，该文件由程序表中左边行号后符号“+”表示。

对这两个程序编译后，打入下列命令进行连接编辑：

```
link invert. com=inver1, invert2
```

它先将两个模块组合起来，再从PL/I库子选择必要的子程序，并将机器码结果存入文件invert.com中，图9-4所示为执行过程。

在会话举例中，为测试基本操作，首先进入“单位矩阵”，该输入值所形成的转置矩阵仍旧是单位矩阵。然后再进入上述所给的方程组，同时有两个解向量。该方程组的输出值在“Solution”下给出，与上述所给出的值相对应。第二个解集对应于输入的第二个解向量。此后，测试输入了一个非法矩阵，紧跟着输入一个零行矩阵而使程序终止。

所给出的完整举例的其他信息可参阅PL/I-80手册和LINK-80手册。

```
PL/I-80 V1.0, COMPILATION OF:INVERT2
L:List Source Program
%include 'matsize.lib';
   NO ERROR(S) IN PASS 1
   NO ERROR(S) IN PASS 2
PL/I-80 V1.0, COMPILATION OF:INVERT2
   1  a 0000  invert:
   2  a 0000      proc (a, r, c);
  3+c 0000      %replace
  4+c 000D            maxrow by 26,
  5+c 000D            maxcol by 40;
   6  c 000D     dcl
   7  c 000D           (d, a(maxrow, maxcol)) float (24),
   8  c 000D           (i, j, k, l, r, c) fixed (6);
   9  c 000D     do i=1 to r;
  10  c 0023     d=a(i, 1);
  11  c 0042        do j=1 to c-1;
  12  c 0059        a(i, j)=a(i, j+1)/d;
  13  c 00B2        end;
  14  c 00B2     a(i, c)=1/d;
```

```
15  c 00E4          do k=1 to r;
16  c 00FA          if k^=i then
17  c 0104             do;
18  c 0104             d=a(k, 1);
19  c 0123                do l=1 to c-1;
20  c 013A                a(k,l)=a(k, l+1)-a(i, 1)*d;
21  c 01B9                end;
22  c 01B9             a(k, c)=-a(i, c)*d;
23  c 021C             end;
24  c 021C          end;
25  c 021C    end;
26  a 021C  end invert;
CODE SIZE=021C
DATA AREA=0016
```

图 9-3　矩阵求逆子程序清单

```
A>b:invert
Solution of Simultaneous Equations
Type rows, columns: 3, 3
Type Matrix of Coefficients
Row      1: 1    0    0
Row      2: 0    1    0
Row      3: 0    0    1
Type Solution Vectors
Solutions:
a=
b=
c=
Inverse Matrix is
     1.00      0.00      0.00
     0.00      1.00      0.00
     0.00      0.00      1.00
Type rows, columns: 3, 5
Type Matrix of Coefficients
Row      1: 1    -1     1
Row      2: 1     1    -1
Row      3: 2    -1     0
Type Solution Vectors
Variable a :2     0     0
Variable b :3.5   1    -1
Solutions:
a=    1.00      2.25
```

```
b=     2.00      5.50
c=     3.00      6.75
Inverse Matrix is
       0.50      0.50      0.00
       1.00      1.00     -1.00
       1.50      0.50     -1.00
Type rows,   columns: 41,0
Matrix is Too Large
Type rows, columns: 0
End of Execution
```

图 9-4　矩阵求逆程序对话

第十节　PL/I-80 在商业处理中的使用

本节对用PL/I-80处理商业数据的一些技术，特别是对各种十进制算术操作作了一些详细描述；对定点十进制与浮点二进制之间的转换作了检查，包括ftc（浮点到字符）库函数的使用；讨论也包括了形象格式化输出的举例及使用四种基本的十进制操作时精度和标度的计算。

10.1　十进制和二进制操作的比较

我们从小就学会了用十个数字进行算术操作，那时允许的数字范围是从0到9；进而应用语言：如Basic、Fortran、Cobol和PL/I，允许我们以简单可读的形式写处理基本常数和数据项的程序，当然，计算机内部通常用二进制或十进制数字完成算术操作。由于 0 和 1 可以直接地由电子开关的开或关进行处理，如在算术处理器中见到的那样，所以，对于计算机的内部计算，用二进制更自然。但我们的程序通常是处理十进制值，所以就有必要在输入时转换为二进制形式而在输出时再转回到十进制形式。我们下面将看到，这种转换有可能发生截断错误，这在商业处理中是不能接受的，因此，为避免整个计算中的错误蔓延，常常要十进制计算。

在大多数语言中，程序员不必控制数字处理所用的内部格式。事实上，微处理器中两种最普通的Basic解释程序的区别主要是在内部数字格式上。一个使用浮点二进制，而另一个则用十进制算术计算。Pascal语言翻译程序通常用浮点和定点二进制格式，精度由版本确定。而Fortran则总是使用浮点或定点二进制进行计算。另一方面，Cobol是为商业应用而设计的，在所有计算中，必须保持有确切的美元和美分，因而用十进制计算处理数据项。

PL/I-80给了程序员选择表示法的灵活性，以使每一个程序能根据实际应用的确切需要进行选择。在PL/I-80中，定点十进制数据项用来完成商业功能，而浮点二进制项则用于科学处理，此时，计算速度是最重要的因素。下面给出的两个程序说明了两种计算形式的基本区别：

```
dec_comp:
    proc options (main);
    dcl
```

```
bin_comp:
    proc options (main);
    dcl
```

```
    i fixed,
    t decimal(7,2);
t=0;
    do i=1 to 10000;
    t=t+3.10;
    end;
put edit(t) (f(10,2));
end decimal_comp;
```

```
    i fixed,
    t float(24);
t=0;
    do i=1 to 10000;
    t=t+3.10;
    end;
put edit(t) (f(10,2));
end bin_comp;
```

这两个程序完成同样的功能，对3.10累加10000次，它们仅有的区别是“dec_comp”用定点十进制变量计算结果，而“bin_comp”则使用浮点二进制变量计算结果。dec_comp得出一个正确的结果31000.00，而bin_comp则产生一个近似值30997.30，这种不同是由于某些十进制常数，如3.10，转换为相应的二进制近似值时发生了内在的截断。使用定点十进制变量时并不发生截断，所以dec_comp产生了一个正确的结果。

这两个程序可以简化为一种更通用的情形来考虑，假定某银行在某一天中处理了10000个$3.10的存款，使用浮点二进制计算的程序，在最后一天将会有额外的$2.70无法入帐。这是由于.10不能表示为有限的二进制小数。也就是说，3.10实际近似为浮点形式3.0999999E+00，每一次加法都在和中增加一个小的误差，导致一个不正确的合计。在科学应用中，内在的截断错误通常无关紧要，因而被忽略。在商业应用中，这种内在错误则不能接受。

要注意，有些情况下，十进制计算也会产生截断错误，蔓延到整个计算中。例如，表达式1/3就不能表示为有限小数，因而近似为

0.33333……

到最大可能的精度。总之，由于我们对十进制计算有丰富的经验，可以预期这种错误的发生并调整程序来计入这种情况。事实上，我们知道，这种错误仅在有明显的除法操作时才发生。我们期望1/10，正好表示为.10而不调整其近似值。但这里出现一个用浮点二进制表示的困难，一些能用定点十进制表达为有限小数的十进制常数不能写作有限的二进制小数，因此，就必然会在转换为浮点二进制时发生截断。

有了这些介绍，现在我们确切地解释表示和处理定点十进制数字的方法。

10.2 PL/I-80 中的十进制计算

在PL/I-80程序中，可以进行定点十进制计算。与浮点格式相比，选用定点十进制计算有某些优点，也有缺点。首先，定点十进制运算保证不会丢掉有效数位，即所有的数位在计算中都作为有效数考虑，这样，在类似乘法这样的运算中不致使最低有效位截断。其次，定点十进制运算排除了指数处理的必要性，这样比较另一种十进制运算格式，操作相对快些。缺点是由于将所有数位都作为有效位，程序员必须始终掌握操作数进行运算的取值范围。下面的内容将对用定点十进制格式的程序提供必要的基本情况。

在PL/I-80中，十进制变量和常数具有“精度”和“标度”，精度指的是变量或常数中数字的位数，而标度则定义为小数部分数字位数。对于定点十进制变量和常数，精度不能超过15，标度不能超过精度。PL/I-80变量的精度和标度是在变量说明中定义的：

```
declare x fixed decimal (10,3);
```

而常数的精度和标度则由编译根据常数中的数位推导出来，且数值都带有小数点，例如常数

$$-324.76$$

的精度为5，标度为2。在内部，定点十进制变量和常数存作BCD对，其中每一个BCD数位或者在字节的高4位，或在低4位，BCD数位的最高有效位定义数值或常数的符号，其中0表示正数，按10的补码形式，9定义为负数（下面将要描述），由于数字是按8位的字节单元存放，为和偶数字节的边界对齐，将有额外的填充数据。例如，数字83.62存储为：

0 8	3 6	2 0

其中每一数位在8位值中表示为4位的半字节单元，前面的BCD对放在内存最低位。

为简化算术操作，负数按10的补码形式存放。10的补码其数值类似于2的补码的二进制表示，仅是数字x的补码值是9－x，数字的10的补码值可以先求得每一数位的补码（用9减去该数），再在最后结果上加1而得到。如－2的10的补码是：

$$9-2+1=7+1=8$$

该值已含有了数字的符号，内部是一个单字节值：

9 8

例如，做－2和＋3的加法如下：

$$98+03=101$$

忽略符号位左边的进位，则得相加的一个正确结果01。由此，在PL/I-80中，加法和减法是等价的。在做减法时，将减数转为补码，再进行加操作。在做算术运算前，所有的数值都扩充为带符号的15位值。为标记方便，负数前加有符号“-”，假定下列的表示法采用10的补码形式，这样，上述的数字写作：

－2

要注意，在内存中不需将小数点存入，因为每一变量和常数的精度和标度在编译时都成为已知的。每次算术运算前，编译的代码对操作数作必要的对正，但在后面的举例中，为更容易地确定对正的影响，常常给出了小数点的位置。例如，数字－324.76表示为：

－3	2 4	7 6

在准备对该值作算术处理时，首先将它装入一个8字节的堆栈结构，组成15位十进制数并在高位带符号。在这种情形下，－324.76表示为：

－ 0 0 0 0 0 0 0 0 0 0 3 2 4 7 6
^

讨论各种算术操作较为方便的形式是设想一个15位的机械或电子计算器，并带有手动的，小数点，在每次操作开始，必须对操作数进行排队以进行运算，且在完成运算时，决定结果中小数点的位置。实际上，小数点的位置由编译进行对正并进行计算，但设想一下实际发生的情况是有用的，这样可以避免发生上溢和下溢条件。在某些情况下，在用DECIMAL或DIVIDE内部构造函数计算期间，可能对精度和标度作一些改变。在下面讨论的样板程序中，将给出这种函数的例子。

首先，为确定在各种情形下产生的对正、精度和标度，我们先对每一算术函数进行检查。

10.3 加法和减法

如上所述，在PL/I-80中，加法和减法在功能上是等价的，因为减法是把减数变为10的补码再由加法运算完成的。给定两数x和y，精度分别是（p,q）和（r,s），加法的执行如下，首先将两操作数装入堆栈并对正，对正时将具有较小标度的操作数左移，直到两操作数的小

数点位置相同。设x的标度大于y的标度，则将y左移q-s位，并在低位补零。对正后，y的精度为r+(q−s)，标度为q（如果在对正处理时，有效数位移至符号位置，则发定点溢出信号)。

为提供一个例子，假定x=31465.2437，y=9343.412，这样，x的精度p=9，标度q=4，而y的精度r=7，标度s=3，在对正之前，数字的形式为：

```
                          |<-------p=9------->|
                                    |<q=4>|
x=+0 0 0 0 0 0 3 1 4 6 5 2 4 3 7
                                    ^
y=+0 0 0 0 0 0 0 0 9 3 4 3 4 1 2
                                    ^
                                    |<s=3>|
                                |<---r=7--->|
```

y的值将被左移q−s=4−3=1位而与x对正，结果为：

```
                          |<-------p=9------->|
                                    |<q=4>|
x=+0 0 0 0 0 0 3 1 4 6 5 2 4 3 7
                                    ^
y=+0 0 0 0 0 0 0 0 9 3 4 3 4 1 2 0
                                    ^
                                    |<  q  >|
                              |<---r+(q−s)=8--->|
```

注意，x的整数部分的数位是p−q，而y的整数位是r−s位。

```
|< p−q=5 >|
  3 1 4 6 5
    9 3 4 3
 |<r−s=4>|
```

这样，和的整数部分必须含有 p−q 位。

```
    3 1 4 6 5
+     9 3 4 3
  ------------
    4 0 8 0 8
  |<p−q=5 >|
```

注意，值足够大时将会产生溢出，在整数部分要再加一倍。

```
    9 9 9 9 9
+   9 9 9 9 9
  ------------
  1 9 9 9 9 8
|<(p−q)+1=6>|
```

这样，x与y的和的合计数位将是整数部分的数位 (p−q)+1=6再加小数部分的数位q，故结果的精度为：

$$(p-q)+1+q=p+1$$

给定两个具有任意精度和标度的值x和y，我们可按如上所述的特殊情况，导出通用的结果精度和标度。首先，标度必须是q和s中的较大者，由下式给出：

$$\max(q,s)$$

这样，结果的精度必须有max(q,s)的小数数位。其次，x的整数部分含有p−q位，而y的整数部分含有r−s位。结果含有p−q、r−s中较大者的位数加小数位数及一位溢出数位，则共有

$$\max(p-q,r-s)+\max(q,s)+1$$

个数位。由于在PL/I-80中，精度不能超过15，故结果的精度必须是：

$$\min(15,\max(p-q,r-s)+\max(q,s)+1)$$

将加法和减法结果的精度和标度写作一对，即是：

$$(\min(15,\max(p-q,r-s)+\max(q,s)+1),\ \max(q,s))$$

用上述的举例

```
                        |<------p=9------->|
                                  |<q=4>|
    x=+0 0 0 0 0 0 3 1 4 6 5 2 4 3 7
                                  ^
    y=+0 0 0 0 0 0 0 9 3 4 3 4 1 2 0
                                  ^
                                  |<  q  >|
                             |<r+(q-s)=8>|
  x+y=+0 0 0 0 0 0 4 0 8 0 8 6 5 5 7
                                  ^
                                  |<-4->|
                          |<-------10-------->|
```

其精度是（10,4），上面的图示可用下列表达式导出：

$$(\min(15,\max(9-4,7-3)+\max(4,3)+1),\ \max(4,3))$$

或

$$(\min(15,\max(5,4)+4+1),4)$$
$$=(\min(15,10),4)=(10,4)$$

10.4 乘法

乘法结果的精度和标度的计算类似于加法和减法，在做乘法之前不需要作小数点对正。两个分别具有精度和标度为（p，q）和（r，s）的操作数x和y，逐位相乘而得到结果。与简单的手工计算类似，在结果中小数点后位数的数量是标度因子q和s的和，结果的位数是两操作数精度之和。与PL/I标准相符合，在最后结果中包括有一位附加数位。这样，乘法结果的精度和标度由下式给出：

$$(\min(15,p+r+1),q+s)$$

例如，假定x=924.5，y=862.33，其精度和标度值为（4,1）和（5,2）

```
    x=+0 0 0 0 0 0 0 0 0 0 0 0 9 2 4 5
                                    ^
    y=+0 0 0 0 0 0 0 0 0 0 0 8 6 2 3 3
                                  ^
```

x和y的乘积及结果的精度和标度给出如下：

```
  x*y=+0 0 0 0 0 0 7 9 7 2 2 4 0 8 5
                                  ^
                                  |<3>|
                          |<-------10-------->|
```

其中精度计算如下：

$$(\min(15,4+5+1),1+2)=(\min(15,10),3)=(10,3)$$

如果乘积的有效位数大于15，则发出定点溢出条件信号。在前一小节的举例中，x=31465.2437，y=9343.412，乘积x*y的精度为（17,7），具有16位有效数字，将导致定点溢出。在这个特定的例子中，必须用内部构造函数DECIMAL，减少x或y的有效位数。即可执行如下的运算：

```
DECIMAL (x,9,3) * y
```

在进行乘操作前，装入堆栈的两个值为：

```
DECIMAL(x,9,3)=+0 0 0 0 0 0 0 3 1 4 6 5 2 4 3
                                      ^
            y=+0 0 0 0 0 0 0 0 9 3 4 3 4 1 2
                                      ^
```

乘积的精度和标度是：

```
          +2 9 3 9 9 2 7 2 9 0 2 9 1 1 6
                              ^
                              |<--- 6 --->|
          |<------------15----------------->|
```

应注意，计算的精度p+r+1值为16，而后减到PL/I-80的最大精度位15，即

$$\min(15,p+r+1)=\min(15,16)=15$$

由于含有乘法运算时精度增长很快，因而，程序员要确保所含的操作数的精度不致产生溢出。再次说明，精度可以由计算中包含的变量显式地定义，或用DECIMAL函数来减少临时结果的精度。

10.5 除法

如在10.1中所描述的那样，除法运算是四种唯一可能产生截断错误的运算。这样，每一除法运算都产生一个由15位十进制数组成的最大精度值，结果的标度值依赖于两个操作数的标度值。假定x和y有精度（p,q）和（r,s）且用x除以y，除法运算进行如下：首先，将x数左移到最左边，在右边加15－p个0，放在栈中的被除数为：

```
|<------ p ------>|<------ 15-p ------>|
     |<---q--->|
 _____________________________________
|x x · · · x x|0 0 · · 0 0|
  ^
```

然后再将x的小数点右移适当的量值与结果中的小数点位置对正，形成操作数：

```
|<------ p ------>|<---15－p--->|
       |<q－s>|<---15－p--->|
 _____________________________________
|x x · · · x x|0 0 · · 0 0|
  ^
                    |<------ r ------>|
                                |<s>|
 _____________________________________
|0 0 · · · 0 0|y y y · · y y|
                                  ^
```

然后再继续用x的有效数位除以y的有效数位，直到产生15位十进制数。参看上面的框图，应注意，由除法产生的小数部分的位数是由在x中调整的小数点位置所决定的。小数点后边

的字段含有（q－s）加（15－p）位。除法结果所得的精度和标度为：

$$(15,(q-s)+(15-p))$$

或 $(15,15-p+q-s)$

假定 x＝31465.243，y＝9343.41，其精度和标度分别是（8,3）和（6,2），装入后x的值为：

```
                    |<------ 8 ------>|
                              |<-3->|
x＝＋0 0 0 0 0 0 0 3 1 4 6 5 2 4 3
                            ^
```

而后，将x的值左移到极限位置并装入y的值形成如下的值：

```
        |<----- 8 ------> |<-15－8＝7->|
              |<-3-> |
x＝＋3 1 4 6 5 2 4 3 0 0 0 0 0 0 0
            ^
y＝＋0 0 0 0 0 0 0 0 0 9 3 4 3 4 1
                                ^
                            |< 2 >|
                      |<----- 6 ------>|
```

为与结果值的小数点对正，设想小数点向右移

```
        |<----- 8 ----->|<------ 7 ------>|
        |               |1|                |
x＝＋3 1 4 6 5 2 4 3 0 0 0 0 0 0 0
                  ^
y＝＋0 0 0 0 0 0 0 0 0 9 3 4 3 4 1
                          |<-----6------ >|^
```

二位，用形成的x的有效数位除以y的六位有效数，结果是：

```
                |<-------------------15-------------------->|
                |                          |<---1＋7＝8--- >|
x/y＝＋0 0 0 0 0 0 0 0 3 3 6 7 6 4 0 1
                          ^
```

在这种情况下，结果的精度和标度由下式给出：

$$(15,15-p+q-s)=(15,\ 15-8+3-2)=(15,8)$$

在十进制除法中考虑的重点是确保在所进行的计算中能形成足够多的小数位。小数数位是由二种方式产生的。第一，当被除数左对正以提供15－p位小数时要充填补零，这样，即可使被除数的精度减少而形成更多的小数位；第二，如果q大于s，如上所示，将形成q－s位附加的小数数位，另一方面，如果被除数所含的小数数位比除数的少，那么q小于s，则浪费（s－q）位小数位。经常出现的一种简单情形是q＝s，在这种特定条件下，小数数位完全取决于被除数的精度，即有15－p个小数位。

在特定的计算中，可以使用DIVIDE内部构造函数对结果截掉或扩充填零（参看“PL/I-80语言手册”），形式是：

DIVIDE (x,y,p,q)

其中p和q是字面常数，它可以作为算术运算中的表达式或子表达式出现，与下列语句具有

同样效果：

DECIMAL (x/y,p,q)

如上所述，值x被y除，但精度和标度值是（p,q)。注意，该运算按上面描述的方法进行，然后，为得到所期望的精度和标度，再将结果值进行适当的移位。

10.6 定点十进制与浮点二进制之间的转换

定点十进制值与浮点二进制之间的转换是很有用的。在PL/I-80中，这种转换是通过先转换为字符格式，再转换为定点十进制或浮点二进制来完成的。虽然在语言中直接提供了从定点十进制转换为字符再转换为浮点二进制这样的转换功能，但对从浮点二进制到字符的转换，提供了称作“ftc”的特殊库函数。该特殊函数在其他的应用中是很有用的，本节将对它作全面的描述。

作为数据格式之间转换的例子，考虑下面的程序：

```
conv:
  proc options (main);
  dcl
    ftc entry (float)
        returns (char(17) var);
  dcl
    d fixed decimal (8,2),
    f float binary;
d=-123456.78;
f=char(d);
f=0.314159265e1;
d=ftc(f);
end conv;
```

在该例中，定点十进制值d先被初始化为 −123456.78，然后，对该定点十进制值用CHAR内部构造函数形成一个字符串常数：

'b−123456.78'

其中b是空格字符。转换为字符后的赋值存入，实际是从定点十进制转换为浮点二进制。如前所述，由于转换为二进制，可能会产生截断错误。下一步，将π值存入浮点二进制值f中，通常，从f到d的赋值将使得小数部分截掉。因为PL/I标准要求先转换为定点二进制。代之，对f用了ftc函数，形成一个长度为17的字符串变量：

'b3.14159200000000'

其中如果数值为正，则前面是一个空格（用b表示），如果是负值，则是符号“−”。其后存入d中形成一个截掉后的值3.14，d说明小数点后有两位数字的标度值。要注意，浮点二进制表示法允许有约$7^1/_2$个有效十进制数位，这样，转换时有可能发生截掉错误。

第二章　PL/I-80语言手册

本PL/I-80语言手册提供了一个相对详细而又简要的关于PL/I-80语言的描述，供有一定经验的PL/I程序员使用。它作为本书中“PL/I-80应用指南”一章的补充。PL/I-80形式上以ANSI PL/I通用子集（子集G）为基础，该子集是按ANS PL/I标准化委员会X3J1说明的。PL/I-80与子集G规则的不同之处如下：

PL/I-80不包括下列属性：

DEFINED
FLOAT DECIMAL (FIXED DECIMAL 保留)
LIKE
PICTURE
FILE（在PL/I-80中，仅允许在OPEN语句中使用）
星号（*）域长和动态数组

PL/I-80不包括下列内部构造函数：

ATANH
DATE
STRING
TIME
VALID

PL/I-80增加了%REPLACE语句。

增加了处理ASCII文件的I/O功能：

变长ASCII记录的READ和WRITE格式
GET EDIT扩充为用A格式输入整个记录
在字符串常数中允许有控制字符

增加了下列的内部构造函数：

ASCII
RANK

本手册对PL/I-80的语句格式以最通用的形式进行描述，采用下述的标记约定：

大写字母单词表示PL/I-80的关键字。

小写字母单词或由小写字母和数字（间隔连字符）组成的单词将被描述或更明确地定义，这些词表示由用户所选用的变量信息。

中括号中的内容表示选择项。

省略号表示其前面的项可以出现一次，也可以连续出现多次。

除上述提到的字符外，所有其他的标点及特殊字符表示这些字符将实际出现。

第一节　基本结构

基本结构由 PL/I-80 程序源正文的低级组织构成，它所包括的内容除了语言字符集的列表外，还有构成标识符（包括关键字和说明名）、常数、定界符、注释和操作符的规则说明。

1.1　字符集

PL/I-80的字符集由大写字母、小写字母、数字及特殊符号组成。特殊符号及其主要用途描述如下：

=	等号或赋值号
+	正号
−	负号
*	星号或乘号
/	斜杆或除号
(	左括号
)	右括号
,	逗号
.	句点
'	单引号或撇号
%	百分号
;	分号
:	冒号
^	逻辑“非”号
~	选择“非”号
&	“and”号或“与”号
!	逻辑“或”号
\	选择“或”号
|	选择“或”号
>	大于
<	小于
_	下连字符
$	美元符
?	问号

1.2　标识符

标识符是一个由 1 到31个字符构成的字符串。这些字符可以是字母、数字或下连字符，其第一个字符必须是字母。在 PL/I-80 中，字母的内部表示全部用大写，因此，仅是大小写上不同的两个标识符将认作是同一个标识符。PL/I-80 也允许在标识符中嵌用问号字符以访问外部系统入口点，该字符在外部系统入口中是常使用的。通常，应避免嵌用问号字符，以保持与全集语言的向上兼容性。

PL/I-80程序源正文中的每一个标识符必须或者是关键字，或者是说明名。关键字是在PL/I-80语言中有特定意义的标识符。如内部构造函数的名称、语句名及数据属性等。说明名是由程序员在DECLARE语句中定义了其用途或意义的标识符（参看第三节）。关键字也可以作为说明名出现，即作为一个用户定义的标识符出现在说明中。在这种情况下，该标识符在PL/I-80程序中的意义取决于它所出现的形式和位置，即根据上下文而定。

1.3 常数

常数是不同于标识符的正文项，它有固定的字面意义，在PL/I-80程序执行期间是不能改变的。PL/I-80的基本常数有算术常数、字符串常数和位串常数。其中算术常数又可以是FIXED BINARY、FLOAT BINARY和FIXED DECIMAL。关于各种常数类型格式的详细描述参见第三节。

1.4 定界符和分隔符

在PL/I-80源程序正文中，重要的一点是要使程序正文的项是可区分的（如标识符等）。完成这种功能的正文项即称作定界符或分隔符。通常，定界符用来限定一个正文项，而分隔符则用来分隔正文项。在PL/I-80中，每一个标识符或算术常数之前和之后都必须有一个或多个分隔符或定界符。定界符既可以是间隔符、操作符、注释，也可以是一定的图形定界符。

间隔符：一个间隔符是指一个空格、一个制表字符（TAB）或行结束字符。

操作符：在PL/I-80中，有四种类型的操作符：

（1）算术操作符：

+	加或正号
−	减或负号
*	乘
/	除
* *	乘方

（2）比较操作符：

>	大于
~>	不大于
>=	大于等于
=	等于
~=	不等于
<=	小于等于
<	小于
~<	不小于

（3）位串操作符：

~	“非”
&	“与”
!或|	“或”

（4）字符串操作符：

!!或|| 并置

在上述操作符中，其中有些是所谓的复合操作符，如>=，在这种操作符的二个字符间不能留有空格。

注释：注释用来在程序中提供一些解释语句，对于程序的执行没有任何影响。注释可以放在能够出现定界符的任意位置。注释要以字符对/* 开始，以 */结尾。

图形定界符和分隔符：下面的一些特殊字符也可以起到定界符或分隔符的作用。有关它们用法的详细描述在后面章节中给出。

: 冒号。用作入口和标号常数的分隔符。

; 分号。用来结束一个语句。

, 逗号。用来分隔列表元素。

. 句号。用来分隔限定名中的项。

' 单引号。作为字符串常数和位串常数的定界符。

—> 箭号。作为指针限定引用的操作符。

= 等号。用作赋值语句中的操作符。

(左括号。

) 右括号。与左括号一起作为一对定界符使用。

第二节 程序结构

PL/I-80 是一种自由格式的分程序结构语言。其基本程序元素是语句。由语句可以组成更大的程序元素——组或分程序。下面将描述这些程序元素的组成规则。

2.1 PL/I-80语句

除赋值语句外，PL/I-80 所有语句都是由一个任选的标号，后跟一个关键字和语句体，再由分号终止这样的形式组成的。语句可以分为下列几类：

结构语句：定义分程序。

说明语句：描述数据。

可执行语句：定义某种操作。

空语句：不进行任何操作。

复合语句：用来形成一个简单语句的语句集合（如IF语句）。

赋值语句。

预处理语句：是编译时的指令。

每一类语句的具体构成将在本手册后面有关章节中讨论。

2.2 组

组是以DO语句开始，以END语句结尾的一个 PL/I-80 语句序列。有两种格式，迭代DO和非迭代DO。

非迭代DO的格式为：

〔标号:〕 DO;

语句-1

⋮

语句-n
END〔标号〕;

迭代DO的形式为：

〔标号:〕 DO语句
语句-1
⋮
语句 n
END〔标号〕;

下面举例说明组的用法：

```
first:
    do; /* non-iterative do */
     j = x;
    if x>0 then
        do; /* non-iterative do */
         x = z;
         z = j * y;
        end;
    else
         x = y;
    end;
    do i=1 to 10; /* iterative do */
     a(i)= i * j;
    end;
```

DO语句将在第八节中进一步讨论。

2.3 分程序

分程序是由BEGIN和END语句（BEGIN分程序）或由PROCEDURE和END语句（PROCEDURE分程序）所界定的语句序列。一个分程序可以嵌套在另一个分程序中，但不允许重叠。分程序用来限定程序中说明名的作用域。BEGIN分程序具有下列格式：

〔标号:〕 BEGIN;
语句-1
⋮
语句-n
END〔标号〕;

其中**语句-1**到**语句-n**是构成分程序体的任意的PL/I-80语句。注意，使用标号任选项并不能象某些全集语言那样自动地添加足够的END语句来使分程序配平。

PROCEDURE分程序具有下列格式：

标号:PROCEDURE-**语句**
语句-1

```
    ⋮
语句-n
END〔标号〕;
```

其中**标号**用来标识该过程，**语句-1**到**语句-n**是任意的PL/I-80语句。注意，END语句中的标号是任选的，但若包含它时，必须要与命名该过程的标号相同。

2.4 名字的作用域

变量的作用域通常是在程序中所定义的它的区域中。相对于变量所出现的分程序，它可以是局部的、全程的，也可以是外部的（EXTERNAL）。关于数据变量的作用域，有下面二条规则。

变量的作用域包括在说明该变量的分程序中，在该分程序之外是无效的。当某变量在一个分程序中说明后，便成为该分程序内的一个局部变量。

在一个分程序中所说明的变量，可以被嵌套在该分程序中的所有分程序所识别，该变量对于这些嵌套的分程序是全程的。但若在一个次级分程序中说明了一个与该变量相同的名字，则是在该次级分程序中引入了一个新的变量，它也就成为该次级分程序中的一个局部变量。

如果上述两条规则都不符合，则变量是无定义的。下面的程序说明了上述规则：

```
p1:
    procedure;
    declare
        (a, b) fixed bin (7);
        a=2; /*a is local to p1*/
        b=3; /*b is local to p1*/
p2:
    procedure;
    declare
        (c, b) fixed bin (7);
        b=2; /*b is local to p2*/
        c=a*b; /*c is local to p2*/
        a = a * b; /*  a =4 */
        end p2;
    put list (a, b);
    end p1;
```

上例不是可执行的程序段。在分程序 p2 中建立了一个新变量b，这是由于该变量在分程序 p2 中作了说明。put list 语句在 p2 分程序的外部，因此，变量 b 的值是 p1 分程序中的值 3 。由于标识符 a 在 p2中没有说明，则在 p2 中引用的变量 a 是在 p1 中说明过的全程变量。其值由 p2中的赋值语句作了改变。注意，在 p2 中说明的变量 c，在过程 p2 的外部是不能识别的。

任何说明为EXTERNAL属性的变量，对于所有的将该变量说明为EXTERNAL 的分程序及其内含分程序都是可识别的。重新说明它为没有 EXTERNAL 属性的分程序除

外。例如：

```
p1:
    procedure;
    declare
        z fixed binary external;
        ⋮
        p2:
            procedure;
            declare
                z fixed binary external;
                ⋮
                p3:
                    procedure;
                    declare
                        z fixed binary;
                        ⋮
                    end;
                end p2;
            end p1;
```

在p1和p2中的变量z引用同一个外部变量，但在p3中的变量z则是一个局部量，它与外部变量z不是同一个变量。注意，由于连接编辑的格式，所有的外部名将仅截取左边的六个字符。因此，要避免使用长的外部变量名，以免引起矛盾。

2.5 分程序的激活

BEGIN分程序与PROCEDURE分程序的主要区别是激活和终止的方式不同。BEGIN分程序按程序中的正常语句流程激活，在碰到其相应的END语句或程序的控制转到该分程序之外时终止。

PROCEDURE分程序仅在由CALL语句调用时激活。如果是函数过程，则是在表达式中引用该函数时激活。过程分程序在控制转回到调用点或函数引用点时终止。如果在正常的执行流程中遇到PROCEDURE分程序，则将被跳过。

2.6 预处理语句

PL/I-80允许在编译时通过使用预处理语句蕴含或修改源正文。预处理语句的标识为符号“%”后跟关键字：

INCLUDE 或 REPLACE

在编译时，%INCLUDE语句从一个外部的CP/M文件中拷贝PL/I-80源语句。语句形式为：

%INCLUDE '文件名';

其中文件名是要被拷贝到源程序中的CP/M文件的文件名。如果不给出驱动器名，则假定为含有该源程序的驱动器。被拷贝的内容则刚好替换掉%INCLUDE语句。被拷贝的部分本身并不要求是完整的语句。例如，%INCLUDE语句可以用来填写结构说明部分或格

式表。下面给出的程序段提供了一个例子：

```
F:
 PROC;
 DCL A FIXED;
 %INCLUDE    'STRU.CLIB';
 DCL C FLOAT;
 ...
 END F;
```

该程序将使PL/I-80编译在 %INCLUDE 语句处，从文件 STRUC.LIB 中拷贝源语句。

%REPLACE语句用来对所指定程序中的标识符替换为常量。其格式为

```
%REPLACE 替换名 BY 常数表达式;
```

其中**替换名**是一个标识符，其后，每当编译遇到该标识符时，便由**常数表达式**所指定的常量替换。**常数表达式**可以是任意的串或算术常数表达式。多个%REPLACE语句可以写作一个%REPLACE语句，其中各元素由逗号分隔。

注意，在 %REPLACE 语句中所定义的名字，并不遵守通常的作用域规则。因此，PL/I-80要求所有的%REPLACE 语句要出现在外部分程序级的任何嵌套的内部分程序之前。通常，所有的%REPLACE 语句紧接着过程语句书写，这样便可以容易地确定其作用域。例如：

```
%REPLACE TRUE BY '1'B;
```

将遇到的所有的**替换名**TRUE替换为常数位串'1'b。这样，语句

```
DO WHILE (TRUE);
```

由编译处理为：

```
DO WHILE ('1'B);
```

2.7 程序

如上所述，过程是一个由PROCEDURE语句和END语句界定的语句集合。并非嵌套在其他分程序之中的过程称作一个外部过程，被完全包含在一个分程序中的过程称为一个内部过程。PL/I-80程序的源程序可以由一个外部过程组成，其中可以含有嵌套的内部过程或分程序。每一外部过程可以分别编译并连接在一起形成一个PL/I-80目标程序。在组成一个程序的多个外部过程中，必须有一个外部过程是主过程，而其余的过程则可以是子程序过程或函数过程。主过程的格式为：

```
标号:
    PROCEDURE OPTIONS (MAIN);
      ⋮
    语句或分程序
      ⋮
    END 〔标号〕;
```

有关程序结构的进一步的详细描述和举例及分别编译、连接和装入的方法，参看“PL/I-80应用指南”和“LINK-80用户指南”。

第三节　数据项

PL/I-80程序中的数据项可以是常数，也可以是变量。常数是在程序的执行期间其值不能改变的数据项，而变量的值则是可以在程序执行期间改变的，程序中的数据项具有一定的附加特性，诸如下标值的范围、施用的操作或请求存储的量值等。这些特性称之为属性。它们是在DECLARE语句中指派给数据变量的。在某些情形下，如常数，其属性由系统缺省给定。数据变量可以是简单数据项，也可以是结构或数组。结构或数组是作为一个数据集合加以引用的。简单数据项变量或常数也称作标量。在PL/I-80中，有六种基本的数据类型：算术、串、指针、标号、入口和文件数据。下面各节将对每一种数据类型作详细的描述。

3.1 算术数据

PL/I-80支持三种类型的数字数据：FIXED BINARY，FLOAT BINARY和FIXED DECIMAL。每一种数字数据项都有相应的精度和标度值，由标记在括号中的整常数p和q表示。精度p指定数据可以含有的十进制或二进制数的位数，标度q则指定十进制或二进制的小数位数。变量的精度和标度可以在变量说明时显式指定，也可以由缺省规则隐含给定。

1. 定点二进制

说明为FIXED BINARY〔(p)〕的变量是一个具有p位二进制数的整数。p的范围是：

$$1 \leqslant p \leqslant 15$$

由于该类型的数据其内部表示为2的补码形式，故FIXED BINARY数的取值范围是－32768至32767。FIXED BINARY数的存储分配取决于p，如果$p \leqslant 7$，则给该项分配一个字节，否则，分配二个字节。其缺省精度是(15)，若赋予FIXED BINARY变量的值超出合法范围，其结果是不确定的。

FIXED BINARY常数以十进制整数方式书写。此时，仅有在由上下文关系中要求是定点二进制数时，这些常数才被认为是FIXED BINARY数据。如下标或与其他的FIXED BINARY数进行算术运算。其余的情况则缺省为FIXED DECIMAL，从其他类型的数据转换时，通常要发生数据的截断（转换规则参见第七节）。例如，在下面的程序段中，变量I的实际值将等于1。

```
DECLARE I FIXED BINARY;
I =1.99;
```

对于一个变量，说明为FIXED BINARY或FIXED BINARY与说明为FIXED BINARY(15)是等价的。

2. 定点十进制

除了由上下文确定使用为FIXED BINARY的常数以外，其余的十进制常数不管是否有小数点都将缺省为FIXED DECIMAL。说明为FIXED DECIMAL〔(p〔,q〕)〕的变量是一个带符号的十进制数，共有p个十进制数位，其中有q位小数位。FIXED DECIMAL数的取值范围r为：

$$-10^{**}(p-q) < r < 10^{**}(p-q)$$

其中：

$$1 \leqslant p \leqslant 15$$

$$且 \quad 0 \leqslant q \leqslant p$$

缺省精度及标度为（7,0)。十进制常数的缺省精度和标度则由常数本身的形式确定。例如：

3.25　缺省为　(3,2)

302　缺省为　(3,0)

十进制数的内部表示为压缩BCD格式，若用一个具有较大标度的值赋给一个较小标度的FIXED DECIMAL变量，则多出的标度位将被截掉，如果给变量所赋值的小数点左边的有效位多于所赋给的变量，则会发生FIXED OVERFLOW错误。

3．浮点二进制

FLOAT BINARY数有二部分：小数部分（尾数）表示数的有效数位；指数部分给出标度因子。对于一个说明为FLOAT BINARY(p)的变量，它包含有符号S，整指数e和p位二进制数给出数的小数部分。FLOAT BINARY数r的取值范围约为：

$$5.88 * 10 * * -39 \leqslant r \leqslant 3.40 * 10 * * 38$$

p的范围是

$$1 \leqslant p \leqslant 24$$

p的缺省精度是24。

FLOAT BINARY常数按科学计数法书写，它有一串十进制数字，带有任选的十进制小数点，紧跟着字母E（可以小写也可以大写），后面是一个带符号任选的十进制整数的指数部分。例如：

A=2.3E2;

即把230赋给A，变量说明为FLOAT与说明为FLOAT BINARY是等价的。

4．内部构造算术函数

除算术操作符外，PL/I-80还提供了下列一些内部构造算术函数作为语言的一部分：

ABS	ACOS	ASIN	ATAN
BINARY	CEIL	COS	COSD
COSH	DECIMAL	DIVIDE	EXP
FIXED	FLOAT	FLOOR	LOG
LOG10	LOG2	MAX	MIN
MOD	ROUND	SIGN	SIN
SIND	SINH	SQRT	TAN
TAND	TANH	TRUNC	

上述内部构造函数的详细描述将在第十二节中给出。

3.2 串数据

在PL/I-80中，有两种类型的串数据，字符串数据和位串数据。字符串的值是一个ASCII字符的序列，也可以是空序列，而位串的值则是非空的位序列。串的长度是指该串中具有的字符数或位数。串数据的使用规则描述如下。

1．字符串数据

说明为CHARACTER(n)的变量是一个长度为n的字符串。其中n可以是1～254。例如：

DECLARE A CHARACTER(10);

定义变量A是10个字符长的字符串。如果赋给A的字符串比A短，则在右边填空格直到A的长度；若字符串长于A，则右边的多余部分被截掉。

字符串常数写作由单引号引起来的一个字符序列。如果字符串本身的字符含有单引号，则以两个连续的单引号表示。这样，其值为 What's Happening? 的字符串常数表示为：

'What''s Happening?'

空字符串由连续的两个单引号定义，它的长度为0。

字符串变量也可以具有VARYING属性。它指定该变量可以表示变长串，其最大长度为n。如：

DECLARE A CHARACTER(10) VARYING;

定义了A表示任意的长度不大于10的字符串值。

PL/I-80允许在字符串常数中含有控制字符。通常，在字符串常数中出现上箭号"^"时，即表示后面跟随控制字符。它将字符的高位的三个bit置为0。这样，串'^M'或'^m'被转换为回车符。同理，'^I'转换为横向的tab字符。字符串中连续出现二个'^'字符时，转换为单个的'^'字符。注意，上箭号字符的这种用法通常在全集语言中是行不通的。当要求兼容性时，要避免使用。

2. 位串数据

说明为BIT(n)的变量是一个含有n位二进制数的位串数据项。其中n的值可以是1～16。

DECLARE A BIT(3);

定义了一个长度为3的位串。位串的赋值与字符串的赋值规则大致相同，但是在填充时用的是零而不是空格。位串变量不可以有VARYING属性。

位串常数写作由数字0～9和字母A～F组成的序列，并用单引号引起来，其后接着写字母B和任选的一位数字。数字范围为1～4。它指定序列中每一位所表示的位数。也就是说位串常数可以写作基数2(B或B1格式)、基数4（B2格式)、基数8（B3格式）或基数16（B4格式)。序列中的字母或数字必须与格式中给出的基数相容。下面举例说明各种选择格式与等价的基数2格式表示：

'101'B1 等价于 '101'B
'101'B2 等价于 '010001'B
'101'B3 等价于 '001000001'B
'101'B4 等价于 '000100000001'B
'9A'B4 等价于 '10011010'B
'77'B3 等价于 '111111'B

3. 并置

中缀操作符|| 或!!可用来并置位串或字符串，其两个操作数必须是同类型的，该类型也就是结果的类型。结果串的长度是操作数长度的总和。对于字符串的并置，若任意一个中有

VARYING属性，则其结果的属性也将是VARYING。在下面的举例中：

```
DECLARE
  A  CHARACTER(3),
  B  CHARACTER(6),
  C  CHARACTER(20)  VARYING;
A='ABC';
B='ABCDEF';
C=A || B;
```

长度为9的字符串'ABCABCDEF'赋值给了变量C。

4. 内部构造串函数

在PL/I-80中，包括有下列内部构造串函数。它们将在第十二节中描述。

ASCII	BITS	BOOL
CHARACTER	COLLATE	INDEX
LENGTH	RANK	SUBSTR
TRANSLATE	UNSPEC	VERIFY

3.3 控制数据项

控制数据项与问题数据项不同，它的作用是控制程序的流程。用来作为PL/I-80语句或过程的标号的标识符便是控制数据项的一种例子。

1. 标号数据

标号数据由标号常数和标号变量组成。标号常数是一个在可执行语句前的标号标识符，标号变量是一个在DECLARE语句中定义为具有LABEL属性的变量。标号变量可以由标号常数或另外的标号变量进行赋值，其赋值规则与别的类型的变量赋值相同。

标号常数和标号变量具有与说明名相同的作用域。一个标号数据项仅在用DECLARE语句显式说明或作为标号常数使用而隐含说明的分程序中才可识别。尤其是使用GO TO语句进行控制的传递，仅当给出的目标标号是在出现该GO TO语句的分程序中能够被识别时，才是合法的。这样，用GO TO语句和标号进行控制的转移时，就局限于当前活动的分程序或内含的分程序中。

标号常数可以带有由简单整常数（可以有符号）组成的下标。在一个单一的分程序中出现的所有具有相同标识符的带下标的标号，构成该分程序内的一个隐含说明的常数标号数组。而在其他分程序及内含分程序中所出现的相同的标号名则定为相应分程序中的一个新的局部说明标号。所有这种隐含定义的常数标号数组仅对于那些在相应的分程序中出现的下标才有定义。标号变量可以在DECLARE语句中显式定义为下标数组。

标号数据所能使用的操作符仅有等于和不等于的比较操作符。

2. 入口数据

入口数据项可以是入口常数，也可以是入口变量。PROCEDURE语句的标号即称作入口常数。入口常数可以是外部的（入口指向一个外部过程）或内部的（入口指向一个嵌套过程）。说明为ENTRY VARIABLE的变量即是一个入口变量，它可以被赋予一个入口常数或其他入口变量值。和标号数据一样，用于入口数据的操作符仅是等于或不等于的比较操作符。入口变量也可以是带下标的。下面的例子说明了入口数据项的用法：

```
DECLARE
  A ENTRY VARIABLE,
  (X, Y) FLOAT BINARY,
  F(3) ENTRY (FLOAT)
       RETURNS(FLOAT) VARIABLE,
  ZZ  ENTRY (FLOAT) RETURNS(FLOAT);
P 1:
  PROC;
  X=5;
  END;
P 2:
  PROC;
  X=25;
  END;
Y=9;
IF Y=5 THEN
  A=P 1;
  ELSE
  A=P 2;
CALL A;
F(2)=ZZ;
Y=F(2)(X);
PUT LIST(Y);
```

ENTRY数据项将在第八节中作进一步的描述。

3.4 指针数据

指针数据用来寻址内存中的某一单元，它的值是程序中某一变量的地址。指针变量按下列方式说明：

```
DECLARE X POINTER;
```

指针变量可以被赋值予其他的指针变量。但如同标号和入口数据一样，对于指针数据所定义的操作也只有=和~=。特殊类型的变量在引用时必须用指针数据来限定。指针限定引用的格式是：

指针变量→有基变量

其中**指针变量**指向**有基变量**的地址。例如：

```
DECLARE  P  POINTER;
DECLARE  X  CHARACTER(2) BASED;
    ⋮
P->X='AA';
```

指针数据项也将在第六节中讨论。

3.5 文件数据

文件数据项用来表示与外部设备相关联的信息。PL/I-80使用文件常数和变量来存取外部数据集。文件常数的说明如下：

DECLARE **文件名** FILE;

其中**文件名**是一个表示文件名的PL/I标识符。在文件名不是一个参数的情况下，文件名自动地作为EXTERNAL，这样便使得在说明过它的所有模块中存取同一个数据集。此外，如果没有执行具有TITLE任选项的OPEN语句，则存取缺省驱动器上的命名为"**文件名**.DAT"的CP/M磁盘文件。

文件变量的说明如下：

DECLARE 文件名 FILE VARIABLE;

文件数据在第九节及"PL/I-80应用指南"一章中有更详细的介绍。

第四节　数据集合

在PL/I-80中，数据可以组合在一起形成数组或结构。能够表示一组数据元素的变量或者是数组变量，或者是结构变量。它们是作为数据集合而被引用的。

4.1　数组

数组是数据项的有序集合，它的各元素具有相同的属性。整体的数组可以由数组名引用，而数组的元素则通过使用整形下标而引用。下标表示元素在数组中的相对位置。在定义一个数组时，必须给出下列特征：元素的数据类型，数组的维数及每一维的域长或元素的数量。数组的每维的总和决定了数组中元素的总数。数组的维数是在DECLARE 语句中用变量名后的维数属性表来定义的。简单的维数属性表由一个正的整常数表构成。每一数字对应一维，指定了该维的下标域长，数字之间由逗号分隔，整个维数表括在圆括号中。例如：

DECLARE A(3, 4) CHARACTER(2);

定义了一个数组。它的元素是长度为2的字符串，其维数是2。第一维的域长是3，第二维的域长是4。这样，A便是一个由长度为2的字符串组成的具有3行4列的数组。

对应于某一维的下标的取值范围通常由该维的域长所隐含。在数组A中，行的下标可取1～3的值，而列下标的取值则为1～4。每一维下标的取值范围也可以用一对格式为m:n的整数代替域长而显式定义。其中m表示该维下标的下界，n表示上界。m和n可以是任意的整数值，只要m的值不大于n即可。例如：

DECLARE B (−2:5, −5:5, 5:10)
FIXED BINARY;

定义了数组B是一个三维数组，它的下标范围分别是－2到5，－5到5和5到10。其相应的域长分别是8，11和6。因此，B包含有定点二进制类型的528个数据项。

数组元素的内部存储是按横向为主的顺序存放的。数组元素的引用由数组的名字后跟括在括号中的下标表达式表，各表达式由逗号分隔。表达式的值指定元素在数组中的位置，下标表达式的数据类型必须是FIXED BINARY，对于数组的每一维必须有一个表达式，每一下标表达式的值必须在该维下标所定义的范围内。

对于定义有相同的数据类型、维数和下标范围的两个数组变量，一个数组变量可以直接赋值给另一个数组变量而不用下标。

内部构造函数DIMENSION、HBOUND和LBOUND可分别用来取得数组的每一维的域长、上界和下界。有关这些函数的详细描述，参见第十二节。

4.2 结构

结构是数据项的分层有序集合。包括在结构中的数据项称作该结构的成员，它们并不要求具有相同的类型，甚至可以是数组或其他的结构(子结构)，第一层的结构称作主结构，其他层次的结构称作子结构。

定义一个结构要指定主结构的名字及其各成员的名字和数据属性。每一名字要带有层号以定义它在分层序列中的层次。层号置于名字之前，并以一个或多个空格分隔。主结构的层号一定是1，各成员（包括层号、名字和属性）的定义必须以逗号分隔。子结构的成员的层号必须要大于子结构的层号。结构名不能有数据类型属性，但可以有维数属性（见下节）、EXTERNAL、STATIC或INITIAL属性。例如，对于一个帐单结构可以定义如下：

```
declare
   1 bill,
      2 name,
         3 last_nm      character(20),
         3 first_nm     character(20),
         3 mid_init     character(1),
      2 address,
         3 street       character(20),
         3 city         character(10),
         3 state        character(3),
         3 zip          character(5),
      2 charges
         3 shop    fixed    decimal (10, 2),
         3 snkbar  fixed    decimal (10, 2),
         3 misc    fixed    decimal (10, 2),
         3 dues    fixed    decimal (10, 2);
```

由于一个结构的成员的名字可以作为另一个结构的成员的名字或是同一结构中的一个子结构的数据项的名字，则在引用该名字时会有二义性。也只有在成员的名字所定义的公共作用域内时，会产生二义性。为解决这种二义性问题，在引用结构成员时使用限定名。在限定名中，成员的名字前面按层号的升序置以结构名字表。每一项由一个句号和任意个空格分隔，所需的结构名仅能确定对成员名唯一引用即可。例如，考虑下列的结构：

```
DECLARE
   1 A,
     2 B,
       3 C FIXED,
       3 D FIXED,
     2 BB,
       3 C FIXED,
```

```
3 D FIXED;
```

对于项C或D的引用是二义的。限定名B·C或B·D或BB·C能唯一地标识结构元素。注意，全限定名应是：

```
A.B.C
A.B.D
A.BB.C
A.BB.D
```

4.3 结构数组

正如结构的成员可以是数组一样，也可以定义结构数组。定义一个以简单结构类型为元素的数组，只要在定义结构时对结构名给出维数属性即可。类似地，子结构也可以给出维数属性。例如：

```
DECLARE
  1 STULIST (100),
    2 STUNAME,
      3 LASTNM    CHARACTER (10),
      3 FIRSTNM   CHARACTER (10),
      3 MID_IN    CHARACTER (1),
    2 SSN         CHARACTER (9),
    2 COUNTRY     CHARACTER (10),
    2 GRADES (5) CHARACTER (2);
```

定义了一个结构数组，其主结构名是STULIST，数组的每一结构元素具有数组GRADES作为其一个成员。引用数组中的一个入口时，对于具有维数属性的结构名和有维数属性的成员名、限定名要与下标一起使用。下标并不一定要求与其相应的名字出现在一起，但必须要写在括号中并用逗号分隔，且必须按正确的顺序出现。

例如，对于数组STULIST的第61个入口的第3个GRADES入口的引用可按下列任意形式出现：

```
STULIST (61) .GRADES (3)
STULIST.GRADES (61, 3)
STULIST (61, 3) .GRADES
```

第五节 数据属性和说明语句

除了常数数据和内部构造名字外，每一个程序数据项必须通过使用DECLARE语句及数据属性显式地定义。

5.1 说明语句

在程序中，除了内部构造函数名或伪变量名外，其他所有的变量名必须要在DECLARE语句中定义。文件常数和变量也必须要在DECLARE语句中定义。控制常数，诸如语句标号和过程名，由它们在程序中的使用而隐含说明。

与一名字（描述为一个数据项）相关联的特性称作该名字的属性。属性可以由缺省而隐

含定义或在DECLARE语句中由说明项显式地定义。DECLARE语句的通用格式是：

```
DECLARE | DCL
    〔层〕名〔属性表〕…
    〔，〔层〕名〔属性表〕〕；
```

其中，“名”是变量标识符，“层”是用于结构说明中的一个正整数，如上节所述。

属性表描述名字的特征。如果忽略属性表，则属性缺省为FIXED BINARY (15,0)，DECLARE语句举例如：

```
DECLARE
    A CHARACTER (2) BASED,
    B FIXED BINARY INITIAL (0),
    C (100) FIXED DECIMAL (5,2);
DCL
    1 BOOK,
      2 TITLE       CHAR (20),
      2 AUTHOR,
        3 LASTNM    CHAR (10),
        3 FIRSTNM   CHAR (10),
      2 PUBLISHER CHAR (20);
```

对于每种数据类型和数据集合的单独说明举例已在第三节和第四节中给出。

被说明名的作用域是程序的一个区域，在该域中，名字及其相关的属性可以被识别。通常，名字出现在DECLARE 语句中即隐含地定义了说明名的作用域是内部的，即在出现该说明的分程序中。如果名字具有EXTERNAL 属性，那么它的作用域将是每一个把它说明为具有EXTERNAL属性的分程序。如果一个变量被说明有 EXTERNAL 属性，则它也必须有 STATIC 属性，这是因为该名字将与一个单独生成的存储相关联。除了作为结构的成员名字外，一个名字在同一分程序中的说明不能多于一次。这样，才能对所出现的每一名字明确地引用。对于具有EXTERNAL属性的名字，在各分程序中对它的说明不能有相冲突的属性。

在PL/I中，为简单方便起见，DECLARE 语句允许具有替换形式。首先，一般的如下形式的DECLARE语句：

```
DECLARE  定义-1；
DECLARE  定义-2；
   …
DECLARE  定义-n；
```

可以被写作如下的等价形式：

```
DECLARE 定义-1，定义-2，…定义-n；
```

其中，每个定义项用逗号和任意个空格分隔。DECLARE语句以分号终止。其次，由几个项定义所共有的属性可以用因子形式提到右边。也就是说，下列形式的一系列定义：

```
项-1 属性-A，项-2 属性-A，…项-n 属性-A
```

可以写作等价的因子形式：

(项-1，项-2，…项-n)属性-A

也允许重复应用本规则。如：

```
DECLARE
  ((A, B)FIXED BINARY,
    C FLOAT BINARY)
    STATIC EXTERNAL;
```

等价于：

```
DECLARE
  A FIXED BINARY STATIC EXTERNAL,
  B FIXED BINARY STATIC EXTERNAL,
  C FLOAT BINARY STATIC EXTERNAL;
```

通常，属性的顺序是无关紧要的，有一点例外，即数组的维数表属性必须要跟在数组名的后面并置于其他属性之前，而结构成员的层号则必须要在其成员名字之前。它们的属性都可以用因子形式书写。由于维数表要求在所描述的名字的右边，它作为因子时，要置于右边，由于层号在它们的成员名字的前面，作为因子时便置于左边。下面形式的序列：

层-k 项-1，层-k 项-2，…层-k 项-n

等价于：

层-k (项-1，项-2，…项-n)

例如：

```
DECLARE
  1  A BASED,
  2  (B FIXED BINARY, C CHARACTER(2));
```

等价于：

```
DECLARE
  1  A BASED,
    2  B FIXED BINARY,
    2  C CHARACTER(2);
```

属性表不能包含有相冲突的属性，如两种数据类型或二种存储类别属性。但可以省略一系列属性，即有足够的属性刻划数据项的特性即可。在这种情况下，由PL/I-80编译根据缺省规则提供。

如果没有给出属性，则假定为FIXED BINARY(15)。

如果给出了DECIMAL或BINARY而没有FIXED或FLOAT，则假定为FIXED。

如果给出了FIXED或FLOAT而没有DECIMAL或BINARY，则假定为BINARY。

若对于FIXED BINARY没有指定精度，则假定为FIXED BINARY(15)。

若对于FIXED DECIMAL没有指定精度，则假定为FIXED DECIMAL(7, 0)。

若对于FLOAT BINARY没有指定精度，则假定为FLOAT BINARY(24)。

5.2 数据属性

下面将给出与程序数据相关的允许使用的属性及其属性的缩写。更详细的属性将在与其有关章的节中讨论。

ALIGNED是一个数据属性，它使得变量与存储边界对正。在PL/I-80中，它是没有

影响的。

AUTOMATIC是一个存储类别属性，它指定根据含有变量说明的分程序的激活来为变量分配存储。在PL/I-80中，除递归外，自动存储是静态地进行分配的。

BASED 或 **BASED**(p)或**BASED**(q( ))是一个存储类别属性。它指定对变量的分配是用户控制的。在这种情形下，p是指针变量，q是指针值函数。

BINARY 或 **BINARY**(p)和 **BIN** 或 **BIN**(p) 定义一个精度为p的BINARY变量。

BIT(n)定义一个长度为n的位串。

BUILTIN表示被说明的名字是语言的内部构造函数之一。如果在某一分程序中对内部构造函数进行了重定义，而后，为了要在其内含分程序中引用该内部构造函数，该内部构造函数名必须在该内含分程序用上述属性重新说明。

CHARACTER(n)和CHAR(n)定义一个长度为n的字符串。

DECIMAL〔(p〔,q〕)〕和DEC〔(p〔,q〕)〕定义一个具有精度(p,q)的DECIMAL数。若不给出q，则假定它为零。缺省精度是(7,0)。

ENTRY〔(**属性-1，属性-2，…，属性-n**)〕定义入口值。其中的**属性-1**至**属性-n**是参数的属性表，这些参数是在PROCEDURE入口值的定义中给出的。

ENVIRONMENT(**任选**)或ENV(**任选**)定义定长和变长的RECORD文件。其中**任选**可以是下列之一：

F (lrecl)

B (lbuff)

F (lrecl), B (lbuff)

表达式lrecl是定长记录文件的记录长度，lbuff是系统缓冲区的大小。

EXTERNAL或**EXT**定义所说明的项的作用域是EXTERNAL，即该项在所有的将其说明为EXTERNAL的分程序中是可以识别的。

FILE定义文件数据。文件属性将在第十节中讨论。

FIXED〔(p〔,q〕)〕定义精度为(p,q)的定点算术数据。

FLOAT〔(p)〕定义精度为p的浮点算术数据。

INITIAL(**值表**)和**INIT**(**值表**)使得在程序执行之前给一个STATIC变量赋初始值。"**值表**"是常数表，以逗号分隔。它们可以被转换为将要初始化的变量类型。表中任一常数都可以在前面置有括在括号中的重复因子。

LABEL定义LABEL变量。**POINTER**定义POINTER变量。

RETURNS(**属性表**)与ENTRY属性一并给出，描述由函数所返回的值的属性。

STATIC是一个存储类别属性，它使得在程序执行之前分配存储。

VARIABLE与FILE或ENTRY属性一起使用。它将该项定义为一个变量，而不是一个FILE或ENTRY常数。

VARYING和**VAR**定义变长字符串。

第六节　存储管理

PL/I 允许对数据域的存储分配进行控制。存储可以在编译时静态分配，也可在执行 期间动态分配。动态分配的存储其后可以释放以供新的使用。在程序中每一变量都有一个存储类别属性，存储类别决定了变量分配存储的方式和时刻。PL/I-80 包括三种不同的存 储 类别：

STATIC
AUTOMATIC
BASED

为改善内部寻址结构，AUTOMATIC存储按与STATIC存储同样的方式处理，但标为RECURSIVE的过程除外。存储类别属性是元素、数组和主结构变量的一种特性，这些属性不能定义给入口名、文件名或数据集合的成员。

6.1 STATIC属性

说明有STATIC属性的变量是在主过程执行之前分配的。属于这种存储类别的变量可以有初始数据值。下面的说明中给出了STATIC属性：

```
DECLARE A FIXED STATIC;
DECLARE B (100) CHAR(2) STATIC;
DECLARE 1 C STATIC;
          2 D CHAR (10),
          2 E FIXED;
```

6.2 INITIAL属性

INITIAL 属性用来在分配存储时对数据项指定初始常数值。该初始化数据将成为程序模块的一部分在程序开始执行时装入。INITIAL属性的形式是：

INITIAL | INIT （**值**〔,**值**〕…）

其中初始值(**值**)的形式为：

〔(**重复因子**)〕**常数表达式**

重复因子是字面常数重复次数，它将**常数表达式**的值重复指定整数次。**常数表达式**必须是字面常数值，且要与被初始化的数据类型相容，由任选的带符号算术常数、串常 数 或 NULL 指针值组成。数组数据项的初始化可以用单个的语句，以数组的第一个元素开始，再按以行为主的顺序继续直到初始常数集的末尾。数据的项数不能超过所初始化的数组的大小。结构成员必须分别地进行初始化。给变量赋以常数与赋值语句的规则相同。如用一个较短的串赋值给一个较长的串变量时，在右边填充空格。另外，为与全集语言相兼容，仅STATIC变量可以有INITIAL属性。INITIAL属性的举例如下：

```
DECLARE A FIXED BINARY STATIC INITIAL (0);
DECLARE B (8) CHARACTER (2)
          INITIAL ((8)'AB') STATIC;
DECLARE
  1  FCB  STATIC,
```

```
2  FCBDRIVE FIXED (7)   INITIAL (0),
2  FCBNAME CHAR (8)   INITIAL ('EMP'),
2  FCBTYPE CHAR(3)  INITIAL('DAT'),
2  FCBEXT BIT(8)  INITIAL ('00'B4),
2  FCBFILL (19) BIT(8);
```

6.3 AUTOMATIC属性

通常，AUTOMATIC属性使数据存储根据入口分配给变量所出现的PROCEDURE或BEGIN分程序。在PL/I-80中，AUTOMATIC存储是静态地分配的，以改善变量的寻址和执行速度。有一点例外是有递归时，其AUTOMATIC变量必须使用动态存储技术，以防止在递归调用时数据的覆盖。注意，对于一个没有被说明为STATIC或BASED的变量，其缺省的存储类别属性是AUTOMATIC，除非对其另有说明。

6.4 BASED属性

说明具有BASED属性的变量，其存储是通过ALLOCATE语句显式地给出的，每当分配一个BASED变量时，相应的POINTER变量即被置为所分配的BASED变量的地址。BASED属性的格式为：

BASED〔(**指针引用**)〕

其中"**指针引用**"是一个无下标的POINTER变量或返回一个POINTER值的无变元函数调用。

如果一个变量说明有BASED属性，但省略了指针引用，那么，对于该变量的每次引用，都必须要包含一个如下格式的指针限定：

指针表达式->变量

其中**指针表达式**是一个指针值表达式，它为所引用的变量确定存储。如果包含有指针引用任选项，那么，每当在没有指针限定而引用该变量时，它便隐含地用作基值。在这种情况下，指针引用在每次非限定变量出现时都要重新求值。在指针引用中所给出的指针变量或指针值函数名是由BASED说明确定其作用域的。即使存在有多个具有相同的指针引用名字的局部说明时也是如此。指针限定的引用具有如下格式，BASED变量说明举例如下：

```
DECLARE A CHAR(8) BASED;
DECLARE B POINTER BASED(Q);
DECLARE C FIXED BASED(P);
DECLARE D BIT(8)  BASED(F( ));
```

6.5 ALLOCATE语句

ALLOCATE语句用来为BASED变量指配存储。具有形式：

ALLOCATE **有基变量** SET （**指针变量**）;

其存储是从动态存储域获得的，大小足以容纳有基变量的值。如果所要求的存储不能满足，则会产生ERROR条件。**有基变量**必须是一个无下标的变量引用，变量的说明要在有BASED属性的ALLOCATE语句的作用域内，所分配的地址存放于在SET项中所命名的**指针变量**中。重要的一点值得注意，用这种方法所分配的存储要在相应的FREE操作发生后才能释放，该FREE操作是用指针变量所持有的分配地址作为操作数的。下面的程序段说明了具有ALLOCATE语句的BASED属性的用法：

```
DECLARE
  (P,Q) POINTER,
  X CHARACTER(2) BASED,
  Y FIXED BINARY BASED(P);
    ...
  ALLOCATE X SET(Q);
  ALLOCATE Y SET(P);
    ...
  Q->X='AB';
  Y=Y+1;
```

6.6 NULL内部构造函数

NULL内部构造函数返回一个指针值。该值是一个唯一的非法存储地址。它在标志各种指针值为空时是很有用的。特别是用在构造连接表时标志表元素结束。NULL 函数的两种调用格式是：

NULL 和 NULL( )

应当注意，在程序启动时，指针值并非必须以NULL值开始，除非NULL 值被用在变量说明的INITIAL任选项中。BASED变量及NULL函数的使用在“PL/I应用指南”中有更多的描述。

6.7 ADDR内部构造函数

ADDR 内部构造函数返回一个指针值。它取得作为变元给出的变量名所占据的存储段的地址，具有形式：

ADDR（**变量名**）

注意，**变量名**必须已分配内存地址，且不能是通过使用某些函数或操作所建立的临时结果。

在PL/I-80中，BASED变量与ADDR 内部构造函数一起使用，可以使得存储共享。在这种时候，有基变量并不用ALLOCATE 语句显式地给出存储，而是覆盖在一个已存在的变量上。使用ADDR函数，有基变量的指针基值被置为在ADDR函数中所用的已存在的变量的地址。其后，存取该有基变量便是存取它所覆盖的变量。下列的程序段说明了存储的共享，此时，字符串的值由位串矢量所覆盖，程序所输出的是字符串值的十六进制位串格式。

```
DCL
  I FIXED,
  P POINTER,
  A CHAR (8),
  B (8)  BIT (8)  BASED (P);
  P=ADDR (A);
  GET LIST (A);
  PUT EDIT ((B (I) DO I=1TO8))
       (B4(2));
```

6.8 FREE语句

BASED变量保留所分配的存储，直至使用FREE语句后才释放。FREE 语句的格式是：

FREE〔指针变量—>〕有基变量；

其中**指针变量**指定了存储的地址。该存储必须是先前用ALLOCATE 语句从动态存储域获得的。如果在FREE语句中不指出指针变量，那么，有基变量必须是用指针引用任选项说明的。此时，由指针引用所定址的存储退回给动态存储域。当通过FREE语句释放后，维护动态存储域的运行子程序自动地将相邻的存储段连接起来。下面以一个无实际功能的程序段给出FREE语句的举例：

```
DECLARE
  (P, Q, R) POINTER,
  A CHARACTER (10) BASED,
  B FIXED BASED (R) ;
    ...
  ALLOCATE A SET (P) ;
  ALLOCATE B SET (R) ;
  ALLOCATE A SET (Q) ;
    ...
  FREE P—>A;
  FREE Q—>A;
  FREE B;
```

第七节　赋值与表达式

赋值语句用来将变量置为表达式或常数的值。赋值语句不含有区别性的关键字．具有通用形式：

变量 = 表达式；

其中**变量**可以是一个标量元素、一个数组、一个结构名或一个伪变量。**表达式**的结构将在下面给出。

7.1 表达式

表达式是操作数和操作符的组合，它在运行时的计算中产生一个单一的值。语法规则决定着在表达式中所用的参照符、操作符和括号的排布。参照符可以是一个常数、变量或一个函数。操作符定义用操作数所进行的计算。括号可用来括住表达式的各部分。关于操作数、操作符和括号的完整的阐述将在下面给出。

1．前缀表达式

前缀表达式由一个一元操作符后跟一个操作数构成。先计算操作数的值，再将操作符应用于结果。前缀表达式的两个举例如下：

~A　和　−SQRT (B)

2．中缀表达式

中缀表达式由二个操作数间隔一个操作符组成。两操作数本身又分别可以是表达式。先计算两个操作数的值，然后再将操作符应用于其结果。注意，在PL/I中，并不指定计算的顺序，以便编译可以选择最有效的计算顺序。下面给出中缀表达式的两个例子。

A+B　　C**2

3. 操作符的优先顺序

在一个无括号表达式或子表达式中，操作符所作用的顺序由一套操作符优先顺序确定。在PL/I中所给的优先顺序列表如下，优先顺序为从最高到最低，具有相同优先顺序的操作符列在同一行中。

指数	**
非	～或^
前缀操作符	+，－
乘、除	*，/
加、减	+，－
并置	||，!!或\\
关系操作符	=，～=，<，～<，>，～>，<=，>=，
与	&
或	|，!或\

操作符的优先顺序其效果如同从最高操作符开始，按降序插入括号对，直到最低优先顺序的操作符，整个表达式成为完全括号化的。当相等优先顺序的操作符出现在同一级时，前缀操作符和指数操作符从右到左计算，其余操作是从左到右。例如，无括号的表达式：

2+z*x**y**2/5－Q

将由编译处理为：

(2+((z*(x**(y**2)))/5))－Q

4. 关系操作符

关系操作符用来对非计算值比较其相等或不相等。计算值也可以根据通常的算术规则进行比较，如果在比较之前操作数的类别不同，则将首先转换成通用类型（将在下节描述）。如果比较结果为真，则产生长度为1的位串，其值为'1'B。反之，若比较为假，则结果为'0'B。在字符串比较时，其较短的操作数将在右边填以空格直到与较长的操作数等长为止。比较方法为从左到右逐字符按附录A中所定义的ASCII码顺序进行比较。位串比较时，较短的位串将在右边用0填入，而后再从左到右逐位进行比较，其0位小于1。

5. 位串操作

位串操作包括

^　～　|　!　\　&

其中前二个符号表示逻辑非（NOT）操作符，其后的三个表示逻辑或（OR），最后一个符号表示逻辑与（AND）。

位串的操作是按位逐一进行的。一元"非"操作对位串操作数中的每一位取反，即每一位0变为1，1变为0，"或"和"与"操作需要有二个位串操作数。如果两操作数的长度不相等，则在较短的操作数右边填以0使与另一操作数等长。结果字位串的长度等于两操作数中较长

的一个。“或”、“与”操作符遵循通常的布尔算术规则：

x	y	x|y	x	y	x&y
0	0	0	0	0	0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	1	1	1	1

（其他的布尔函数很容易用BOOL 内部构造函数而构成）。

6. 指数

如果指数值是一个非负的字面常数时，指数计算是一系列的乘运算。否则，将要使用LOG和EXP超越函数进行计算。指数操作中的一些特殊情形考虑如下：

如果x＝0 且y＞0，则x＊＊y＝0。

如果x＝0 且y＜0，则发生ERROR条件。

如果x～＝0 且y＝0，则x＊＊y＝1。

如果x＜0 且y是非整数，则发生ERROR条件。

7.2 算术转换

在计算表达式时，可能要有各种类型的数据间的转换。所有的转换将牵涉到源数据、中间结果和目标数据。中间结果是由源数据根据下面描述的转换规则来计算的，而目标数据的格式又是从中间结果得到的。

1. 算术到算术转换

算术操作数转换的通用规则是：

如果一个操作数是FIXED，而另一个是FLOAT，则通用类型是FLOAT。FIXED BINARY (p) 转换为FLOAT BINARY (p)，而FIXED DECIMAL (p, q) 则截取转换为FLOAT BINARY (p')：

$$p' = \min(\text{ceil}(p * 3.32)), 24),$$

如果一个操作数是FIXED BINARY，而另一个类型是FIXED DECIMAL，则通用类型是FIXED BINARY。FIXED DECIMAL (p, q) 则变为FIXED BINARY (p)。

在已经转换为通用类型后，如果操作数是FLOAT BINARY，那么结果是FLOAT BINARY。结果的精度是两操作数中的精度较高者。

如果两操作数是FIXED DECIMAL，设第一个操作数的精度为 (p, q)，第二个操作数的精度为 (r, s)，那么对于加减操作，结果精度为：

(MIN (15, MAX (p−q, r−s) +MAX (q, s) +1),MAX (q, s))

对于乘，结果精度为：

(MIN (15, p+r+1),q+s)

对于除，结果精度为：

(15, 15−p+q−s)

当做FIXED DECIMAL除时，需要谨慎。DIVIDE函数可用来控制商的精度。

如果操作数是FIXED BINARY，设 (p) 是第一个操作数的精度，(r)是第二个操作数的精度，那么对于加减操作，其结果精度为：

MIN (15, MAX (p, r) +1))

对于乘，结果精度是：

(MIN (15, p+r+1))

使用DIVIDE内部构造函数，要与全集语言兼容，且标度因子必须给定为零。产生的结果为整数。

如果结果的精度不足以容纳所得数时，将发生截断。对于FLOAT BINARY数据项，其右边的将被截掉，而在FIXED DECIMAL计算中则丢掉小数数位。FIXED BINARY高位将被丢失。许多算术转换函数可用来控制在计算表达式时所发生的转换。详细描述如下。

2. FIXED内部构造函数

FIXED内部构造函数引用为FIXED (x) 或FIXED (x, p) 或FIXED (x, p, q)。其中x是将要转换为FIXED算术数据类型的变量或表达式。p和q给出目标的精度和标度。如果x是FIXED DECIMAL，则结果也是FIXED DECIMAL，否则，结果将是FIXED BINARY。如果x是FLOAT BINARY，结果是FIXED BINARY。如果没有给出p或q，那么，结果取决于x的精度和标度如下：

X FIXED BINARY (r) 产生FIXED BINARY (r).

X FLOAT BINARY (r) 产生FIXED BINARY (MIN (15, r)).

X FIXED DECIMAL (r, s) 产生FIXED DECIMAL (r, s).

3. FLOAT内部构造函数

FLOAT内部构造函数引用为FLOAT (x) 或FLOAT (x, p)。其中x是将要转换为FLOAT算术数据类型的变量或表达式，p给出目标的精度。如果没指出p，那么，其结果如下：

X FIXED BINARY (r) 产生FLOAT BINARY (r)

X FLOAT BINARY (r) 产生FLOAT BINARY (r)

X FIXED DECIMAL (r, s) 产生

FLOAT BINARY (MIN (CEIL((r−s) * 3.32),24))

4. BINARY内部构造函数

BINARY内部构造函数引用为BINARY (x) 或BINARY (x, p)。其中x是将被转换为BINARY算术数据类型的变量或表达式，p是目标的精度。如果 x 是 FIXED BINARY或FIXED DECIMAL，结果为FIXED BINARY；如果x是FLOAT，那么结果是FLOAT BINARY；如果不给出p，则结果如下：

X FLOAT BINARY (r) 产生FLOAT BINARY (r)

X FIXED BINARY (r) 产生FIXED BINARY (r)

X FIXED DECIMAL (r, s) 产生

FIXED BINARY (MIN (CEIL((r−s) * 3.32+1,15)))

5. DECIMAL内部构造函数

DECIMAL内部构造函数引用为DECIMAL (x) 或DECIMAL (x, p)或DECIMAL (x, p, q)。其中x是将被转换为 FIXED DECIMAL 算术数据类型的变量或表达式。p和q是目标结果的精度和标度。如不给出p和q，那么，结果如下：

X FIXED BINARY (r) 产生FIXED DECIMAL (CEIL (r/3.32)+1,0)

X FLOAT BINARY (r) 产生

FIXED DECIMAL (MIN (CEIL(r/3.32), 15),0)

X FIXED DECIMAL (r, s) 产生FIXED DECIMAL (r, s)

6. DIVIDE内部构造函数

DIVIDE内部构造函数用来控制除法操作结果的精度。其形式为：

DIVIDE (x, y, p[,q])

其中x和y是任意的算术表达式，x被y除。p是一个FIXED BINARY表达式，指出要求的精度. q 是一个FIXED BINARY表达式，指出要求的标度，若不给出.则假定q 为零。由于在全集语言中，对于FIXED BINARY除法，非零标度因子是由这种操作产生的，故需要DIVIDE函数。

7.3 串的转换

当表达式中有算术数据和位串数据项时，在它们之间将会发生转换。下面将给出串操作数的各种转换规则。

1. 算术到位串的转换

从算术源数据类型x 转换为串目标时按下述规则进行。根据算术转换规则将ABS (x) 转换为FIXED BINARY (p)，再将中间结果FIXED BINARY转换为长度为 p 的位串。如果目标的长度大于 p .则在中间结果右边填 0 字位；如果目标长度小于 p，中间结果最右边超出的字位被截掉。

2. 算术到字符的转换

各种算术数据类型将按下述方法转换为中间字符串：FIXED DECIMAL (p, q)，q= 0 .结果字符串的长度是p+3 .字符串由源数位去掉前导零，如果数为负值.要在前面加以负号，而后在左边填入空格变为长度为p+3的字符串。例如，FIXED DECIMAL (3) 具有值330. 转换为字符串'bbb330'.其中b表示一个空格位。值 0 将转换为单个零数位的结果。

FIXED DECIMAL (p, q),q＞0，其结果字符串的长度也是p+3，其格式如上所述，只是包括有小数点及小数部分而已。例如，FIXED DECIMAL (5, 2) 数据项具有值－13.25，转换为字符串结果'bb－13.25'，除了紧挨小数点前的一个零外. 其余的前导零忽略。

FIXED BINARY (b),其源将被转换为FIXED DECIMAL(p). 其中p=CEIL (b/3.32) +1，再将结果FIXED DECIMAL (p) 用如上描述的格式转换为长度p+ 3 的字符串。FIXED BINARY (15) 具有值－32时，转换结果为'bbbbbb－32'。

FLOAT BINARY (b)，小数部分被转换为FIXED DECIMAL (p)，其中p= CEIL (b/3.32)。结果字符串的长度是p+6，且具有下述的标准科学记数法格式。如果源值为负，则第一个字符是负号，否则，该位是空格。下一位是数值的最高有效位，紧跟着是小数点及其余的p－1位小数数位，而后是指数指示位 "E".再加一位指数符号位和二位指数值。FLOAT BINARY (24) 具有值250.1E1时，转换的字符串结果是 'bbbb2.501E+03'。

如果目标的长度大于中间结果的长度，则在串的右边填空格；反之，如果目标短于中间结果，串从右边截成较短的字符串。

3. 位串到算术的转换

在一个长度为n的字位串（0≤n≤15）转换为一个算术数据类型时，将首先转换为它的等价的FIXED BINARY (15)，而后再根据前面所描述的规则转换为目标值。'1011'B转换到FIXED BINARY (15) 产生值11。

4. 位串到字符串的转换

长度为n的字位串将转换为长度为n的字符串，其中0位转换为字符0，1位转换为字符1。如果目标长度大于源长，则目标右边填空格；如果目标长度短于源长，右边的超出字符截掉。

5. 字符到算术的转换

从字符串到算术转换时，源字符串必须含有一个合法的算术常数值。如果x是一个字符串，则对x的转换受算术转换内部构造函数的影响如下：

FIXED (x) 或DECIMAL (x) 返回一个FIXED DECIMAL值，如果不给出p，则假定为15。BINARY (x) 产生一个FIXED BINARY值，如果不给出p，则置为15。注意，其结果将仅仅是x的整数部分。FLOAT (x) 产生一个FLOAT BINARY值。如果不给出p，假定p=24，如果x是空值或全部为空格，则转换为0。

如果字符串不是一个合法的算术表示，或目标数据域不够表示转换后的值，则产生ERROR条件。下列的举例表明了从字符串到算术数据类型的各种转换。

字符串		
'00987'	FIXED BINARY (15)	987
'9.87'	FIXED DECIMAL (6,2)	0009.87
'−9.87E2'	FLOAT BINARY (24)	−9.87E2
'−9.87E2'	FIXED DECIMAL (9, 2)	0000987.00
'−9.87E2'	FIXED DECIMAL (5, 0)	00987
'−987.372'	FIXED DECIMAL (4, 2)	出错
'2I3'	FIXED BINARY (15)	出错

6. 字符串到位串的转换

字符串转换为位串时，源字符串必须仅含有字符0和1，每一字符0转换为0位，每一字符1转换为1位。如果目标的长度大于源的长度，那么，在右边填0；如果目标短于源的长度，则从右边截掉；如果源是一个空字符串，或全为空格，那么，结果位串是0。

7.4 伪变量

两个内部构造名SUBSTR和UNSPEC在所有的PL/I—80程序中已预定义。可以用作表达式中的源操作数或赋值语句左端的目标操作数。由于它们的表示作用和一个简单的程序变量一样，因而这些名被称为伪变量。

1. 字符SUBSTR

字符子串操作符允许在一个串中存取个别字符。具有如下两种形式：

SUBSTR (**字符变量**, i)

和SUBSTR (**字符变量**, i, j)

其中**字符变量**是一个CHAR或CHAR VARYING带下标或无下标变量引用，i和j是FIXED BINARY表达式。当SUBSTR出现在一个表达式中时，第一种形式提取字符串

中从位置 i 开始的其余的子字串。其中第一个字符位置数位号是 1，上述的第二种格式和第一种完成同样的功能，但截取的子串长度是 j 。注意，如果 i 或i＋j 超出了串的长度，结果将是不确定的。其中串的长度对于CHAR 变量是其说明的长度大小，而对 CHAR VARYING变量则是当前长度。

当SUBSTR 出现在赋值语句的左端时，它必须是单独出现的。也就是说，当它作为一个字符串赋值的目标时，SUBSTR 操作不能含在字符串表达式中。SUBSTR 操作在这种情形下出现格式为：

SUBSTR（**字符变量**，i）＝**字符表达式**；

SUBSTR（**字符变量**，i，j）＝**字符表达式**；

第一种格式将**字符表达式**所给定的值赋给**字符变量**中的子串，赋值的起始位置为 i ，直至达到字符表达式的长度或字符变量的结束，取二者首先满足的情形。第二种格式具有相同的效果．仅是接受字符域的宽度限定为长度 j 。注意，i和i＋j的值必须落在字符串的固定长度或当前长度内，否则，操作产生不确定的结果。

同一字符变量可以同时出现在赋值语句的左边和右边，赋值时子串不发生重写，虽然这在其他的PL/I版本中可能发生。下列的赋值语句给出了SUBSTR的一种用法：

SUBSTR(C(I),J,K+2) ＝SUBSTR(D，J)||SUBSTR (E，J＋5，3)；

2． 字位SUBSTR

在PL/I－80中，字位子串的操作与上述的字符串 SUBSTR 类似，它们具有相同的限制。首先，PL/I－80 位串的精度范围根据单字节和双字节而限于 1 到16。出于编译期间中间值的考虑，字位子串操作的长度必须是常数。这样，字位子串的格式便为：

SUBSTR（**位变量**，i）

和SUBSTR（**位变量**，i，k）

其中**位变量**是一个带下标或无下标的BIT变量引用，k是一个范围为1～16的字面常数。i是一个FIXED BINARY表达式。SUBSTR操作的效果与上述的字符串操作完全相同，只是当SUBSTR 出现在一个表达式中及出现在左端作为一个位串存储操作的目标被赋值时，取长度为 k 的字位串。字位SUBSTR的举例将在下面给出。

3． UNSPEC

UNSPEC 伪变量将单字节或双字节变量作为 8 位或 16 位的位串数据项进行存取。UNSPEC引用的格式是：

UNSPEC（**变量**）

其中**变量**是对一个数据项带下标或无下标的变量引用。该数据项占有单字节或双字节的存储单元。注意，临时存储的结果不能作为变元。当UNSPEC用在表达式中时，其结果是变元的位串值。当UNSPEC出现在赋值语句的左端时，所赋的值转换为位串并直接地存作单字节或双字节值。

当通常语言性能不能使用某些基础的功能时，常用UNSPEC伪变量作为“换码”机制，但是初学编程者常常落入使用UNSPEC的陷阱而不去使用更合适的高级语言功能。事实上，每当似乎有必要使用 UNSPEC 时，就应当对问题进行综合的检查，看是否可以避免使用它。下面的举例给出了一种情况，对于二个存储单元进行存取。用UNSPEC操作将二个绝对地址装在两个指针变量中，而后两个有基变量再覆盖这两个存储单元，这样它们便可以作

为16和8位数进行存取。然后再应用字位SUBSTR 伪变量将一个子串从一个单元移至另一个单元中。

```
DCL
  (P, Q)  POINTER,
 A  BIT (16)  BASED (P),
 B  BIT (8)  BASED (Q),
 I  FIXED;
 ...
 UNSPEC (P) ='FF80'B4;
 UNSPEC (Q) ='FFF0'B4;
 ...
 SUBSTR (B, 4, 2) = SUBSTR (A, I, 2);
 ...
```

第八节　顺序控制语句

顺序控制指的是程序中的语句所执行的次序。程序语句通常是与分布在程序中的顺序控制语句一起按次序执行的。顺序控制语句用来改变正常的流程。通常，顺序控制语句允许有条件或无条件转移、受控循环，现描述如下。有关过程调用改变正常执行顺序的情形，将在本节以后叙述。

8.1　GO　TO语句

GO　TO语句使控制无条件地转到一个指定标号的语句，有以下几种形式：

```
GO  TO   标号常数
GOTO     标号常数
GO  TO   标号变量
GOTO     标号变量
```

其中**标号常数**是一个字面标号。它作为一些带标号语句的前缀出现。**标号变量**是一个简单的或带下标的标号变量，且已赋予了一个标号常数值。所给的标号值必须是在该 GO　TO 语句作用范围内的一个语句标号，且该标号不能嵌套在任何类型的DO组中。下面给出GO　TO语句的三个例子：

```
GO  TO  LAB1;
GOTO    WHERE;
GO  TO  L (J);
```

8.2　IF语句

IF　语句根据一个布尔测试值的真或假，有条件地执行一个语句或一组语句。任选的ELSE 子句提供了当布尔测试为假时执行的替换语句或语句组。IF语句的通用格式如下：

```
IF  条件  THEN  语句组1;
  〔ELSE  语句组2〕;
```

其中**条件**是一个标量表达式，它产生一个字位串值。而**语句组1**和**语句组2**则可以是单个语句或

包含在DO组或BEGIN组中的语句组。如果语句组1或语句组2是一单个语句，它不能是DECLARE、END、ENTRY、FORMAT或PROCEDURE语句，如果条件的结果中每一位都是1，那么，便执行THEN语句组1；否则，控制转到语句组2(如果存在时)或转到IF语句顺序流程中的下一语句。

IF语句可以自身嵌套。在这种情况下，每一个ELSE与其最内层的IF-THEN组配对，也可以用空语句使所要求的IF-ELSE语句成对。考虑下面的嵌套IF语句，

```
IF A=Y THEN
   IF Z=X THEN
      IF W>B THEN
         C=0;
      ELSE  C=1;
   ELSE;
ELSE  A=Y2;
```

其中第三个ELSE语句前的空语句与第二个IF-THEN测试相对应。

8.3 迭代DO语句

最简单形式的DO组即执行一次的语句集合。这种非迭代式的DO组在前面已有叙述。本节描述迭代DO组，它是由下面两种通用格式开头的语句组。

```
DO WHILE (条件);
DO 控制变量=do指明表;
```

其中**控制变量**是一个无下标变量，**条件**是一个布尔表达式，**do指明表**是下列之一：

〔起始表达式〔TO 终止表达式〕〔BY 增量表达式〕〕〔WHILE (条件)〕
〔起始表达式〔BY 增量表达式〕〔TO 终止表达式〕〕〔WHILLE (条件)〕
〔起始表达式〔REPEAT (重复表达式)〕〔WHILE (条件)〕

在这些通用格式中，**起始表达式**是一个表达式，它给出控制变量的初始值。**终止表达式**是一个表示控制变量终止值的表达式，**增量表达式**是在每次循环执行后，加到控制变量上的一个表达式，**重复表达式**是在每次迭代后，用来替换控制变量的表达式，**条件**是一个产生位串值的表达式，如果字位串中的每一位都为1，则认为是真值。如果含有TO终止表达式，但省略了BY增量表达式，那么，假定增量表达式是1。使用TO和BY的两种格式具有同样的执行方式，区别仅是这两个元素的顺序不同。

WHILE表达式是在DO组执行之前求值的。如果条件为假，则循环终止，控制转到相应的END语句之后的语句。

除了REPEAT任选项外，在do指明表中的表达式是在循环执行之前求值的，以使对起始、终止、增量值的改变不致影响循环的执行次数。在有REPEAT任选项的情况下，每次迭代后，要重新计算重复表达式的值。重求得的表达式存入控制变量，如果它还包括在WHILE测试中，则还存入测试中，再重新求值测试。

为严格定义迭代组的作用，将其分解为等价的IF和GO TO语句序列如下，在分解中，表达式e_1、e_2、e_3和e_4相应于起始表达式、终止表达式、增量表达式、重复表达式和条件值，而i则表示一个合法的控制变量。首先，DO-WHILE语句：

```
DO WHILE (e1);
```

```
    ...
    END;
```

可以等价地表示为语句序列：

```
LOOP:
    IF~e1 THEN
      GOTO ENDLOOP;
    ...
    GOTO LOOP;
ENDLOOP:;
```

类似地，DO-REPEAT组：

```
DO i=e2 REPEAT (e2);
    ...
    END;
```

变作：

```
    i=e1;
LOOP:
    ...
    i=e2;
    GOTO LOOP;
```

注意，在这种情形下，循环将无限制地继续进行，直到由其中的语句终止，如GO TO或STOP语句。也可以添加WHILE任选项：

```
DO i=e1 REPEAT (e2) WHILE (e3);
    ...
    END;
```

其等价语句为：

```
    i=e1;
LOOP:
    IF  ^e3 THEN
      GOTO ENDLOOP;
    ...
    i=e2;
    GOTO LOOP;
ENDLOOP:;
```

简单的迭代DO组

```
DO i=e1 TO e2;
    ...
    END;
```

被处理作：

```
    DO i=e1 TO e2 BY e3;
```

```
        ...
        END;
```

它可以被表示为等价的序列：

```
        i=e1;
        LAST=e2;
        INCR=e3;
    LOOP:
        IF endtest THEN
            GO TO ENDLOOP;
            ...
            i=i+INCR;
            GO TO LOOP;
    ENDLOOP:;
```

其中IF语句包含了一个控制变量与LAST值的终止测试比较 (endtest)。比较将与增量值INCR的符号有关，如果INCR为负值，测试为：

```
        IF i<LAST THEN
            GO TO ENDLOOP;
```

否则，测试变为：

```
        IF i>LAST THEN
            GO TO ENDLOOP;
```

最后，加WHIE 任选项：

```
        DO i=e1 TO e2 BY e3 WHILE (e4);
        ...
END:
```

则可得等价语句：

```
        i=e1;
        LAST=e2;
        INCR=e3;
LOOP:
        IF ^e4 THEN
            GO TO ENDLOOP;
        IF endtest THEN
            GO TO ENDLOOP;
        ...
        i=i+INCR;
        GOTO LOOP;
ENDLOOP:;
```

注意，在这些等价的语句序列中，LAST 和 INCR 的值具有表达式e2和e3的特性，进而，在每一步中，要根据通常的PL/I-80规则进行算术转换和比较。

8.4 条件处理

ON、REPEAT 和 SIGNAL 语句提供了运行时截取错误条件的便利手段。这些错误条件大部分会使程序终止。下列条件是PL/I-80能识别的：通用错误条件（ERROR）、一定的计算错误条件（FIXEDOVERFLOW、OVERFLOW、UNDERFLOW和 ZERODIVIDE）和一定的 I/O 条件（ENDFILE、ENDPAGE、KEY 和 UNDERFINEDFILE）。关于例外处理的其他细节，参看下面的叙述及"PL/I-80 应用指南一章"。

8.5 ON 语句

ON语句定义程序执行期间所处理的条件及条件发生后所采取的操作。ON语句具有格式：

ON 条件 中断续元

其中**条件**可以是下列之一：

ERROR	FIXEDOVERFLOW
OVERFLOW	UNDERFLOW
ZERODIVIDE	ENDFILE
ENDPAGE	UNDEFINEDFILE
KEY	

中断续元可以是一条PL/I-80语句或包含在BEGIN-END分程序中的多条PL/I-80语句。在 ON 语句中所命名的特定条件发生时，将执行这些语句。该 BEGIN 分程序不能通过 RETURN 语句退出，尽管该限制并不妨碍 BEGIN-END 分程序中的过程定义。转出该 BEGIN分程序可用非局部GO TO语句。如果在 BEGIN-END 组中的所有语句执行后，没有遇到非局部转移语句，控制返回到原条件发出点。

中断续元不能释放条件发生时正在使用的变量，如果是 I/O条件，也不能关闭文件。只有在包含该中断续元的分程序终止或被REVERT语句恢复或在其后执行顺序中又建立了另外的中断续元后，该中断续元才失掉其作用。在最后这种情形，中断续元被压入堆栈，直到由REVERT 语句重新激活。REVERT 语句取消最末一个中断续元。注意，在任意给定点，最多可以有16个ON条件被激活。

8.6 SIGNAL 语句

SIGNAL 语句可以使特定的条件有计划地发生并调用相应的中断续元。如果没有被激活的中断续元，则缺省为系统处理。在多数情况下，缺省处理为打印 "backtrace" 并终止程序的执行。SIGNAL语句的形式是：

SIGNAL 条件

其中**条件**是上述ON语句中所列的条件之一。例如：

SIGNAL ZERODIVIDE

调用当前的ZERODIVIDE中断续元。

8.7 REVERT 语句

REVERT语句用来撤销当前的中断续元。如果先前已存在，则重新建立原中断续元。REVERT语句的格式是：

REVERT 条件;

其中**条件**是在上述ON语句中所列的条件之一。例如：

REVERT OVERFLOW;

取消OVERFLOW条件的当前中断续元。注意，在一个PROCEDURE或BEGIN分程序的出口处，将对该分程序中所有的中断续元自动地执行REVERT语句。

8.8 缺省中断续元

除了FIXEDOVERFLOW和ENDPAGE外，缺省的中断续元通常是打印适当的错误信息随后终止程序。FIXEDOVERFLOW条件对于FIXED BINARY溢出是不发信号的，只在FIXED DECIMAL计算超出它们分配域的大小时发生。当发生ENDPAGE条件时，缺省中断续元在输出文件中插入一个换页字符并将当前行号置为1。

8.9 条件处理内部构造函数

PL/I-80包括了几个为协助例外处理而设的内部构造函数。列出如下：

ONCODE	ONLINE
ONKEY	PAGENO
LINENO	

ONCODE函数返回一个FIXED BINARY值，它表示最近发生的ERROR条件的错误类型。如果没有条件发生，则返回值为0，错误代码随着版本而有变化，可在“PL/I-80命令一览表”中查到。

ONFILE、ONKEY、PAGENO和LINENO内部构造函数将在后面章节中详细解释。

8.10 过程分程序

过程分程序即由PROCEDURE与相应的END语句界定的分程序。它是由子程序CALL语句或函数调用语句调用的。用过程可以对同一个程序执行一次或多次，而不必在程序中对该段进行多次复制。传递到过程中的通信数据称为实在参数，在PROCEDURE语句中所定义的由过程所接收的变量表，则称为形式形式参数。

8.11 调用一个过程

过程有二种：子程序过程和函数过程。二者的区别在于：子程序通过CALL语句调用，而函数过程的调用则是简单地使用其函数名。需要时，后跟括在括号中的一组实在参数，调用处的上下文要求是一个表达式。此外，函数过程给调用程序段返回一个标量值。

CALL语句用来将控制转到一个子程序过程，并在必要时给该过程传递信息。CALL语句的格式是：

CALL **过程名**〔(**下标1**, …, **下标n**)〕〔(**变元1**, …, **变元m**)〕;

其中**过程名**是要调用的过程名。**下标1**到**下标n**表示一组任选的下标表。当过程名是一个带下标的入口变量时，要求有该表。**变元1**到**变元m**表示传递到该过程的实在参数。实在参数可以是下列的任意量：算术或串变量、数组或结构、常数、表达式、标号、文件名或指针 多维数组的截面不能作为实在参数，注意，实在参数的数量是由相应的PROCEDURE句首所定义的，如果不需要参数，则必须给出一个空的括号对。CALL语句的举例如：

```
CALL P ( );
CALL Q (A, (B), B+C);
CALL V (I, J) (A, (B), (B+C));
```

函数以同样方法调用，只是不需要关键字 CALL，且结果值必须用作表达式，函数调用举例如下：

```
I=F ( );
IF G (A, (B), B+C) = 5 THEN
    I=H (I, J)(A, (B)) + SQRT (X);
```

8.12 过程定义的构成

前面描述了过程调用的方法，本节的目的是描述子程序或函数定义的总体构成。过程可以在一个特定程序的任意点进行定义。当在顺序控制流程中遇到时，将被跳过。也就是说，过程只能通过上节所描述的调用方法激活。通常，所有的过程定义在主程序开始或结尾位置的一个单独的节段中，这取决于程序的风格。主过程本身是一个单独的过程定义，进而，分离的过程可以定义并连接在一起而形成一个单独的模块。任何过程定义的总体结构是：

```
过程名:
    过程语句;
    过程体;
END 〔过程名〕;
```

其中**过程名**是一个命名该过程的标识符，过程名将成为一个入口常数，它可以在它所出现的作用域的全部范围内进行存取。一个过程的入口点由**过程语句**标识。**过程体**可以由零条或多条语句序列构成。过程定义由一个相应的END语句终止，它也可以是过程的出口点。在过程体中，也可以出现RETURN语句。

过程语句限定过程分程序的开始，定义形式参数表，给出函数过程返回值的属性。其通用格式是：

```
PROCEDURE 〔(参数1, …, 参数n)〕
    〔OPTIONS (MAIN)〕
    〔RETURNS 属性表〕
    〔RECURSIVE〕;
```

其中**参数1**到**参数n**是过程的形式参数。所有的形式参数必须在过程体的基本分程序层中说明。形式参数可以是下列之一：无下标变量、数组、主结构、标号变量、文件名、入口名或指针变量。形式参数不能有下列属性：

```
STATIC      AUTOMATIC
BASED       EXTERNAL
```

OPTIONS (MAIN) 标识该过程为程序开始时接受控制的第一个过程。函数过程要求有RETURNS属性表，以给出由函数返回值的特性。RECURSIVE属性表示该过程可以在执行时直接或间接地激活其本身。

8.13 RETURN 语句

RETURN 语句将控制返回到原调用分程序中过程调用点之后的位置。若是函数过程，还要返回一个值。RETURN语句的格式是：

```
RETURN 〔(返回表达式)〕;
```

其中**返回表达式**是要返回到调用点的值。必要时，返回值转换为在过程语句的 RETURNS

任选项中所指定的属性。RETURN语句举例如下：

```
RETURN;
RETURN (X**2);
RETURN (F (A,(B)));
```

RETURN语句使其所在的过程分程序终止。如果RETURN语句是在主过程中，控制将返回到操作系统。

8.14 非局部GO TO语句

当目标标号的值常数在含有该GO TO语句的最内部分程序的外部时，便形成一个非局部GO TO语句。通常，由于它会使程序的结构性很差，应避免使用非局部GO TO。当然，有些场合，非局部GO TO是适宜的，尤其是在终止错误条件的情况下，直接地转到使程序重新开始执行的全程错误恢复标号，它常常是很有用的。在这种情况下，所有嵌套的中断续元被自动地撤消，且过程返回信息丢弃。下面的程序给出了从一个过程定义中转出的非局部GO TO举例。

```
P:
  PROCEDURE;
  GO TO L;
  END P;
  CALL P( );
L: ;
```

8.15 STOP语句

STOP语句终止程序的执行，关闭所有打开的文件，将控制返回到操作系统。通常STOP语句只在主程序级出现，但也可以在一个嵌套过程调用中执行而提前暂停程序的执行。其格式是简单的：

```
STOP;
```

8.16 变元和参数

如前所述，由调用分程序所传递的数据项称为实在参数，由被调过程所接收的数据项则称为形式参数。在调用时，每一个实在参数要与相应的形式参数配对。当实在参数与相应的形式参数共享存储时，则称实在参数由引用而传递。在这种情况下，在被调分程序中对形式参数所作的任何改变都将使调用分程序中的实在参数值发生变化。

当实在参数和形式参数并不共享存储时，称实在参数是由值传递的。在这种情况下，送给被调过程的是实在参数的拷贝。这样，对于形式参数的任何改变将仅仅影响其拷贝，而不影响实在参数的值。

由引用传递的变元是一些变量，这些变量的属性与形式参数的属性相符合。由数组和结构组成的集合表达式，总是必须在下标范围、类型、精度和标度上相符合，因而，总是由引用传递的。

当实在参数是下列情形之一时，它是由值传递的：常数、入口名、由变量引用和操作符组成的表达式、括在括号中的变量引用、函数调用或与形参指明表不相符合的表达式。在最后一种情形，实在参数将被转换为形参的类型、精度和标度，给定由下列语句开始的过程P：

```
P:
```

```
PROCEDURE (A, B, C);
DCL
  A CHAR(10),
  B FIXED,
  C FLOAT,
  ...
```

如下格式的CALL送给该过程三个实在参数，与三个形式参数A、B和C相对应。在该例中，假定X是CHAR(10)，Y和Z都是FIXED：

```
CALL P(X, (Y), Z);
```

由于第一个实在参数X与它的形式参数A相符合，故它是由引用传递的；由于第二个参数作为一个表达式出现，故由值传递；第三个参数将转换为FLOAT类型，由值传递。

8.17 ENTRY属性

入口数据项用来标识过程名，由入口常数和入口变量组成。入口常数相应于内部过程或分别编译的外部过程。入口变量是可以在程序执行期间赋予入口常数值的数据项。

对于外部编译的过程，形式参数和返回值的特性必须要在调用程序中用 ENTRY 说明加以定义。程序员要保证入口说明与外部定义的过程相一致。这一点是必要的，以便在用连接编辑对程序段组合时能正确地连接。另外，持有入口常数值的变量也必须用 ENTRY 说明定义。若应用程序需要，入口变量可以带有下标。ENTRY属性用来将一个标识符定义为入口数据项，并给出形式参数的属性。如果入口项是一个函数，需要时可给出返回值的属性。ENTRY说明的格式为：

DECLARE过程名〔(**下标1**，…，**下标n**)〕
〔VARIABLE〕
〔ENTRY (**属性1**，…,**属性n**)〕
〔RETURNS(**返回属性**)〕;

其中的属性可以按任意顺序列写，且至少必须有一个ENTRY或RETURNS。由过程名所给出的标识符是入口数据项名。**下标1**到**下标n**是任选的下标表，**属性1**到**属性n**是形式参数属性表，**返回属性**是任选的函数入口项的返回值属性。VARIABLE属性表示数据项是一个入口变量，它必须在程序执行期间赋给一个入口常数值。注意，仅在具有VARIABLE属性时，下标表才是合法的。如果该过程不要求有参数，形式参数属性表省略。在这种情况下，如果有RETURNS属性，ENTRY属性也可省略。

如果对于特定的参数指出了维数属性，它必须是说明的第一个属性，如果形式参数是一个结构，要在属性定义前加层号来指出构成信息。在属性1到属性n中，不允许使用因子属性。举例如下：

```
DECLARE X ENTRY;
DECLARE Y ENTRY VARIABLE;
DECLARE P (0:10) ENTRY (FIXED, FLOAT) VARIABLE;
DECLARE Q ENTRY (1,2 FIXED,2 FLOAT,(5:10)DECIMAL);
DECLARE R RETURNS (CHAR (10));
```

第九节　输入/输出处理

本节介绍PL/I-80的输入/输出功能。用它可进行内存和外部设备间的数据传输。PL/I优越功能的相当大的部分是在它的I/O处理能力上。有关I/O功能应用的更完整的论述，包括在“PL/I-80应用指南”一章中，下面只打算作简单的和概念性的讨论，而不是完整解释。完整的举例请见第一章。

9.1 FILE数据项

一台外部设备可以是控制台、行打印机或磁盘文件。传送至或取自外部设备的数据元素的集合称作数据集。而相应的内部文件常数或变量则简单地标为文件。除了预定义的标准输入输出文件（称作SYSIN和SYSPRINT）外，所有在特定程序中所要存取的文件必须用下列格式的文件说明进行描述。

DECLARE 文件名 FILE [VARIABLE];

其中文件名是文件标识符。如果不指出VARIABLE任选项，则该说明定义一个文件常数；如果给出了VARIABLE属性，那么该说明定义一个文件变量。该文件变量可以由赋值语句使之持有一个文件常数值。

一个文件常数说明建立一个文件参数块，它是一个含有关于文件信息的存储段。文件变量说明并不建立参数控制块，因而，只有在它被赋予了一个文件常数值以后，文件变量在输入/输出中才是合法的。文件数据可以使用等于和不等于的比较操作符。如果二者引用同一个文件参数块，则项是相等的。最后应注意，文件常数被定为EXTERNAL属性，而文件变量则是说明它的分程序中的一个局部量。除非它用了EXTERNAL说明。

9.2 文件类型

PL/I-80识别三种基本文件类型：流式、顺序记录和直接记录文件。文件的类型决定了数据存放的方式及传送或存取的方法。

在流式I/O中，数据作为组织成行和页的ASCII字符序列处理，行由行标志分隔、页由页标志分隔。行标志即回车、换行字符对或单个换行符。而页标志则是一个换页字符。流式文件只可以顺序存取，也就是说，数据项的读写是按顺序直到文件尾或文件关闭。

对于流式文件，可以用格式化I/O和自由格式I/O。当数据流传送到内存时，输入的字符转换为接收变量的数据类型。反之，送至流式文件的数据则转换为它们的ASCII码表示。I/O转换按前面所描述的数据类型转换规则进行。

在记录式I/O中，所传送的数据项的大小随项的大小和文件特性而变。记录传送时并不进行转换，其结果是以机器内存中表示数据项的二进制位串方式进行传送。

顺序记录文件仅可以顺序存取。也就是说，记录是按线性顺序读写的，另一方面，直接记录文件则是根据在READ或WRITE语句中所提供的键值存取，而不要求顺序进行。在直接记录文件中，每一个记录都有一个相应的键值，它为后继存取提供了一个唯一的记录标识。

9.3 打开一个文件

在数据集上进行任何I/O事务之前，必须先打开文件。文件可以用OPEN语句显式地打开，也可通过GET、PUT、READ或WRITE语句存取文件时隐含打开。在打开一个

文件时，要做下列工作：首先，文件的属性要进行合并，其次，将文件与一个外部数据集联系起来。如果是输入文件而不存在外部数据集，则将发生UNDEFINEDFILE条件；如果是输出文件，则已存在的具有相同名字的数据集将被抹掉，而后再建立一个新的数据集。类似地，如果打开文件进行更新存取，而数据集不存在，则建立一个数据集。如果文件已处于打开状态，则没有任何作用。

OPEN语句用来显式地打开一个文件，OPEN语句的格式为：

OPEN FILE（**文件名**）〔**文件属性**〕；

其中**文件名**是一个在文件说明语句中出现过的文件名。**文件属性**表示下列关键字的任意合法组合。

STREAM　RECORD
INPUT　OUTPUT
KEYED　DIRECT
SEQUENTIAL　UPDATE
PRINT　TITLE
PAGESIZE　LINESIZE

OPEN语句的解释及上述关键字的用法部分与具体实现有关。因此，每一关键字的详尽格式及相应的参数在“PL/I-80应用指南”中给出。

9.4 文件参数块

每一个文件参数提供了对存储在文件参数块（FPB）中的值的存取。每一个FPB含有如下一些信息：文件状态表示文件是打开或是关闭。文件标题给出与文件常数相关联的外部设备或数据集命名。在STREAM文件中，为了对要取（GET）或送（PUT）的下一个位置进行定位，要保存有字符列的位置。在STREAM文件中要对当前的行进行计数，对于PRINT文件，还要对当前页计数，且也保持有当前记录位置。在打开一个文件时，FPB还要增加对CP/M文件控制块（FCB）以及行大小、页大小、定长记录大小、内部缓冲区大小和文件描述符定址。其他与实现版本有关的信息也可以从FPB中取得。一个成功的OPEN操作之后，文件描述符将对文件定义下述的文件属性集之一。

STREAM INPUT
STREAM OUTPUT
STREAM OUTPUT PRINT
RECORD INPUT SEQUENTIAL
RECORD OUTPUT SEQUENTIAL
RECORD INPUT SEQUENTIAL KEYED
RECORD OUTPUT SEQUENTIAL KEYED
RECORD IPPUT DIRECT KEYED
RECORD OUTPUT DIRECT KEYED
RECORD UPDATE DIRECT KEYED

9.5 输入/输出 ON 条件

在I/O操作期间，可能会发生许多条件。每当一个输入操作读至文件结束时，便发生ENDFILE条件。对于STREAM文件，它是control-Z，对于RECORD文件，则是文

件的物理结束。对于OUTPUT文件，当所有的磁盘空间用完时，也会发生ENDFILE条件。

对于一个具有PRINT属性的STREAM输出文件，当行号超过页大小时，便发生ENDPAGE条件。其缺省的中断续元是在输出中送一个换页符，并将行号复位到1。

每当遇到一个非法键值时，便会发生KEY条件。在PL/I-80中，如果FIXED BINARY键值和定长记录大小的倍数超过磁盘的容量，便发生该条件。

每当一个输入文件不存在或由于目录空间不够而不能建立输出或更新文件时，会发生UNDEFINEDFLE条件。如果在TITLE任选项中所提供的文件名表达不恰当，该条件也会发生。

有许多内部构造函数可用于I/O异常操作中，它们将在后面各节中详细描述。它们是：LINENO、PAGENO、ONFILE和ONKEY，此外，关于其他与实现版本有关的信息请参阅“PL/I-80应用指南”一章。

9.6 CLOSE语句

CLOSE语句使得文件与外部数据集相离脱，CLOSE语句的格式是：

CLOSE FILE（**文件名**）；

其中**文件名**的值是一个文件常数。执行该操作时，如果文件不是打开状态，则忽略关闭操作，否则，清除缓冲区，将输出文件永久性地记录在磁盘上。其后，文件还可以使用上面讲的OPEN语句重新打开。

9.7 预定义文件

文件常数SYSIN和SYSPRINT是所有的PL/I-80程序的一部分，除了在OPEN、GET、PUT、READ或WRITE语句中显式地引用了这些文件外，是不需要说明的。在GET或READ语句中不给出FILE任选项的情况下，自动地存取文件SYSIN。类似地，在PUT或WRITE语句中没有FILE任选项时，存取SYSPRINT。在这种情况下，SYSIN文件变为控制台键盘，具有80个字符的行宽，而SYSPRINT则变为控制台输出显示，行宽也是80字符，且页大小不限。

第十节 流式输入/输出

STREAM文件由以行标志和页标志分隔的ASCII字符序列构成，STREAM I/O语句提供了在STREAM文件中存取字符数据的工具。通常，流式I/O具有下列规则：文件的初始列位置为1；每遇到一个行标志或页标志，则列位置复位到1；否则，如果输入或输出字符是图形符，列位置前进一列。如果在输出中列的位置超过行大小，则写一个行标志，行号增1，列位置复位到1。当行号超过页大小时，写一个页标志，列位置和行号都复位到1。

在PL/I-80中，提供了三种形式的STREAM I/O，称作列表式、编辑式和行式。列表式I/O传送数据项时没有格式指明表，编辑式I/O可以对存取的字符编排格式，而行式I/O则可用非编辑形式存取变长字符数据。注意，在PL/I-80中提供的行式I/O可用READ和WRITE语句处理变长ASCII记录，但在其他版本的PL/I中则是不可使用的。

下面描述各种STREAM I/O语句中所用的名称限制。

文件名 文件标识符。

行数 一个FIXED BINARY表达式。它定义输入时跳过的行标志数或输出时写

在数据之前的行标志数。

输入表 一个由逗号分隔的变量表，从输入流中传送的数据项将赋给这些变量。输入表决定了在数据流中由输入数据所赋值的变量数和顺序。在PL/I-80中，变量必须是标量值。

在输入表中，可以包括有迭代DO循环。DO语句首的格式可以是前面所述的DO语句格式，但不能含有REPEAT子句。其通用格式为：

（项1，…，项n DO迭代）

如语句：

```
GET LIST ((A(I), B(I)  DO I=1 TO 10));
```

输出表 一个由变量、常数或表达式组成的输出项表，各项由逗号分隔。输出表可以含有迭代DO组，如上述输入表所示。

10.1 列表式I/O

列表式I/O输入流必须具有下列特性：数据流中的数据项可以是算术常数、字符串常数或位串常数。每一数据项后面必须有分隔符，分隔符由一系列的空格或前后含有空格的逗号或行结束符组成。内含的制表标记（ctl-I）作为空格处理，含有空格或逗号的字符串数据必须引在单引号中，否则，空格或逗号将作为分隔符处理。

在输入流中，用逗号作为输入行中的第一个非空格字符或用两个连续的逗号（中间也可以间隔一个或多个空格）表示一个空字段。空字段表示相应的输入表中的数据项不传送数据。这样，目标数据值仍旧不变。当在GET语句中出现SKIP任选项时，对行标志计数，除此之外，它仅作为分隔符。

10.2 GET LIST语句

GET LIST语句供用列表式I/O读取数据。GET语句的格式是：

```
GET 〔FILE（文件名）〕〔SKIP〔(行数)〕〔LIST （输入表）);
```

其任选项必须至少给出一个。除LIST任选项必须最后出现外，其余项顺序可以任意。如果省略FILE任选项，则假定为FILE(SYSIN)。如果在SKIP任选项中不给出行数，则跳过一个行标志。

输入表中所有的数据项传送到输入变量后，输入流中的列位置在所读的最后一项数据的字符后面。

输入流中的字符串可以用单引号引起来，也可以不用。如果使用了引号，该引号并不传送到输入变量中。同样地，对于位串常数，引号及尾部的B并不传送到输入变量中。

输入的字符串限制在一行之内。这样，对于从控制台输入的字符串，当以回车作为终止时，仅仅前面的引号是必要的。

10.3 PUT LIST语句

PUT LIST语句供使用列表式I/O写数据，PUT LIST语句的格式是：

```
PUT 〔FILE（文件名）〕〔SKIP（行数）〕〔PAGE〔(p)〕〕
    〔LIST（输出表）);
```

如同GET LIST一样，必须至少有一个任选项，LIST任选项必须最后出现。如果省略FILE任选项，则假定为FILE(SYSPRINT)。如果用了SKIP任选但不给出行数，那么，行数缺省为1。如果行数为0，则不写行标志，但列位置要复位到1，在任何时候任何

情况下，使用SKIP任选项都将使列位置复位到1。PAGE任选项仅对PRINT文件是合法的。如果不给出p，其缺省值是1。注意，每当写页标志时，列位置和行号都要复位到1。输出表中的数据项将转换为它们的字符串表示法并写到STREAM文件中。空格用来分隔输出文件中的数据。如果数据项长于输出行中剩余的字符数，该项将写在下一行的开始。如果数据项所表示的字符串长度超过行的大小，则该数据项将单独写在超过行宽的一行中。如果在输出传送期间超过了页的大小，将会发生ENDPAGE条件。

通常，字符串要用单引号引起来，字符串本身包含的单引号字符写作一对单引号，如果文件具有PRINT属性，则额外的单引号将被消掉。位串数据则总是带引号的，且后面跟有字母B。

10.4 编辑式I/O

编辑式I/O的输入表和输出表与列表式的相同。读或写数据的方式是由GET EDIT和PUT EDIT语句的格式表所列的格式项而决定的。

10.5 格式表

格式表是由逗号分隔的一系列格式项，它描述将被读入的数据项(数据格式项)，指出数据项在数据流中的排布(控制格式项)或引用另外的格式表(间接格式项)。格式表的通用形式为：

〔n〕格式项…〔,〔n〕格式项〕

其中n是一个字面常数值，范围为1～254，它给出其后的格式项的重复因子。如果省略n，则假定重复因子为1。格式项可以是一个数据格式项，也可以是一个控制格式项。为能对多个格式项重复，也可将格式项列为一组：

(格式表)

格式项也可以是间接格式项。在PL/I-80中，间接格式项必须仅是在格式表中唯一的格式项，不能在前面加重复因子。

10.6 数据格式项

数据格式项用来对外部STREAM数据集读取或写入数字或字符字段。PL/I-80支持下列的数据格式项：

F(w,〔d〕)　用于定点算术数据。其中w是宽度(字段中的字符数目)，d是小数点右边的字符数目。在输入时，将读入由w所指定数量的字符，如果字符串含有小数点，则小数点决定其标度，否则，标度由d决定。前导及尾部空格忽略。如果字段中仅含有空格字符，所读的值为0。在输出时，d指定输出值的标度。如果忽略d，则标度为0，输出值被舍入到整数，除非变量有精度15(最大精度)。除了在小数点前的第一个零外，其余的前导零被消去。

E(w〔,d〕)　用在输出中以科学记数格式表示算术数据。用于输入时将十进制字符转换为浮点二进制值。w指定字段的宽度，而d则给出小数点右边数字的位数。在输出时，w必须比d值大7，由于输出域将显示为：

+n.ddddE+ee

其中+表示符号位置，n是前导数位，dddd表示长度为d的小数部分，E+ee表示指数域。

A〔(w)〕 读或写字符串数据的w个字符，输入时，为要与全集 PL/I 相兼容，必须要包含w。对于PL/I-80，允许在输入中省略 w，在这种情况下，当前行其余的字符全部读完，但不包括回车、换行符。在输出时，如果省略w，那么假定w是输出串的长度。如果w大于输出串的长度，那么在右边填加空格。如果w小于输出串的长度，右边的长出部分将被截掉。

B〔b〕〔(w)〕 表示位串数据表示法，在输入时，必须要包括w，且在输入数据流中，必须仅含有 0 和 1，否则将发生 ERROR 条件。对于每一数位所占的位（bit）由b给出。在输出时，将变量转换为位串类型，然后再转换为它的字符串表示。如果不包含w，则输出其结果字符串。如果所指定的w大于字符串长，那么，在右边填空格，如果结果字符串长于w值，则发生ERROR条件。

10.7 控制格式项

控制格式项用于布置行、页和空隔。PL/I-80支持下列的控制格式项。

COLUMN（nc）或COL（nc） 在输入或输出数据流中，将格式指针移至nc列。在输入时，由设置位置到nc列所越过的字符被忽略。如果当前的列位置小于nc，格式指针移至nc列，如果当前列位置大于nc，指针移到下一行的nc位置。如果nc超过行的最右位置，格式指针移至下一行的第一列。在输入时，格式指针的移动丢弃输入字符，在输出时，指针前移时，数据流中加入空格。

在输出时，在将位置设到nc列的处理中，写入空格。同样，如果当前位置大于nc，程序输出一个行标志，然后再输出空格直到新行的nc列。如果nc超过行的大小，则写行标志并将列位置设置为 1。

X（sp） 在输入或输出数据流中将格式指针前移sp个位置。

在输入时，sp是将要空过的字符数。遇到行标志时，忽略不计，在下一行继续进行。

在输出时，sp是要写的空格数。如果到达行末，则写行标志并继续在下一行写空格。

SKIP〔(nl)〕 指出将跳过或写nl个行标志。如果省略nl，则假定为 1。列位置复位到 1。

输入时，nl是在移到下一个格式项之前要跳过的行标志数。注意，如果在显式或隐式 OPEN 操作后面紧接着被执行的第一个格式项便是SKIP（1），则第一行被丢弃。另外，对于输入流，SKIP(0)是无定义的。

输出时，nl是要写的行标志数。对于PRINT文件，如果在写行标志的过程中超过页的大小，则发生ENDPAGE条件，在从中断续元返回时，SKIP操作中止。

LINE（ln） 仅用于PRINT文件，指出所写的下一个数据项的行号。常数ln必须要大于零。

如果当前行号等于ln，LINE不起作用。如果当前行号小于ln，则输出行标志，直到当前行等于ln。如果所写的行标志超过了当前页的大小，将发生ENDPAGE条件。

PAGE　仅用于PRINT文件。PAGE任选项将使得写一个页标志，页号加1，行号和列号复位到1。

注意，当在格式表中遇到控制格式项时，才执行它。在输入或输出之后所剩的控制格式项是不起作用的。

10.8 间接格式项

间接格式项用FORMAT语句的格式表替代间接格式项。间接格式项的形式为：

R（格式标号）

其中格式标号是FORMAT语句前的标号常数。该标号要在间接格式项的作用域内。如上所提及的，PL/I-80有些限制，即仅间接格式项本身可以出现在格式表中，前面不能有重复因子。示例如下：

```
PUT SKIP EDIT (A, B, C) (R (ELSEWHERE)) ;
```

10.9 FORMAT语句

FORMAT语句定义一个间接格式项，形式如下：

格式标号：FORMAT(格式表)；

其中格式标号是对应于FORMAT的标号常数。格式表是如前所描述的格式项表。例如，FORMAT语句：

```
L1 : FORMAT (A(5), F(6,2), SKIP(3), A(2));
```

作为下列语句的间接格式项引用：

```
GET EDIT (A, B, C) (R(L1));
```

10.10 GET EDIT 语句

GET EDIT语句用格式表读取数据，通用形式为：

GET〔FILE（文件名）〕SKIP〔(行数)〕〔EDIT（输入表）（格式表）〕；

如GET LIST一样，数据项读到在输入表所给出的变量中，直到输入表完或到达文件的末尾。GET EDIT语句中的每一个输入表项都与其后的一个格式表项成对。当处理中遇到控制格式项时，要执行其功能。如果在格式表结束之前输入已完，其余的格式项被忽略。如果在输入表结束之前格式表已完，则格式表从开始重新处理。

10.11 PUT EDIT语句

PUT EDIT语句按格式表输出数据项。PUT EDIT语句的形式是：

```
PUT 〔FILE (文件名)〕
    〔SKIP 〔(n)〕
    〔PAGE 〔(p)〕〕
    〔EDIT (输出表) (格式表)〕;
```

类似于GET EDIT语句，PUT EDIT语句输出表的输出表达式与格式表中的格式项是成对的。处理中遇到控制格式项时，将被执行。在PUT语句结束时没有处理的格式项被忽略。另外，如果在处理期间格式表结束，则将重新从头开始使用格式表。

10.12 行式I/O

在 PL/I-80 中,处理 STREAM 文件的变长 ASCII 记录可用两种形式的 READ 和 WRITE 语句。这两种形式在其他的 PL/I 版本中并不通用。因此，如果着重考虑向上兼容性的话，要避免使用。这两种形式称作变长READ和变长WRITE。

10.13 变长READ语句

READ语句可以用来读变长STREAM INPUT文件，变长READ语句的形式为：

```
READ 〔FILE (文件名)〕 INTO (v);
```

其中 v 是一个 CHAR VARYNG 串变量。如果不给出 FILE 任选项，则假定为 FILE (SYSIN)。

在下面的讨论中，有很重要的一点值得注意：即READ语句不同于变长READ语句，其区别仅在于它的目标变量具有VARYING属性。

从文件中读取数据直到 v 的最大长度或碰到换行字符。v 的长度被置为所读字符的长度，其中包括换行符。注意，当变长 READ 语句有缺省 OPEN 时，导致文件属性包括有 STREAM INPUT。对属性STREAM INPUT，说明如下：

```
1  BUFFER,
2  BUFFCH CHAR (254) VARYING;
```

例如，语句：

```
READ FILE(F) INTO (BUFFER);
```

将产生一个RECORD式的数据传送，这是因为其目标是一个结构，而不是CHAR VARYING变量。语句：

```
READ FILE(F) INTO (BUFFCH);
```

则由于其目标是一个CHAR VARYING 而作为一个 ASCII STREAM INPUT 数据传送处理。

10.14 变长WRITE语句

变长WRITE语句用来写变长ASCII STREAM数据，变长WRITE语句的形式是：

```
WRITE 〔FILE (文件名)〕 FROM(v);
```

其中v是一个CHAR VARYING 串变量。在输出中将不添加附加的控制字符，若在应用中需要有控制字符，它们必须要包含为串的一部分。前面已讲过，PL/I-80 是允许在串常数中含有控制字符的，即在串中的字符前加“^”。

再次强调，变长WRITE在PL/I语言中是不通用的，若要求向上兼容，必须避免使用。如变长READ一样，WRITE 和变长 WRITE 的不同也仅在于它的源变量具有 VARYING属性。另外，要注意由该语句产生的缺省OPEN将包括有STREAM OUTPUT属性。基于前面所给的说明，语句：

```
WRITE FILE(F) FROM (BUFFER);
```

将用RECORD式的数据传送，而

```
WRITE FILE(F) FROM (BUFFCH);
```

则处理为变长WRITE语句，是对ASCII STREAM OUTPUT文件进行操作，这是由于该源变量具有VARYING属性。

第十一节　记录式输入／输出

记录文件中含有未经转换而从连续的存储中传入或传送到存储中的二进制数据。RECORD 处理有二种形式：SEQUENTIAL，其记录按出现的顺序存取；DIRECT，记录通过键存取，下面将讨论各种通用形式。对于每种情形，文件名是一个文件变量或文件常数。x是一个连续的集合或标度数据类型，而不能是CHAR VARYING，k是FIXED BINARY键值或变量。另外，在本节中所描述的大多数操作具有设备依赖性。因此，读者需参考在第一章“PL/I-80应用指南”详细给出的CP/M和MP/M文件系统接口描述。

11.1 READ语句

READ语句用来读定长或变长的RECORD SEQUENTIAL文件。READ语句的形式是：

```
READ FILE (文件名) INTO(x);
```

如果文件没有打开，READ语句提供一个自动的OPEN，并具有RECORD SEQUENTIAL INPUT属性。

所读的字节数由x的长度确定，除非在打开时用了ENV任选定义为定长记录文件。在后一种情形，所读数据的量是说明的固定长度。如果x的长度与记录长度不匹配，则x或者用零填充或将右边的多余字符截掉。

11.2 WRITE语句

WRITE语句不加转换地将数据从内存传送到数据集。对于RECORD SEQUENTIAL文件，WRITE语句的格式是：

```
WRITE FILE (文件名) FROM(x);
```

该语句将伴随有缺省的OPEN，并添加SEQUENTIAL OUTPUT RECORD属性。另外，输出记录的大小刚好是x的长度，除非文件打开时用ENV任选项给定了定长记录的大小。在这种情况下，该语句将写一个定长记录的大小。如果x与定长记录的长度不匹配，则或者填零或者从右截掉。

11.3 带KEY的READ语句

具有KEY任选项的READ语句用来直接地在文件中存取每个记录。带KEY的READ语句的格式是：

```
READ FILE (文件名) INTO(x)    KEY(k);
```

其中k是一个FIXED BINARY表达式，它定义要存取的相关的记录。键值从零开始，直到键值与定长记录长度的乘积达到磁盘的容量。在PL/I-80下不能用带有KEY的READ语句存取变长记录。

11.4 带KEYTO的READ语句

带有KEYTO任选项的READ语句使得在顺序存取时从输入文件中获得键值。这些键值通常保存在内存中或是在另一个文件中，以使输入文件的记录其后可以直接存取。带有KEYTO的READ语句的形式是：

```
READ FILE (文件名) INTO(x)
          KEYTO (k);
```

其中k是一个FIXED BINARY变量。详细描述参见“PL/I-80应用指南”。

11.5 带KEYFROM的WRITE语句

带有KEYFROM的WRITE语句用来直接访问输出文件。格式为：

```
WRITE FILE (文件名) FROM(x)
          KEYFROM(k);
```

其中k是一个FIXED BINARY 表达式，产生一个键值。它的处理与上述带KEY的READ语句相同。

第十二节 内部构造函数

内部构造函数是一个计算子程序，作为PL/I-80库的一部分提供。内部构造函数可以象用户定义的函数引用一样使用，只是使用内部构造函数名时可以不说明。如果内部构造函数的名字已在程序中重新说明，则该内部构造函数在所说明的作用域内不能调用，但它可以在一个内含分程序中再次用BUILTIN属性说明后在该内含分程序中使用。内部构造函数可以根据它们在PL/I-80 中的用途分为下列类别。

算术
数学
串处理
转换
条件处理
其他

下面，将对每一内部构造函数的特定格式、参数、属性、目的和特性进行描述。

12.1 算术函数

1. ABS

格式：ABS(X)

参数：X可以是任意的算术表达式。

结果：返回X值的绝对值。

算法：如果X≥0，返回X，否则，返回−X。

结果类型：与X相同。

2. CEIL

格式：CEIL(X)

参数：X是任意算术表达式。

结果：返回大于或等于X的最小整数。

算法：−FLOOR(−X)

结果类型：与X的类型相同的一个整数值。

3. DIVIDE

格式：DIVIDE(X, Y, P)

或DIVIDE(X, Y, P, Q)

参数：X和Y是算术表达式。

结果：返回X除以Y的商。其结果的精度为P，标度因子为Q。其中P和Q是常数。若不包括Q，则假定Q是0，如果X和Y是FIXED BINARY，Q必须略去或等于零。

结果类型：X和Y的通用算术类型。

4 FLOOR

格式：FLOOR(X)

参数：X是任意的算术表达式。

结果：计算小于或等于X的最大整数。

结果类型：与X的类型相同的一个整数值。

5. MAX

格式：MAX (X,Y)

参数：X和Y是算术表达式。

结果：返回较大的值。

算法：如果X≥Y返回X，否则，返回Y。

结果类型：X和Y的通用算术类型。

6. MIN

格式：MIN (X,Y)

参数：X和Y是算术表达式。

结果：返回较小的值。

算法：如果X≤Y返回X，否则，返回Y。

结果类型：X和Y的通用算术类型。

7. MOD

格式：MOD (X,Y)

参数：X和Y是算术表达式。

结果：返回X对Y的模值。

算法：如果Y=0，返回X，否则，返回 X-ABS (Y) *FLOOR (X/ABS(Y))。

结果类型：具有X和Y的通用算术类型的值。

举例：MOD (7,3)　　返回1

MOD (−7,3)　　返回2

MOD (7,−3)　　返回1

MOD (−7,−3)　　返回2

注：除Y=0外，MOD (X Y) 总是返回一个小于ABS (Y) 的非负值。

8 ROUND

格式：ROUND (X,K)

参数：X是一个算术表达式，K是一个带符号的整常数。

结果：返回X四舍五入到小数点右的K位，如果K小于零，则到小数点左面K位。

算法：返回：

SIGN (X) *FLOOR (ABS (X) *B**N) +0.5) /B**N

其中：如果X是BINARY，B=2，如果X是DECIMAL，B=10。

如果X是FIXED，N=K，否则，如果X是ELOAT，E是X的指数则N=K－E。

结果类型：与X相同。

举例：

ROUND (1234.24698, 3)　返回12345.24700

ROUND (34567.12345, －3)　返回35000.00000

9. SIGN

格式：SIGN (X)

参数：X是任意的算术表达式。

结果　返回－1、0或1，表示X的符号。

算法：如果X＜0返回－1

如果X＝0返回0

如果X＞0返回＋1

结果类型：FIXED BINARY

10　TRUNC

格式：TRUNC (X)

参数：X是任意的算术表达式。

结果：返回X的整数部分。

算法：如果X＜0返回 (CEIL (X))

如果X≥0返回 (FLOOR (X))

结果类型：与X类型相同的带符号整数值。

举例：

TRUNC (52.146) 返回52

TRUNC (－52.146) 返回－52

12.2 数学函数

在PL/I-80库中所提供的数学函数由最常用的三角函数及其反函数，以e为底（自然）、以2为底和以10为底（常用）的对数函数，自然指数函数，双曲正弦和余弦函数，平方根函数等组成。每个函数都定义单一的FLOAT BINARY变元(其他类型的变元也可接受，但要自动地转换为这种类型)，返回一个FLOAT BINARY结果。

除SQRT外的所有的函数子程序都基于采用切比雪夫（Chebyshev）多项式近似值的算法，SQRT函数子程序基于牛顿法，其典型算法是将给定的变元均分为有限的区间（通常－1≤X≤1），然后用适当的递归关系式求出切比雪夫近似值。使用这些子程序产生错误的最大根源是由于在均分处理时丢失了有效位，除此之外，子程序平均精度为7.5位有效十进制数字。

注：对于所有的数学函数，假定参数X是一个算术表达式，则它将被转换为FLOAT BINARY，结果类型也是FLOAT BINARY。

1　ACOS

格式：ACOS (X)

参数：X是一个算术表达式，－1≤X≤1。

结果：返回 arccos (X) ；也即 ACOS (X) 是一个角度，以弧度计，它的余弦值是X。

$0 \leqslant \text{ACOS (X)} \leqslant \pi$

结果类型：FLOAT BINARY

算法：$\text{ACOS (X)} = \frac{\pi}{2} - \text{ASIN (X)}$

错误条件：如果X不在区间〔−1，1〕，发生ERROR条件。

2. ASIN

格式：ASIN (X)

参数：X是一个算术表达式，

$$-1 \leqslant x \leqslant 1$$

结果：返回 (arcsin (X) ，即 ASIN (X) 是一个以弧度计的角度，它的正弦值是X。

$$-\frac{\pi}{2} \leqslant \text{ASIN (X)} \leqslant \frac{\pi}{2}$$

结果类型：FLOAT BINARY

算法：切比雪夫多项式逼近。

错误条件：如果X不在区间〔−1，1〕，产生ERROR条件。

3. ATAN

格式：ATAN (X)

参数：X是任意的算术表达式。

结果：返回arctg (X) ，即ATAN (X) 是一个以弧度计的角度，它的正切值是X。

$$-\frac{\pi}{2} \leqslant \text{ATAN (X)} \leqslant \frac{\pi}{2}$$

结果类型：FLOAT BINARY

算法：切比雪夫多项式逼近。

4. ATAND

格式：ATAND (X)

参数：X是任意的算术表达式。

结果：返回以度计的arctg (X) ，即是一个以度计的角度，其正切值是X。

$-90 \leqslant \text{ATAND (X)} \leqslant 90$

结果类型：FLOAT BINARY

算法：$\text{ATAND (X)} = \frac{180}{\pi} * \text{ATAN (X)}$

5. COS

格式：COS (X)

参数：X是一个算术表达式。

结果：返回cos (X) X以弧度计。

结果类型：FLOAT BINARY

算法：切比雪夫多项式逼近。

6 COSD

格式：COSD (X)

参数：X是一个算术表达式。

结果：返回cos (X)，X以度计。

结果类型：FLOAT BINARY

算法：$\mathrm{COSD}(X) = \mathrm{COS}\left(X * \frac{\pi}{180}\right)$

7. COSH

格式：COSH (X)

参数：X是一个算术表达式。

结果：返回X的双曲余弦值。

结果类型：FLOAT BINARY

算法：COSH (X) = (EXP (X) +EXP (−X))/2

8. EXP

格式：EXP (X)

参数：X是一个算术表达式。

结果：返回一个e的X次幂的值，其中e是自然对数底。

结果类型：FLOAT BINARY

算法：切比雪夫多项式逼近。

9. LOG

格式：LOG (X)

参数：X是一个算术表达式，X>0。

结果：返回X的自然对数。

结果类型：FLOAT BINARY

算法：切比雪夫多项式逼近。

错误条件：如果X≤0，产生ERROR条件。

10. LOG2

格式：LOG2 (X)

参数：X是一个算术表达式，X>0。

结果：返回以2为底的X的对数。

结果类型：FLOAT BINARY

算法：LOG2 (X) =LOG (X) /LOG (2)

错误条件：如果X≤0，产生ERROR条件。

11. LOG10

格式：LOG10 (X)

参数：X是一个算术表达式，X>0。

结果：返回以10为底的X的对数。

结果类型：FLOAT BINARY

算法 LOG10 (X) =LOG (X) /LOG (10)

错误条件：如果X≤0，产生ERROR条件。

12. SIN

格式：SIN (X)

参数：X是一个算术表达式。

结果：返回X的正弦，X以弧度计。

结果类型：FLOAT BINARY

算法：切比雪夫多项式逼近。

13 SIND

格式：SIND (X)

参数：X是一个算术表达式。

结果：返回X的正弦，X以度计。

结果类型：FLOAT BINARY

算法：SIND (X) =SIN (X * π/180)

14. SINH

格式：SINH (X)

参数：X是一个算术表达式。

结果：返回X的双曲正弦。

结果类型：FLOAT BINARY

算法：SINH (X) = (EXP (X) −EXP (−X))/2

15.SQRT

格式：SQRT (X)

参数：X是一个算术表达式，X≥0。

结果：返回X的平方根。

结果类型：FLOAT BINARY

算法：牛顿法。

错误条件：如果X<0，产生ERROR条件。

16.TAN

格式：TAN (X)

参数：X是一个算术表达式。

结果：返回X的正切，X以弧度计。

结果类型：FLOAT BINARY

算法：如果cos (X) = 0，产生ERROR条件；

否则TAN (X) =SIN (X) /cos (X) 。

错误条件：如果cos (X) = 0，产生ERROR条件。

17. TAND

格式：TAND (X)

参数：X是一个算术表达式。

结果：返回X的正切，X以度计。

结果类型：FLDAT BINARY

算法：TAND (X) =TAN (X * π/180)

错误条件：如果cos (X * π/180) = 0，产生ERROR条件。

18. TANH

格式：TANH (X)

参数：X是一个算术表达式。

结果：返回X的双曲正切。

结果类型：FLOAT BINARY

算法：TANH (X) = (EXP(X)−EXP(−X)/(EXP (X) +EXP(−X))

12.3 串函数

1. BOOL

格式：BOOL (X, Y, Z)

参数：X是一个位表达式。

Y是一个位表达式。

Z是一个位串常数，4位长。

结果：根据X和Y返回一个布尔函数。由位串常数Z指定如下：设Z_1、Z_2、Z_3、Z_4是Z的从左到右的值。那么，位值A、B和四位串Z确定布尔函数BOOL(A, B, Z)：

A	B	BOOL (A,B,Z)
0	0	Z_1
0	1	Z_2
1	0	Z_3
1	1	Z_4

由此根据位串X、Y归纳函数BOOL(X、Y) 如下：如果X Y长度不同，那么在较短的位串后填0，直到二者长度相同。而后，BOOL (X, Y, Z) 的定义将是一个位串，使A是X的第N位，B是Y的第N位，可由上表得出结果的第n位。

结果类型：BIT (n)，其中n等于MAX (LENGTH (X), LENGTH (Y))

举例：BOOL ('0011'B, '0101'B, '1001'B) 返回'1001'B。

BOOL ('01011'B, '11', '1001') 返回'01100'

2. COLLATE

格式：COLLATE ( )

参数：无。

结果：返回一个长度为128字节的字符串，它由ASCII字符集的一组字符按升序排列组成，ASCII字符集在附录A中给出。

结果类型：CHARACTER (128)

3. INDEX

格式：INDEX (X, Y)

参数：X和Y是同一类型的串表达式，或者是位，或者是字符。

结果：返回一个整数值，指出在X中出现串Y的最左边的位置，如果X或Y是空 的或Y

并不出现在X中，则返回值是0。

结果类型：FIXED BINARY

4. LENGTH

格式：LENGTH (X)

参数：X是一个串表达式，或者是位串，或者是字符串。

结果：返回串X的字符数或位数，如果X具有属性VARYING LENGTH (X)，返回X的当前长度。

结果类型：FIXED BINARY

5. SUBSTR

格式：SUBSTR (X, I [, J])

参数：X是一个位串或字符串。

I是一个FIXED BINARY值。

J是一个FIXED BINARY值。

结果：返回从第I个元素开始的串X的拷贝，长度为J。如果未给出J，则假定它等于LENGTH (X) −I+1。

结果类型：与X相同。

错误条件：无，如果参数超出范围将得到一个不确定的结果。

6. TRANSLATE

格式：TRANSLAE (X, Y [, Z])

参数：X是一个字符表达式。

Y是一个字符表达式。

Z是一个字符表达式。

结果：如果Z不给出，则假定为COLLATE ( )；如果Y短于Z则在右边用空格填充，直到它的长度等于Z的长度；然后，若在Z中的某字符出现在X中，则由Y中相应于Z中该字符位置的字符去替换X中的相应字符。

结果类型：和X相同。

举例：TRANSLATE ('BDA'，'1234'，'ABC') 返回 '2D1'。

7. VERIFY

格式：VERIFY (X, Y)

参数：X是一个字符表达式。

Y是一个字符表达式。

结果：如果X中的每一个字符都在Y中出现，则返回值为0；否则，返回一个整数值，指出不在Y中出现的X的最左边的字符。

结果类型：FIXED BINARY

举例：

VERIFY ('ABCDE7', '7ABDE7') 返回3。

VERIFY ('ABC123', '1A2B3C4D') 返回0。

VERIFY (' A') 返回0。

VERIFY ('A', '') 返回1。

12.4. 转换函数

这些函数允许用户将一种数据项的类型转换为另一种类型、并被用作内部的自动类型转换。

1. ASCII

格式：ASCII (X)

参数：X是一个FIXED BINARY表达式。

结果：返回单个字符，它是ASCII排列顺序中的第x个字符。关于ASCII代码，参见附录A。

结果类型：CHARACTER (1)

算法：ASCII (X) =SUBSTR (COLLATE, I), MOD (X,128) +1,1)

注：ASCII (X) 是RANK (X) 的反函数。

2. BINARY

格式：BINARY (X [,P])

参数：X是一个算术表达式或可以转换为一个算术值的串表达式。如果X是一个具有非零标度因子的DECIMAL，那么P必须要给出，其中P是一个整常数，它指定了结果的精度。

结果：返回一个等于X的BINARY算术值。

结果类型：如X是FLOAT BINARY，结果是FLOAT BINARY，否则是FIXED BINARY。

3. BIT

格式：BIT (S[, L])

参数：S是一个算术或串表达式。L是一个正的FIXED BINARY表达式。

结果：当给出L时，将S转换为长度为L的位串；否则，S转换为位串，其长度由在第七节中给出的转换规则确定。

结果类型：BIT

4. CHARACTER

格式：CHARACTER (S [, L])

参数：S是一个算术或串表达式。L是一个正的FIXED BINARY表达式。

结果：当给出L时，S被转换为一个长度为L的字符串；否则，S被转换为字符串，其长度由第七节中所描述的规则确定。

结果类型：CHARACTER

5. DECIMAL

格式：DECIMAL (X [, P [, K]])

参数：X是一个算术表达式或是能转换为算术值的串表达式，P是一个整常数，$1 \leqslant P \leqslant 15$，K是一个整常数，$0 \leqslant K \leqslant P$。

结果：X转换为DECIMAL值，P和K是任选项。当指出时，它们分别表示精度和标度因子，若只给出P，则假定K为零；如果P和K都未指出，则结果的精度和标度将由第七节所给出的转换规则确定。

结果类型：FIXED DECIMAL

6. DIVIDE

格式：DIVIDE (X, Y, P [, Q])

参数：X和Y是算术表达式。P、Q是整常数，0≤Q≤P。

结果：返回X除以Y的值，具有精度P和标度Q，Q是任选项。如果X和Y都是FIXED BINARY，不能给出Q。

结果类型：X和Y的通用类型。

7. FIXED

格式：FIXED (X [, P [, K]])

参数：X是一个算术表达式或可以转换为算术值的串表达式。

P是一个整常数。

K是一个整常数。

结果：X转换为FIXED算术值。P和K是任选项，分别指定结果的精度和标度因子。如果只给出P，则假定K为零；如果P、K都未给出，则其精度和标度因子由第七节给出的转换规则确定。

结果类型：如果X是FIXED DECIMAL或CHARACTER，结果是FIXED DECIMAL，否则是FIXED BINARY。

8. FLOAT

格式：FLOAT (X [, P])

参数：X是一个算术表达式或能够转换为算术值的串表达式。P是一个任选的正整常数。

结果：X转换为FLOAT算术值，P是任选的，给出时决定结果的精度。如果不给出P，则精度由第七节中描述的转换规则确定。

结果类型：FLOAT BINARY

9. RANK

格式：RANK (X)

参数：X是一个长度为1的字符串值。

结果：返回一个ASCII字符X的整数表示。参见附录A。

结果类型：FIXED BINARY

算法：RANK(X)=INDEX(COLLATE( ), X)-1

10. UNSPEC

格式：UNSPEC (X)

参数：X是一个数据项引用，它在内存中的内部表示是16位或小于16位（8位）。

结果：返回由X所引用的地址的内容。

结果类型：位串。其长度等于数据项X的内部表示的长度。

12.5 条件函数

这些函数允许PL/I-80用户审查由于条件的发生而产生的中断。每一函数都没有参数。仅当在由于中断而进入的中断续元中执行时，返回一个值。该中断是由所使用的函数条件之一引起或是这样一个条件发出信号时引起的。

1. ONCODE

格式：ONCODE ( )

结果：返回的值是一个最近的PL/I-80运行出错的错误号。该错误将发出一个ERROR条件的信号。

结果类型：FIXED BINARY

2. ONFILE

格式：ONFILE ( )

结果：返回值是一个文件名，对于该文件有一个最近的ENDFILE或ENDPAGE条件信号。

结果类型：CHARACTER

3. ONKEY

格式：ONKEY ( )

结果：返回值是一个字符串，它给出了一个记录的键值，该记录引起了一个输入/输出条件或转换条件。

12.6 其他函数

1. ADDR

格式：ADDR (X)

参数：X是一个变量引用，它的存储是连续的。

结果：返回一个指针，指出变量X所被分配的位置。

结果类型：POINTER

2. DIMENSION

格式：DIMENSION (X, N)

参数：X是一个数组变量。

N是一个正的整数表达式。

结果：返回一个正整数，它表示数组X的第N维的域长。

结果类型：FIXED BINARY

3. HBOUND

格式：HBOUND (X, N)

参数：X是一个数组变量。

N是一个正整数表达式。

结果：返回数组变量X的第N维的上界。

结果类型：FIXED BINARY

4. LBOUND

格式：LBOUND (X, N)

参数：X是一个数组变量。

N是一个正整数表达式。

结果：返回数组X的第N维的下界。

结果类型：FIXED BINARY

5 LINENO

格式：LINENO (X)

参数　X是一个文件值。

结果　返回由X所引用的文件控制块的行号。

该文件控制块必须具有PRINT属性。

结果类型：FIXED BINARY

6. NULL

格式：NULL

结果：返回一个空指针值，即不指向任何单元的指针。

结果类型：POINTER

7　PAGENO

格式：PAGENO (X)

参数：X是一个文件值。

结果：返回一个由X所引用的文件控制块的页号。

该文件控制块必须具有PRINT属性。

结果类型：FIXED BINARY

附　录

附录A　ASCII代码表和转义字符

	N	HEX	ASCII	N	HEX	ASCII	N	HEX	ASCII
^@	0	00	NUL	43	2B	+	86	56	V
^A	1	01	SOH	44	2C	'	87	57	W
^B	2	02	STX	45	2D	−	88	58	X
^C	3	03	ETX	46	2E	.	89	59	Y
^D	4	04	EOT	47	2F	/	90	5A	Z
^E	5	05	ENQ	48	30	0	91	5B	[
^F	6	06	ACK	49	31	1	92	5C	\
^G	7	07	BEL	50	32	2	93	5D	]
^H	8	08	BS	51	33	3	94	5E	^
^I	9	09	HT	52	34	4	95	5F	−
^J	10	0A	LE	53	35	5	96	60	`
^K	11	0B	VT	54	36	6	97	61	a
^L	12	0C	FF	55	37	7	98	62	b
^M	13	0D	CR	56	38	8	99	63	c
^N	14	0E	SO	57	39	9	100	64	d
^O	15	0F	SI	58	3A	:	101	65	e
^P	16	10	DLE	59	3B	;	102	66	f
^Q	17	11	DC1	60	3C	<	103	67	g
^R	18	12	DC2	61	3D	=	104	68	h
^S	19	13	DC3	62	3E	>	105	69	i
^T	20	14	DC4	63	3F	?	106	6A	j
^U	21	15	NAK	64	40	@	107	6B	k
^V	22	16	SYN	65	41	A	108	6C	l
^W	23	17	ETB	66	42	B	109	6D	m
^X	24	18	CAN	67	43	C	110	6E	n
^Y	25	19	EM	68	44	D	111	6F	o
^Z	26	1A	SUB	69	45	E	112	70	p
^[	27	1B	ESC	70	46	F	113	71	q
^\	28	1C	FS	71	47	G	114	72	r
^]	29	1D	GS	72	48	H	115	73	s
^~	30	1E	RS	73	49	I	116	74	t
^_	31	1F	US	74	4A	J	117	75	u

32	20	SP	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	'	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

附录B PICTURE格式项

本附录描述附加性能——PICTURE输出数据格式项。它从PL/I-80的1.3版本开始提供。该性能依照ANSI委员会X3J PL/I子集G标准与全集ANSI PL/I标准一起提供。

B.0 Picture句法

PICTURE数据格式项用来按定点十进制格式编排数字数据进行输出。这种编排后的结果值是一个字符串，其形式将由数字值和PICTURE格式项中的形象指明表确定。PICTURE格式项的句法是：

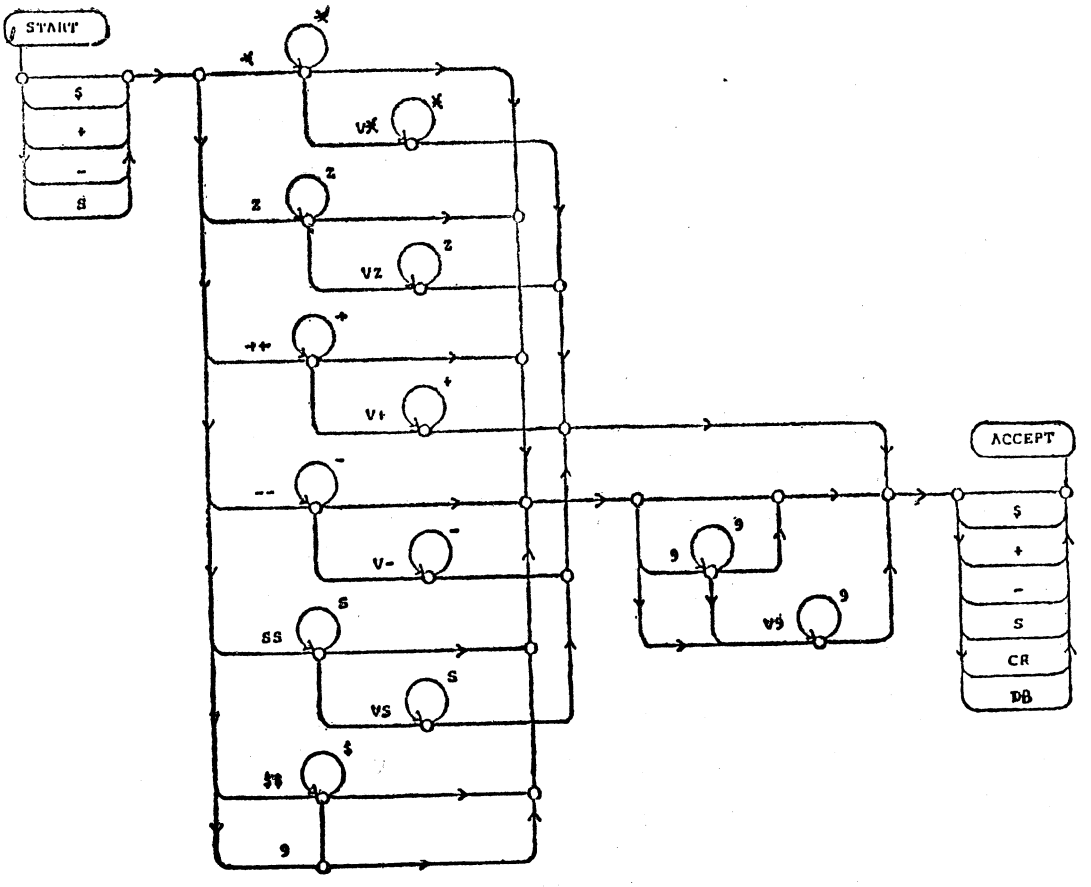
P〈**指明表**〉

其中〈**指明表**〉是一个描述形象指明表的字符串。这种格式项可以用在PUT EDIT语句中，用法与任何其他的数据格式项相同（参见10.6）。用来描述形象指明表的字符串常数必须是用下列的一个或多个字符组成：

$ + − S	固定或浮动字符
* Z	条件数字字符
9	数字字符
V	小数点位置字符
/, : B	插入字符
CR DB	贷方和借方字符

且必须满足一定的句法规则。首先，插入字符可以出现在一合法〈指明表〉中的任何位置，但不能将形象字符对CR和DB分开。若将形象指明表中所有的插入字符去掉，其结果字符串必须是图B.1中的有限状态识别器能够接受的，必须能从START节点开始沿着该示意图到达ACCEPT。其中的转换越过边界时要求或者边界是无标号的，或者边界上的标号是〈指明表〉中的下一字符。例如，下列的字符串常数定义合法的形象指明表：

```
'BB$***, ***V.99BB'
'S----, 999V.99BCR'
'99:99:99'
':BBB$SSSSS, SSS.VSSBBB:'
```



图B.1 形象指明表识别器

B.1 Picture语义

形象指明表将数字值编排为字符串值的方式是由出现在指明表中的形象字符的类型确定的。字符'$'、'+'、'－'和'S'既可作为固定的字符，也可作为浮动字符出现。如果这种字符在形象指明表中仅出现一次，则该字符是固定字符，否则便是浮动的。如果是浮动的，则除与条件数字对应的数位外，其他位都要出现。对于两种情形，这些形象字符都将与数字值的符号一起决定输出字符。输出由下表确定，每一字符在输出中占据一位。

固定/浮动	字符			
符　号	S	+	－	$
正	+	+		$
负	－		－	$

如形象字符是固定的，输出字符将出现在相应的输出位置，如形象字符是浮动的，那么输出字符出现在浮动形象字符范围中第一个非零数字前或在浮动字符的最后一位。其他各浮

动字符则由空格代替，消去对应于数字值中的零数位。

字符'*'和'Z'称作条件数位形象字符或消零字符。在形象指明表中，它们与数字值中的数位相对应。在输出时，如果对应的数位是零，输出字符分别为'*'或空格，若对应位非零，输出为数字字符。

形象字符'B'、'/'、'.'、':'和','称作插入字符（在ANSI中，没有定义字符':'为插入符，但在PL/I-80中，为了显示表示时间的数字数据，添加了该字符）。插入字符将使该字符出现在相应的输出位置（B插入结果为空格）。但出现在浮动字符或消零字符中的插入字符除外。如果插入字符出现于浮动域或能使数字数位消去的消零字符中，插入字符也按上述规则消去。

注意，在某些版本中，'B'是无条件插入符，即它总使相应位输出一个空格。但根据ANSI标准，这种空格在输出中可以由浮动字符或消零字符'*'重写。

形象字符'9'在形象指明表中表示数值的数字将出现在相应的输出位置上。因此，'9'是一个无条件数字位。

数值中的数字与形象指明表中的数字数位之间的对应关系由形象字符'V'建立。该字符仅表示整数位的结束和小数位的开始，由此表明形象指明表与数字值的对齐关系。如果并未出现该字符，则假定由形象指明表所隐含的所有数位位置引用为整数数位(无小数)。这样，结果中将不出现数值的任何小数位。注意，'V'形象字符仅是一个字符而已，并不对应于结果中的字符位置。这样，如果没有'V'，结果的长度是形象指明表的长度，若出现'V'，则比其少1。字符'V'在消去字符中也有一定的作用。例如，对小数数位，除了将所有的字符都消去（输出全是空格）的情况外，相应于V后面的数位是不消去的。在'V'的后面，如果消去状态是ON，则将转换为OFF。由于在'V'形象字符后面有插入字符，如小数点，故除了全被消去的情况外，插入字符并不消去。

字符对'CR'和'DB'表示"贷方"和"借方"，它们看作是符号字符。如果这种字符出现在形象指明表中，且数值的符号为负，那么，指定的字符对将出现在结果中。如果数字值是正的，那么，相应这种字符位置是两个空格。

除上述的规则以外，在缺省情况下有一些通用的规则。如果数值是零，且形象指明表中没有形象字符'9'，那么，若在所有位置都用了形象字符'*'，则输出结果全是'*'，否则，输出全是空格。该规则优先于上述的其他规则。另外，如数值的符号为负，且在指明表中没有指定任一符号形象字符S、+、-、CR或DB，则将发出转换错误信号ERROR(1)。

每一个形象指明表都根据下述规则对结果中的数字值隐含指定了精度和标度。插入字符和字符对'CR'、'DB'对精度和标度没有影响。除结果的精度小于固定/浮动字符数或消零字符数加字符'9'的个数外，若没有'V'，结果标度为零，若存在'V'，结果的标度等于在字符'V'后所出现的浮动字符的数或消零字符数或字符'9'的个数。

图B.2给出了一些使用PICTURE数据格式项的规则。

0.00	BB$***,***V.99BB	$*******.00
0.01	BB$***,***V.99BB	$*******.01
0.25	BB$***,***V.99BB	$*******.25
1.50	BB$***,***V.99BB	$****** 1.50

12.34	BB$***,***V.99BB	$*****12.34
123.45	BB$***,***V.99BB	$****123.45
1234.56	BB$***,***V.99BB	$***1,234.56
12345.67	BB$***,***V.99BB	$*12,345.67
123456.78	BB$***,***V.99BB	$123,456.78
0.00	$SSSSBSSSV.SS	
0.01	$SSSSBSSSV.SS	$ +0.01
0.25	$SSSSBSSSV.SS	$ +0.25
1.50	$SSSSBSSSV.SS	$ +1.50
12.34	$SSSSBSSSV.SS	$ +12.34
123.45	$SSSSBSSSV.SS	$ +123.45
1234.56	$SSSSBSSSV.SS	$ +1 234.56
12345.67	$SSSSBSSSV.SS	$ +12 345.67
123456.78	$SSSSBSSSV.SS	$ +123 456.78
0.00	99/99/99	00/00/00
0.01	99/99/99	00/00/00
0.25	99/99/99	00/00/00
1.50	99/99/99	00/00/02
12.34	99/99/99	00/00/12
123.45	99/99/99	00/01/23
1234.56	99/99/99	00/12/35
12345.67	99/99/99	01/23/46
123456.78	99/99/99	12/34/57
0.00	**:**:**	********
0.01	**:**:**	********
0.25	**:**:**	********
1.50	**:**:**	*******2
12.34	**:**:**	******12
123.45	**:**:**	****1:23
1234.56	**:**:**	***12:35
12345.67	**:**:**	*1:23:46
123456.78	**:**:**	12:34:57
0.00	/++++,+++.V++/	
0.01	/++++,+++.V++/	/ +01/
0.25	/++++,+++.V++/	/ +25/
1.50	/++++,+++.V++/	/ +1.50/
12.34	/++++,+++.V++/	/ +12.34/
123.45	/++++,+++.V++/	/ +123.45/
1234.56	/++++,+++.V++/	/ +1,234.56/

12345.67	/++++, +++.V++/	/ +12,345.67/
123456.78	/++++, +++.V++/	/+123,456.78/
0.00	s***b****.v**	**********
−0.01	s***b****.v**	−*******01
0.25	s***b****.v**	+*******25
−1.50	s***b****.v**	−*****1.50
12.34	s***b****.v**	+*****12.34
−123.45	s***b****.v**	−****123.45
1234.56	s***b****.v**	+**1 234.56
−12345.67	s***b****.v**	−*12 345.67
123456.78	s***b****.v**	+123 456.78
0.00	$SSSSBSSSV.SS	
−0.01	$SSSSBSSSV.SS	$ −01
0.25	$SSSSBSSSV.SS	$ +25
−1.50	$SSSSBSSSV.SS	$ −150
12.34	$SSSSBSSSV.SS	$ +1234
−123.45	$SSSSBSSSV.SS	$ −12345
1234.56	$SSSSBSSSV.SS	$ +123456
−12345.67	$SSSSBSSSV.SS	$ −1234567
123456.78	$SSSSBSSSV.SS	$ +12345678
0.00	***.***S	********
−0.01	***.***S	*******−
0.25	***.***S	*******+
−1.50	***.***S	*****2−
12.34	***.***S	*****12+
−123.45	***.***S	****123−
1234.56	***.***S	**1.235+
−12345.67	***.***S	*12.346−
123456.78	***.***S	123.457+
0.00	$***, ***v**cr	**********
−0.01	$***, ***v**cr	$*******01CR
0.25	$***, ***v**cr	$*******25
−1.50	$***, ***v**cr	$******150CR
12.34	$***, ***v**cr	$*****1234
−123.45	$***, ***v**cr	$****12345CR
1234.56	$***, ***v**cr	$**1.23456
−12345.67	$***, ***v**cr	$*12.3457CR
123456.78	$***, ***v**cr	$123.45678
0.00	/++++, +++.V++/	

−0.01	/++++. +++ V++/	/	01/
0.25	/++++. +++ V++/	/	+25/
−1.50	/++++. +++. V++/	/	1.50/
12.34	/++++. +++ V++/	/	+12.34/
−123.45	/++++. +++ V++/	/	123.45/
1234.56	/++++. +++. V++/	/	+1,234.56/
−12345.67	/++++. +++ V++/	/	12,345.67/
123456.78	/++++. +++ V++/	/	+123,456.78/

图 B 7　编辑数值数据PICTURE实例

附录 C　外部过程

本附录描述在局部过程定义中使用的EXTERNAL的用法。它在 PL/I-80 版本1.3及其后版本中提供使用。它是一个非标准特性，如果要求与其他的子集—G 语言向上兼容，则要避免使用它。

将分别编译的过程集组合成为单一的编译过程常常是很有用的。这些过程引用同一全程数据。根据子集—G标准，每一子程序必须要分别地编译。全程数据则要在每一编译中复制，然后再用连接编辑将单独的模块组合在一起产生一个最终的目标模块。在PL/I-80中，为使该过程能存取外部过程，可在过程句首用EXTERNAL 属性，为与数据研究公司以后的PL/I语言版本兼容，仅在最高层过程标志以EXTERNAL属性，所有的全程数据则标为STATIC。含有一组 EXTERNAL 子程序的编译仅由子程序组成，而没有主程序。下列程序段给出了EXTERNAL 属性的用法：

```
module:
  proc;
  dcl
    1  global  data static,
       2  a_field char (20) var  init (' '),
       2  b_field fixed init (0),
       2  c_field float init (0);
  set  a:
    proc (c) external;
  dcl c char (20) var;
      a_field=c;
    end set_a
  set  b
    proc (x) external;
    dcl x fixed;
    b_field=x;
```

```
  end set_b;
set_c:
  proc (y) external;
  dcl y float;
  c_field = y;
  end set_c;
sum:
  proc returns (float) external;
  return (b_field+c_field);
  end sum;
display:
  proc external;
  put skip list (a_field, b_field, c_field);
  end;
end module;
```

上述程序定义了五个外部过程：set_a、set_b、set_c、sum 和display，在下面给出的程序中存取了四个过程：

```
call_ext.
  proc options (main);
  dcl
    set_a entry (char (20) var);
    set_b entry (fixed),
    set_c entry (float),
    sum returns (float),
    display entry;
  call set_a ('Johnson, J');
  call set_b (25);
  call set_c (5.50);
  put skip list (sum ( ));
  call display ( );
  end call_ext;
```

这两个程序分别地进行编译并连接编辑在一起形成一个单独的模块。注意，由于连接编辑格式的限制，外部名太长时要从右边截掉，因此必须在前六个字符上是唯一的。

SuperSoft C 编译程序使用手册

第一章 引 言

SuperSoft C 编译程序基本上接受全功能C语言，但有少量的例外，在本手册附录A中列出。C语言的标准参考手册是Brian W·Kernighan和Dennis M·Ritchie合著的《程序设计语言C》(The C Programming Language)。该书已有中译本，本手册不再包括其中的内容。

SuperSoft C编译程序是自编译、优化、两遍扫描的编译程序，它以汇编源代码方式生成最终输出文件。它适用于多种操作系统和CPU。目前可用的机器和操作系统配置见附录B。由于C语言本身具有可移植性，对于这个特定的编译程序实现，要移植到其他操作系统配置或其他机器上，还是比较容易的。

SuperSoft C编译程序大部分是以C语言书写的，这样做不但便于开发和维护，而且提供了对编译程序进行测试的最直接工具。

由于执行了代码优化，本编译程序所生成的代码在空间和时间方面的效率足以满足系统级和应用级程序设计的要求。以CP/M操作系统为例，要连接到该操作系统，除了10行过程需要用汇编代码之外，编译程序整个I/O接口都是以SuperSoft C编写的。由此可见其效率之一斑。

SuperSoft C是模块化的两遍扫描编译程序。每一遍扫描由独立的编译程序段执行。这样做有很大的好处。将编译程序分为两个独立的程序使它运行时可占有相对小的可用内存，但仍然具有C语言的绝大部分功能。这种模块化的结构也使得第一遍扫描即语法检查（CC）和第二遍扫描即代码优化/生成（C2）之间有清晰的接口。第一遍扫描的输出文件简单地作为第二遍扫描的输入文件。模块化还使该编译程序更容易适应其他机器和操作系统，因为第一个模块是与机器和系统无关的，只有第二个模块中一部分需要修改。

编译程序第一遍扫描，将SuperSoft C源代码文件接收作为输入，对它作语法检查。它生成的输出文件是以中间代码形式的文件，称为通用代码或U代码（即不是专用于任何一种系统或机器的）。该编译程序的设计思想之一，是其两遍扫描输出都可以被人看懂，因此，可以使用通常的文本编译程序检查和修改U代码。一种与机器无关的U代码到U代码优化程序正在开发之中，目前尚未提供。

编译程序的第二遍扫描接收CC生成的U代码文件作为输入。输入文件经过复杂的优化过程，包括代码重新组织，在函数内部作代码修正以及在某些代码行的“窗口”内部作多次重复的代码转换。转换过程与优化过程结合在一起，最后生成以汇编源代码形式的输出文件。

选择汇编代码作为编辑程序的最终输出有几个好处，有些对于系统程序员特别有价值。因为所生成的代码不是专门用于现有机器中所用的某一种汇编语言，实际上，任何一种硬件上兼容的汇编程序（包括生成浮动代码的汇编）都可使用。所以，该编译程序的输出可以与现有任何用汇编源代码写的软件系统结合，或与之兼容。程序员也可以用编译伪指令#ASM和#ENDASM直接将若干行汇编代码插入C源文件中。对这样插入的代码行，编译程序不以任何方式作修改或优化。

使用汇编源代码还可以满足两遍扫描输出都可被人看懂这个设计要求。编译输出的可读性，既方便调试，又可使用户能看到编译生成的是什么样的代码。程序员可对生成代码进行修改，也可以修改源代码，使之更能适合特殊的应用要求。

对于任何编译程序的设计，主要是在编译一个应用程序所需的时间和执行该程序所需的时间这两者之间进行权衡。既然我们的主要设计目标之一是能有效地用于系统级程序设计，因此，我们更强调执行速度，而牺牲编译的速度（C2 执行的某些优化所需的时间与被优化的最大C函数大小的平方成比例）。这样做的结果，虽然增加了程序开发时间，但却使我们的编译程序能用于更广泛的程序设计任务。SuperSoft C 编译程序使用户能在相对小的硬件系统上进行有效的系统级程序设计，并使程序在结构化和清晰度方面都比较好，这不能不说是它的独到之处。

后面各章节提供了使用该编译程序所需的信息，并对它的特性作了一些描述。其中，第二章主要叙述如何使用SuperSoft C编译程序，以及它支持的预处理伪指令。第三章描述了它所提供的标准库函数。第四章叙述如何把代码插入C2的运行时库C2.RH。关于SuperSoft C 和标准的UNIX C （第7版本）之间的差别，以及SuperSoft C 目前可用的机器和操作系统配置在附录A和B中分别列出。

第二章　使用C编译程序

本章介绍如何使用C编译程序，内容包括：

在编译程序磁盘中包含哪些文件；

哪些文件是编译用户程序所必须的；

还需要其他什么软件；

编译命令行语法；

本编译程序在特定操作系统环境（如CP/M）下的使用；

用于CC和C2的命令行任选；

在SuperSoft C中支持的编译预处理伪指令以及如何重新确定编译最终输出代码在RAM或ROM单元的起始地址。

一、在CP/M下提供的文件

如果你有CP/M版本，则在工作拷贝上应该有下列文件：

CC.ext	:	编译程序第一遍扫描，语法分析程序
C2.ext	:	第二遍扫描，代码优化／生成程序
C2.RH	:	C2用于ASM的运行时库头部文件
C2.RT	:	C2用于ASM的运行时尾部文件
C2.RTM	:	用于RMAC和M80的运行时尾部文件
C.SUB	:	编译用的CP/M SUBMIT文件
STDIO.H	:	标准I/O函数头
STDIO.C	:	标准I/O函数
ALLOC.C	:	动态内存分配函数
CRUNT2.C	:	运行时C语言部分
FUNC.C	:	辅助函数
FORMATIO.C	:	printf、scanf函数等
*.REL	:	相应文件的RMAC（Digital Research）和M80（Microsoft）"REL"格式
SAMP?.C	:	测试C的各种功能的样本程序

CC.ext和C2.ext是编译程序文件，对这两个可执行文件采用什么扩展名，依赖于操作系统。在CP/M-80下，扩展名为".COM"。C2.RT，STDIO.H，ALLOC.H和AUXFN.H是内部构造函数文件。在磁盘上，带扩展名".C"的其他文件，是以SuperSoft C源代码提供的样本程序。

"REL"文件是为方便那些使用Digital Research的RMAC或MicroSoft的M80汇编软件包的用户的。如果你没有与之兼容的汇编程序，则可以不管这些文件。

在8086编译的情况下，第二遍扫描分为COD2COD（优化程序）和C2I86（代码生成程序），C2.RH和C2.RT分别替换为C2I86.RH和C2I86.RT，不包含C2.RTM和".REL"文件。

二、所需的文件

如果不使用浮动汇编程序，则在编译一个程序时，上节列出的前4个文件总是需要的。如果使用浮动汇编程序，则需要C2RT.REL。如果不使用重定位，编译程序的第二次扫描，C2，在编译期间，自动将运行时库中包含的所有代码段和函数以及C2.RH、C2.RT，插入用户程序中。否则，用户必须连接运行时库函数，每个".H"或".C"文件仅当用户程序调用它的一个或多个函数时才需要。但是，这些函数是基本的组成部分，大多数用户程序都要调用多个这样的函数。如果用户程序不调用最后三个文件中定义的函数，则必须将这些文件中所有必要的函数和数据定义插入，包括被用户程序调用的函数所调用的或需要的函数和数据定义。

由于编译程序的最终输出是机器汇编源码形式，因此，还需要与所用操作系统及硬件兼容的汇编程序，以及装入和运行程序所需的软件。

三、在ZMOS下提供的文件

如果你有ZMOS版本，则在工作拷贝盘上应有下列文件：

CC.IMAG	:	编译程序第一遍扫描，语法分析程序
C2Z8001.IMAG	:	第二遍扫描，代码优化/生成程序
WHEADER.REG	:	C2用于ASM的运行时库头部文件
WTRAILER.REG	:	C2用于ASM的运行时尾部文件
ZSTDIO.REG	:	标准I/O函数
ALLOC.REG	:	动态内存分配函数
CRUNT2.REG	:	运行时C语言部分
FUNC.REG	:	辅助函数
FORMATIO.REG	:	printf，scanf函数等
SAMP?.REG	:	测试C的各种功能的样本程序

四、命令格式和顺序

为编译SuperSoft C的源程序，使用下列命令文件和序列：

CC 文件名.C … 任选…

C2 文件名.COD任选…

在调用CC时，可在命令行列出任意个文件名，文件名之间用空格分隔开。文件名指的是你所用操作系统要求的文件说明的前缀部分。所指定文件无须包含完整的功能，因为所有文件均将进行语法分析，按其所列顺序，好象一个文件一样。U代码输出文件赋值给第一个给定文件名。当执行C2时，只能指定一个文件名。

按规则，C源代码文件的扩展名或后缀为".C"，但其他扩展名也是合法的。CC自动地为其U代码输出文件提供扩展名".COD"，C2的汇编代码输出文件自动地给定扩展名".ASM"。

对CC和C2，命令行任选项之间均应用空格分隔开。编译程序通常的操作方式由缺省条件指定，无须列出任选项。但是，如果在非8080、8085、8086或Z80的CPU运行C编译程序，则必须将三个与机器有关的任选项+J、-I和-P，设置为对所用CPU合适的值。

下面我们以8080系列CPU在CP/M-80操作系统下为例，叙述编译运行SuperSoft C程序必须遵循的过程。我们假定已准备好编译运行第一个样本C程序SAMP1.C，首先检

在磁盘上具有全部必须用到的文件，并且你的程序包括所需的全部数据和函数定义（除C2.RT中的定义外），然后就可以开始。

五、使用标准的CP/M-80汇编程序ASM进行编译

为了把SAMP1.C转换成可执行的命令文件，以便在CP/M-80下运行，必须按顺序打入下列每条命令：

```
CC  SAMP1.C
C2  SAMP1.COD+ASM
ASM  SAMP1
LOAD  SAMP1
```

上述每条命令均导致在包含SAMP1.C的磁盘上建立一个新文件，这些文件的名分别为：

```
SAMP1.COD   ; CC的输出
SAMP1.ASM   ; C2的输出
SAMP1.HEX   ; ASM的输出
SAMP1.COM   ; LOAD的输出
```

建立的最后一个文件是对用户程序的可执行命令文件，这是用户程序可在CP/M下运行的唯一形式（其他文件均可删去）。一旦建立了该文件，只需打入：

```
SAMP1   打回车键
```

就可以开始执行你的程序。

CC、C2和ASM还可能产生出错信息，指出用户程序中的毛病。CC和C2产生的错误信息大部分是自解释性的，ASM产生的错误信息在有关的CP/M手册中提供。有两类ASM错误信息可由于C程序编写不好而产生：1)标号值错和多重定义标号通常由于重复定义C外部符（包括C函数）引起。注意，外部符的有效前导字符数取决于所用的汇编程序及装入程序。2)未定义标号通常由于未定义C外部符（包括C函数）引起。

在进行程序研制时，经常要多次重复上述过程，除指定文件名外，不用作其他改变。如果能够使上述过程只用一条命令便可执行，可以节省大量的时间。在CP/M下，SUBMIT命令和SUBMIT文件提供了这种功能。其他操作系统也提供了类似的功能。

一个文件名为C.SUB的文件，包含下列命令行：

```
CC  $1.C
C2  $1.COD+ASM
ASM  $1.AAZ
LOAD  $1
$1
```

其中，"$1"是符号参数，可用SUBMIT命令的第一个实在参数替换。

在磁盘上建立上述文件后，打入：

```
SUBMIT C SAMP1   按回车键
```

如果CP/M在当前磁盘上找到C.SUB，则执行下列五条命令（对每个$1均已由SAMP1替换）：

```
CC  SAMP1.C
```

```
C2  SAMP1.COD+ASM
ASM  SAMP1
LOAD  SAMP1
SAMP1
```

关于SUBMIT文件的详细描述，请参阅CP/M 用户手册。

六、用标准CP/M-86汇编程序ASM86编译

为把SAMP1.C转换成可执行的命令文件，以便在CP/M-86 下运行，可按下列顺序打入每一条命令：

```
CC  SAMP1.C
COD2COD  SAMP1.COD
C2I86  SAMP1.U
ASM86  SAMP1
GENCMD  SAMP1 DATA〔X1000〕
```

上述每一条命令均导致在包含 SAMP1.C 的磁盘上建立一个新文件，文件名分别 为：

```
SAMP1.COD  ;  CC的输出
SAMP1.U    ;  COD2COD的输出
SAMP1.A86  ;  C2I86的输出
SAMP1.H86  ;  ASM86的输出
SAMP1.CMD  ;  GENCMD的输出
```

所建立的最后一个文件是对用户程序的可执行命令文件。这是用户程序可 在 CP/M-86 下运行的唯一形式(其他文件可以删去)。一旦该文件建立，只须打入：

```
SAMP1    按回车键
```

则开始执行你的程序。

CC COD2COD、C2I86和ASM86 还可产生错误信息，指出你的程序中的毛病。CC 和C2产生的错误信息大部分是自解释性的，ASM86产生的错误信息在CP/M 手册中提供。有两类ASM86错误信息可由于C程序编写不好而产生：1)标号值错和多重定义标号 通常由于重复定义C外部符（包括C函数）引起。注意，外部符的有效前导字符数取决于所用的汇编程序及装入程序。2)未定义标号通常由于未定义C外部符（包括C函数）引起。

在程序研制过程中，经常要多次重复上述过程，只是文件名要作改变。在CP/M下，用SUBMIT命令和SUBMIT文件，只须打入一条命令，便可执行上述整个过程。

一个SUBMIT文件，文件名为C.SUB，应包含下列命令行：

```
CC  $1.C
COD2COD  $1.COD
C2I86  $1.U
ASM86  $1 $$PZ
GENCMD  $1 DATA〔X1000〕
$1
```

其中“$1”是符号参数，可被SUBMIT命令的第一个实在参数替换。

在你的磁盘上建立上述文件后，打入：

SUBMIT C SAMP1 按回车键

如果CP/M在当前磁盘上找到C.SUB，则执行下列五条命令：

CC SAMP1.C

COD2COD SAMP1.COD

C2I86 SAMP1.U

ASM86 SAMP1

GENCMD SAMP1 DATA [X1000]

七、用M80和L80（浮动汇编程序）编译

使用M80和L80编译的过程基本上和前面所述相同，不同的是编译产生REL模块，这些模块要进行连接（该过程与RMAC的过程类似），过程如下：

CC SAMP1.C

C2 SAMP1.COD

M80 SAMP1, =SAMP1.ASM

L80 SAMP1, FUNC, CRUNT2, STDIO, ALLOC, C2RT.SAMP1/N/E: CCSTAR

上述每一条命令均导致在当前登录磁盘上建立一个新文件，文件名是：

SAMP1.COD ；CC的输出

SAMP1.ASM ；C2的输出

SAMP1.REL ；M80的输出

SAMP1.COM ；L80的输出

用户使用上述过程时，需要有下列REL格式的文件：

ALLOC.REL ；从ALLOC.C产生

STDIO.REL ；从STDIO.C产生

CRUNT2.REL；从CRUNT2.C产生

FUNC.REL ；从FUNC.C产生

C2RT.REL ；从运行时库C2RT.MAC产生

在盘上还需要有C2.RTM文件。

对浮动汇编程序的SUBMIT文件是

CC $1.C

C2 $1.COD

M80 $1, =$1.ASM

L80 $1, CRUNT2, FUNC, STDIO, ALLOC, C2RT, $1/N/E；CCST-AR

八、CC命令行任选项

CC命令行任选项列出如下（缺省值或者无动作，或者如所指定动作）：

-G 关闭U代码生成，该任选用于对源程序文件进行语法检查。选择-G和+L任选，可产生编号的带宏定义扩展的C源程序清单。缺省是生成U代码。

+CO 强制输出至控制台，建立一个长度为零的输出文件。缺省（-CO）是输出至文件名.COD。

+LNO　置源程序行号于输出文件中相应的U代码首行的前面。输出还指示出由#INCLUDE伪指令组合的每个文件的开始行号。嵌套的多个"#INCLUDE"也清楚地指示出来。

+L　在U代码输出文件中列出每行C源代码作为注释。源程序行是按功能分组的，因此与相应的U代码行只是松散相关。宏定义被展开，行号出现在每行源代码之前。

+F　在处理每一行C源程序之后，立即冲去（写至文件中）生成的所有U代码行。如果同时选择+L和+F任选项，则放入输出文件的每一源程序注释行后面，紧接着是由该源程序行生成的所有U代码行。选择该任选项使CC和C2的执行减慢，并禁止通常由C2执行的某些优化功能。

+CR　在生成的每个U代码输出文件后附加回车符，某些文本编辑程序（如CP/M的ED）需要这种格式。

+A　将CC所用的最后一个内存单元地址放入输出文件中作为注释的每行C源代码前面。如果同时选择了+L任选，则+A任选无效。

+DDT　将其他调试程序信息插入输出文件中。

-N　后面紧跟一个整数，指定为CC生成的顺序编号的标号初值。缺省为初值2。

九、与机器有关的CC命令行任选项

下列三个任选项用于在特定的目标处理机（CPU）下剪裁CC。它们是仅有的与机器有关的任选项，对8080系列CPU，每个任选项均设置了适当的缺省值。如果这些任选项设置的值对你所用的CPU不合适，则编译程序不能正确运行（下面出现的jflag，int-size和pad-size是编译程序内部的变量名）。

+J　置jflag等于1。当表示一个字符（char）的单个字节作为局部参数传递给一个函数时，加宽至int指定的字节数。

置jflag为1使该字节被存放在它指定的存储单元的最右字节，该设置对z8000是合适的。缺省条件是置jflag等于0，这使得代表一个字符的同一字节，被存放在它指定的存储单元的最左字节，这个设置适合于8080系列CPU。

-I　后面紧跟一个整数，指定int的字节数（int-size），缺省是2字节。

-P　后面紧跟一个整数，对所有整数地址指定适当的模数（pad-size），即每个整数地址模pad-size的值等于0。换言之，每个整数地址均为pad-size值的倍数。缺省值为模1。

对特定的目标处理机，剪裁编译程序所需的上述三个任选项，可通过使用下面定义的组合标志，在一个操作中设置。

+i8080　设置jflag=0，int-size=2，pad-size=1。目标处理机为Intel 8080，8085，8086，8088，186，188和286。

+Z80　设置值同上，目标处理机是Zilog Z80。

+Z8000　设置jflag=1，int-size=4，pad-size=2。目标处理机是Zilog Z8000。

+Z8001　设置jflag=1，int-size=4，pad-size=2。目标处理机是Zilog Z8001。

+Z8002　设置jflag=1，int-size=2，pad-size=2。目标处理机是Zilog Z8002。

缺省条件与+i8080与+Z80的结果相同。

十、C2 命令行任选项

C2命令行任选项列出如下（缺省值无动作，或如所指定）：

－G　关闭汇编代码生成，缺省是生成汇编输出代码。

－O　关闭优化，只将U代码转换为汇编代码，缺省是执行优化。

－Q　后面紧跟一个整数，指定优化级别，缺省是全功能优化（目前未实现）。

－X　使生成代码较少，执行速度减慢，优化时以速度换取空间。缺省是以空间换取速度。但两者差别不大，因为大部分优化对于速度和空间利用率均有好处。

＋CO　强制输出至控制台，可以查看输出。只建立长度为零的输出文件。缺省是输出至文件名.ASM。

＋L　将每行U代码放入汇编代码输出文件中作为注释。U代码行按功能分组。

＋PRGL　将当前被优化的函数名显示在控制台上，指示C2执行到什么地方。

＋OC　跟踪优化过程，如果代码改变，则写信息至输出文件，指示出代码的变化。该任选减慢C2执行速度，并大大增加输出量。

＋T　生成代码并插入到输出文件中，这使得在运行时，每个函数每一次进入时，均显示出函数名。该任选项对于调试程序很有用，使程序员可以跟踪程序控制流程（目前未实现）。

＋Z　后面紧跟一个字符串，C2用这个字符串作为它生成的所有标号的前缀。缺省前缀为“C”。

－RH〈文件名〉　优化程序使用指定文件作为运行时头部文件（注意，－RH和文件名之间无空格）。

－RT〈文件名〉　优化程序使用指定文件作为运行时尾部文件（注意，－RT和文件名之间无空格）。

－ENT〈关键字〉　定义一个关键字，用于说明浮动汇编程序中标号的入口点（注意，－ENT和关键字之间无空格）。

－EXT〈关键字〉　定义一个关键字，用于说明浮动汇编程序中的外部标号（注意，－EXT和关键字之间无空格）。

－ORGnnnn　强制输出代码开始地址取nnnn指定的十进制值。

十一、编译预处理伪指令

所有SuperSoft C编译的输入文件均要经过预处理，预处理程序扫描输入文件，查找以“＃”开头的行。这些行本身不被编译，而是用于控制编译过程，称为编译控制行或预处理指示句。这些行以新行字符结束，而不用分号。在SuperSoft C中支持的预处理伪指令有：＃DEFINE，＃INCLUDE和一对＃ASM，＃ENDASM语句。除＃DEFINE伪指令外，其他每个指示语句在语法上与C语言其他语句无关，可以出现在程序中任何地方。所有指示语句直至编译结束均有效。下面分述各指示句的作用。

1.　＃DEFINE　伪指令

如果在输入源程序中发现下列形式的行：

＃DEFINE　标识符　记号串

则预处理程序将后继出现的所有指定标识符均用指定的记号串替换（记号是程序设计语言中的词法单位，诸如标识符、操作符、关键字等等）。这是最简单的宏代入形式。如果＃DE-

FINE后的标识符在双引号或单引号内，则不作替换。每个替换记号串均被再次扫描，查找是否还有其他的#DEFINE标识符，因此，可以允许嵌套的#DEFINE指示句。如果重复定义一个标识符将导致编译错误，则对参数化的#DEFINE不支持。

该指示句最常用于在程序头部定义一组符号常数，供后面的函数使用（这远比在程序语句中插入文字常数要好）。下面是这种用法的一个简化例子：

```
#DEFINE BSIZ 0×100
char Buf1 〔BSIZ〕,Buf2 〔BSIZ〕;
main ()
{
    register int i;
    for (i=0; i++<BSIZ;){
            Buf1 〔i〕 = Buf 〔2〕 = 0;
}
```

通过#DEFINE指示句作宏代入还有许多其他用途，详细说明请参阅《程序设计语言C》。

2. #INCLUDE伪指令

如果下列形式的行：

#INCLUDE "驱动器名：文件名"

#INCLUDE〈驱动器名：文件名〉

输入至编译程序，则编译预处理程序将以指定驱动器中名为指定文件名的整个文件内容替换该行。如果文件找不到，则预处理程序在当前磁盘上查找该文件。将来（现版本中没有这个功能），如果使用#INCLUDE〈文件名〉指示句，则预处理程序将在所有可用的驱动器上（即迄今为止已用到的所有驱动器）查找指定的文件。在#INCLUDE文件中发现任何#INCLUDE指示句，也按同样方法处理。因此，允许嵌套的#INCLUDE指示句。

3. #ASM，#ENDASM伪指令

使用SuperSoft C的这一特殊功能，可将汇编代码行直接插入C语言源程序文件中，如下所示：

```
⋮
putchar ('y');
#asm
      mvi a, 88
      call output

#endasm
crlf (^);
⋮
```

在#ASM和#ENDASM之间的所有程序行，经过编译程序的两个模块，不作任何改变，被插入到最终输出文件的相应位置上。在被插入的汇编代码中，不应该使用以"C"开头的标号。因为编译程序生成以"C"为前缀的标号，这样，在程序被汇编时，可能导致重复标号错。

十二、重新确定编译生成代码的起始地址

浮动汇编生成的目标代码中，内存地址是以相对程序原点的位移或作为外部引用给出的。外部引用是使用符号标号，引用另一程序或外部数据区中的某一单元。相对程序原点是特定程序的开始位置，该程序的所有内部地址都相对于它来计算。在该程序被装入到某一绝对开始地址之前，其地址一直是相对的。程序装入的绝对开始地址就是该程序的绝对原点。相对于绝对原点计算出来的所有地址，都是绝对地址。

浮动汇编程序可用一些方法指定你的程序的相对地址或绝对地址单元应该是什么。在CP/M绝对汇编（ASM）的情况下，使用ORG指令。ORG语句的地址单元被ASM用作该语句后面程序代码的原点，直至下一条ORG语句或该文件结束。当要把程序装入ROM时，这种能力特别重要，因为ROM中的程序要写的所有数据区一定在RAM中。ORG指令的变元为汇编生成用于CP/M装入程序LOAD的代码指定绝对原点。ORG指令还可用于为不同程序段和数据区建立不同的相对原点和绝对原点。因此，如果你想装入到ROM中的程序需要可重写的数据区，则在你的程序文件中至少必须有两条ORG语句：一条在程序开始处，指定它在ROM中的绝对原点；另一条在可写数据区，指定它在RAM中的绝对原点。

本编译程序生成的最终汇编代码输出包含两条ORG语句，分别在程序和数据区开始处，如下所示：

```
ORG *   ; REORIGIN PROGRAM HERE
:
ORG *   ; REORIGIN DATA HERE
:
```

变元"*"预示当前地址单元，这使每条ORG语句没有操作数，只有插入适合于你的情况的变元值时，这些语句才有效。

十三、列表函数的说明

SuperSoft C编译程序现在支持列表函数。列表函数的自变量数目不预先确定。标准的库函数包括下列内部构造的列表函数：printf，scanf，fprintf，fscanf，sprintf和sscanf。

为使用户自己建立的列表函数能正确执行，这些列表函数必须在用户程序第一次调用它们之前进行说明如下：

```
extern type listf ( . );
```

其中listf是任意的列表函数，type是任意的存储类型。这就使得C编译程序在每次调用该函数时，生成自变量计数并推至参数栈作为最后一个自变量。如果该说明放在用户程序开始，作为全程外部符，则只须在程序中出现一次。

第三章　标准库函数

随SuperSoft C编译程序提供的标准库函数在下列文件中定义。

C2.RH：运行时库头部

C2.RT：运行时库尾部

STDIO.C：标准UNIX型I/O函数

ALLOC.C：动态内存分配函数

CRUNT2.C：辅助的运行时公用函数

FUNC.C：辅助函数

FORMATIO.C　printf和scanf

在第二次扫描期间，编译程序自动地将其运行时库C2.RT中预编译、预优化的汇编代码，插入到用户程序中。包含其余标准库函数定义的另外三个文件，以SuperSoft C源代码形式存放。

一、将标准库函数插入用户程序

如果用户程序调用上面列举的最后三个文件中的函数，则在编译之前必须将该函数的定义插入你的程序中。如果该函数又调用其他函数，则这些函数的定义也必须插入。将SuperSoft C函数或代码段插入用户程序的方法有五种，你可以使用下面所述的四种方法，将任意的SuperSoft C代码（而不只限于标准的库函数）插入你的程序中。

第一种方法：#INCLUDE指示句

第一种，也是最简单的方法是使用#INCLUDE预处理指示句，将包含所需函数的整个文件插入你的程序中。最好是把用于标准库函数文件的所有#INCLUDE指示句放在程序头部，紧接在外部数据定义之后。

FORMATIO.C使用STDIO.C；

STDIO.C使用STDIO.H和ALLOC.C；

ALLOC.C使用CRUNT2.C；

FUNC.C单独使用。

将所有必须的模块都包括进来的主要缺点是：有些从来不被调用的函数也插入到用户程序中，使程序不必要地增大，编译时间也因此增加。

第二种方法：CC命令行文件名表

如果在编译程序第一次扫描CC命令行时列出若干文件名，则编译程序将对这些命名文件按列出顺序进行语法分析，好象它们是一个文件一样，其结果放入单个输出文件中。因此，既然在C源代码文件中函数可以按任意顺序出现，只须在CC的命令行列出文件名，就可以将任何SuperSoft C源代码文件插入用户程序中。例如，将所有标准库函数插入程序Y.C中的CC命令行格式是：

CC Y.C CRUNT2.C STDIO.C ALLOC.C FUNC.C FORMATIO.C

这种方法的优缺点与第一种方法类似，但比较灵活。要改变插入你的程序中的文件，而不改变程序，只须在CC命令行打入一组不同的文件名。

第三种方法：预编译和插入到C2.RH中

可以将一个或全部SuperSoft C源码形式的标准库函数文件进行编译，结果文件（经过很少的编辑）插入编译程序运行时库C2.RH中。该过程在第四章中详细描述。用这种方法可以加快含有标准库函数文件的用户程序的编译速度，但是这种方法同样存在上述两种方法的缺点。

第四章中描述的过程还有一个重要的用途。如果用户知道有一组或几组函数在程序中最常使用，则可以将这些函数进行预编译，插入到C2.RH中，建立用户自己的运行时库版本。生成这样一组函数的方法在后面叙述，用这种方法插入的函数可以从提供的标准库函数中挑选，也可以由用户自己建立，以满足不同的要求。程序设计经验越丰富，运行时库的用户化就能做得越好。这种方法既能节省用户程序的编译时间，又能使程序中不必要的函数尽量少些。

第四种方法："剪贴"

这种方法概念上很简单，但做起来很费力。指导思想是：从标准库函数文件，建立一个或多个只包含用户程序所需的数据和函数定义的文件（"剪"），然后，用某种方法把所建立的文件在编译前合并到用户程序中（"贴"）。

这种方法的优点是用户程序的源码和目标码都可以尽量小。缺点是每个细节都要顾及，你必须检查是否执行你的程序所需的全部数据和函数定义都存在并且完整无缺。很显然，这是很费时间的事情，而且在处理过程中会不知不觉地出现一些很麻烦的错误。

另一方面，在许多应用中，使用户程序能最有效地使用内存空间，是很重要的。有些程序可能已达到或超过你的系统最大内存容量，而另一些程序可能要求在执行期间，有较大量的数据需要存放。

你还可能发现，有时要反复多次使用某一组标准库函数，以及用户自己的某些函数。在这种情况下，使用该方法的第一步，就是建立你自己的"用户化"的一组或几组SuperSoft C标准库函数。

如果你不很熟悉C语言和所用的操作系统，最好不要使用这种方法。下面我们只是简单扼要地介绍一下"剪贴法"的几个主要步骤。

1. 把包含有打算插入用户程序中的数据或函数定义的全部文件拷贝到备份盘上。

2. 用编辑程序，把不打算插入用户程序中的数据或函数定义从文件中删去。要十分小心，不要把被用户程序调用的函数所调用的那些函数定义，或者把打算插入的那些函数所需的数据定义删掉。

3. 如果现在已经有多个包含所需的函数或数据定义的文件，可以将它们连接在一起，也可以留在不同的文件中。

4. 现在又面临如何把已建立的文件插入到用户程序中去的问题。你可以用前面所说的三种办法中的任一种，因为每种方法对任何文件都是适用的。只不过现在所包含的数据和函数定义都是用户程序所需要的，没用的东西已经删去。因此，第四步就是使用编辑程序把已建立的文件中的定义插入用户程序中适当的地方。

第五种方法：浮动汇编程序

使用浮动汇编程序（如Microsoft M80-L80）的用户，可以将所需的子例程连接到用户程序中。其过程是，分别编译一组函数，然后连接在一起。

SuperSoft C允许对外部变量（即全程变量）的EXTERN指示句。而且，所有未定义函数均自动地被说明为外部的，并在LINK时解决全部引用。

浮动是将用户函数和库函数插入最终COM文件中的最好方法，但这种方法只能用于有浮动汇编程序的用户。

上面所述的五种方法究竟选用哪一种，主要取决于你在C语言程序设计方面的经验。初学C语言的程序员无疑应使用第一种方法，因为这种方法出错机会最少。即使是最有经验的C语言程序员也应尽量少用第四种方法。第五种方法最好，但只限于有浮动汇编程序的用户。总的说来，选用哪种方法，要结合实际情况，看哪种方法能最有效地利用编程时间。

二、动态内存分配函数的初始化

在用户程序直接或间接调用动态内存分配函数alloc之前，必须对它进行初始化。如果未作初始化，则alloc和free函数以及所有调用它们的函数都不能正确执行。对alloc函数的间接调用，可能由于调用用户程序中定义的函数，或调用STDIO.C中定义的有缓冲区文件I/O函数而产生。

如果用户程序通过对其内部定义的函数调用而直接或间接地调用alloc，则在第一次直接或间接调用alloc之前，必须说明(隐式的)外部变量"allocs"，指针型,指向字符(char*)并赋值为0。

如果用户程序用"#INCLUDE"包括标准库函数文件STDIO.C，并调用它定义的有缓冲区文件I/O函数（后者又调用alloc），则在直接或间接调用alloc之前（包括通过调用有缓冲区文件I/O函数所作的调用），仍然必须将alloc设置为0。

三、SuperSoft C 标准库函数描述

本节按字母顺序逐一描述SuperSoft C标准库函数。这些函数大部分是以SuperSoft C源代码形式提供的，其余的以汇编代码形式提供，用户可以从源代码了解这些函数实现的细节。

下面列举的函数，都采用了一种缩写形式，说明每个函数返回的数据和传递给每个函数的变元的类型。每个函数描述的前面几行是它在SuperSoft C中的定义的前几行，也就是函数本身及其变元说明。函数的类型是它返回值的类型。如果没有指出函数类型，则它不返回任何确定的值。函数的每个变元类型必须指明。

注意：STDIO.H文件包含#DEFINE语句，该语句建立一对预定义的符号返回值（实际返回数值如括号内所示）：SUCCESS（0）和ERROR（−1）。在STDIO.C中定义的许多函数均返回这两个值。使用这些符号值可以使STDIO.C中的函数编码更容易理解。STDIO.C包含（#INCLUDES）STDIO.H。

下面列出标准库函数：

ABS

```
int abs(i)
    int i;
```

abs返回i的绝对值（如果i小于0，abs返回−i，否则返回i）。

ALLOC

```
char * alloc(n)
    int n;
```

alloc分配长度为n的连续内存区，它分配的每块内存区都从偶地址开始。

alloc如果成功，则返回指针值，指向该内存块的第一个单元。应该将这个指针值存放起来，供用户程序后继使用。如果要求分配的连续内存区复盖：1)已存放在内存中的程序，2)运行时堆栈，3)前面已分配的内存块，则 alloc 返回空指针值(0)，并且不作任何内存分配。

注意：为正确执行alloc或调用alloc的函数，用户程序在第一次调用alloc函数或alloc所调用的函数之前，必须清除外部变量（赋以 0 值）allocs，为指针型，指向字符(char *)，这将使alloc在首次被调用时，对其本身作初始化。allocs在ALLOC.C中说明。

ATOI

```
int atoi (s)
    char  *s;
```

atoi返回十进制整数值，相应于S指向的以空字符结尾的ASCII字符串。如果该字符串包含除前导标记、空格和减号（全部均为任选）以外的字符，后面跟以连续的十进制数字，则atoi返回 0 值。

BDOS

```
int bdos (c, de)
    int c, de;
```

bdos使CP/M用户能将BDOS功能的直接调用插入以SuperSoft C编写的程序中。使用bdos的程序在CP/M和8080系列CPU范围之外是不可移植的。bdos将机器寄存器C置为在 c 中给定的值，而将寄存器对DE置为在de中给定的值，并启动BDOS 调用。BDOS要求寄存器C包含有效的BDOS功能号。关于BDOS功能调用的详细介绍，请参阅“CP/M接口指南”。

bdos返回单字节值（作为一个整数），该值等于A寄存器内容（BDOS通过A寄存器返回它的值）。

BIOS

```
int bios (jmpnum, bc, de)
    int jmpnum, bc, de;
```

bios使CP/M用户能将BIOS的直接调用插入以SuperSoft C 编写的程序中。调用bios的程序在CP/M和8080系列CPU范围之外是不可移植的。bios将机器寄存器对 BC 置成在bc中给定的值，将寄存器对DE置成在de中给定的值，并将控制转移到jmpnum 指定的BIOS转移向量入口点，启动适当的BIOS调用。该入口点可以数值方式或符号方式指定。

每个入口点的助记名和数字值给出如下：

WBOOT… 0	READER… 6
CONST… 1	HOME… 7
CONIN… 2	SELDSK… 8
CONOUT… 3	SETTRK… 9
LIST… 4	SETSEC…10
PUNCH… 5	SETDMA…11

READ…12　　LISTST…14

WRITE…13　　SECTRAN…15

如果jmpnum是SELDSK(8)或SECTRAN(15)，则在执行BIOS调用后，bios返回保留在寄存器HL中的值；否则，返回保存于寄存器A中的值，无符号扩展（亦即为0～255之间的值）。

BRK

```
char * brk(p)
    char * p;
```

brk将外部变量名CCEDATA置为p指向的内存字节，并返回CCEDATA，等价于它传递的指针值。CCEDATA的初值设置为紧接在用户程序外部数据区最后一个字节后面的那个字节。因为CCEDATA被其他内存分配函数用作基值，这就可以正确地将用户程序中的这些函数初始化。在用户自己的程序中，几乎根本不需要调用brk。

CCALL

```
int ccall (addr, hl, a, bc, de)
    char * addr, a;
    int hl, bc, de;
```

ccall将机器寄存器HL、A、BC、DE分别置为在hl、a、bc、de中给定的值，并调用在addr地址开始的汇编语言子程序。执行该子程序后，ccall返回存于HL寄存器中的值。调用ccall的程序在8080系列CPU范围之外不可移植。

CCALLA

```
int ccalla (addr, hl, a, bc, de)
    char * addr, a;
    int hl, bc, de;
```

ccalla与ccall相同，只是在执行addr地址开始的子程序后，返回存于A寄存器中的值。调用ccalla的程序，在8080系列CPU范围之外也是不可移植的。

下面是ccalla函数的使用实例：

```
int bdos (c, de)
  char c;
  int de;
{
        return ccalla (5, 0, 0, c, de);
 }
```

很显然，上面列出的C函数是通过ccalla来实现BDOS功能（这不是BDOS实现的方式）。ccall和ccalla均可以用于调用用户自己建立的汇编语言子程序。

CLOSE

```
int close(fd)
    FILE * fd;
```

close关闭由指针fd所指向的文件描述符说明的文件。如果指定文件关闭成功，则返回值为SUCCESS(0)；如果：1)fd不指向一个有效的文件描述符，或者2)该文件由于操作

系统级的错误没有被关闭，则close返回值为ERROR(−1)。

close不将文件结束字符（EOF）放入缓冲器，并且不调用fflush。如果在用close调用一个通过fopen打开的有缓冲区输出文件时，没有首先调用fflush，则保存在该文件的I/O缓冲区的数据将会丢失。

CODEND

```
char * codend ()
```

codend返回一个指针值，指向跟在用户程序的根段代码结束后面的字节。除非重新确定用户程序的外部数据区开始地址，否则codend返回的值指向该数据区的开始（目前未实现）。

CPMVER

```
int cpmver ()
```

在执行对BDOS功能12的调用之后，cpmver返回存放在寄存器HL中的值。该返回值不带符号扩展；如果调用程序在CP/M2.0以前的版本下运行，则返回值为0；如果在CP/M2.0版本下运行，返回值为0x0020；如果在CP/M2.0后的各版本下运行，返回值为0x0021～0x002F；如果在MP/M下运行，则返回值为0x0100。该函数在编写运行于CP/M或MP/M下的C语言程序时，是很有用的。调用cpmver的程序在CP/M和MP/M范围之外是不可移植的。

CREAT

```
FILE * creat (fspec)
    FILESPEC * fspec;
```

creat在磁盘上建立一个文件，其文件描述由fspec给出，并且打开该文件，用于直接输出。任何具有相同文件描述的现存文件均被删除。

creat调用如果成功，返回一个指针值，指向给定文件的有效描述符。应该把指针值存放起来，供用户程序后继使用。如果：1）没有足够的用户内存区用于存放新的文件描述，2）给定的文件描述无效，3）由于操作系统级的错误，不能建立和打开文件，则creat返回ERROR（−1），不建立或打开任何文件。

ENDEXT

```
char * endext ()
```

endext返回一个指针值，指向紧接在用户程序外部数据区最后一个字节后面的字节。该值应等于CCEDATA的初值（目前未实现）。

EVNBRK

```
char * evnbrk (n)
    int n;
```

evnbrk执行与sbrk相同的功能，不同的是它的返回值总是偶数。如果对于给定变元n、sbrk应返回一个奇数值，则evnbrk执行时“跳过”一个字节，使返回偶数值。如果调用成功，evnbrk返回一个指针值，指向加到用户内存区的字节块中第一个存储单元；如果指定加到用户内存区的字节数使用户内存区：1）复盖已存放在内存中的程序，2）复盖运行时堆栈，或3）超出可用内存，则evnbrk返回值−1，不增加用户内存区字节数。

注意：在alloc调用之间，不能调用具有负变元的evnbrk。

EXEC

```
int exec (fspec)
    FILESPEC * fspec;
```

exec将fspec指向的描述为空字符串结束的文件装入，并执行该文件包含的可执行代码（串常数，如“nextprog”也可以在fspec中传递，因为它求值为指向空结束字符串的指针）。如果未指定扩展名或文件型，则在查找文件之前，exec将“.COM”附加到指定的文件名后面。

如果该文件不存在，或者由于操作系统级的错误，不能打开或读入该文件，则exec返回值－1。如果exec成功，则装入文件复盖调用程序的内存映象，不可能再返回该调用程序。但是，被调用程序可以通过另一个exec调用链回调用程序。exec使用户可以链接或成功执行一系列程序文件。这些程序文件中，每一个程序均可复盖调用它的前一程序的内存映象。

数据可以在文件内部，或通过外部数据区，从调用程序传递到被调用程序。为在一个文件内部最有效传递数据，调用程序应关闭该文件，并由被调用程序重新打开它。为通过外部数据区传递数据，该数据区起始地址必须与调用和被调用程序的起始地址相同。

EXECL

```
int execl (fspec, arg1, arg2, ···, 0)
    FILESPEC * fspec;
    char * arg1, * arg2, ···;
```

execl能将命令行参数以一系列由arg1，arg2，…指向，以零终止的字符串传递到所调用的程序，除此之外，execl与exec完全相同。所传递的最后的变元必须是零。execl用由它的变元指向的那些字符串建造一个命令行，在这些字符串间插以空格。

另一方面，在CP/M下，整个命令行可以在arg1指向的字符串内部传递。空格应出现在字符串合适的位置，且arg2应为零。

在这两种情况中，参数都应在字符串或字符串组内依次出现，与对被调用的程序键入的命令行中它们出现的次序一样（字符串常量可用来代替那些指向对arg1，arg2，…，以及fspec字符串的指针）。CP/M要求所建造的命令行的长度不超过126个字符。其他操作系统可能对此有不同的限制。

EXIT

```
exit ()
```

exit将控制从程序送回操作系统（使退出至系统级）。exit不冲掉（写到磁盘）任何输出缓冲器，不关闭任何开文件。

EXTERNS

```
char * externs ()
```

externs返回一个指针，指向用户程序外部数据区中的第一个字节。这个值与用codend返回的相同，除非重新确定该区的起始地址。

FABORT

```
int fabort (fd)
    FILE * fd;
```

fabort 释放对一开文件分配的文件描述符，而不关闭该文件。文件由指向它的文件描述符的指针fd指定。调用fabort 不影响只作为输入打开的文件内容，但对于作为输出打开的文件，调用fabort会导致写到该文件的部分或全部数据的丢失（为与 BDS C 兼容，我们包括了这个函数，但建议最好不要用它）。

如果fabort成功，则返回SUCCESS(0)。如果fd不是指向一有效的文件描述符，则fabort返回ERROR (−1)。

FCLOSE

```
int fclose (fd)
    FILE *fd;
```

fclose关闭通过fopen打开的有缓冲的输出文件。文件由指向其文件描述符的指针fd指定。fclose把一文件结束字符（EOF）放置在文件输入/输出缓冲器的当前位置，并在关闭文件前调用fflush。

如果指定的文件被成功地关闭，fclose返回SUCCESS(0)。如果：1）指定文件没有通过fopen被打开作为有缓冲的输出文件，2）fd不是指向一有效的文件描述符，或3）由于一操作系统级的错误，文件不能被关闭，该fclose均返回ERROR(−1)且不关闭该文件。

FFLUSH

```
fflush(fd)
    FILE *fd;
```

fflush冲掉或将输入/输出缓冲器的当前内容写入文件中，此缓存器与通过fopen打开作为有缓冲输出的文件相联系，文件由指向其文件描述符的指针 fd 指定。当打开文件时，输入/输出缓冲器的大小被设置为系统记录长度的正整数倍（在文件输入/输出操作期间，系统记录是所传送的数据的最小单位。CP/M下，系统记录长度为128个字节；UNIX 下，为1字节。关于详细情况，请查阅操作系统资料）。在调用fflush之后，文件输入/输出指针将指向在最后写入的一个系统记录之后的那个系统记录的开头。

如果缓冲器被成功地写到文件中，fflush 返回 SUCCESS（0）（缓冲器空时调用fflush不起作用）。如果：1）文件没有通过 fopen 打开作为有缓冲的输出文件，2）fd 不是指向一有效的文件描述符，或3）由于操作系统级的错误，整个缓冲器内容不能写入文件，则fflush均返回ERROR（−1）且不冲掉缓冲器。

注意：因为每当文件的输出缓冲器满时，它被自动冲掉，所以每一次调用 fflush时，缓冲器会含有若干字节希望保存的数据以及一些无用数据（未定义的字节）。如果是处在一文件的结尾，在调用 fflush 和关闭该文件之前，可简单地写一个文件结束字符（EOF）作为数据的最后一个字节（fclose就是这样做的）。在调用fflush后但在关闭文件前，仍然可以查找和读刚刚写的任何文件记录。可是，如果不是处在文件末尾并且希望查找另一个记录，就会产生问题，因为对于打开的有缓冲的输出文件，在每一次调用seek之前必须调用fflush，以避免丢失缓冲器的内容，这就难免将无用数据转储到文件中。

FGETS

```
chan *fgets (s, n, fd)
    chan *s;
    int n;
```

```
FILE * fd;
```

fgets将最多n－1个字符从通过fopen打开有缓冲的输入文件读入起始于s的字符串。文件由指向其文件描述符的指针fd指定。fgets读入时不越过新行字符或超过n－1个字符。然后fgets将一个null字符添加到所读的那些字符后以建立一个以零终止的字符串。

如果fgets成功，则返回一指针，指向该字符串(与传递到s中的值相同)。如果：1) fd不是指向一个有效的文件描述符，2) 由于操作系统级的错误，文件不能读，或3) 已到达文件结尾，则fgets返回一空指针值（0）。

FOPEN

```
FILE * fopen (fspec, mode, buffer-size)
    FILESPEC * fspec;
    char * mode;
    int buffer-size;
```

fopen建立和/或打开一文件，作为有缓冲的输入/输出文件并带有在fspec给出的文件说明。如果fopen成功，它返回一指针，指向关于所指定文件的一有效的文件描述符。应把这个指针存储起来供你的程序以后使用。如果：1) 对buffer-size指定的值小于系统记录长度；2) 没有足够的用户内存可用作新的文件描述符；3) 所给的文件说明是无效的；或4) 由于操作系统级出错，文件不能被打开或建立，则fopen均返回ERROR（－1）且不建立或打开任何文件。

对mode，必须指定为“w”，“a”，“r”或“rw”四者之一。究竟指定为哪一个，决定了文件的输入/输出方式，如下表所示：

"w"	只写方式
"a"	只写方式，将输出附加到文件结尾
"r"	只读方式
"rw"	读-写方式

如果一个文件的输入/输出方式是w、a或rw，我们就说，该文件作为有缓冲输出打开；如果一个文件的输入/输出方式是v、rw两者之一，则称该文件作为有缓冲输入打开。

对buffer-size指定的值决定与文件相联系的输入/输出缓冲器的大小，这个值应是系统记录长度的正整数倍（系统记录是文件输入/输出操作期间所传送的数据的最小单位。在CP/M下，系统记录长度为128字节；UNIX下，为1字节。关于详细情况，请查阅操作系统资料)。指向这个输入/输出缓冲器中第一个字节的指针被存在文件描述符中，因而仅有指向此文件描述符的指针需要被传递到任何其他有缓冲的文件输入/输出函数。

FPRINTF

```
int fprintf (fd, format, arg1, arg2, ···)
    FILE * fd;
    char * format;
    ···
```

fprintf与printf是相同的，只是fprintf将它的格式化输出字符串写到输入/输出缓冲器并开始于该缓冲器的当前单元，而不是写到标准输出，该缓冲器是与通过fopen打开、有缓冲输出的文件相联系的。每当缓冲器满时，它被自动冲掉（即它的整个内容被写到文件中）。文件由指向它的文件描述符的指针fd指定。

fprintf是一个列表函数，它在使用前必须被说明。

如果整个输出字符串被成功地写到缓冲器，fprintf返回SUCCESS(0)。可是由于文件输入/输出操作的缓冲作用，这样的返回值不能保证相同字符串被成功地写到该文件中。因为对fprintf的某一次调用产生的错误及其对输出结果的影响直到一些后面的函数调用引起文件的输入/输出缓冲器被冲掉时，才表现出来。如果：1)fd不是指向一个有效的文件描述符；2)文件没有通过fopen打开作为有缓冲的输出文件；3)由于发生操作系统级的错误，文件不能被写；或4)由于缺少磁盘空间，不能将整个字符串写到文件时，fprintf返回ERROR(−1)。

FPUTS

```
int fputs (s, fd)
    char * s;
    FILE * fd;
```

fputs将由s指向的以零终止的字符串写到输入/输出缓冲器的当前单元，此缓冲器与通过fopen打开作为有缓冲的输出文件相联系。每当缓冲器满时，它自动被冲掉（即它的整个内容被写到文件）。文件是由指向其文件描述符的指针fd指定的。每一出现在字符串中的新行字符（'\n'），一个回车和一个新行（"\r\n"）被写到缓冲器。末尾的null字符不写入。

如果：1)fd不指向一有效的文件描述符；2)文件没有通过fopen打开作为有缓冲的输出文件；或3)由于发生操作系统级的错误，文件不能写，则fputs返回ERROR（−1）并且不写字符串。否则，fputs返回实际写到缓冲器的字节数，减去插入的回车字符数。可是，由于文件输入/输出操作的缓冲作用，这样一个返回值不能保证那些相同的字节将成功地写到该文件，因为对fputs的某一次调用产生的错误及其对输出结果的影响直到一些后面的函数调用引起文件的输入/输出缓冲器被冲掉时，才表现出来。

FREE

```
free (p)
    char * p;
```

free释放内存中由调用alloc预先分配的一块存储区。变元p是指向该块中第一个存储单元的指针，它应与调用alloc返回的值相等。所分配的块可按任意次序释放。用一个不是经过调用alloc预先得到的变元来调用free是一个严重的错误。

FSCANF

```
int fscanf (fd, format, arg1, arg2, ···)
    FILE * fd;
    char * format, * arg1, * arg2, ···;
```

fscanf与scanf相同，只是输入字符串是从与通过fopen打开的有缓冲的输入文件相联系的输入/输出缓冲器读入，而不是从标准输入源读入。文件由指向其文件描述符的指针

fd 指定。 fscanf 从输入/输出缓冲器的当前位置开始读。当它成功地赋值给格式字符串中所列的每一项相对应的字节，或者到达文件结尾时停止读入。

fscanf 是一个列表函数，在使用前必须被说明。请参阅第二章最后的“列表函数的说明”。

如果没有出现错误，fscanf返回成功地赋给的值数。如果：1) fd 不指向一有效的文件描述符；2)文件没有通过fopen打开作为有缓冲的输入文件；或 3)由于发生操作系统级的错误，文件不能读，则fscanf返回ERROR（－1）并且不执行输入。

GETC

```
int getc (fd)
    FILE * fd;
```

getc顺次从通过fopen打开作为有缓冲的输入文件返回一个作为整数（0至255之间）的字符（字节）。文件由指向其文件描述符的指针fd 指定。回车和换行被显式地返回。如果：1)文件未打开作为有缓冲输入，或 2)已到达文件末尾，getc返回ERROR（－1）。

GETCHAR

```
char getchar ()
```

getchar从标准输入源（CP/M CON：设备——通常是控制台键盘）返回下一字符。如果getchar遇到一个Control-Z（CP/M的EOF标志符），则返回一null 字符。如果含有对getchar调用的程序正在CP/M下运行，且getchar遇到一个Control-C，程序将被异常终止，控制将返回到启动该程序的例行程序。

GETS

```
gets (s)
    char * s;
```

gets从标准输入源（CP/M CON：设备——通常是控制台键盘）读下一行到起始于 s 的字符串。gets置换新行字符（'\n'）或回车/新行组合（"\r\n"），它用一个null字符终止输入行以建立一个以零终止的字符串。因为gets不测试起始于 s的字符串长度是否足以容纳输入行，所以应该定义这个字符串，以使它能容纳合理要求的最长的输入行。

GETVAL

```
int getval (s)
    char * * s;
```

必须传递给getval一个指针，这个指针指向以null 终止的由用逗号分隔的子字符串组成的ASCII字符串的指针，每个子字符串应只含有引导空白、标记，或一个负号（均为任选），随后是连续的十进制数字。

当用满足上面要求的一个特殊的参数初次调用getval时，它返回对应于第一个子字符串的十进制整数值，并且对该参数指向的指针（辅助指针）加一个增量，以使它指向下一子字符串的第一个字符。以后每用同一参数调用getval，都返回与由辅助指针目前指向的子字符串对应的十进制整数值并且对该指针增值，以使它指向下一子字符串的第一个字符。

因为getval返回一个16位带符号的二进制整数，在任何子字符串中描述的值不应大于＋32,767或小于－32,768。

如果getval在子字符串的起始处遇到一无效字符，它返回 0 值并且对辅助指针增值，如

前所述。如果getval在一子字符串内遇到一无效字符，它终止该子字符串，返回对应于直至该点为止所读的有效字符的值并且对辅助指针取增量，如前所述。当getval遇到末尾的null字符时，它将辅助指针设置成0并返回0值。

GETW

```
int getw (fd)
    FILE * fd;
```

getw顺次从通过fopen打开作为有缓冲的输入文件中返回一个整数。文件由指向其文件描述符的指针fd指定。回车和换行被显式地返回。如果：1）文件未打开作为有缓冲的输入文件，或2)已到达文件结尾或在输入文件中出现整数－1，则getw返回ERROR(－1)。因而errno应作为真实的错误条件来检查。getc的调用可与getw的调用交替进行。getc和getw读的信息可分别由putc和putw来写。这里，不需要对输入文件中信息进行任何特殊的调整。

INDEX

```
char * index (s, c)
    char * s, c;
```

index返回一指针，指向起始于s的字符串中第一个出现的字符c。如果字符串中没有出现c，index返回空指针值（0）。

INITB

```
initb (array, s)
    char * array, * s;
```

initb使字符数组的初始化较为方便。应传递给它两个参数：第一个是array，它应为指向一字符数组的指针；第二个是s，也应为一指针，它指向一个以null终止、以逗号分隔的表示十进制整数值的ASCII字符串。当调用initb，它将起始于s的字符串中的每一个十进制整数值顺次转换成二进制整数值，并把该值的最低有效8位赋给由array指向的字符数组中的相应元素。

如果字符串中有n个整数值且数组中多于n个元素，那么数组中只有前n个元素被赋值，而其余元素的内容未改变。如果字符串中有n个整数值，但数组元素数小于n，那么超出数组结尾的字节将被赋值，就象它们是数组中的元素一样，数据可发生重写错误。防止这些情况发生或者提供对这些情况的处理是程序员的职责。

INITW

```
initw (array, s)
    int * array;
    char * s;
```

initw使整型数组的初始化较为简便，应传递给它两个参数：第一个为array，它应是指向一整型数组的指针；第二个是s，也为一指针，它指向一以null终止用逗号分隔的表示为十进制整数值的ASCII字符串。当调用initw时，它将起始于s的字符串中的每一十进制整数值顺次转换成二进制整数值，并把该值赋给由array指向的整型数组中的相应元素。

如果字符串有n个整数值，且数组元素数目大于n，那么只有数组中前n个元素被赋值，而其余元素的内容不改变。如果字符串有n个整数值，但数组元素数目小于n，那么超

指数组结尾的字节将被赋值，就象它们是数组中的元素一样，数据可发生重写错误。防止这些情况发生，或者提供对这些情况的处理，这是程序员的职责。

INP

```
int inp(port)
    char port;
```

在对一个输入端口执行机器指令IN之后，inp返回在port处指明的输入端口的当前值（这个函数只在IN或相等的指令有意义的机器上使用）。

ISALNUM

```
int isalnum (c)
    char c;
```

如果c是一个ASCII字母数字字符，isalnum返回真（1），否则返回假（0）。

ISALPHA

```
int isalpha (c)
    char c;
```

如果c是一ASCII字母字符，isalpha返回真（1），否则返回假（0）。

ISASCII

```
int isascii (c)
    char c;
```

如果c是一ASCII字符，isascii返回真（1），否则返回假（0）。

ISCNTRL

```
int iscntrl (c)
    char c;
```

如果c是一ASCII控制字符，iscntrl返回真（1），否则返回假（0）。

ISDIGIT

```
int isdigit (c)
    char c;
```

如果c是一表示十进制数字0～9之一的ASCII字符，isdigit返回真（1），否则返回0。

ISLOWER

```
int islower (c)
    char c;
```

如果c是一ASCII小写体字母字符，islower返回真（1），否则它返回0。

ISNUMERIC

```
int isnumeric (c, radix)
    char c;
    int radix;
```

如果c是一ASCII字符，且此字符表示由radix指定基数的数制中的一有效数字，则isnumeric返回真（1），否则返回假（0）。例如：

```
isnumeric ('A', 15)
```

返回真，仅当1<radix<36时，isnumeric有定义。

ISPRINT

```
int isprint (c)
    char c;
```

如果c是一可打印的ASCII字符，isprint返回真（1），否则它返回假（0）。

ISPUNCT

```
int ispunct (c)
char c;
```

如果c是一表示标点符号的ASCII字符，ispunct返回真（1），否则它返回假（0）。

ISSPACE

```
int isspace (c)
    char c;
```

如果c是一表示空格、标记或一新行的ASCII字符，isspace返回真（1），否则返回假（0）。此函数是为与BDS C兼容而设立的，标准的UNIX C函数为iswhite。

ISUPPER

```
int isupper (c)
    char c;
```

如果c是一ASCII大写体字母字符，isupper返回真（1），否则它返回0。

ISWHITE

```
int iswhite (c)
    char c;
```

如果c是一表示空格、标记、新行或一格式馈给的ASCII字符，iswhite返回真（1），否则返回假（0）。

KBHIT

```
int kbhit ()
```

kbhit测试一字符是否已在控制台键盘上打入——如果是，返回真，否则返回假。更确切地说，如果一个字符出现在标准的输入源（CP/M CON：设备——一般是控制台键盘），kbhit返回真（非0），否则它返回假（0）。此函数在不具备这类函数的系统（例UNIX）上是不可用的。

MOVMEM

```
movmem (source, dest, n)
    char *source, *dest;
    int n;
```

movmem将存储器中从source起始的n个连续字节的内容复制到从dest起始的n个连续的字节。这两个区域可以重迭。Source指向的区域中的那些字节只要不被作为区域间重迭操作的结果重写，它们是没有改变的。

OPEN

```
FILE *open (fspec, mode)
    FILESPEC *fspec;
    int mode;
```

open打开在fspec处指定的文件作直接输入/输出文件，这个文件必须已通过creat建立。如果open成功，返回一指针，指向所指定文件的有效的文件描述符。应存储这个指针供用户程序后继使用。如果：1）没有足够的用户存储用作新的文件描述符，2）所给的文件说明是无效的，3）指定的文件或不存在或没有通过creat建立，或4）由于操作系统级的错误，文件不能被打开，则open返回ERROR（－1），且不打开文件。

对于mode必须指定0、1、2三者之一，它们决定文件的输入/输出方式，如下表所示：

0	只读方式
1	只写方式
2	读-写方式

OTELL

```
unsigned int otell (fd)
    FILE * fd;
```

otell返回字节区距，此区距是从文件中当前被存取的512字节块的起点算起到对于该文件的下一次文件输入/输出操作将开始的点。文件由指向其文件描述符的指针fd指定。otell不指明输入/输出操作从哪一个512字节块开始。见rtell和tell。

OUTP

```
outp(port,b)
    char port, b;
```

outp将字节b放置在由port指明的输出端口，并对该端口执行一机器指令OUT（此函数仅对OUT指令有意义的机器有用）。

PAUSE

```
pause ()
```

pause暂停调用程序的执行，直到在控制台键盘上键入一字符。pause执行一空循环，并测试输入是否存在于标准输入源(CP/M CON:设备——一般为控制台键盘)。仅当存在时，退出循环，将控制返回到调用程序。在pause返回之前，必须从标准输入读一个或多个字符。

PEEK

```
char peek(addr)
    char * addr;
```

peek返回位于addr的存储字节的内容。包括此函数是为与BDS C兼容。它在C中是多余的，因为间接是C语言的一个特性。

PGETC

```
char pgetc(fd)
    FILE * fd;
```

pgetc与getc是相同的，只是每当遇到一回车字符（'r\'）后接一新行字符('\n')

时，它只返回新行。于是pgetc将文件内部的行从CP/M格式转换成UNIX格式。

POKE

```
poke (addr, b)
    char *addr, b;
```

poke将字节b写到位于addr的存储字节。b必须是一lvalue表达式。为与BDS C兼容，已包括了函数peek，它在C中是多余的，因为间接是该语言一个特性。

PPUTC

```
int pputc (c, fd)
    char c;
    FILE *fd;
```

pputc是与putc相等的，只是每当在c中传递给它一新行字符（'\n'）时，它首先将一回车字符（'\r'）写入文件的输入/输出缓冲器，然后写已传递给它的那个新行字符。于是pputc将写到文件的行从UNIX格式转换成CP/M格式。

PRINTF

```
printf (format, arg1, arg2…)
    char *format;
    …
```

printf将一格式化的输出字符串写到标准输出（CP/M CON:设备——一般是控制台屏幕）。必须把一指向以null终止的字符串的指针format传递给printf（对于format，字符串常量也是有效的，因为它求值结果为以null终止的字符串）。此字符串控制输出字符串的产生。可传递给printf一系列其他变元：arg1，arg2，…。这个系列中各个单独的变元可以是字符、整数、无符号整数或字符串指针。只有第一个变元format是要求的，其它均为任选。

printf是一列表函数，在使用前必须被说明。请参阅第二章最后"列表函数的说明"。

由format指向的字符串可含有普通字符或含有专用子字符串，它们以字符%起始，称作转换说明。当printf遇到每一普通字符时，它从右到左扫描该字符串，字符串被简单地写到标准输出。当遇到每一转换说明，都引起系列arg1，arg2，…中的下一变元值被按照说明进行转换和格式化并写到标准输出。

在每一转换说明中的字符%后可能出现：

1) 一任选的负号'－'，它如果出现，使被转换的值在它的域中按左对齐。缺省为右对齐。

2) 一任选的十进制数字串，它指明写入该值的域中字符的最小数目。被转换的值不截断。可是如果它具有的字符少于这里指定的，则在左边用空格把它填充到指定的宽度。如果指定为左对齐则在右边填充空格。如果这个数字串以0开始，那么转换的值要用0，而不是用空格填充。

3) 另一任选的十进制数字串，它前面必须是一句点'.'，指明要从以null终止的字符串复制的字符的最大数目。

4) 一字符，称作转换字符，指明要完成的转换的类型。

以上只有转换字符必须存在于转换说明中。所有其他项若出现，必须依照上面所列的次

序。

它们指定的（有效的）转换字符和转换类型是：

c　变元的最低有效字节被解释为一字符。该字符仅当它是可打印的，才被写。

d　变元，应为一整数，并转换为十进制表示法。

o　变元，应为一整数，转换为八进制表示法。

x　变元，应为一整数，转换成十六进制表示法。

u　变元，应为一无符号整数，转换为十进制表示法。

s　变元，它被解释为一字符串指针。所指向的字符串中的字符被读、写，直至读到一null字符，或已写入任选指定的最大字符数，见上面第3项。

%　字符%被写入。这是一个换码序列，与变元无关。

PUTC

```
int putc (c, fd)
    char c;
    FILE * fd;
```

putc将字符c写到输入/输出缓冲器的当前单元，此缓冲器与通过fopen打开有缓冲的输出文件相联系。每当该缓冲器满时，它自动被冲掉（即它的整个内容被写到文件）。文件是由指向其文件描述符的指针fd指定。

如果：1) fd不是指向一有效文件描述符；2)文件没有通过fopen打开作为有缓冲的输出；或3) 由于操作系统级的错误，缓冲器不能被写，则putc返回ERROR(－1)并且不写字符。否则，putc将字符写到文件的输入/输出缓冲器并返回SUCCESS（0）。可是，由于文件输入/输出操作的缓冲作用，这样的返回值不能保证相同字符成功地写到文件，因为对putc的某一次调用产生的错误及其对输出结果的影响直到一些后面的函数调用使文件的输入/输出缓冲器被冲掉时，才表现出来。

PUTCHAR

```
putchar (c)
    char c;
```

putchar将字符c写到标准输出（CP/M　CON：设备——一般为控制台屏幕）。

PUTS

```
puts (s)
    char * s;
```

puts将起始于s的字符串写到标准输出（CP/M CON：设备——一般是控制台屏幕）。所有回车命令必须在该字符串中显式地出现。

PUTW

```
int putw(i,fd)
    int i;
    FILE * fd;
```

putw将整数i写到输入/输出缓冲器的当前单元，此缓冲器与通过fopen打开有缓冲的输出文件相联系。每当该缓冲器满时，它自动被冲掉（即它的整个内容被写到文件）。文件由指向其文件描述符的指针fd指定。

如果：1)fd不是指向一有效的文件描述符；2)文件没有通过fopen打开作为有缓冲的输出；或3)由于操作系统级的错误，缓冲器不能被写，则putw返回ERROR（−1）并且不写整数。否则，putw将整数写到文件的输入/输出缓冲器，并返回SUCCESS(0)。可是由于文件输入/输出操作的缓冲作用，这样的返回值不能保证相同的整数成功地写到文件，因为由对putc的某一次调用产生的错误及其对输出结果的影响直到一些后面的函数调用使文件的输入/输出缓冲器被冲掉时才表现出来。putc和putw的调用可交替进行。用putc和putw写的文件可使用getc和getw来读。

RAND

```
int rand ()
```

rand返回由前一个srand调用初置的伪随机数字序列中的下一个值。序列中值的取值范围为0到65,535。

C表达式

```
rand () % n
```

将对一大于或等于0但小于n的整数求值。

READ

```
int read (fd, bufr, n)
    FILE * fd;
    char * bufr;
    int n;
```

read将打开用于直接输入或有缓冲输入的文件中，从文件输入/输出指针当前单元开始的最多n个字节读入由bufr指向的内存缓冲器。文件由指向其文件描述符的指针fd指定。

你应该定义由bufr指向的缓冲器，以使它至少能含有n个字节。n必须是系统记录长度的正整数倍（系统记录是在文件输入/输出操作期间传送的数据的最小单位。在CP/M下，一个系统记录为128字节。至于详细情况，请查阅操作系统资料）。

文件输入/输出指针将总是指向系统记录的起点。在对read的一次调用后，文件输入/输出指针将指向所读的最后一个记录后的系统记录的起点。

如果没有出现错误，read返回实际所读的字节数。如果某些字节正从一文件读出，read返回系统记录长度的倍数或返回（0）。只有在已到达文件结尾时，才返回0。如果字节正从一个作为文件打开的串行设备（例如CP/M CON:或RDR:设备）中读出，read返回（1），因为每次调用read只能从一串行设备读一字节。如果：1)文件未打开作为输入；2)n小于系统记录长度；3)由于操作系统级的错误，文件不能被读，则read返回ERROR(−1)并且不读文件。

RENAME

```
int rename (fname, fspec)
    char * fname;
    FILESPEC * fspec;
```

rename重新命名在fspec处指定的文件，给予它含在由fname指向的以null终止的字符串中的名（对于fname，一个字符串常量，例如“newname”也是有效的，因为它对一指向以null终止的字符串的指针求值）。若有驱动器名和号，它们是不改变的。

RESET

```
reset(n)
    int n;
```

reset使程序执行返回到由先前调用 setexit设置的点。这种转移具有从setexit返回的外部特征。传递给reset的参数n作为由setexit返回的值出现。

reset和setexit在一起对重复地退出至一公共点的编码比较简明——特别是当这类传送要求拆开若干层的函数调用时。例如，在写一个交互式编辑程序过程中，你可以在命令循环的顶部调用setexit并测试它的返回值是否等于零(0)。每一非0值可被用于指明一不同的错误条件。可以打印错误号，继续执行命令循环。对reset的调用可散布在整个循环的合适的地方。在每一情况下，传递到 reset的参数都指明某一特定错误条件存在(非0) 或不存在(0)。

虽然reset和setexit在使用和语法方面类似于函数，但它们被作为编译预处理程序伪指令执行，而不是作为函数。因此，在标准库函数文件中找不到它们。

RINDEX

```
char * rindex (s,c)
    char * s,c;
```

rindex返回一指针，该指针指向起始于s的字符串中最后出现的一个 c字符。如果字符串中没有出现c，rindex返回一空指针值 (0)。

RTELL

```
unsigned int rtell (fd)
    FILE * fd;
```

rtell返回512字节块区距，从文件起点开始，到对该文件的下一次文件输入/输出操作将在其内开始的512字节块。rtell不指明在该块内I/O操作将从哪一个字节开始。文件由指向它的文件描述符的指针fd指定。见otell和tell。

SBRK

```
char * sbrk(n)
    int n;
```

sbrk增加n个字节到用户存储区 (即将CCEDATA 增值n)。sbrk如果成功，返回一指向所增加的内存区中第一个字节的指针。如果指定的内存块大小：1)覆盖已存储的程序，2)覆盖运行时堆栈，或3)超过可用的内存，sbrk均返回 (−1) 并且不增加字节到用户存储区。

切记：在对alloc的调用之间不得以负变元调用sbrk。

SCANF

```
int scanf(format, arg1, arg2,…)
    char * format;
```

scanf从标准输入 (CP/M CON：设备——通常是控制台键盘) 读一格式化的输入字符串。在由它的第一个变元 format 指向的格式字符串的控制下，scanf从它的输入字符串抽取一系列的子字符串，称为输入字段，转换每一个字段中所表示的值，称为输入值，并把这些转换了的值顺次赋给由它的其余变元arg1，arg2,…指向的数据对象。

scanf 是列表函数，在使用前必须被说明。请参阅第二章最后“列表函数的说明”。

必须传递给scanf一指向恰当的以 null 终止的格式字符串的指针format，作为它的第一个变元（对于 format，字符串常量也是有效的，因为它对一指向以 null终止的字符串的指针求值）。一系列其他变元arg1，arg2,…，可传递到 scanf，所有这些变元均必须是指针。由arg1，arg2,…指向的各数据对象，可以是字符、字符数组或是整数。

格式字符串可含有“空白”字符（即空格、标记和新行），普通字符，或含有以字符%起始，称为转换说明的特殊子字符串。格式字符串中第一个转换说明相应于并决定输入字符串中第一个输入字段的边界，它也决定对该字段中表示的输入值所要完成的转换类型。每一对相继的转换说明和输入字段有着相同的关系。

跟在每一转换说明中字符%后可出现：

1) 一任选的取消赋值字符'*'，它如果出现，便跳过相应的输入字段。

2) 一任选的十进制数字串，它指定相应输入字段中字符的最大数。

3) 一转换字符，指明要对相应的输入值完成的转换类型。

以上只有转换字符必须存在于转换说明中，所有其他项若出现，必须依照上面所列出的次序。

有效的转换字符和转换类型是：

% 一单独的%字符，它是输入字符串在这点所要求的。这是一个换码序列——不完成赋值。

c 输入值解释为一字符。相应的变元应为一字符指针。常规的跳过空格字符被取消。要读下一个非空白字符，须使用%1s。若字段宽度也被指定，相应的变元应是指向一字符数组指针，且读入指定数目的字符。

s 输入值解释为一字符串。相应的变元应是指向一字符数组的指针，此数组大得足以容纳该字符串外加一个由scanf增加的结尾null字符。当读入空格、新行或已读入指定最大数目的字符时，输入字段终止。

〔 输入值解释为一字符串，相应的变元应为指向一字符数组的指针，该数组大得足以容纳此字符串加上由scanf增加的结尾null字符。输入字段的终止位置确定如下。上面的左括弧随后是一组字符和一右括弧。若该组中第一个字符不是上箭头'^'，输入字段由不在括弧内组中的第一个字符来终止。若第一个字符是上箭头，输入字段由括弧内组中的第一个字符（排除'^'）来终止。

d 输入值解释为十进制整数并被转换成二进制整数，与其相应的变元应为一整数指针。

o 输入值解释为八进制整数并被转换成二进制整数，相应的变元应为一整数指针。

x 输入值解释为十六进制整数并被转换成二进制整数，相应的变元应为一整数指针。

scanf的中心任务是确定它的输入字符串中输入字段的边界，该输入字段含有要转换及要赋值的输入值。为找到这些子字符串，scanf扫描它的输入字符串的那些字符，把它们每一个与由format指向的字符串中相应的字符比较。如果输入字符串中的一字符与格式字符串中相应的字符相匹配，则它被舍弃并读输入字符串中的下一个字符。如果相应的字符不匹配，scanf立即返回。注意，输入字符串中任意数目的空白与格式字符串中任意数目的空白相匹配。格式字符串中的空白是任选的（它被忽略），而在输入字符串中，它可用来对输入

字段定界。因此，所谓相应的字符不简单是那些从它们各自的字符串的起点算起具有相同字节数的字符。在格式字符串中遇到的字符%，它引入一转换说明，输入字符串中的相应字符被认为是输入字段的第一个字节。输入域或延伸直到在输入字符串中遇到一空格字符，或延伸到已读入对于字段宽度所确定的字节数。上面的转换字符c和[只是与此不同的一般规则的例外。输入字段中任何不适当的字符都会引起scanf立即返回。

scanf返回它所赋予的经转换的输入值数，但如果在标准输入源没有输入，则返回常量EOF。

SEEK

```
int seek (fd, offset, origin)
    FILE *fd;
    int offset;
    int origin;
```

seek设置与一开文件相联系的文件输入/输出指针的值。文件由指向它的文件描述符的指针fd指定，它可能已打开作为直接输入/输出或作为有缓冲的输入/输出。seek主要用于与tell和直接文件输入/输出函数read及write配合。在更多的情况下，为了防止数据丢失，seek必须与有缓冲的文件输入/输出函数一起使用。

对区距所赋的值有不同解释，依据赋给起始地址（origin）的值而定：

如果origin=0，文件输入/输出指针将指向文件起点加区距字节。

如果origin=1，文件输入/输出指针将指向文件当前位置加区距字节。

如果origin=2，文件输入/输出指针将指向文件结尾加区距字节。

如果origin=3，文件输入/输出指针将指向文件起点加区距×512字节。

如果origin=4，则文件输入/输出指针将指向文件当前地址加区距×512字节。

如果origin=5，则文件输入/输出指针将指向文件结尾加区距×512字节。

SETEXIT

```
int setexit ()
```

setexit设置它的单元作为“reset”点——对reset的后继调用将程序转向此点执行。以后每次调用reset引起来自函数setexit的一个明显的返回。setexit返回参数n的值，它已被传递到reset。见reset。

SETMEM

```
setmem (p,n,b)
    char *p;
    int n;
    char b;
```

setmem将起始于p的n个连续的字节设置成b指定的值。你可使用setmem预置各种缓冲器和数组。

SLEEP

```
sleep
```

sleep对以4MHz运行的Z80CPU暂停执行零点几秒。你可以通过更改一或两个常数的值使此函数适合不同的CPU和/或时钟速率。

SPRINTF

```
sprintf (s, format, arg1, arg2,…)
char * s, * format;
…
```

sprintf 与 printf 相同，只是它将格式化的输出写入起始于s的字符串。与printf对比，printf是将它的输出写到标准输出，而 sprintf 是将它的输出写到文件。sprintf把一null字符添加到格式化的输出字符串。sprinf是列表函数。

SRAND

```
srand (seed)
    int seed;
```

srand将rand函数的返回值预置为seed中所传递的值。

```
SSCANF (s, format, arg1, arg2,…)
    char * s, * format, * arg1, * arg2,…;
```

sscanf与scanf（及fscanf）相同，只是它的格式化输入字符串是从起始于s以null终止的字符串读，而不是从标准输入读。sscanf不读结尾null字符。sscanf是列表函数。

STRCAT

```
char * strcat (s1, s2)
    char * s1, * s2;
```

strcat将起始于s2字符串的一份拷贝添加到起始于s1的字符串尾部，建立单个以null终止的字符串，注意，由此产生的字符串起始于s1并且只含有一个结尾null字符。

strcat返回一指向结果字符串的指针，该指针与传递给它的参数s1相同。

STRCMP

```
int strcmp (s1, s2)
    char * s1, * s2;
```

strcmp比较起始于s1的字符串与起始于s2的字符串。这个比较类似于字母的比较，只是它是以两字符串中相应字符的数值为基准的。当在这两字符串任一个中遇到第一个null字符时，此比较结束。

根据起始于s1的字符串大于、等于或小于起始于s2的字符串，strcmp分别返回一正整数、零（0）或一负整数。

STRCPY

```
strcpy (s1, s2)
    char * s1, * s2;
```

strcpy将起始于s2的字符串复制到起始于s1的字符串，在复制了null字符后，函数终止。如果起始于s2的字符串的长度大于起始于s1的字符串的长度，跟在后面的字节中的数据可发生重写错误。

strcpy 返回传递给它的参数s1。

STREQ

```
int * streq (s1, s2)
    char * s1, * s2;
```

streq比较起始于s1和s2的字符串中的前n个字符，n是起始于s2的字符串中的字符数(不包括结尾null)。如果两字符串中相应的字符是相同的，streq返回n，否则，返回零(0)。

STRLEN

```
int strlen (s)
    char * s;
```

strlen返回起始于s的字符串中的字符数（不包括结尾null)。

STRNCAT

```
char * strncat (s1, s2, n)
    char * s1, * s2;
    int n;
```

strncat与strcat相同，只是strncat从起始于s2的字符串中至多添补n个字符（从右截取）到起始于s1的字符串的尾部。

STRNCPY

```
char * strncpy (s1, s2, n)
    char * s1, * s2;
    int n;
```

strncpy与strcpy相同，只是strncpy将正好n个字符复制到起始于s1的字符串，如果必要的话，它截断或用null填充起始于s2的字符串。如果起始于s2的字符串含有n个或n个以上的字符，结果产生的字符串则不是以null终止。

TELL

```
unsigned int tell (fd)
    FILE * fd;
```

tell返回从一文件起始处算起至该文件的下一次输入/输出操作开始点的字节区距。文件由指向它的文件描述符的指针fd指定。如果对一长度大于64K的文件调用tell，它的返回值容易发生算术溢出。见otell和rtell。

TOLOWER

```
int tolower(c)
    char c;
```

如果c是一大写体字母ASCII字符，tolower返回c的等价小写体字符，否则它返回c。

TOPOFMEM

```
char * topofmem ()
```

topofmem返回CCEDATA(见brk，evnbrk和sbrk)。

TOUPPER

```
int toupper(c)
    char c;
```

如果c是一小写体字母ASCII字符，toupper返回c的等价大写体字符，否则，它返回c。

UNGETC

```
ungetc(c, fd)
    char c;
    FILE * fd;
```

ungetc将字符c写进最新读的输入/输出缓冲器的字节，该缓冲器与通过fopen打开，有缓冲输入的文件相联系。ungetc还对指向要从文件输入/输出缓冲器读的下一字节之指针取减量，以使它指向刚写过的字节。

ungetc如果成功，返回一未定义的值。若ungetc不能完成其功能，它返回ERROR（－1），例如，指定的文件未通过fopen打开作为有缓冲输入。只有在对一文件调用fgetc，fscanf，getc或getw（有缓冲的文件输入函数）之一后，才能对此文件调用ungets。对一给定的文件调用那些有缓冲的文件输入函数当中，只有对ungetc一次调用能保证得到所期望的结果。

UGETCHAR

```
ugetchar(c)
    char c;
```

ugetchar使下一次调用getchar返回c。在连续的调用getchar之间多次地调用ugetchar将不影响标准输入的状态。

UNLINK

```
int unlink(fspec)
    FILESPEC * fspec;
```

unlink从文件系统中删去fspec处指定的文件。如果文件被成功地删去，unlink返回SUCCESS(0)。如果：1)所给文件说明是无效的，或2)由于一操作系统级的错误，文件不能被删去，则unlink返回ERROR(－1)并且不删除文件。

WRITE

```
int write(fd, buffer, num-bytes)
    FILE * fd;
    char * buffer;
    int num-bytes;
```

write将在num-bytes处指定数目的字节从由buffer指向的缓冲器写到一个为直接（没有缓冲的）输出打开的文件。文件由指向它的文件描述符的指针fd指定。

赋予num-bytes的值必须至少等于一个系统记录中的字节数。在CP/M下，一个系统记录含有128个含节，对于其他操作系统，将有变化。你必须确切地使由buffer指向的缓冲器至少含有在num-bytes中指定的字节数。write不测试缓冲器。

write返回实际所写字节的数目，由于物理磁盘分块的输入/输出操作，它将总是系统记录长度的倍数。如果你指定要写的字节数目不是系统记录长度的倍数，所写的最后记录的一部分会含有其内容是你不曾指定的一些字节。如果由write返回的值小于在num-bytes处指定的字节数，则很可能是磁盘空间用完。这应视为一个错误。如果文件描述符是无效的，或文件不能被读，则返回值－1以指明出错。

每个文件描述符都含有一个指针，此指针指向下一个要在文件输入/输出操作中存取的记录。即使你不指定所写的最后的记录中所有字节的内容，一个对write的调用也把该指

针向前移动所写的字节数。后面对 read 或write的调用将从该指针的新位置开始。因此，所写的那些不是系统记录长度倍数的字节可将未指定的数据(无用数据)留在你的文件中。可通过调用seek来修改文件输入/输出指针位置而不作读或写，但它仍然总是指向一个记录的起点。

第四章　将代码插入运行时库

SuperSoft C 允许用户把操作码传送到优化程序的“C2.RH”文件，此文件自动置位于C程序的起始处，并提供必要的系统运行时间例行程序。这样做的优点是在重新编译程序时，不需要每一次都重新编译和重新优化运行的例行程序。

要完成此项工作，用户需要编译必要的例行程序，产生一汇编语言文件。然后把此汇编语言文件与C2.RH合并。做完这些后，就可以使用那些例行程序，不需要再做更多的工作。

下面是子程序的CRUNT2.C文件被移入运行时间例程的分步详述（不必使用#include “crunt2.c” 伪指令）。

1) 现在我们准备编译“CRUNT2.C”文件。发出如下命令：

```
A>CC CRUNT2.C
⋮
A>C2 CRUNT2.COD +ASM +ZA -RH -RT
```

“+ZA”任选通知翻译程序以字母“A”开始所有标号，这避免了与正常的编译产生的代码冲突，后者每一标号是以一个“C”起始的。-RT和-RH任选免去头部和尾部文件。

2) 接下来，将文件REName(重新命名)为CRUNT2.LIB，准备合并入C2.RH文件。

3) 在AUXFN.LIB文件能被合并前，必须做少量的编辑。键入：

```
A>ED CRUNT2.LIB
*#a
*Op
```

接着你将看到：

```
; C Optimizer V1.1
        ORG     256
; C Compiler V1.1
iswhite:
        lxi     h, 2
        dad     sp
        mov     l, m
        mvi     h, 0
        mov     a, h
        ora     l
        jz      A2
        lxi     h, 2
        dad     sp
```

```
        mov     l, m
        mvi     h, 0
        lxi     d,-32
        dad     d
        mov     a, h
        ora     l
        jz      A2
        lxi     h, 2
        mov     l, e
```

你的文件的开始部分应与在此列出的相同(除版本号之外)。此刻你必须删去下面的行:

ORG 256

4) 至此你必须编辑C2.RH文件。屏幕应示出:

A>ED C2.RH

*#A

*4410p

CCSTART:

```
;
;           which CPU?
            mvi     a, 2
            inr     a
            jpe     c$is$8080
;           is Z80  —this code will not work in ROM
cldir       equ     0b0edh  ;       ldir instruction
            lxi     h, cldir
            shld    cldir1
            shld    cldir2
            xra     a
            sta     cldir1+2
            sta     cldir2+2
c$is$8080:
            lxi     d, ccidstr
            MVI     C, 9
            CALL    CCBDOS
            LHLD    6
            SPHL
            call    main
            rst     0
exit        equ     0       ; exit()—for C
ccidstr:
```

```
        DB      'SuperSoft C Copyright 1981'
        db      0dh,0ah,'Run Time Package V2.3',0DH,0AH,'$'
; * * * * * insert any user code here * * * * *
cc0err:
        pop     d
        push    b
        mvi     c, 9
        call    ccbdos
        lxi     h, 0
        pop     b
        ret
```

把当前编辑程序指针前进到下行之上：

```
; * * * * insert any user code here * * * *
```

并发出命令：

```
*rcrunt2
```

这将使编辑程序读入“CRUNT2.LIB”文件。

5) 用一个“e”退出编辑程序。到此，你已将AUXFN.H含有的全部函数移入称为C2.RH的系统运行时文件，这意味着不必再使用#include “CRUNT2.C” 伪指令。可是无论是否需要，这些函数仍在编译。最好是有几个不同的C2.RH文件可用，以便在需要某些例程时可以使用它们。

6) 除了“main( )”，任何函数都可被移入系统运行时间库。

附 录

附录A SuperSoft C与标准Unix C的区别

标准的C语言特性尚未实现的有：STATIC 存储级；TYPEDEF 说明；LONG，FLOAT和DOUBLE数据类型；位字段的说明和使用；及初始化。

函数的所有形式变元必须在该函数内部说明。即：

```
func(aa, bb, cc)
  int bb;
  int aa, cc;
{
```

…被接受，但

```
func (aa, bb, cc)
{
```

…将产生一警告消息。

代码生成程序（优化程序）还没有把任何前缀连到变量名，因此，你的C源代码中的全程标识符有可能与汇编程序的名字和关键字冲突。所以必须避免在C源代码中使用你的汇编程序保留的任何符号或关键字。这两点缺陷会在以后的版本中得到补救。

参数化的定义，#if，#ifdef，#ifndef，#undef，#else，#endif和#line预处理程序伪指令还不支持。

附录B SuperSoft C编译程序配置

SuperSoft C编译程序目前可用的机器和操作系统配置如下：

目前的操作系统是：CP/M-80，MP/M-80，CP/M-86，MP/M-86(及与其兼容的)；PC-DOS（和兼容的）；UNIX，XENIX（和兼容的）；Central Data ZMOS。

目前的主机或目的CPU是：Intel8080，Intel8085，Intel8086，Intel8088，Intel186，Intel188，Intel286，Zilog Z80，Zilog Z8001，Zilog Z8002和Zilog Z8003。

SuperSoft适用于主CPU，目的CPU和操作系统的任一有效的组合。

附录C　几个共同的问题及解答

问　　题	解　　答
在汇编阶段一些函数未定义	记住要#include含有所需文件的文件
汇编阶段一些变量未定义	记住说明所有要使用的变量
在汇编阶段，一些函数被标记为重复定义错	可能某个函数或变量定义了两次
当使用磁盘输入/输出时，程序不完成预定动作	记住，你必须将“allocs”预置成0（即allocs=0)
当进行地址分配时，你的程序不完成预定动作	记住将allocs 设置成0。（见上）
汇编期间标出“P”错误	一变量或函数已被定义好几次

附录D　提供的函数设置

设置在ALLOC.C的函数：

alloc　　evnbrk　　free

设置在C2.RH的函数：

bdos　　brk　　ccall　　ccalla
exit　　reset　　setexit　　streq

设置在CRUNT2.C的函数：

getchar　　gets　　isalpha　　isdigit
islower　　isupper　　iswhite　　movmem
putchar　　puts　　sbrk　　setmem
strcpy　　strlen　　toupper　　ungetchar
xmain

设置在FUNC.C的函数：

abs　　atoi　　bios　　getval
index　　initb　　initw　　isalnum
isascii　　iscntrl　　isnumeric　　isprint
ispunct　　isspace　　kbhit　　pause
peek　　poke　　qsort　　rand
rindex　　sleep　　srand　　strcat

strcmp	strncat	strncpy	tolower

设置在STDIO.C的函数：

close	creat	exec	execl
fabort	fclose	fflush	fopen
getc	getw	open	otell
pgetc	pputc	putc	putw
read	seek	tell	ungetc
unlink	write		

设置在FORMATIO.C的函数：

fprintf	fputs	fscanf	printf
scanf	sprintf	sscanf	

编译校对：刘维平、刘运基、高江虹、丁　平、赵成武

责任编辑：谢雪峰、王宏成、暴雪琴、吴雪莹、王晓东