PL/I-80™

APPLICATIONS GUIDE

DIGITAL RESEARCH™

PL/I-80™ APPLICATIONS GUIDE

PL/I-80 APPLICATIONS GUIDE

Digital Research
P.O. Box 579
801 Lighthouse Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

The "PL/I-80 Applications Guide" was prepared using the Digital Research TEX Text formatter.

****
* Second Printing: December, 1980 *
****

TABLE OF CONTENTS

1. INTRODUCTION 1

2. PL/I-80 SYSTEM OPERATION 4

3. PL/I-80 PROGRAMMING STYLE 11

4. PL/I-80 INPUT/OUTPUT CONVENTIONS 14
   4.1. The OPEN Statement 14
   4.2. The PUT LIST Statement 18
   4.3. The GET LIST Statement 20
   4.4. The PUT EDIT Statement 21
   4.5. The GET EDIT Statement 23
   4.6. The FORMAT Statement 25
   4.7. The WRITE Statement 25
   4.8. The READ Statement 27

5. PL/I-80 PROGRAMMING EXAMPLES 31
   5.1. Polynomial Evaluation 31
   5.2. The File Copy Program 34
   5.3. Name and Address File Processing 37
   5.4. An Information Management System 40

6. LABEL CONSTANTS, VARIABLES, AND PARAMETERS 55

7. EXCEPTION PROCESSING 59
   7.1. The ON Statement 60
   7.2. The REVERT Statement 61
   7.3. The SIGNAL Statement 62
   7.4. The ERROR Exception 64
   7.5. FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, and ZERODIVIDE 66
   7.6. ENDFILE, UNDEFINEDFILE, KEY, and ENDPAGE . . . 66
   7.7. ONCODE, ONFILE, ONKEY, PAGENO, and LINENO . . . 68
   7.8. An Example of Exception Processing 69

8. APPLICATIONS OF CHARACTER STRING PROCESSING 75
   8.1. The OPTIMIST Program 75
   8.2. A Free-Field Scanner 78

9. APPLICATIONS OF LIST PROCESSING 84
   9.1. Managing a List of Words 84
   9.2. A Network Analysis Program 90

10. USES OF RECURSION IN PL/I-80 104
   10.1. Evaluation of Factorials 104
   10.2. Evaluation of the Ackermann Function 114
   10.3. An Arithmetic Expression Evaluator 117

11. SEPARATE COMPILATION AND LINKAGE 126
   11.1. Data and Program declarations 126
   11.2. An Example of Separate Compilation 129

12. COMMERCIAL PROCESSING USING PL/I-80 135
12.1. A Comparison of Decimal and Binary Operations 135
12.2. Decimal Computations in PL/I-80 137
12.3. Addition and Subtraction 139
12.4. Multiplication 141
12.5. Division 143
12.6. Conversion Between Fixed Decimal and Float Binary 146
12.7. A Simple Loan Payment Schedule 147
12.8. Ordinary Annuity 150
12.9. Formatted Loan Payment Schedule 156
12.10. Computation of Depreciation Schedules 167

1. INTRODUCTION TO PL/I-80

The PL/I-80 system is a complete software package for application programming under the Digital Research CP/M and multiprogramming MP/M operating systems (the name, by the way, is pronounced PL-ONE, but is spelled with the Roman numeral "I", so don't be confused when you see lower case "pli" in various commands and program examples). The PL/I-80 language is based upon the new Subset G language defined by the ANS PL/I Standardization Committee X3J1. The subset contains all necessary application programming constructs of full PL/I, discarding seldom-used or redundant forms. The resulting language constraints encourage good programming practices while simplifying the compilation task.

PL/I-80, like all programming languages (and most natural languages) is most easily learned by studying working examples. The purpose here is to introduce the mechanics of compiling, linking, and executing programs, and to introduce useful facilities of the language. The presentation is followed by detailed sample programs which illustrate Input/Output processing, scientific computation, business applications, along with string and list processing.

The best way to learn PL/I-80 is to study these examples by reading the associated text, examining the programs, and cross-checking with the reference manual when necessary. Once you understand the operation of a particular sample program, you may wish to modify the program to enhance its operation and further your experience with the language. If you are a beginner, check with your local universiy or community college: programming courses are often available which specifically cover the PL/I language (you'll find you have a particular advantage over your classmates, since your turnaround time is only a few minutes). Alternatively, you may wish to purchase one of the hundreds of textbooks which are currently available on the subject. Most of these textbooks are found in university bookstores or through special orders, and cover the basics of PL/I.

Your PL/I-80 system diskette does not contain a CP/M operating system, so you must first make a copy of the PL/I-80 programs for everyday use, and generate a CP/M system on the first two system tracks (be sure you have read your licensing agreement - you have certain responsibilities when you make copies of Digital Research programs). Load your newly created diskette into drive A, reboot CP/M, and type a DIR command. You'll find several types of files, including:

COM	CP/M Command Files or Composite Programs (PLI.COM is one of these)
DAT	Default Data File Type
IRL	Indexed Relocatable Code (PLILIB.IRL is the library)

OVL	PL/I-80 Compiler Overlays (PLI0, PLI1, and PLI2)
PLI	PL/I-80 Source Programs (e.g., type OPTIMIST.PLI)
PRL	Page Relocatable Object (Used in MP/M Partitions)
PRN	Printer Disk File (Program Listing to Disk)
REL	Relocatable Object Code (Such as Developed Programs)

The only files which contain printable characters are the "PLI" source programs and "PRN" printer listing files. Several programs are included on the PL/I-80 system disk which correspond to various examples in this manual, along with additional programs of increasing complexity. To begin with, try running a program which has already been compiled and linked to the PL/I-80 runtime library. Type the command

```
OPTIMIST
```

the OPTIMIST program will load and respond with

```
What's up?
```

Answer by typing the sentence

```
None of these programs make sense.
```

(be sure to end your input with a period, followed by a return). After you get the response from the OPTIMIST, you can type a few more sentences if you wish, then type a control-C to stop the OPTIMIST.

The OPTIMIST is a PL/I program which is included on your PL/I-80 system diskette in source form. Display the program using the type command

```
TYPE OPTIMIST.PLI
```

As an example, go through a complete compilation and test of the OPTIMIST program by following the steps shown below. Note that although you can run the OPTIMIST program in any memory size, the PL/I-80 compiler needs at least a 48K CP/M system for operation. Be sure that the PLI.COM and overlay files are on your default disk, otherwise you'll get the error message "NO FILE: PLI0.OVL" when you start the compiler. Compile the OPTIMIST program by typing

```
PLI OPTIMIST
```

The compiler will process the program in three steps, referred to as "passes," marked by the messages

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

If you examine your directory, you'll find the file

```
OPTIMIST.REL
```

which contains the relocatable machine code produced by the PL/I-80 compiler for the OPTIMIST program. If you wish, you can recompile with the listing option so you can view the program as it is being compiled. This is accomplished by typing

```
PLI OPTIMIST $L
```

The compiler will proceed as before, but this time it produces the program listing in the last pass.

The relocatable machine code resulting from the compilation is not directly executable, so you'll have to link the REL file with the PL/I-80 runtime subroutine library by typing

```
LINK OPTIMIST
```

The LINK-80 program produces an OPTIMIST.COM file which replaces the one that came with your diskette. Your new OPTIMIST program should operate in the same manner as the original program.

2. PL/I-80 SYSTEM OPERATION

First it's necessary to expand upon the compiler and linker operations presented in the previous section. In general, the PL/I-80 compiler reads program files prepared under CP/M or MP/M using the standard program editor (ED). The program is processed by the PL/I-80 compiler, linked using LINK-80, and subsequently tested. As an example, consider the simple payroll program compiled and listed in Figure 2-1. The compiler proceeds through the first two passes and lists each line containing an error, with the line number to the left, a short error message, and a "?" below the position in the line where the error occurred. You can, at any time, abort the compilation by typing a carriage-return at the console. This particular facility is useful if the number of error diagnostics is excessive, and you wish to make certain corrections before proceeding. The program line number is listed on the left, followed by a letter a-z which denotes the nesting level for each line. The main program level is "a", and each nested BEGIN advances the level by one letter, while each nested PROCEDURE level advances by two. The relative machine code address for each line is listed next as a four digit hexadecimal number. This address is useful in determining the amount of machine code generated for each statement and the relative machine code address for each line of the program. The source statement is printed on the line following the relative machine code value.

The $L parameter provided on the command line which starts the compiler is called a "compiler switch" and enables the listing option. A list of compiler switches is shown below. In each case, the single letter command follows the "$" symbol given in the command line, with a maximum of seven command letters following the dollar sign. The default when no parameters are specified results in a compilation with no listing, where all error messages are sent to the console.

B Builtin Subroutine Trace
shows the library functions which are called-out by your PL/I program

D Disk File Print
sends the listing file to disk, using the file type PRN

I Interlist Source and Machine Code
decodes the machine language code produced by the compiler in a pseudo-assembly language form

L List Source Program
produces a listing of the source program with line numbers and machine code locations (automatically set by the I switch)

N Nesting Level Display
enables a pass 1 trace which shows exact balance of DO, PROC, and BEGIN

```
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
```

Figure 2-1. Wage Program Listing.

exact balance of DO, PROC, and BEGIN statements with their corresponding END statements

P Page Mode Print
inserts form feeds every 60 lines, and sends the listing to the printer

S Symbol Table Display
shows the program variable names, along with their assigned, defaulted, and augmented attributes

PL/I-80 allows separate compilation of individual procedures, where each compilation produces a "REL" file. Only one procedure can be included with "options(main)" and this becomes the main program for the module, while all other subroutines have the usual PL/I procedure header.

The file PLILIB.IRL contains the subroutines which can be called-out by your PL/I-80 program, and as shown in the previous section, the relocatable machine code is linked with the PL/I run-time library subroutines by typing the command:

```
link wage
```

producing a Composite Program. If you are operating under the MP/M system, the command

```
link wage[op]
```

again produces a Composite Program, but in this case the machine code is in page relocatable format which executes in an MP/M partition. In the first case, LINK-80 produces a "wage.com" file for execution under CP/M or in an absolute segment under MP/M. In the second case, LINK-80 produces a file named "wage.prl". In addition to the machine code files, LINK-80 also produces the symbol table file, named "wage.sym" which can be loaded for debugging purposes under SID or ZSID.

Figure 2-2a shows the output from LINK-80 for the simple wage program. By convention, the subroutines extracted from the PL/I-80 library are preceded by the "?" symbol in order to avoid conflicts with user-defined symbol names. Symbols enclosed within slashes ("/") are EXTERNAL variables (COMMON in Fortran), and symbols followed by "*" are undefined. It is important to note that LINK-80 implements the Microsoft linkage editing format in order to be compatible with a variety of other language processors. This format, however, restricts the length of external names to 6 characters, so even though your internal variable names can be as long as 31 characters, make sure all your externally defined names are unique in the first 6 positions.

By default, LINK-80 does not list the "?" symbols from the library. If you want a complete listing of these symbols, type

```
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
```

Figure 2-2a. A Simple Link Edit for the Wage Program.

```
A>link wage[q]
LINK V0.4

PAYROL   0100   ?START   1D60   ?SYSPR   03BB   ?SLCTS   1756
?PNCOP   02F3   ?QIOOP   1CDF   ?SYSIN   03B7   ?GNVOP   08CB
?IM22N   17C1   ?SSVFS   177F   ?QCDOP   12F1   ?DSTOP   1563
?SCVCM   16E7   ?SKPOP   0526   ?DLDOP   153C   ?DMUOP   15DC
?QDDSR   1446   ?DSUOP   15BC   ?QDCOP   14A0   ?PNVOP   0317
?SLVTS   1754   ?STOPX   1E71   /?FILAT/ 1ECF   /?FPB/   1ED8
?PNBOP   02ED   ?PNCPR   05C5   ?IS22N   1823   ?SIOOP   03C0
?SIOPR   03DE   /?FPBST/ 1F06   /SYSIN/  1F19   /SYSPRI/ 1F3E
?OIOOP   069D   ?FPBIO   084E   ?OIOPR   06BC   ?BSL16   16D6
?SIGNA   197E   ?SKPPR   052F   ?GNCPR   0A45   ?WRBYT   0F2C
?PAGOP   08BD   ?NSTOP   16DC   ?SMVCM   179E   ?SJSVM   171C
?SSCFS   1769   ?QB08I   12DD   ?OPNFI   0E09   /?FMTS/  1F66
?FPBOU   1D33   ?FPBIN   1CEB   ?GNVPR   0908   ?RDBYT   0F19
?RDBUF   0F52   ?WRBUF   0F75   ?CLOSE   105E   ?GETKY   108F
?SETKY   10B5   ?PATH    1042   ?BDOS    0005   ?DFCB0   005C
?DFCB1   006C   ?DBUFF   0080   ?ALLOP   182A   ?FREOP   18C0
?ADDIO   1DBC   ?SUBIO   1DD3   ?WRCHR   1D49   ?RFSIZ   11BA
?RRFCB   122C   ?RWFCB   1231   ?QB16I   12E0   ?QDDOP   13BF
?DNGOP   15A5   ?QDDSL   13D7   ?DOVER   16BC   ?BSL08   16D0
?SCCCM   16EE   ?SJSCM   171E   ?SJSTS   1730   ?SMCCM   17A2
?IM22    17C1   ?IM11    17F5   ?IS22    1823   ?ERMSG   1E8C
?BEGIN   3055   /?ONCOD/ 1F6E   ?SIGOP   196E   ?STACK   304F
?ONCPC   1CA3   /?CONSP/ 1F71   ?ONCOP   1C06   ?REVOP   1C5B
/?CNCOL/ 1F92   ?RECLS   3BFC   ?BOOT    0000   ?CMEM    1ECF
?DMEM    3059

ABSOLUTE      0000
CODE SIZE     1DCF (0100-1ECE)
DATA SIZE     10C5 (1F94-3058)
COMMON SIZE   00C5 (1ECF-1F93)
USE FACTOR      4E
```

Figure 2-2b. Link Editing using the LINK-80 "Q" Switch

```
link wage[q]
```

and a listing of the form shown in Figure 2-2b results.

Execution proceeds by typing the name of the COM or PRL file, as shown in Figure 2-3. The program executes, and prompts the console for input. As discussed in the I/O section which follows, input from the console is free-field with the full line editing facilities of CP/M and MP/M. The message

```
End of Execution
```

is displayed upon completion of the program before returning to the console command level.

Various run-time errors terminate program execution if not explicitly intercepted within the PL/I program. In this case, the message form shown below is displayed:

```
error-condition (code), file-option, auxiliary-message
Traceback:  aaaa bbbb cccc dddd # eeee ffff gggg hhhh
```

where the "error-condition" is one of the standard PL/I conditions

```
ERROR   FIXED OVERFLOW   OVERFLOW   UNDERFLOW
  ZERODIVIDE   END OF FILE   UNDEFINED FILE
```

and "(code)" is an error subcode which identifies the origin of the error. The "file-option" is printed when the error involves an I/O operation, and takes the form:

```
internal=external
```

where "internal" is the internal program name which references the file involved in the error, and "external" is the external device or file name associated with the file. The "auxiliary-message" is printed whenever the preceding information is insufficient to identify the error. Finally, the "traceback" portion lists up to eight elements of the internal stack in order to help identify the program statement which produced the error. If the stack depth exceeds eight elements, the "#" separates the topmost four elements on the left from the lowermost four elements on the right. In the form shown above, element aaaa corresponds to the top of stack, while hhhh corresponds to the bottom of the stack. Unless the statement in error has filled the low end of the stack with a character or decimal temporary, the value hhhh determines the main program statement in error, as described below.

An execution of the wage program, shown in Figure 2-4, gives an example of the diagnostic form. In this case, the first console input is entered properly, but the second line terminates console input with an end-of-file (control-Z). The END OF FILE condition is raised for the SYSIN file which is standard console input. The external device connected to SYSIN is, in this case, the operator's console, denoted by CON.

```
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
```

Figure 2-3. Execution of the Wage Program.

```
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
```

Figure 2-4. Error Traceback for the Wage Program.

by CON.

The traceback shows the lowest stack location as 0146 (hexadecimal), corresponding to the main program statement in error. Referring back to Figure 2-2a, the PAYROL program address is shown in the upper left corner as 0100, which is the normal beginning location under CP/M. The difference 0146-0100 = 0046 is the relative location of the error. The Figure 2-1 listing shows that the address 0046 falls between the code addresses listed alongside line 18 (003A to 00A0-1), and thus it was within this line that the error occurred.

3. PL/I-80 PROGRAMMING STYLE

Before we get into PL/I-80 programming details, it's worthwhile discussing the topic of programming style. PL/I is a "free-format" language, which means that you can write programs without regard to column positions and specific line formats. Each line can be up to 120 characters in length (terminated by a carriage return), and is logically connected to the next line in sequence. The compiler simply reads the source program from the first through the last line, disregarding line boundaries. With this freedom of expression comes a responsibility on your part to adhere to some stylistic conventions so that your programs can be easily read and understood by other programmers. Professional programmers know that it's not enough to just have a program that produces the proper output (although that's a desirable quality!). The program must also be consistent in form, and divided into logical segments which are easy to comprehend. A well-constructed program is a work of art which is appreciated for its structure as well as its function.

There are many stylistic conventions wnich are used throughout the industry. The rules given below illustrate one set of conventions which we'll use fairly consistently throughout the examples this manual.

First, note that PL/I programs can be written in either upper or lower case. Internally, the PL/I compiler translates all characters outside of string quotes to upper case. We generally prefer the use of lower case throughout programs since it decreases the program density and generally improves readability. Second, indentation is used throughout PL/I to set off various declarations and statements. In order to simplify indentation, the PL/I compiler expands tabs (control-I characters) to every fourth column position. Be aware, however, that CP/M utilities, such as ED, expand tabs to multiples of eight columns, so the line will appear wider during the edit and display operations. Note also that the TRUNC (truncate) error is issued if the expanded line length exceeds 120 columns. Program statements start at the outer block level in the first column position. Each successive block level, initiated by a DO, BEGIN, or PROCEDURE group is started at a new indentation level, either four spaces or one tab stop. Statements within a group are given at the same indentation level, with procedure names and labels on a single line by themselves. An IF statement should be directly followed by the condition and the THEN keyword, with the next statement indented on the next line. When the IF statement has an associated ELSE, the ELSE starts at the same level as the IF. Further, the statement following the ELSE is indented and placed on the next line. Finally, the declaration statement should be formed by placing the DECLARE keyword on a single line, followed by the declared elements indented on the following line. Complicated attribute factoring should be avoided since this reduces program readability. Blank lines (i.e., lines containing only a carriage return) are inserted when necessary to improve paragraphing, and most often used to separate logically distinct segments of the program. Many of the longer PL/I keywords have abbreviations (e.g., DCL is equivalent to DECLARE). Inconsistent use of abbreviations produces awkward programs, so within a project

use either the long or short forms, but not both.

In general, large programs are divided into several logical groups, or "modules," where each module performs a specific primitive function. These modules are expressed as PL/I subroutines which are either locally or externally defined. Local subroutines become a part of the same main or subprogram, while external subroutines are separately compiled and linked together using LINK-80. Locally defined subroutines are placed at the end of the program so that the beginning contains only declarations and top-level statements which call the local subroutines. As a general rule, neither the top-level statements, nor the locally defined subroutines, should exceed one or two pages in length. If you are just learning to program PL/I-80, you'll probably want to use just a main program with locally defined subroutines, following the form of most of the examples of this manual. When your application programs increase in size, however, it may be more effective for you to break programs into separate modules so that individual segments can be compiled and linked in pieces, thus reducing overall development time.

Comments are a welcome sight within programs, but avoid introducing them at random spots throughout the source file since they detract from the overall structure. Again, consistency is the watchword: a good practice is to place the comment at the head of subroutines or logical statement groups, and if you've properly decomposed your program you'll find that these explanatory remarks, along with your well-formed program, provide the required information to understand program operation. The program shown in Figure 3-1 illustrates the conventions presented in this section.

```
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
```

Figure 3-1. An Illustration of Stylistic Conventions.

4. PL/I-80 INPUT/OUTPUT CONVENTIONS

We'll start with a detailed discussion of the PL/I-80 I/O system. This will provide the necessary foundation for the examples that are presented later. If this section becomes too detailed for you, skip to the GET and PUT statements where the simplest I/O facilites are found. Scan the example programs in later sections and then return to reread the details - they'll make more sense next time.

PL/I-80 provides a device independent I/O system which interfaces PL/I-80 programs with the CP/M and MP/M file system. The parameters for this interface are provided in the OPEN statement and through the defaulting mechanisms of the GET, PUT, READ, and WRITE statements.

4.1. The PL/I-80 OPEN Statement.

The OPEN statement is optional, and takes place automatically when a file is accessed using GET, PUT, READ, or WRITE when an explicit OPEN has not occurred. If you do not want the file to take the default attributes, it's necessary to explicitly OPEN the file before it is accessed. The form of the open statment is:

```
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)
```

where the attributes may be listed in any order. The value f denotes the value of a file constant or file variable and must be named in the open statement. All other attributes are optional, and take default values shown below. The values i and j denote FIXED BINARY expressions, while c represents a character expression. Attributes shown on the same line are in conflict and, if not included, the first attribute on each line with multiple attributes becomes the default value. The last four attributes take default values as shown below:

```
ENV (B(128))
LINESIZE (80)
PAGESIZE (60)
TITLE ('f.DAT')
```

A STREAM file contains variable length ASCII data, while a RECORD file generally contains pure binary data. The lines of an

ASCII data file are defined by the interspersed carriage return line feed sequences. Note that the line feed is included following each carriage return when the file is created using the ED program. Files created using PL/I-8Ø can, however, contain a series of line feeds without preceding carriage returns. In this case, the end of line is sensed when the line feed is encountered. The PRINT attribute applies only to STREAM files, and generally suggests that the data is eventually destined for display on a line printer device.

INPUT files are expected to exist at the point of the OPEN statement, while OUTPUT files are deleted, if they exist, and created at the OPEN statement. An UPDATE file cannot have the STREAM attribute, and can be both written and read. An UPDATE file is created if it does not exist.

SEQUENTIAL files are read or written from beginning to end, while DIRECT files can be accessed randomly. A DIRECT file automatically receives the RECORD attribute.

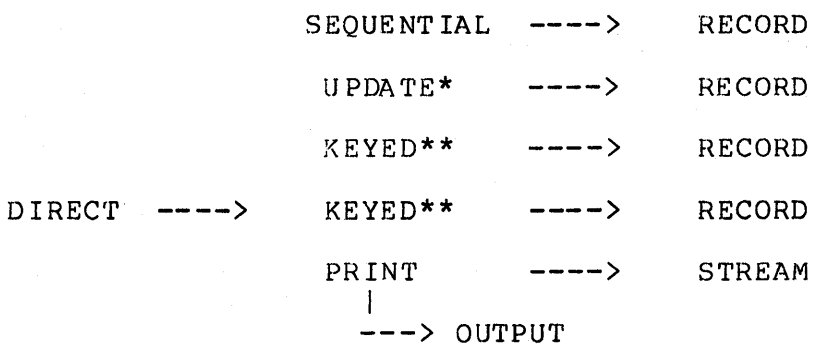
A KEYED file can be accessed through the use of keys, and automatically receives the RECORD attribute. In PL/I-8Ø, a KEYED file is simply a fixed-length record file, where the key is the relative record position of the record being accessed, based upon the fixed record size.

The ENV (Environment) attribute defines fixed and variable length record files, along with the internal buffer sizes. The form ENV(B(i)) causes the I/O system to buffer i bytes of storage, where i is internally rounded-up to the next multiple of 128 bytes. In this case, the file is assumed to have variable length records and, in PL/I-8Ø, cannot have the KEYED attribute since the record size is not fixed.

The ENV(F(i)) form defines a file with fixed length records containing i bytes each, which is internally rounded to the next multiple of 128 bytes. In order to comply with the PL/I standard, you are also required to define files with fixed-length records as KEYED. The default buffer size is, in this case, i bytes rounded to the next higher multiple of 128 bytes.

The form ENV(F(i),B(j)) defines a file containing fixed length records of i bytes (rounded up, as above), with a buffer size of j bytes (again, rounded up). Note that you can specify a fixed length record larger than the buffer size. Again, you are required to include the KEYED attribute to maintain compatibility with the standard.

If you specify the KEYED attribute, then the record length must be given using either the ENV(f(i)) or ENV(F(i),B(j)) form. Further, PL/I-8Ø requires all UPDATE files to be declared with the DIRECT attribute in order that the individual records may be located. After applying the default values, the following attributes are added:



* In PL/I-80, UPDATE must also be DIRECT
** In PL/I-80, KEYED must have ENV(F(i)) or ENV(F(i),B(j))

That is, the attribute RECORD is added to SEQUENTIAL, UPDATE, and KEYED files, while STREAM is added to PRINT files. PRINT files are also automatically given the OUTPUT attribute. The KEYED attribute is added to DIRECT files (which, in turn, adds the RECORD attribute).

An OPEN statement cannot contain conflicting attributes obtained in the OPEN statement itself or through the default or implied mechanisms.

So, what does all this mean? Basically, if you want to read a file containing ASCII characters, you have to define it as a STREAM file, otherwise it must be a RECORD file. Normally, this is all you have to deal with. If you want to perform random access, define the file as DIRECT and use ENV to define the record size. If you just want to read the keys, you can define the file as KEYED, and leave off the DIRECT attribute. You'll get quite a bit more insight by reading the examples in the sections which follow.

The LINESIZE option applies only to STREAM files, and defines the maximum input or output line length. The PAGESIZE option applies only to STREAM OUTPUT files, and defines the length of a page.

The TITLE(c) option allows programmatic connection between an internal file name and an external device or CP/M file. When not specified, the external file name becomes the value of the file reference, with the type "DAT". Otherwise, the character string c is evaluated to produce either a device name:

```
$CON   System Console
$LST   System List Device
$RDR   System Reader Device
$PUN   System Punch Device
```

or a disk file name

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

where "d:" is an optional drive name, and x and y represent the file

name and file type, respectively. Note that either x or y, or both, may be $1 or $2. If $1 is specified, then the first default name is taken from the command line and filled into that position of the title. Similarly, $2 is taken from the second default name and filled into the position in which it occurs. The file name x cannot be blank, nor can x, y, or d contain "?" symbols. The physical I/O devices $CON, $RDR, $PUN, and $LST can be opened as STREAM files only, $RDR must be INPUT, and $PUN and $LST must have the OUTPUT attribute.

Several examples of the OPEN statement are shown below, assuming each file fi has been declared elsewhere as a file constant. In each case, the source statement is listed, with the default and augmented attributes shown below the statement.

```
open file (f1);
  STREAM INPUT LINESIZE(80) TITLE('f1.DAT') ENV(b(128))

open file (f1) print;
  STREAM OUTPUT LINESIZE(80) PAGESIZE(60)
  TITLE('F2.DAT') ENV(B(128)

open file (f3) sequential title('new.fil');
  RECORD INPUT ENV(B(128))

open title('a:'||c) file (f4) direct keyed env(f(2000));
  RECORD INPUT ENV(f(2048),b(2048))

open update keyed file (f5) env(f(300),b(100));
  RECORD ENV(f(384),b(128)) TITLE ('f5.DAT')

open input direct title(c||'OUT') env(f(100),b(2000));
  RECORD ENV(f(128),b(128))
```

Integer expressions are allowed wherever a constant is shown above. Thus the statment

```
open file (f1) linesize(k+3) pagesize(n-4) env(b(x+128));
```

is a valid form of the open statement. Finally, note that when an OPEN statement references a file which is already open, the statement is ignored.

the form of the close statement is

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

where f is a file variable or file constant. All open files are automatically closed at the end of the program or upon execution of the STOP statement.

Files opened with the STREAM attribute can be accessed through GET and PUT statements, while files with the RECORD attribute are accessed through READ and WRITE, with one exception noted later.

4.2. The PL/I PUT LIST Statement.

The PUT LIST statement takes the form:

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

where all elements are optional (although at least one must be specified). PUT LIST options can be given in any order, but the LIST option, if specified, must occur last. In the form shown above, f is a file variable or constant, and i is an integer expression. The LIST option includes a data list, denoted by d, and described in the paragraphs which follow.

The PUT statement writes data or control characters to the file given by FILE(f), or to the standard console file SYSPRINT which is implicitly declared in all PL/I-80 programs. If the file has not been previously opened, it is automatically opened when the PUT statement is executed. The SYSPRINT file is implicitly opened as:

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

The SKIP option can take one of the forms:

```
SKIP    SKIP(i)
```

where i is a FIXED expression. The first form causes a carriage return line feed sequence to be inserted into the output file, and resets the column position of the output file to 1. The form SKIP(i) inserts a single carriage return into the output stream, followed by i line feed characters. Note that SKIP(0) moves the column position to 1 (e.g., the cursor is positioned to the left of the line on a CRT display), without a line feed.

The PAGE option causes an automatic SKIP(0), and places a form-feed character into the output stream. The order in which the PAGE and SKIP options are listed in the PUT statement is of no consequence: if specified, the PAGE option is executed first, followed by the SKIP option.

The data list d given in the LIST option takes the general form:

```
LIST (d1,d2, ..., dn)
```

where each di is either a simple constant, scalar expression, or iterative group. An iterative group takes the form

```
(e1,e2, ... ,em DO iteration)
```

where, again, e1 through em are themselves constants, scalar expressions, or iterative groups. The "iteration" portion of the list takes the same form as a PL/I DO-group header, and controls the number

of times each embedded group is written. The iterative group has the same effect as the PL/I-80 DO-group shown below:

```
DO iteration;
PUT LIST(e1,e2, ..., em);
END;
```

Each element of the data list is evaluated, and converted to a string constant according to the normal PL/I-80 conversion rules. If the data item is a string value, and the output file does not have the PRINT attribute, then quote symbols are placed around the string, and each embedded single quote is changed to a double quote value. If the data item is a bit string, then the character "b" is appended to the end of the output value. Values written to a disk file in this manner are suitable for subsequent input using a GET LIST statement.

Upon converting the data item to a string value, the current column position is compared to the linesize to ensure that the data item will fit in the current line. If not, an automatic SKIP is issued, and the data item is written on the following logical line. If the item is not the first on a line, a preceding blank is written to separate each data item (this blank is included in the data item length when multiple data items are written).

Examples of the PUT statement are shown below, followed by a short explanation of their effect:

```
put skip;
```

moves to a new line in the file SYSPRINT (usually the console).

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

writes a string to the standard output file SYSPRINT, producing either

```
Type Name:      or 'Type Name:
```

The first form is produced if the PRINT attribute is present.

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

writes a form-feed to the file specified by f, followed by a carriage return and three line feed characters. The three variables a,b, and c are then converted to varying character strings and sent to the file f.

Additional valid forms are shown below.

```
put skip list( x||'+'||y,((x+y)));
put list ((a(i),b(i) do i=1 to 10));
put list (x,((a(i,j) do j=1 to n) do i=1 to m));
put list((x(i) do i = 1 to k+m while(x(i) < 10)));
```

4.3. The PL/I-80 GET LIST Statement.

Similar to the PUT statement, the GET statement is used to read files with the STREAM attribute. The form of the GET statement is:

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

where the FILE, SKIP, and LIST options obey the constraints of the PUT statement shown above. If the FILE(f) option is not included, the standard input file SYSIN is accessed with the automatic OPEN statement:

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

The file f must have the STREAM INPUT attributes. The SKIP option causes the input stream to be flushed to the next end of line, while the SKIP(i) statement reads through the next i line feed characters. The data items given in the LIST option must be scalar variables or iterative groups, as given in the PUT statement, and must be valid targets of assignment statements.

When the console is accessed through a GET statement, the PL/I-80 I/O system accesses the console and waits for input. The operator can type up to 80 characters, using the normal line editing facilities of CP/M, before issuing a carriage return (an automatic carriage return is issued following the 80th character). In this case, the carriage is returned to the left side, followed by a line feed. This buffered line (including the line feed) is then used for subsequent GET statement input.

External data read by the GET statement is taken by PL/I-80 as a sequence of characters, or as a bit or character string surrounded by string quotes. Each data item is separated by one or more blanks and an optional comma character. It must be possible to convert the data read in this manner to the type of the target item. If, for example, a decimal number is specified in the GET statement, then the input value must contain a valid decimal number.

Note that if a data item is empty (i.e., a pair of commas is encountered, possibly separated by intervening blanks), the value of the target data item is not altered. This particular feature of PL/I is useful when displaying data at a console which is then reread and optionally changed.

The carriage return found at the end of each input line serves as a delimiter (blank or comma). Further, string constants cannot go beyond a line boundary, and, in fact, are automatically closed when an end of line is encountered. (Thus, only the leading quote is necessary when typing string data at the console.) The normal CP/M end of file character (control-Z) can be typed at the console, but it must be the first character on the line.

4.4. The PL/I-80 PUT EDIT Statement.

The PUT EDIT statement is similar to PUT LIST described above, except data is written into particular fields of the output line, as described by a list of format items. The form of the PUT EDIT is

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

where the data list specifies a number of values to be written in fixed fields defined by the format list fl. The list of data items to write, denoted by d, obeys the same rules as the PUT LIST. One or more format items are given in the list fl, separated by commas, and optionally grouped within parentheses. Any format item may be preceded by a positive constant integer value not exceeding 254, which determines the number of times to apply the format item or group of format items. Each element of the data list is paired with a format item which determines the column position and interpretation of the data element. See the Reference Manual, as well as the GET EDIT statment, for additional details. The format items are:

A	Writes the next alphanumeric field using the size of the (converted) character data as a field width.
A(n)	Similar to the A format, except the field width is n, with truncation or blank pad on the right.
B	Writes a bit string value to the output, where the field width is determined by the precision of the data item.
B(n)	Similar to B, except the field width is given by the constant n, with truncation or blank pad on the right.
B1	Equivalent to the B format shown above.
B1(n)	Equivalent to B1 shown above.
B2	Equivalent to B, except the digits are written in radix 4 notation (Ø,1,2,3).
B2(n)	Equivalent to B(n), except radix 4 digits are printed.
B3	Equivalent to B, except radix 8 notation is used for output (Ø to 7).
B3(n)	Equivalent to B(n), except radix 8 digits are printed.

B4	Equivalent to B, except radix 16 digits are written to the field (Ø-9, A-F).
B4(n)	Equivalent to B(n), except radix 16 digits are written.
COLUMN(n)	Moves to column position n before writing the next data item. This may cause the current line to be flushed.
E(n)	Writes a data item into a field of n characters in scientific notation, with maximum precision allowed within the field width (n must be at least 6).
E(n,m)	Writes a data item into a field of n characters, with m decimal places of precision. The number is written in scientific notation with one digit to the left of the decimal point.
F(n)	Write a numeric value in a field of n digits, with no fractional part. The value is rounded before it is printed.
F(n,m)	Write a numeric value in a field of n digits, with m fractional digits. The value is rounded in the m+1 fractional position before printing.
LINE(n)	Moves to line n in the output before writing the next data item.
PAGE	Performs a page eject for print files.
R(fmt)	Specifies a remote format. In PL/I-8Ø, if the R format appears, it must be the only format item in fl.
SKIP	Skips to the next output line before writing the next data item.
SKIP(n)	Skips n lines in the output before printing the next data item.
TAB(n)	Moves to the nth tab position in the output line, where tabs are defined at multiples of eight columns.
X(n)	Inserts n blank characters into the output stream before writing the next data item.

Unlike the PUT LIST statement, data fields are written to the end of the line, without a "pre-fit" test. If the entire field cannot be written, the portion which does fit on the current line is sent to the output, a carriage return line feed is written, and the remainder of the field is written on the following line. COLUMN, LINE, PAGE, SKIP, TAB, and X format items which occur at the end of the format list have no effect after the entire data list has been written. Valid PUT EDIT statements are shown below:

```
        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. The PL/I-80 GET EDIT Statement.

The GET EDIT statement is similar to the GET LIST statement, except data is read from particular fields in the input stream. While GET LIST is more appropriate for console input, GET EDIT is often used to read data that has been written by another program. The form of the GET EDIT statement is

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

where the FILE and SKIP options are identical to the GET LIST statement. The EDIT option specifies a list of target variables to receive the data which, again, matches the data specification of the GET LIST. The EDIT option is followed by a format list, consisting of a sequence of format items defined as follows:

A	Read the next alphanumeric field up to the next carriage return, line feed or end of file (not standard PL/I).
A(n)	Read the next n characters as an alpha-numeric field.
B(n)	Read the next n characters and inter-pret as a bit string. The field must be all blank, or contain a sequence of 1's and 0's with right or left blank pad.
B1(n)	Interpreted in the same manner as B.
B2(n)	Similar to B1(n), but the sequence must contain digits selected from 0,1,2,3.

B3(n)	Similar to B1(n), but the sequence must contain digits from Ø through 7.
B4(n)	Similar to B1(n), but the sequence must contain digits from Ø to 9, and A to F.
COLUMN(n)	Move to column position n in input, may require a read past end of line.
E(n)	Read the next n fields as a numeric value, with possible leading and trailing blanks. The number must be a properly formed constant, but may take a simple signed or unsigned integer form, a number with a decimal fraction, or a number in scientific notation.
E(n,m)	Equivalent to E(n), the scale factor m is ignored on input.
F(n)	Equivalent to the E(n) form shown above.
F(n,m)	Equivalent to E(n), except that the decimal point is assumed m positions to the left of the least significant digit if there is no decimal point in the field.
LINE(n)	Moves to line n in the input before reading the next field.
R(fmt)	Specifies a remote format. In PL/I-8Ø, if the R format item appears, it must be the only format item in fl.
SKIP	Clears the current input line before reading additional data items.
SKIP(n)	Clears the current input line, and moves n-1 additional lines through the input before reading additional data.
X(n)	Moves n characters through the input stream before reading the next field.

The carriage return line feed sequences are ignored in the A(n), B(n), B1(n), B2(n), B3(n), E(n), E(n,m), F(n), and F(n,m) formats: when encountered, the next input line is read to obtain the remaining characters of the field. Each format item can be preceded by a repetition count, and groups of items can be enclosed within parentheses and separated by commas with a preceding repetition count. The repetition count r must be a positive constant value, not exceeding 254, and is equivalent to writing the same format r times.

In processing the GET EDIT statement, the PL/I-80 I/O system keeps track of the next data item to read, along with the next format item to use in input processing. As each data item is read, the next successive format item in the list is selected, repeating each item if a repetition count is present, until the data list is exhausted. Format items which remain in the list are left unprocessed: in particular, the control format items which remain (COLUMN, LINE, SKIP, and X) have no effect when the data list is exhausted. If the list of format items is exhausted before all data items have been read, the format list is restarted at the beginning. (See the Reference Manual for exact details on EDIT directed input operations.) The following examples show a number of valid GET EDIT statements.

```
get edit(hours,pay) (f(4),f(5,2));
get file(employee) (hours,pay) (r(fmt1));
get edit((a(i) do i = 1 to 10) (8e(6),skip);
get skip(2) edit(u,v,w) (b3(4),x(4),2a(5));
get file(input) edit((mat(i) do i=1 to mat(1)))
  (line(3),4(10(f(4),x(2),2f(4),skip(2)),skip));
```

4.6. The PL/I-80 FORMAT Statement.

The FORMAT statement allows a list of format items to be shared among various GET and PUT EDIT statements. The form is

```
fmtname:
    FORMAT (fl)
```

where fl denotes a list of format items, as shown in the GET and PUT EDIT statements above. The list of format items is then referenced using the R format within the GET or PUT format list. Again, note that PL/I-80 restricts the use of the remote format: if it appears in a GET or PUT EDIT, it must be the only format item in the list. Valid FORMAT statements are shown below, and referenced in the examples of the previous two sections.

```
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. The PL/I-80 WRITE Statement.

The WRITE statement is used primarily to transmit data from memory to an external file without conversion to character form. The basic form of the WRITE statement appears as follows:

```
WRITE
    FILE(f)
```

FROM(x)

where both the FILE and FROM elements must be present, but may appear in any order, f is a file reference, and x is a scalar or connected aggregate data type. The file f is opened automatically as:

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

If already open, file attributes of f must not conflict with these default values. Thus, for example, a KEYED file is allowed (since this only implies fixed length records), but DIRECT is not.

If file f has been previously opened with the KEYED attribute, then each record length is fixed, and determined by the ENV(F(i)) option given in the OPEN statement. Otherwise, the file is assumed to contain variable length records, where each record length is determined by the aggregate data size of x. Given a KEYED file with records of length i, each record is written from x for a maximum of i bytes. The record is padded with zeroes if the length of x is less than i. If f is not KEYED, then the record length is exactly the size of x.

An alternative form of the write statement is:

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

where the elements can appear in any order. If the file f is not already open, the default statement

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

occurs before the file is accessed. Note that the DIRECT attribute implies a KEYED file (which, in turn, implies a RECORD file). The file may have previously been opened with either OUTPUT, the INPUT, or UPDATE attributes, but must have the DIRECT attribute. Recall that in the case of OUTPUT, the file is deleted, if it exists, and a new file is created. If the file is marked as INPUT, then the file must already exist. An UPDATE file is opened for access if it exists, and created if it does not exist.

When the KEYFROM option is included, each record is accessed through a key k which, in PL/I-80, is a FIXED expression providing the relative record number of the record to write, based upon the fixed length of each record. The lowest key value is k = 0, while the maximum key value depends upon the record length obtained from the ENV(f(j)) attribute: if j is the fixed record size, and j' is the rounded record size, then the largest key times j' cannot exceed the capacity of the drive.

A special form of the WRITE statement is supported by PL/I-80 for processing variable-length STREAM data, delimited by carriage return line feed sequences. Given a file f with the STREAM OUTPUT

attributes, and a varying character string v, the statement:

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

writes the characters of v to the STREAM file f, including any embedded control characters. The form:

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

writes the string value v to the standard output device, and is equivalent to:

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

In order to facilitate control character processing, PL/I-80 allows control characters to be entered into string constants. In general, the character "^" within a string constant denotes that a control character follows. The occurrence of a double "^" within a string, however, is reduced to a single "^" character. The effect of a leading "^" is to mask the high-order four bits of the character which follows to zero. Thus, the sequence "^m" within a string constant is converted to a carriage return. Embedded control characters are shown in the examples given in sections which follow.

To summarize, let f be a file, x be a scalar or connected aggregate data type, v be a varying character string, and k be a fixed binary expression. The following forms show the required file attributes in each case:

```
write file(f) from(x);
  SEQUENTIAL OUTPUT (Optionally 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 (automatically SYSPRINT)
```

4.8. The PL/I-80 READ Statement.

The READ statement is used, with one exception, to read fixed or variable length RECORD files without conversion from character form. That is, data is transmitted from an external file to data elements in memory, where the external file is assumed to contain binary data. It is the responsibility of the programmer to interpret the meaning of the data which is transmitted.

The form of the basic READ statement is:

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

where f is a file reference, and x is a connected aggregate or scalar data type (e.g., a structures, array, or simple variable). Both the FILE and INTO elements must be present, but may occur in any order. If the file f is not already open, then it is automatically opened as:

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

As in the case of the WRITE statement, if f is already open, then its attributes must not conflict with those shown above.

If the file has been opened with the KEYED attribute, then each record is assumed to be of fixed length, as defined in the ENV(f(i)) attribute. Otherwise, the record length is assumed to be variable, depending upon the size of the target data x specified in the INTO element. Given a KEYED file, if the record length i is greater than the size of x, all remaining bytes in the record are ignored. If the record length is less than the size of x, then only i bytes are read into x. If the file is not KEYED, then the number of bytes read is exactly the size of x.

The keys for a particular file can be optionally extracted as the file is read sequentially using the form:

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

where the elements may be specified in any order. The effect of this form is exactly the same as the previous READ statement, except that the key value for the record is stored into the FIXED BINARY variable reference denoted by k. Note, however, that in order to read the key as well as the data, the file must be KEYED. Thus, the automatic OPEN statement which applies to this second form of the READ is:

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

If a previous open has occurred, the attributes of f must not conflict with these default attributes. Note, in particular, that KEYED must be present, and DIRECT is not allowed since the KEYTO option simply extracts the key, but does not specify the keyed record to read. This form of the READ statement is most often used in the situation where the file is first read sequentially to determine the keys, and later accessed directly to read, write, or update specific records within the file.

The third form of the READ statement specifies the keyed record to read:

```
READ
```

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

where the elements may be specified in any order. If the file is not already open, the default OPEN shown below is executed:

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

If the file is already open, the open attributes must not conflict with these default values, except that the file may have been opened with the UPDATE attribute. Note that the DIRECT attribute also implies that the file is KEYED.

The effect of this READ statement is to directly access the record which has the key value k. Since the file is KEYED, the record length must be fixed, as defined by the ENV(f(i)) attribute, and data transfer takes place according to the above rules for fixed length records.

A special form of the READ statement is allowed in PL/I-80 to process variable length STREAM INPUT files:

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

and

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

where v is a varying character string, and f is an ASCII data file (or character device) with records delimited by carriage return line feed sequences. If FILE(f) is not specified, then the standard output file SYSIN is assumed. If f is not open, it is opened with the statement:

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

The effect of this statement is to read data from the file until either the maximum length of v is reached, or a line feed character is read. The length value of v is set to the number of characters processed, including control characters, which specifically includes the carriage return and line feed characters. If the standard SYSIN file is attached to the console, then a maximum of 80 characters is read before an automatic carriage return and line feed is issued.

In summary, if f is a file, x is a scalar or aggregate data reference, v is a varying character string, and k is a fixed binary key, the following forms show the required file attributes:

```
read file(f) into(x);
  SEQUENTIAL INPUT (Optionally KEYED) RECORD

read file(f) into(x) keyto(k);
  SEQUENTIAL INPUT KEYED RECORD

read file(f) into(x) key(k);
```

```
  DIRECT INPUT or DIRECT UPDATE

read file(f) into(v);
  STREAM INPUT

read into(v);
  STREAM INPUT (Automatically SYSIN)
```

The following section contains a number of examples which show the use of the various PL/I-80 I/O statements.

5. PL/I-80 PROGRAMMING EXAMPLES

The purpose of this section is to introduce the various PL/I-80 I/O statements through several sample programs. The programs themselves are simple in nature, but illustrate the basic techniques for stream and record processing.

5.1. Polynomial Evaluation.

Two programs for polynomial evaluation are shown in Figures 5-1 and 5-2. Each program interacts with the system console by reading three values: x, y, and z, which are then used in the evaluation of

$$p(x,y,z) = x^2 + 2y + z$$

The programs have one main loop, bounded by a single DO-END group. On each successive loop, the values of x, y, and z are read from the standard SYSIN (console) file, and used in the polynomial evaluation. The value produced by p(x,y,z) is written to the SYSPRINT file (again, the console) in the middle of the loop. The STOP statement is executed if all input values are zero, thus terminating the indefinite loop.

The console interaction is shown below the program listing in each Figure. Referring to Figure 5-1, note that the initial values for x, y, and z are 1.4, 2.3, and 5.67, respectively. The next input, however, takes the form

```
,4.5,,
```

which changes only the value of y. On this loop, the values of x, y, and z are 1.4, 4.5, and 5.67. The third input line changes y and z, while the fourth line changes only x.

These two programs illustrate a number of points which should be noted in passing. The "%replace" statement is used on line 6 to define the literal value of "true" as a bit string constant '1'b which is substituted by the compiler whenever the name "true" is encountered. In particular, the DO group beginning on line 12 is interpreted by the compiler as

```
do while('1'b);
...
end;
```

which loops until the contained STOP statement is executed.

The only essential difference between the programs of Figures 5-1 and 5-2 is that the first uses float binary data items, while the second program defines the variables as fixed decimal types. Although the float binary computations execute significantly faster that their fixed decimal equivalents, the binary computations are carried out to only about 7-1/2 decimal places and involve truncation errors which are inherent in floating binary computations.

```
 1 a 0000 poly:
 2 a 0006     procedure options(main);
 3 a 0006
 4 a 0006     /* evaluate polynomial */
 5 a 0006
 6 c 0006     %replace
 7 c 0006         false by '0'b,
 8 c 0006         true  by '1'b;
 9 c 0006     dcl
10 c 0006         (x,y,z) float binary;
11 c 0006
12 c 0006         do while(true);
13 c 0006         put skip(2) list('Type x,y,z: ');
14 c 0022         get list(x,y,z);
15 c 005B
16 c 005B         if x = 0 & y = 0 & z = 0 then
17 c 008E             stop;
18 c 0091
19 c 0091         put skip list('       2');
20 c 00AD         put skip list('      x  + 2y + z =',p(x,y,z));
21 c 00DA         end;
22 c 00DA
23 c 00DA     p:
24 c 00DA         proc (x,y,z) returns (float binary);
25 e 00DA         dcl
26 e 00E7             (x,y,z) float binary;
27 e 00E7         return (x * x + 2 * y + z);
28 c 0109         end p;
29 c 0109
30 a 0109     end poly;


Type x,y,z: 1.4, 2.3,5.67

       2
      x  + 2y + z =  1.223000E+01

Type x,y,z: ,4.5,,

       2
      x  + 2y + z =  1.663000E+01

Type x,y,z: ,.6e-3, 7

       2
      x  + 2y + z =  0.896119E+01

Type x,y,z: 2.3,,,

       2
      x  + 2y + z =  1.229119E+01

Type x,y,z: 0,0,0
```

Figure 5-1. Floating Point Polynomial Evaluation.

```
 1 a 0000 poly:
 2 a 0006     procedure options(main);
 3 a 0006
 4 a 0006     /* evaluate polynomial */
 5 a 0006
 6 c 0006     %replace
 7 c 0006         true by '1'b;
 8 c 0006     dcl
 9 c 0006         (x,y,z) fixed decimal(15,4);
10 c 0006
11 c 0006         do while(true);
12 c 0006         put skip(2) list('Type x,y,z: ');
13 c 0022         get list(x,y,z);
14 c 0067
15 c 0067         if x = 0 & y = 0 & z = 0 then
16 c 00B2             stop;
17 c 00B5
18 c 00B5         put skip list('       2');
19 c 00D1         put skip list('      x  + 2y + z =',p(x,y,z));
20 c 0100         end;
21 c 0100
22 c 0100     p:
23 c 0100         proc (x,y,z) returns (fixed decimal(15,4));
24 e 0100         dcl
25 e 010D             (x,y,z) fixed decimal(15,4);
26 e 010D        return (x * x + 2 * y + z);
27 c 0153        end p;
28 c 0153
29 a 0153     end poly;

Type x,y,z: 1.4, 2.3, 5.67

       2
      x  + 2y + z =            12.2300

Type x,y,z: , .0006, 7

       2
      x  + 2y + z =             8.9612

Type x,y,z: 723.445, 80.54, 0

       2
      x  + 2y + z =        523533.7480

Type x,y,z: 0,0,,

End of Execution
```

Figure 5-2. Fixed Decimal Polynomial Evaluation.

5.2. The File Copy Program.

A general purpose file-to-file copy program is shown in figure 5-3. The program defines two file constants on line 4, called input and output. The files are opened on lines 6 and 9, followed by a continuous loop which reads data from the input file, and copies the line to the output file.

Both OPEN statements define STREAM files containing ASCII data, with internal buffers of 8192 characters each. The first OPEN statement has the default value of INPUT, while the second file explicitly defines an OUTPUT file (otherwise, it would also be considered an INPUT file). The TITLE options connect the internal file names to external CP/M devices and files: the first file name is taken as the first default name typed in the command tail when the copy program executes (denoted by $1.$1). Similarly, the second file name is taken from the second default name on the command line (denoted by $2.$2). The input file must exist, while the output file is erased, if it exists, and re-created.

This particular program shows the special use of READ and WRITE to process STREAM files: line 15 reads a STREAM file into "buff" which is a varying character string. The line of input, up to the next line feed, is read into buff, and the length of buff is set to the amount of data which was read, including the line feed character. The next statement performs the opposite action: a WRITE statement sends data to a STREAM file from buff, which is a varying character string. The output file receives all characters from the first position through the LENGTH(buff).

The program terminates by reading through the input file until the STREAM end of file (control-z) is reached. At this point, the END OF FILE condition is raised, and the program stops. All files are automatically closed (and internal buffers are emptied), preserving the newly created output file.

A sample execution of the copy program is shown in Figure 5-4, using the command line

```
copy copy.pli $con
```

In this case, the input file is taken as "copy.pli" (which just happens to be the original source file), while the output file is the system console. The result is that the copy.pli program is listed at the operator's terminal. The command

```
copy a:x.dat c:u.new
```

would, for example, copy the file x.dat from drive "a" to the new file u.new on drive "c".

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

L: List Source Program

   NO ERROR(S) IN PASS 1

   NO ERROR(S) IN PASS 2


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

   1 a 0000 copy:
   2 a 0006      proc options(main);
   3 c 0006      dcl
   4 c 0006          (input,output) file;
   5 c 0006
   6 c 0006      open file (input) stream env(b(8192))
   7 c 0023          title('$1.$1');
   8 c 0023
   9 c 0023      open file (output) stream output env(b(8192))
  10 c 0040          title('$2.$2');
  11 c 0040      dcl
  12 c 0040          buff char(254) varying;
  13 c 0040
  14 c 0040          do while('1'b);
  15 c 0040          read  file (input)  into (buff);
  16 c 0058          write file (output) from (buff);
  17 c 0073          end;
  18 a 0073      end copy;

CODE SIZE = 0073
DATA AREA = 0109
```

Figure 5-3. File to File Copy Utility.

```
A>b:copy copy.pli $con
copy:
    proc options(main);
    dcl
        (input,output) file;

    open file (input) stream env(b(8192))
        title('$1.$1');

    open file (output) stream output env(b(8192))
        title('$2.$2');
    dcl
        buff char(254) varying;

        do while('1'b);
        read  file (input)  into (buff);
        write file (output) from (buff);
        end;
    end copy;

END OF FILE (3), File: INPUT=COPY.PLI
Traceback: Ø44B Ø3AF Ø155
End of Execution
```

Figure 5-4. Execution of the File Copy Utility.

5.3. Name and Address File Processing.

Two programs are shown in Figures 5-5 and 5-7, called "create" and "retrieve," which manage a simple name and address file. The create program produces a STREAM file containing individual names and addresses which are subsequently accessed by the retrieve program.

The create program, shown in Figure 5-5, contains a structure which defines the name, address, city, state, zip code, and phone number format. The console is prompted for each data input, and each successive entry is written to the output file until the name "EOF" is entered by the operator.

The record structure is read and merged with the source program from a separate file, using a "%include" statement which is a statement in the source file, but is not shown in the listing. The presence of a "%include" statement is indicated by the "+" symbols to the right of the source line number. The source program, in fact, appears as follows:

```
create:
    procedure options(main);
    /* create name and address file */

%include 'record.dcl';

    %replace
        true by '1'b,
    ...
```

The file given in the "%include" statement can be any valid CP/M file name, and is copied from the file at the point of the "%include" statement.

In this particular program, the input file name is entered by the operator on line 25 and listed in the TITLE option in the OPEN statement on line 27. The PRINT attribute is not specified in the OPEN statement, and thus the output file is in a form suitable for later input using a GET LIST statement.

The console interaction and subsequent program output is shown in Figure 5-6. In this case, the output file is specified by the operator as "names.dat" in the first input line. Recall that LIST input is delimited by blanks and commas, unless the delimiters are included within a quoted string. Thus, the input line

```
'Aaron Appleby
```

is taken as a single string value with the implied closing quote automatically inserted at the end of the line. The second entry includes the three input values

```
Don't-Know,  'Won''t Know', 99999
```

which are assigned to the variables city, street, and state. The

```
 1 a 0000 create:
 2 a 0006     procedure options(main);
 3 a 0006     /* create name and address file */
 4 a 0006
 5+c 0006     dcl
 6+c 0006         1 record,
 7+c 0006           2 name  character(30) varying,
 8+c 0006           2 addr  character(30) varying,
 9+c 0006           2 city  character(20) varying,
10+c 0006           2 state character(10) varying,
11+c 0006           2 zip   fixed decimal(6),
12+c 0006           2 phone character(12) varying;
13 c 0006
14 c 0006     %replace
15 c 0006         true by '1'b,
16 c 0006         false by '0'b;
17 c 0006     dcl
18 c 0006         output file;
19 c 0006     dcl
20 c 0006         filename character(14) varying;
21 c 0006     dcl
22 c 0006         eofile bit(1) static initial(false);
23 c 0006
24 c 0006     put list ('Name and Address Creation Program, File Name: ');
25 c 001D     get list (filename);
26 c 0037
27 c 0037     open file(output) stream output title(filename);
28 c 0051
29 c 0051         do while (^eofile);
30 c 0058         put skip(3) list('Name: ');
31 c 0074         get list(name);
32 c 008E         eofile = (name = 'EOF');
33 c 00A0         if ^eofile then
34 c 00A7             do;
35 c 00A7             /* write prompt strings to console */
36 c 00A7             put list('Address: ');
37 c 00BE             get list(addr);
38 c 00D8             put list('City, State, Zip: ');
39 c 00EF             get list(city, state, zip);
40 c 012A             put list('Phone: ');
41 c 0141             get list(phone);
42 c 015B
43 c 015B             /* data in memory, write to output file */
44 c 015B             put file(output)
45 c 01A9                 list(name,addr,city,state,zip,phone);
46 c 01A9             put file(output) skip;
47 c 01C0             end;
48 c 01C0         end;
49 c 01C0     put file(output) skip list('EOF');
50 c 01DF     put file(output) skip;
51 a 01F3     end create;
```

Figure 5-5. File CREATE Program.

```
A>b:create
Name and Address Creation Program, File Name: names.dat


Name: 'Aaron Appleby
Address: '32 West East St.
City, State, Zip: Claustrophobia, Ca., 92995
Phone: 123-4567


Name: 'Bugsy Burton
Address: 'Good Question
City, State, Zip: Don't-Know,'Won''t Know', 99999
Phone: 333-9999


Name: 'Zwiggy Zittsmacher
Address: 2323-W-2nd#201
City, State, Zip: Lincoln, Wa., 98177
Phone: 345-5432


Name: EOF

End of Execution
```

Figure 5-6a. Interaction with the CREATE Program.

```
A>type names.dat
'Aaron Appleby' '32 West East St.' 'Claustrophobia' 'Ca.'     92995 '12
'Bugsy Burton' 'Good Question' 'Don''t-Know' 'Won''t Know'     99999 '3
'Zwiggy Zittsmacher' '2323-W-2nd#201' 'Lincoln' 'Wa.'     98177 '345-54

'EOF'
```

Figure 5-6b. Output from the CREATE Program.
(Note: output listing is truncated on right.)

first value does not begin with a quote, so the data item is scanned until the next blank, comma, or end of line occurs. The second data item begins with a quote, causing all input through the trailing balanced quote to be consumed, with all embedded double quotes reduced to a single quote. The last value, 99999, is assigned to a decimal number, and must contain only numeric data.

The CP/M TYPE command is used, following program execution, to display the STREAM file which was created. The (truncated) output shows the quoted strings which were produced for each input entry.

The retrieve program shown in Figure 5-7 reads the previously created file and displays the name and address data according to an operator request. The "record.dcl" structure is included in the retrieve program, matching the create program discussed above.

In general, the retrieve program works as follows: the main loop between 30 and 60 reads two string values corresponding to the lowest and highest names to print on each iteration. The embedded loop between 41 and 58 reads the entire input file and lists only those names between the lower and upper bounds.

Similar to the create program, retrieve reads the name of the source file from the console, but opens and closes this source file each time a console retrieval request occurs. The OPEN statement on line 38 sets-up the input file, with internal buffer size of 1024 bytes. After the file has been processed, the CLOSE statement on line 59 is executed, and all internal buffers are reclaimed. As a result, the input file is effectively set back to the beginning on each retrieval request.

Program interaction is shown in Figure 5-8. Again, the input file is given as "names.dat" which is assumed to exist on the disk in the form produced by "create." The input values

```
B,D
```

set lower to 'B' and upper to 'D' which causes retrieve to list only 'Bugsy Burton'. The second input line consists only of a comma pair, leaving the lower bound as the sequence 'AAA...A' while the upper bound remains at 'zzz...z'. These two bounds include all of the alphabetic range, resulting in a display of the entire list of names and addresses.

It should be noted that the sysprint file was explicitly opened with the PRINT attribute on line 26 to illustrate the form of the resulting output. This statement is, however, superfluous since the PUT statement on line 27 would have provided the same information.

5.4. An Information Management System.

The example of this section provides the model for an information management system consisting of a set of four programs. The four programs work together to manage a file of employee names, addresses, wage schedules, and wage reporting mechanisms. In general,

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


L: List Source Program

%include 'record.dcl';
   NO ERROR(S) IN PASS 1

   NO ERROR(S) IN PASS 2


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

   1 a 0000 retrieve:
   2 a 0006     procedure options(main);
   3 a 0006     /* name and address retrieval program */
   4 a 0006
   5+c 0006     dcl
   6+c 0006         1 record,
   7+c 0006           2 name  character(30) varying,
   8+c 0006           2 addr  character(30) varying,
   9+c 0006           2 city  character(20) varying,
  10+c 0006           2 state character(10) varying,
  11+c 0006           2 zip   fixed decimal(6),
  12+c 0006           2 phone character(12) varying;
  13 c 0006
  14 c 0006     %replace
  15 c 0006         true  by '1'b,
  16 c 0006         false by '0'b;
  17 c 0006
  18 c 0006     dcl
  19 c 0006         (sysprint, input) file;
  20 c 0006
  21 c 0006     dcl
  22 c 0006         filename character(14) varying,
  23 c 0006         (lower, upper) character(30) varying,
  24 c 0006         eofile bit(1);
  25 c 0006
  26 c 0006     open file(sysprint) print title('$con');
  27 c 0022     put list('Name and Address Retrieval, File Name: ');
  28 c 0039     get list(filename);
```

Figure 5-7a. RETRIEVE Program Listing, Part A.

```
 29 c 0053
 30 c 0053          do while(true);
 31 c 0053          lower = 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
 32 c 005F          upper = 'zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz';
 33 c 006B          put skip(2) list('Type Lower, Upper Bounds: ');
 34 c 0087          get list(lower,upper);
 35 c 00AF          if lower = 'EOF' then
 36 c 00BD              stop;
 37 c 00C0
 38 c 00C0          open file(input) stream input environment(b(1024))
 39 c 00DB              title(filename);
 40 c 00DB          eofile = false;
 41 c 00E0              do while (^eofile);
 42 c 00E7              get file(input) list(name);
 43 c 0104              eofile = (name = 'EOF');
 44 c 0116              if ^eofile then
 45 c 011D                  do;
 46 c 011D                  get file(input)
 47 c 0177                      list(addr,city,state,zip,phone);
 48 c 0177                  if name >= lower & name <= upper then
 49 c 0194                      do;
 50 c 0194                      put page skip(3)
 51 c 01B1                          list(name);
 52 c 01B1                      put skip list(addr);
 53 c 01CB                      put skip list(city,state);
 54 c 01EE                      put skip list(zip);
 55 c 0211                      put skip list(phone);
 56 c 022E                      end;
 57 c 022E                  end;
 58 c 022E              end;
 59 c 022E          close file(input);
 60 c 0237          end;
 61 a 0237      end retrieve;

CODE SIZE = 0237
DATA AREA = 0141
```

Figure 5-7b. RETRIEVE Program Listing Part B.

```
A>b:retrieve
Name and Address Retrieval, File Name: names.dat


Type Lower, Upper Bounds: B,D




Bugsy Burton
Good Question
Don't-Know Won't Know
    99999
333-9999

Type Lower, Upper Bounds: ,,




Aaron Appleby
32 West East St.
Claustrophobia Ca.
    92995
123-4567



Bugsy Burton
Good Question
Don't-Know Won't Know
    99999
333-9999




Zwiggy Zittsmacher
2323-W-2nd#201
Lincoln Wa.
    98177
345-5432

Type Lower, Upper Bounds: EOF,,

End of Execution
```

Figure 5-8. Interaction with the RETRIEVE Program.

a file is initially prepared using a data entry program, called enter, which establishes the data base. A second program, called keypr, reads this data base and prepares an index file for direct access to this data base for information and update. A third program, called update, interacts with the console to allow access to the data base. Finally, the report program reads the data base to produce a final report. Although these programs are, themselves, simplistic in nature, they contain all the elements of a more advanced data management system, thus demonstrating the power of the PL/I-80 programming system, while providing the basis for custom programs.

The "enter" program interacts with the operator's console and constructs the initial data base, as shown in Figure 5-9. The basic input loop appears between lines 36 and 49 where the operator is prompted for an employee name, age, and hourly wage. The "employee" data structure is filled with this variable information, and, for simplicity of the example, the address fields are filled with default values on line 44. Operator input is terminated when the name "EOF" is entered.

The employee record names a number of fields which total 84 bytes in length (the $S compiling parameter can be used to verify this value). For expansion, a record size of 100 bytes is specified in the OPEN statement on line 33, where each record of the "emp" file holds exactly one employee data structure.

The OPEN statement names "emp" as a KEYED file, which makes each record a fixed size as specified in the environment option. In this case the fixed size is 100 bytes, but is internally rounded to 128 bytes. The buffer size is also given in the OPEN statement as 8000 bytes, again rounded up to 8192. Each employee record is filled from the console and written to the employee file named in the command line, with the file type "EMP" given on line 34.

The WRITE statement itself is included in a separate subroutine, named WRITE, which is called from lines 41 and 48, and is defined starting at 51. The WRITE statement was placed into a separate subroutine to reduce program size.

Interaction with the enter program is given in Figure 5-10. Each employee record is entered including the name, age, and hourly wage. The program terminates when the EOF entry is typed, and the file "plant1.emp" is closed and recorded on the disk.

The "keypr" program constructs a key file by reading the data base file created by "enter." The key file is a sequence of employee names, followed by the key corresponding to that name. In this particular case, the key file is written in STREAM mode so that it can be displayed at the console. Referring to Figure 5-11, the "EMP" employee file is OPENed on line 16 with the KEYED attribute, where each record length is given as 100 bytes, with a buffer size of 10000 bytes. The "keys" key file is then OPENed in STREAM mode, with with LINESIZE(60) and a TITLE option which appends "KEY" as the file type.

The keypr program reads successive records on line 23, extracts

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

L: List Source Program

   NO ERROR(S) IN PASS 1

   NO ERROR(S) IN PASS 2


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

   1 a 0000 enter:
   2 a 0006     proc options(main);
   3 a 0006
   4 c 0006     %replace
   5 c 0006         true  by '1'b,
   6 c 0006         false by '0'b;
   7 c 0006
   8 c 0006     dcl
   9 c 0006         1 employee static,
  10 c 0006           2 name     char(30) varying,
  11 c 0006           2 addr,
  12 c 0006             3 street char(30) varying,
  13 c 0006             3 city   char(10) varying,
  14 c 0006             3 state  char(7)  varying,
  15 c 0006             3 zip    fixed dec(5),
  16 c 0006           2 age      fixed dec(3),
  17 c 0006           2 wage     fixed dec(5,2),
  18 c 0006           2 hours    fixed dec(5,1);
  19 c 0006
  20 c 0006     dcl
  21 c 0006          1 default static,
  22 c 0006            2 street char (30) varying
  23 c 0006              initial('(no street)'),
  24 c 0006            2 city   char(10) varying
  25 c 0006              initial('(no city)'),
  26 c 0006            2 state  char(7)  varying
  27 c 0006              initial('(no st)'),
  28 c 0006            2 zip    fixed dec(5)
  29 c 0006              initial(00000);
  30 c 0006     dcl
  31 c 0006         emp file;
  32 c 0006
  33 c 0006     open file(emp) keyed output environment(f(100),b(8000))
  34 c 0026         title ('$1.EMP');
  35 c 0026
```

Figure 5-9a. ENTER Program Listing Part A.

```
36 c 0026          do while(true);
37 c 0026          put list('Employee: ');
38 c 003D          get list(name);
39 c 0057          if name = 'EOF' then
40 c 0066              do;
41 c 0066              call write();
42 c 0069              stop;
43 c 006C              end;
44 c 006C          addr = default;
45 c 0078          put list ('  Age, Wage: ');
46 c 008F          get list (age,wage);
47 c 00C1          hours = 0;
48 c 00D1          call write();
49 c 00D7          end;
50 c 00D7
51 c 00D7      write:
52 c 00D7          procedure;
53 e 00D7          write file(emp) from(employee);
54 c 00F0          end write;
55 a 00F0      end enter;
```

Figure 5-9b. ENTER Program Listing Part B.

```
A>b:enter plant1
Employee: Abercrombie
  Age, Wage: 25, 6.70
Employee: Fairweather
  Age, Wage: 32, 15.00
Employee: Eggbert
  Age, Wage: 45, 5.50
Employee: Willowander
  Age, Wage: 27,,
Employee: Millywatz
  Age, Wage: ,7.20
Employee: Quagmire, 23, 4.30
  Age, Wage:  Employee: EOF

End of Execution
```

Figure 5-10. Interaction with the ENTER Program.

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

L: List Source Program

   NO ERROR(S) IN PASS 1

   NO ERROR(S) IN PASS 2


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

    1 a 0000 keypr:
    2 a 0006     proc options(main);
    3 a 0006
    4 a 0006     /* create key from employee file */
    5 a 0006
    6 c 0006     dcl
    7 c 0006         1 employee static,
    8 c 0006           2 name char(30) varying;
    9 c 0006
   10 c 0006     dcl
   11 c 0006         (input, keys) file;
   12 c 0006
   13 c 0006     dcl
   14 c 0006         k fixed;
   15 c 0006
   16 c 0006     open title('$1.emp') keyed
   17 c 0026       env(f(100),b(10000)) file(input);
   18 c 0026
   19 c 0026     open file (keys) stream output
   20 c 0044         linesize (60) title('$1.key');
   21 c 0044
   22 c 0044         do while('1');
   23 c 0044         read file(input) into(employee) keyto(k);
   24 c 0062         put skip list(k,name);
   25 c 0087         put file(keys) list(name,k);
   26 c 00AA         if name = 'EOF' then
   27 c 00B9             stop;
   28 c 00BF         end;
   29 a 00BF     end keypr;

CODE SIZE = 00BF
DATA AREA = 0030
```

Figure 5-11. Listing of the KEYPR Program.

the key with the KEYTO option, and writes the name and key to both the console and to the key file. The sample interaction of Figure 5-12 shows the output from keypr using the "plantl.emp" data base. Note that the key values extracted by the READ statement are just the relative record number corresponding to the record's position in the file. Following program execution, the CP/M TYPE command is used to display the actual contents of the "plantl.key" file.

The third program, shown in Figure 5-13, allows access to the data base created by enter and indexed through the file created by keypr. The update program first reads the STREAM key file into a vector which cross-references the employee name with the corresponding key value in the data base. The dimensioned structure which holds these cross-reference values is defined on line 17, and filled between lines 30 and 33.

The main program loop between lines 30 and 55 accesses the individual records of the employee file OPENed on line 25. The OPEN statement marks this file as DIRECT, which allows both READ and WRITE operations where the individual records are identified by a key value. The operator enters an employee name as "matchname" which will be directly accessed in the data base.

The direct access is accomplished by searching the list of names read from the key file, between lines 40 and 54. If a match is found, the employee record is brought into memory from the employee file through the READ with KEY statement on line 43. Various fields are then displayed and updated from the console, and the record is rewritten to the data base using the WRITE with KEYFROM statement on line 51. Execution terminates when the operator enters the name "EOF" as an input value.

Three successive update sessions are shown in Figure 5-14. The employee name is entered by the operator, the record is accessed and displayed, and the fields are optionally updated. In particular, note that the GET statement is quite useful here: if the operator wishes to change a value then the new value is typed in the field position, otherwise a comma delimiter leaves the field unchanged. During these three interactions, various addresses and work times are updated.

The final report program uses the updated employee file to produce a list of employees along with their paycheck values according to their hourly wage and number of hours worked, as shown in Figure 5-15. The report program again accesses the "EMP" file, but reads the file sequentially to produce the desired output information. The main loop between lines 37 and 53 reads each successive employee record and constructs a title line of the form:

[name]

followed by a dollar amount. For illustration, the STREAM oriented form of the WRITE statement is again used to produce the output line. Note, however, that the embedded control-m (^m) and control-j (^j) characters are included at the end of "buff" to cause a carriage return and line feed when the buffer is written. The report program

```
A>b:keyfile plantl

        Ø Abercrombie
        1 Fairweather
        2 Eggbert
        3 Willowander
        4 Millywatz
        5 Quagmire
        6 EOF
End of Execution
A>type plantl.key
'Abercrombie'          Ø 'Fairweather'          1 'Eggbert'
        2 'Willowander'          3 'Millywatz'          4
'Quagmire'          5 'EOF'          6
```

Figure 5-12. Interaction with the KEYPR Program.

```
PL/I-8Ø V1.Ø, COMPILATION OF: UPDATE

L: List Source Program

   NO ERROR(S) IN PASS 1

   NO ERROR(S) IN PASS 2


PL/I-8Ø V1.Ø, COMPILATION OF: UPDATE

   1 a ØØØØ update:
   2 a ØØØ6     proc options(main);
   3 c ØØØ6     dcl
   4 c ØØØ6         1 employee static,
   5 c ØØØ6           2 name     char(3Ø) var,
   6 c ØØØ6           2 addr,
   7 c ØØØ6             3 street char(3Ø) var,
   8 c ØØØ6             3 city   char(1Ø) var,
   9 c ØØØ6             3 state  char(7)  var,
  1Ø c ØØØ6             3 zip    fixed dec(5),
  11 c ØØØ6           2 age      fixed dec(3),
  12 c ØØØ6           2 wage     fixed dec(5,2),
  13 c ØØØ6           2 hours    fixed dec(5,1);
```

Figure 5-13a. Listing of the UPDATE Program Part A.

```
 14 c 0006      dcl
 15 c 0006          (emp, keys) file;
 16 c 0006      dcl
 17 c 0006          1 keylist (100),
 18 c 0006            2 keyname char(30) var,
 19 c 0006            2 keyval  fixed binary;
 20 c 0006      dcl
 21 c 0006          (i, endlist) fixed,
 22 c 0006          eolist bit(1) static initial('0'b),
 23 c 0006          matchname char(30) var;
 24 c 0006
 25 c 0006      open file(emp) update direct env(f(100))
 26 c 0025          title ('$1.EMP');
 27 c 0025
 28 c 0025      open file(keys) stream env(b(4000)) title('$1.key');
 29 c 0044
 30 c 0044          do i = 1 to 100 while(^eolist);
 31 c 005C          get file(keys) list(keyname(i),keyval(i));
 32 c 00A4          eolist = keyname(i) = 'EOF';
 33 c 00CC          end;
 34 c 00CC
 35 c 00CC          do while('1'b);
 36 c 00CC          put skip list('Employee: ');
 37 c 00E8          get list(matchname);
 38 c 0102          if matchname = 'EOF' then
 39 c 0110              stop;
 40 c 0113              do i = 1 to 100;
 41 c 0125              if matchname = keyname(i) then
 42 c 013E                  do;
 43 c 013E                  read file(emp) into(employee)
 44 c 016C                      key(keyval(i));
 45 c 016C                  put skip list('Address: ',
 46 c 01B5                      street, city, state, zip);
 47 c 01B5                  put skip list('         ');
 48 c 01D1                  get list(street, city, state, zip);
 49 c 021A                  put list('Hours:',hours,': ');
 50 c 024E                  get list(hours);
 51 c 026D                  write file(emp) from (employee)
 52 c 02A8                      keyfrom(keyval(i));
 53 c 02A8                  end;
 54 c 02A8              end;
 55 c 02A8          end;
 56 a 02A8      end update;

CODE SIZE = 02A8
DATA AREA = 0D97
```

Figure 5-13b. Listing of the UPDATE Program Part B.

```
A>b:update plant1

Employee: Willowander

Address:  (no street) (no city) (no st)        Ø
         '123 E Willow', Williams, Ca., 98344
Hours:      Ø.Ø : 43.5

Employee: Quagmire

Address:  (no street) (no city) (no st)        Ø
         '321 W Q St', Quincy, Ca., 98222
Hours:      Ø.Ø : 38.6

Employee: EOF

End of Execution
A>b:update plant1

Employee: Quagmire

Address:  321 W Q St Quincy Ca.    98222
         ,,,,
Hours:     38.6 : 5Ø.5

Employee: Abercrombie

Address:  (no street) (no city) (no st)        Ø
         ,,,,
Hours:      Ø.Ø : 46.7

Employee: Fairweather

Address:  (no street) (no city) (no st)        Ø
         345-W-8th#3Ø4  Bloomberg  Wa. 33455
Hours:      Ø.Ø : 38.6

Employee: EOF

End of Execution
A>b:update plant1

Employee: Quagmire

Address:  321 W Q St Quincy Ca.    98222
         ,,,,
Hours:     5Ø.5 : 67.4

Employee: Millywatz

Address:  (no street) (no city) (no st)        Ø
         '345 6th St', Mipville, Ca. 98444
Hours:      Ø.Ø : 6Ø.2

Employee: EOF
```

Figure 5-14. Interaction with the UPDATE Program.

```
 1 a 0000 report:
 2 a 0006     procedure options(main);
 3 a 0006
 4 c 0006     dcl
 5 c 0006         1 employee static,
 6 c 0006           2 name     character(30) varying,
 7 c 0006           2 addr,
 8 c 0006             3 street character(30) varying,
 9 c 0006             3 city   character(10) varying,
10 c 0006             3 state  character(7)  varying,
11 c 0006             3 zip    fixed dec(5),
12 c 0006           2 age      fixed dec(3),
13 c 0006           2 wage     fixed dec(5,2),
14 c 0006           2 hours    fixed dec(5,1);
15 c 0006
16 c 0006     dcl
17 c 0006         dashes character(15) static initial
18 c 0006             ('$--------------'),
19 c 0006         buff character(20) varying;
20 c 0006
21 c 0006     dcl
22 c 0006         i fixed,
23 c 0006         (grosspay, withhold) fixed dec(7,2);
24 c 0006
25 c 0006     dcl
26 c 0006         (repfile, empfile) file;
27 c 0006
28 c 0006     open file(empfile) keyed env(f(100),b(4000))
29 c 0026         title ('$1.EMP');
30 c 0026
31 c 0026     open file(repfile) stream print title('$2.$2')
32 c 0045         environment(b(2000));
33 c 0045
34 c 0045     put list('Set Top of Forms, Type Return');
35 c 005C     get skip;
36 c 006D
37 c 006D         do while('1'b);
38 c 006D         read file(empfile) into(employee);
39 c 0085         if name = 'EOF' then
40 c 0094             stop;
41 c 0097         put file(repfile) skip(2);
42 c 00AB         buff = '[' !! name !! ']^m^j';
43 c 00CD         write file(repfile) from (buff);
44 c 00E5         grosspay = wage * hours;
45 c 0105         withhold = grosspay * .15;
46 c 0125         buff = grosspay - withhold;
47 c 0147             do i = 1 to 15
48 c 0180                 while (substr(buff,i,1) = ' ');
49 c 0180             end;
50 c 0180         i = i - 1;
51 c 0187         substr(buff,1,i) = substr(dashes,1,i);
52 c 01AC         write file (repfile) from(buff);
53 c 01C7         end;
54 c 01C7
55 a 01C7     end report;
```

Figure 5-15. Final Report Generation Program.

```
A>b:report plantl $con
Set Top of Forms, Type Return


[Abercrombie]
$----265.96

[Fairweather]
$----492.15

[Eggbert]
$------Ø.ØØ

[Willowander]
$----2Ø3.37

[Millywatz]
$----368.43

[Quagmire]
$----246.35
End of Execution
```

Figure 5-16a. Report Generation to the Console.

```
A>b:report plantl plantl.prn
Set Top of Forms, Type Return

A>type plantl.prn

[Abercrombie]
$----265.96

[Fairweather]
$----492.15

[Eggbert]
$------Ø.ØØ

[Willowander]
$----2Ø3.37

[Millywatz]
$----368.43

[Quagmire]
$----246.35
```

Figure 5-16b. Report Generation to a Disk File.

then computes the pay value using the expression

```
grosspay - withhold
```

which is assigned to the varying character string called buff. The assignment causes automatic conversion of the decimal value to string type, with leading blanks. The leading blanks are then scanned and replaced by a dollar sign dash sequence, and written to the report file.

Figures 5-16a and 5-16b show the output from the report program. In the first case, the report is sent to the console for debugging purpose, while "plantl.prn" receives the data in the second example.

6. LABEL CONSTANTS, VARIABLES, AND PARAMETERS.

You probably noticed that all of the programs shown above either stop by encountering an end of file condition, with a corresponding ENDFILE traceback, or use a special data value which signals the end of data condition. The POLY program in Figure 5-1, for example, detects the end of data by checking for the special case where all three input values, x, y, and z, are zero.

There are, fortunately, more elegant ways to sense the end of data condition in PL/I-80. In fact, sensing the end of data is just one facility among many, under the general topic of "exception processing." As a prelude to the discussion of exception processing, we need some background in label processing, since labelled statements are often involved when handling exceptional conditions. As always, if the discussion becomes too detailed, skip to later sections and return when you've seen some examples.

Contemporary programming practices advocate the general avoidance of labelled statements and GO TO's due to the unstructured programs which often result from using such statements. The resulting programs are often difficult to comprehend by another programmer and become unreadable, even to the author, as the program grows in size. PL/I-80 provides a comprehensive set of control structures in the form of iterative DO groups with REPEAT and WHILE options which preclude the necessity for labelled statements in the general programming schema.

There are occasions, however, when judicious use of labelled statements is considered appropriate. One particular situation, for example, is found in program exception processing where the occurrence of a catastrophic error, such as a mistyped input data line, is most easily handled by simply transferring control to an outer block label where program recovery takes place. In this case, the program flow is considerably simpler to comprehend than the alternative system of flags, tests, and return statements.

Generally, one should avoid labelled statements and GO TO's whenever the normal PL/I-80 program control structures are directly applicable, limiting their use to exception processing and locally defined computed GO TO's.

Program labels, like other PL/I-80 data types, fall into two broad categories: label constants and label variables. Label constants are those which appear literally within the source program, and do not change as the program executes. Label variables, however, have no initial value and must be assigned the value of a label constant through a direct assignment statement, or through the actual to formal parameter assignments implicit in a subroutine call. The simplest form of a label constant precedes a PL/I-80 statement as shown below.

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

In this case, "lab" has the constant label value corresponding to the

particular statement address where the PUT statement starts.

A label constant can also contain a single positive or negative literal subscript, corresponding to the target of a n-way branch (i.e., a "computed GO TO"). The program segment which follows shows a specific example.

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

In this case, four label constants q(-1), q(Ø), q(2), and q(3) are defined within the program. The label constant vector

```
q(-1:3) label constant
```

is automatically defined to hold the values of these label constants. You must ensure that program control does not transfer to a subscript which does not have a corresponding label constant value. In the above case, for example, a branch to q(i) produces an undefined value if i is below -1, equal to 1, or above 3.

Label constants are either only locally referenced, or non-locally referenced. A locally referenced label constant occurs as the target of a GO TO statement only within the PROCEDURE or BEGIN block in which it occurs. A label constant is non-locally referenced if it occurs on the right side of an assignment to a label variable, as an actual parameter to a subroutine, or as the target of a GO TO statement within an inner nested PROCEDURE or BEGIN block. Although there is no functional difference between a locally referenced and non-locally referenced label constant, there is additional space and time overhead required to handle non-locally referenced labels. For this reason, the PL/I-8Ø assumes that subscripted label constants will be only locally referenced: the results are undefined if control transfers to a subscripted label constant from outside the current scope.

The non-functional program segment shown below provides an example.

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

The label constant "labl" is only locally referenced within the procedure p1, while "lab2" is the target of both a local reference within p1 and a non-local reference within p2.

A label variable takes on the value of a label constant through an explicit assignment statement, or through the implicit assignment performed when a subroutine is called. Similar to other PL/I-80 variables, label variables must be declared and may be optionally subscripted.

The skeletal program of Figure 6-1 shows various label constants and variables. The label constants in this program are c(1), c(2), c(3), labl, and lab2, and are defined by their literal occurrence within the program. The label variables are x, y, z, and g defined by the declarations on lines 5 and 33. At the start of execution, the label variables have undefined values. The variable x is first assigned the constant value labl. Label variable y then (indirectly) receives the constant value labl through the assignment on line 7. As a result, all three GO TO statements beginning on line 9 are functionally equivalent: each transfers control to the null statement following the label labl on line 27.

The subroutine call on line 13 shows a different form of variable assignment. Lab2 is an actual parameter which is sent to the procedure p, and assigned to the formal label variable g. In this particular program, the subroutine call transfers program control directly to the statement labelled "labl."

The DO-group beginning on line 15 initializes the variable label vector z to the corresponding constant label vector values of c. Note that since c is a vector of label constants, the reversed assignment

```
c(i) = z(i);
```

would be invalid within this program. Due to this initialization, the two computed GO TO statements starting on line 20 have exactly the same effect.

```
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/I-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(1);
  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
```

Figure 6-1. An Illustration of Label Variables and Constants.

7. EXCEPTION PROCESSING.

An important facility of any production programming language is its ability to intercept run-time error conditions in order that a program-defined action can take place to handle the error. An exceptional condition takes place, for example, when input data is read from an interactive console, and the operator inadvertently types a data value which does not conform to the input data variable. Under normal circumstances, a "conversion" exception is raised by the run-time system and, in the absence of any program-defined action, execution terminates with a traceback. In a production environment, however, this premature termination could occur after hours of data entry, resulting in a considerable amount of wasted effort.

Thus, PL/I-80 incorporates a comprehensive set of operations for exception processing in the form of ON, REVERT, and SIGNAL statements. The ON statement defines the actions which take place upon encountering an exception, the REVERT statement disables the ON statement, and the SIGNAL statement allows various conditions to be raised by the program.

There are a total of nine major exception categories which are, by name, ERROR, FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, ZERODIVIDE, ENDFILE, UNDEFINEDFILE, KEY, and ENDPAGE. The first five categories include all arithmetic error conditions and miscellaneous conditions which can arise during I/O setup and processing, as well as conversion between the various data types. The last four categories apply to a specific file which is being operated upon by the run-time I/O system. Each condition has an associated subcode which provides information as to the source of the exception, as described below.

As a simple example, consider the file-to-file copy program shown below:

```
copy:
    proc options(main);
    dcl
        buff char(254) var,
        (input, output) file;

    open file(input) stream title('$1.$1');
    open file(output) stream output
        title('$2.$2');

    on endfile(input)
        stop;

        do while('1'b);
        read file(input) into(buff);
        write file(output) from(buff);
        end;
    end copy;
```

As described in Section 5.2, this program opens an input file, called "input," and transfers each record from this file to an output file,

called "output." Under normal circumstances, the program in Figure 5-3 terminates with an END OF FILE error message with a program traceback, which, although of no harm, can be somewhat distressing to an uninitiated user of the program. The ON statement given above, however, intercepts the end of file condition on the input file and executes a single STOP statement. In this case, the program terminates normally with the message

Execution Terminated

The various statement forms for ON, REVERT, and SIGNAL are discussed first, using the ENDFILE condition as an example, followed by a description of each of the conditions.

7.1. The ON Statement.

The ON statement is used to programmatically intercept a particular condition when it is raised by the run-time system or a SIGNAL statement. The form of the ON statement is

ON condition on-body;

where "condition" is one of the exception categories given above, and "on-body" is a PL/I-80 statement or statement group to execute when the condition occurs. In order to avoid ambiguity, the statement must be a simple statement (not a conditional), or a BEGIN-END group which itself can contain any valid PL/I-80 statement, other than a RETURN (a RETURN is allowed, of course, within any procedure definitions which occur inside the BEGIN-END group).

Control returns from the ON statement at the end of the statement or BEGIN-END group. Alternatively, control may be transferred to a non-local label outside the on-body. If the condition is already set, execution of yet another ON statement with the same condition saves the previous condition in "stack order" and institutes the new condition. A stacked condition is reinstituted when a REVERT statement is executed, or the block containing the ON statement is exited. Three examples of valid ON statements are given below.

```
on endfile(input)
    eofile = '1'b;

on endfile(input)
    go to exit;
```

```
on endfile(input)
    begin;
    if input = sysin then
        stop;
    put list('Ok to Stop?');
    get list(ok);
    if ok = 'y' then
        stop;
    go to retry;
    end;
```

7.2. The REVERT Statement.

Execution of a REVERT statement disables the currently active named condition, and recovers the previously stacked condition, if any. The form of the REVERT statement is

```
REVERT condition;
```

where "condition" is one of the categories discussed above. An automatic REVERT statement takes place for any ON conditions set within a procedure when the procedure is exited. The following program segment, for example, shows balanced ON and REVERT statements used within a DO group:

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

In this particular case, the ON statement is executed at the beginning of each iteration, enabling the ENDFILE condition. The REVERT statement at the end of the group disables the condition which was set at the beginning of the block. The ON and REVERT statements do, however, require some simple run-time processing and thus the above group is more efficiently written as:

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

Note that no more than 16 ON conditions can be active or stacked at any given point in the program execution. The message:

```
Condition Stack Overflow
```

occurs if the stack size is exceeded, and the program terminates.

The (rather unstructured) program shown in Figure 7-1 illustrates the automatic REVERT statements which take place upon procedure exit. The procedure "p" is called from line 8 with the DO group index, along with the label constant "exit" as actual parameters. The ON statement within p is executed upon each invocation and, without the automatic REVERT statements, would overflow the condition stack when the index i reaches 17. There are three possible ways to exit p: first, if the operator types an end of file character (control-z, followed by return) the enabled ON condition is executed, sending control through the label variable "lab" to the statement labelled "exit." Since this GO TO takes control outside the environment of p, the ON condition automatically REVERTs.

The second possible exit follows the test on line 21. If the operator types a value equal to the index, then the GO TO statement on line 22 is executed, again sending control to the non-local label "exit" which REVERTs to the original condition.

Finally, control can return normally by reaching the end of the procedure p. In this case, the automatic REVERT is again executed, disabling the ON condition which was set on line 17. Thus, the enabled ON condition is always disabled, no matter how program control leaves the environment of p.

7.3. The SIGNAL Statement.

The on-body which corresponds to a particular ON statement can be activated through execution of a SIGNAL statement which takes the form:

```
SIGNAL condition;
```

The effect is the same as if the condition had been enabled externally: the topmost stacked ON condition is executed, if it exists. If no ON condition is active, the default system action takes place. The following program segment illustrates a particular use of the SIGNAL statement:

```
on endfile(sysin)
    stop;

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

In this example, the SIGNAL statement is executed whenever the value "END" is read from the SYSIN file. The ON condition set at the

```
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
```

Figure 7-1. Program Illustrating REVERT Processing.

beginning of the program thus receives control upon a real end of file, or when the "END" value is read.

7.4. The ERROR Exception.

The ERROR condition is the broadest category of all PL/I-80 exceptions and includes, through its subcode, both system defined and programmer defined conditions. The form of the ERROR condition is

```
ON ERROR on-body;
  SIGNAL ERROR;
  REVERT ERROR;
```

or

```
ON ERROR(integer-expression) on-body;
  SIGNAL ERROR(integer-expression);
  REVERT ERROR(integer-expression);
```

In the first three cases, the ERROR subcode is assumed to be zero, while the second set includes a specific subcode in the range 0 to 255. The forms

```
ON ERROR on-body;
ON ERROR(0) on-body;
```

intercept any error condition, no matter what subcode is set. The form

```
ON ERROR(3) ... ;
```

for example, intercepts the ERROR condition only when it is accompanied by subcode 3. In general, ERROR conditions with a subcode in the range 0-127 are considered catastrophic. As a result, the on-body for these conditions must not return, but instead must execute a GO TO to a non-local label. Subcodes in the range 128-255 are considered harmless, and may return after performing some local action.

Subcodes for the ERROR condition are partitioned into four groups:

(a)	0	-	63	Reserved for PL/I-80
(b)	64	-	127	Programmer Defined
(c)	128	-	191	Reserved for PL/I-80
(d)	192	-	255	Programmer Defined

The subcodes which are presently assigned from group (a) are:

ERROR(1)	-	Data Conversion: data types do not conform during assignment, computation, or input processing.

ERROR(2) - I/O Stack Overflow

ERROR(3) - Argument to transcendental function is out of range.

ERROR(4) - I/O Conflict: the attributes of an open file do not match the attributes required for a particular GET, PUT, READ or WRITE.

ERROR(5) - Format stack overflow, nested format evaluation exceeds 32 levels.

ERROR(6) - Invalid format item, data item does not conform to format item, or unrecognized format item encountered.

ERROR(7) - Free space exhausted, no more space is available in dynamic storage area.

The following program segment provides a simple example of the use of the ERROR condition:

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

The GET statement reads a variable x from the SYSIN file. If the operator types invalid data during the input operation, ERROR(1) is signalled by the run-time system. In this case, the on-body gets control, and error message is written to the console, and execution re-commences at the "retry" label.

The SIGNAL statement can be used in conjunction with the ON statement to flag either terminal or non-terminal conditions. The statement

```
signal error(64)
```

raises the ERROR(64) condition. If there is an ON ERROR(64) active, the corresponding on-body receives control. Otherwise, the program terminates with an error message. The statement

```
signal error(255)
```

performs a similar action except that the program does not terminate if the ERROR(255) condition is not active. Note that an ON ERROR or ON ERROR(Ø) statement will intercept any subcode in the range Ø-255. The particular error subcode can be extracted, however, using the ONCODE function discussed below.

7.5. FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, and ZERODIVDE.

The arithmetic exceptions are

```
FIXEDOVERFLOW or FIXEDOVERFLOW(i)
     OVERFLOW or OVERFLOW(i)
    UNDERFLOW or UNDERFLOW(i)
   ZERODIVIDE or ZERODIVIDE(i)
```

where i denotes the optional integer expression. Similar to the ERROR function, ON, REVERT, and SIGNAL statements can specify any of these conditions. Further, if the integer expression is absent, then a zero value is assumed. An ON statement with a zero valued subcode intercepts a subcode of any value from Ø-255. Subcode values are divided into system defined and user defined values, as listed with the ERROR function. Note, however, that all arithmetic faults are considered terminal. That is, if an ON condition is set for an arithmetic exception, then the on-body must contain a transfer to a global label. Otherwise, the program is terminated upon return from the ON unit.

Currently defined system subcodes are listed below:

FIXEDOVERFLOW(1)	-	Decimal Add, Multiply, or Store
OVERFLOW(1)	-	Floating Point Pack
UNDERFLOW(1)	-	Floating Point Pack
ZERODIVIDE(1)	-	Decimal Divide
ZERODIVIDE(2)	-	Floating Point Divide
ZERODIVIDE(3)	-	Integer Divide

7.6. ENDFILE, UNDEFINEDFILE, KEY, and ENDPAGE.

Several exceptional conditions may arise during I/O processing which are related to particular file access. These conditions are

denoted by

```
ENDFILE(file-reference)
UNDEFINEDFILE(file-reference)
KEY(file-reference)
ENDPAGE(file-reference)
```

where "file-reference" denotes a file-valued expression. The file value which results need not denote an open file.

The ENDFILE condition is raised whenever the end of file character (control-z) is read from a STREAM file, or the physical end of file is encountered in a RECORD file which is processed in SEQUENTIAL mode. A DIRECT READ with a key beyond the end of file also raises the ENDFILE condition. Similarly, RECORD or STREAM OUTPUT operations will signal ENDFILE if the disk capacity is exceeded.

The UNDEFINEDFILE condition is raised whenever a file is accessed for INPUT or OUTPUT and the file does not exist on the specified disk. This condition will also be raised if a physical device ($CON, $LST, $RDR, $PUN) is accessed as a KEYED or UPDATE file.

The KEY condition is raised when a program attempts to access a key value beyond the capacity of the disk.

The ENDPAGE condition is raised for PRINT files when the value of the current line reaches the PAGESIZE for the specified file. The current line begins at zero, and increased by one for each line-feed which is sent to the file. If the file is initially opened with

```
PAGESIZE(Ø)
```

then the ENDPAGE condition is never raised. (PL/I-8Ø opens the default console input, SYSIN, with a zero PAGESIZE.) The current line is reset to one whenever a form-feed is sent to the output file, through a PAGE option in a PUT statement within an ON-unit, or through the default system action which automatically inserts the form-feed. In any case, if the ENDPAGE condition is raised during the execution of a SKIP option, then the SKIP is terminated.

It must be noted that if an ON unit intercepts the ENDPAGE condition, but does not execute a PUT statement with the PAGE option, then the current line is not reset to one. The result is that the ENDPAGE will not be signalled since the current line continues unbounded until a PUT statement with the PAGE option is executed. The current line will, in reality, count as high as 32767 and then begin again at 1. Due to fact that the line count is always greater than zero, the ENDPAGE condition will never be signalled for files with a PAGESIZE of zero.

As described above, if no ENDPAGE ON-unit is active, the default system action is to insert a form-feed into the output file. The default system action for ENDFILE, UNDEFINEDFILE, and KEY, however, is to terminate the program execution with an error message.

If an ON-unit receives control for ENDFILE, UNDEFINEDFILE, or

KEY, and returns to the point where the signal occurred, then the current I/O operation is terminated, and control is passed to the statement following the OPEN, GET, PUT, READ, or WRITE which caused the condition to be raised.

7.7. ONCODE, ONFILE, ONKEY, PAGENO, and LINENO.

Several built-in functions are provided in PL/I-80 which aid in exception processing. Specifically, five function declarations exist in the scope of all PL/I-80 programs:

```
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);
```

The ONCODE function returns the most recently signalled subcode, or zero if no condition has been raised. The function can be used, for example, to determine the exact source of an error after the ON unit is activated:

```
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:
```

The ONFILE function returns the string value of the internal file name which was last involved in an I/O operation which raised a condition. In the case of a conversion error, the ONFILE function produces the name of the file which was active at the time. If no file was involved in a signalled condition, then the ONFILE function returns a string of length 0. An example of the ONFILE function is shown below.

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

The ONKEY function returns the value of the last key involved in an I/O operation which raised the ONKEY condition, and is only valid within the on-body of the activated unit. An example of the use of ONKEY is

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

The last two functions, PAGENO and LINENO, return the current page number and current line number for the PRINT file named as the parameter. Note that when the ENDPAGE condition is signalled by other than a SIGNAL statement, the line number is one greater than the page size for the file. The PAGENO and LINENO functions are illustrated in the example which follows.

7.8. An Example of Exception Processing.

Figure 7-2 shows a rather extensive example of I/O processing in PL/I-80 using ON conditions. The purpose of this program is to copy a STREAM file from the disk to a PRINT file, while properly formatting the output line with a page header and line numbers. The program interacts with the console to obtain the parameters for the copy operation. The statements which appear between lines 10 and 47 read the parameter values from the console, and provide error exits and retry operations for each input value. Lines 49 through 66 setup various ON units which intercept errors during the copy operation. The actual copy operation itself is enclosed in the iterative DO group between lines 69 and 74. These individual sections of the program are discussed in detail below.

Functionally, the program setup involves reading five values: the number of lines on each page, the width of the printer line, the line spacing (normally single or double spaced output), and the destination and source files or devices. During entry of these parameters, the operator may type an end of file character (control-z), and the prompting is restarted. The PUT statement on line 10 writes the initial sign-on message. Note that the first character of the message is a control-z which, when using an ADM-3 CRT device, clears the screen. The ON statement shown on line 12 traps the ENDFILE condition for SYSIN so that execution begins at "typeover" whenever an end of file is read from the console. Lines 16 through 20 read the first two parameters, with no error checking (other than detecting the end of file). Line 22, however, intercepts conversion errors for all operations which follow. If the operator types a non-numeric field during execution of the GET statement on line 29, the on-body between 23 and 26 gets control, writes an error message, and then branches to "getnumber" where the input operation is reattempted. Following successful input of the "spaces" parameter, the REVERT statement on line 30 disables the conversion error handling.

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

L: List Source Program

   NO ERROR(S) IN PASS 1

   NO ERROR(S) IN PASS 2


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

   1 a 0000 copy: procedure options(main);
   2 c 0006     dcl
   3 c 000D          (sysin, sourcefile, printfile) file;
   4 c 000D     dcl
   5 c 000D          (pagesize, pagewidth,
   6 c 000D           spaces, linenumber) fixed;
   7 c 000D     dcl
   8 c 000D          (line char(14), buff char(254)) varying;
   9 c 000D
  10 c 000D     put list('^z    File to Print Copy Program');
  11 c 0024
  12 c 0024     on endfile(sysin)
  13 d 002C          go to typeover;
  14 d 0032
  15 c 0032     typeover:
  16 c 0039     put skip(5) list('How Many Lines Per Page?   ');
  17 c 0055     get list(pagesize);
  18 c 006D
  19 c 006D     put skip list('How Many Column Positions? ');
  20 c 0089     get skip list(pagewidth);
  21 c 00A6
  22 c 00A6     on error(1)
  23 d 00AD          begin;
  24 e 00B0          put list('Invalid Number, Type Integer');
  25 e 00C7          go to getnumber;
  26 d 00CA          end;
  27 c 00CA     getnumber:
  28 c 00D1     put skip list('Line Spacing (1=Single)?   ');
  29 c 00ED     get skip list(spaces);
  30 c 010A     revert error(1);
  31 c 0111
  32 c 0111     put skip list('Destination  Device/File:  ');
  33 c 012D     get skip list(line);
  34 c 014C
  35 c 014C     open file(printfile) print pagesize(pagesize)
  36 c 016C          linesize(pagewidth) title(line);
  37 c 016C
```

Figure 7-2a. Listing of the COPYLPT Program Part A.

```
 38 c 016C     on undefinedfile(sourcefile)
 39 d 0174         begin;
 40 e 0177         put skip list('"',line,'" isn''t a Valid Name');
 41 e 01A7         go to retry;
 42 d 01AA         end;
 43 c 01AA     retry:
 44 c 01B1     put skip list('Source File to Print?      ');
 45 c 01CD     get list(line);
 46 c 01E7     open file(sourcefile) stream title(line)
 47 c 0204         env(b(8000));
 48 c 0204
 49 c 0204     on endfile(sourcefile)
 50 d 020C         begin;
 51 e 020F         put file(printfile) page;
 52 e 0221         stop;
 53 d 0224         end;
 54 d 0224
 55 c 0224     on endfile(printfile)
 56 d 022C         begin;
 57 e 022F         put skip list('^g^g^g^g Disk is Full');
 58 e 024B         stop;
 59 d 024E         end;
 60 d 024E
 61 c 024E     on endpage(printfile)
 62 d 0256         begin;
 63 e 0259         put file(printfile) page skip(2)
 64 e 028F             list('PAGE',pageno(printfile));
 65 e 028F         put file(printfile) skip(4);
 66 d 02A3         end;
 67 d 02A3
 68 c 02A3     signal endpage(printfile);
 69 c 02AC         do linenumber = 1 repeat(linenumber + 1);
 70 c 02B2         get file (sourcefile) edit(buff) (a);
 71 c 02D2         put file (printfile)
 72 c 0306             edit(linenumber,'|',buff) (f(5),x(1),a(2),a);
 73 c 0306         put file (printfile) skip(spaces);
 74 c 0326         end;
 75 c 0326
 76 a 0326     end copy;

CODE SIZE = 0326
DATA AREA = 021A
```

Figure 7-2b. Listing of the COPYLPT Program Part B.

The input and output files are opened between lines 35 and 47. The program assumes that the output file can always be opened, but detects an UNDEFINED input file so that the operator can correct the file name.

Two ON ENDFILE statements are executed between lines 49 and 59. The first ON statement traps the input end of file condition and performs a page eject on the output file. This ensures that the printer output will be at the top of a new page upon completion of the print operation. The STOP statement included in this ON unit completes the processing with a normal exit. The second ON unit intercepts the end of file condition on the print file. Since this can only occur if the disk file fills, the message "Disk is Full" is printed, followed by normal termination. The preceding control-g characters send a series of "beeps" to the CRT as an alarm. Note that the run-time system closes all files upon termination so that the print file is intact to the full capacity of the disk.

An ENDPAGE ON unit is executed on line 61 which intercepts the end of page condition for the print file. Whenever this condition is raised, the ON unit moves to the top of the next page, skips two lines, prints the page number, and skips 4 more lines before returning to the signal source. The SIGNAL statement on line 68 starts the print file output on a new page by explicitly sending control to the ON unit defined on line 61. All subsequent ENDPAGE signals are generated by the run-time I/O system at the end of each page.

The DO group beginning on line 69 initializes and increments a line counter on each iteration. The GET EDIT statement on line 70 reads with an A (Alphanumeric) format which fills "buff" with the next input line up to, but not including, the carriage return line feed sequence. The PUT EDIT statement on line 71 writes the line to the destination file with a preceding line number, a blank, a vertical bar, and another blank (resulting from the A(2) field). Note that the SKIP(SPACES) operation may be partially aborted if the ENDPAGE condition is raised during the execution of the PUT statement.

Operator interaction is shown in Figure 7-3, where the original copy.pli program is used as the source file, and the $LST (physical printer) is used as the destination. Figure 7-4 shows the first two pages of output produced by this program.

As you can see, there is quite a bit of error handling that can be done within your program. Even this last example could be further enhanced to handle errors in the first two input lines (currently a CONVERSION error could be raised) or errors in the destination file name. In fact, a good exercise at this point is to add exception handlers for these errors, and then retest the program. To gain further experience, go back over the previous examples and add ON units to trap invalid input data and end of file conditions. By the time you finish changing these examples, you'll have a good foundation in exception processing.

```
A>b:copylpt
    File to Print Copy Program


How Many Lines Per Page?   1Ø

How Many Column Positions? 4Ø

Line Spacing (1=Single)?   zot
Invalid Number, Type Integer
Line Spacing (1=Single)?   1

Destination  Device/File:  $lst

Source File to Print?      $zap

" $zap " isn't a Valid Name
Source File to Print?      b:copylpt.pli
```

Figure 7-3. Console Interaction with COPYLPT.

```
PAGE         1


    1 | copy: procedure options(main);
    2 |      dcl
    3 |           (sysin, sourcefile, prin
tfile) file;









PAGE         2


    4 |      dcl
    5 |           (paqesize, pagewidth,
    6 |            spaces, linenumber) fix
ed;
```

Figure 7-4. Output from the COPYLPT Program.

8. APPLICATIONS OF CHARACTER STRING PROCESSING.

This section is devoted entirely to giving the details of two sample programs which illustrate ways in which PL/I-80 character string functions can be used to manipulate character data. The intention is to again provide a basis for gaining fluency in the language. As before, read the explanations, examine the sample programs, and make changes to these programs in order to expand your own working knowledge of PL/I-80.

8.1. The OPTIMIST Program.

Recall the first Pl/I-80 program in Section 1, called the OPTIMIST? The OPTIMIST has the task of turning a negative statement into a positive statement, based upon a few commonly used words in the English language. The OPTIMIST performs its job using the character string facilities of PL/I-80.

Figure 8-1 shows the OPTIMIST listing. The first segment, between lines 7 and 20, defines the data items referenced within the program. The remaining portion reads sentences from the console, terminated by a period, and retypes the sentences in their positive form. A sample console interaction is shown in Figure 8-2. As you can tell, the program does a pretty good job if the sentences are of the proper form, but reveals itself as just another computer program when things get complicated.

The sequence of negative words is defined starting on line 8, with the corresponding positive words beginning on line 10. Thus, "never" becomes "always," while "none" becomes "all," and so-forth. Note that the word "not" is replaced by the empty string. The upper and lower case alphabets are also included for case translation in the sentence processing section.

The OPTIMIST doesn't want to stop talking, so the DO group between lines 22 and 42 loops indefinitely, and terminates only through some unnatural influence, such as a control-z (end of file) or control-c (system warm start) at the beginning of an input line. Each successive input sentence is constructed between lines 24 and 29, where the DO group reads another word, and concatenates the word onto the end of the sentence. The SUBSTR test in the DO WHILE heading checks for a period at the end. Note that the maximum length of a sentence is 254 characters (additional characters are discarded).

Upon reading the complete sentence, all upper case characters are translated to lower case so that the negative words can be scanned. This case translation is performed by the TRANSLATE built-in function shown on line 30. As an additional illustration of string processing, the VERIFY function is used on line 31 to ensure that the sentence consists only of letters and a period. If not, VERIFY returns the first (non-zero) position which mismatches, and the OPTIMIST responds with

Actually, that's an interesting idea.

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

   1 a 0000 optimist:
   2 a 0006     proc options(main);
   3 c 0006     %replace
   4 c 0006         true   by '1'b,
   5 c 0006         false  by '0'b,
   6 c 0006         nwords by 5;
   7 c 0006     dcl
   8 c 0006         negative (1:nwords) char(8) var static initial
   9 c 0006             (' never',' none',' nothing',' not',' no'),
  10 c 0006         positive (1:nwords) char(10) var static initial
  11 c 0006             (' always',' all',' something','',' some'),
  12 c 0006         upper char(28) static initial
  13 c 0006             ('ABCDEFGHIJKLMNOPQRSTUVWXYZ. '),
  14 c 0006         lower char(28) static initial
  15 c 0006             ('abcdefghijklmnopqrstuvwxyz. ');
  16 c 0006     dcl
  17 c 0006         sent char(254) var,
  18 c 0006         word char(32)  var;
  19 c 0006     dcl
  20 c 0006         (i,j) fixed;
  21 c 0006
  22 c 0006         do while(true);
  23 c 0006         put skip list('What''s up? ');
  24 c 0022         sent = ' ';
  25 c 002F             do while
  26 c 004B                 (substr(sent,length(sent)) ^= '.');
  27 c 004B             get list (word);
  28 c 0065             sent = sent !! ' ' !! word;
  29 c 0088             end;
  30 c 0088         sent = translate(sent,lower,upper);
  31 c 00AB         if verify(sent,lower) ^= 0 then
  32 c 00C2             sent = ' that''s an interesting idea.';
  33 c 00CF             do i = 1 to nwords;
  34 c 00E1             j = index(sent,negative(i));
  35 c 00F9             if j ^= 0 then
  36 c 0102                 sent = substr(sent,1,j-1) !!
  37 c 0162                    positive(i) !!
  38 c 0162                    substr(sent,j+length(negative(i)));
  39 c 0162             end;
  40 c 0162         put list('Actually,'!!sent);
  41 c 017F         put skip;
  42 c 0193         end;
  43 a 0193     end optimist;

CODE SIZE = 0193
DATA AREA = 01F2
```

Figure 8-1. Listing of the OPTIMIST Program.

```
A>b:optimist

What's up? Nothing is up.
Actually,  something is up.

What's up? This is NOT fun.
Actually,  this is fun.

What's up? Programs like this never make sense.
Actually,  programs like this always make sense.

What's up? Nothing is easy that is not complicated.
Actually,  something is easy that is complicated.

What's up? Nobody cares, and its none of your business.
Actually,  somebody cares and its all of your business.

What's up? No, no, no, no.
Actually,  some no no no.

What's up? You no, no, no.
Actually,  you some no no.

What's up? NO, NO, NO!

Actually, that's an interesting idea.

What's up? No it is not.
Actually,  some it is.

What's up? ^Z

END OF FILE (1), File: SYSIN=CON
Traceback: Ø9B7 Ø962 Ø157 6EØØ # 1BD7 Ø521 8Ø82 Ø157
End of Execution
```

Figure 8-2. Interaction with the OPTIMIST Program.

If VERIFY returns a zero value, then the sentence contains only (translated) lower case letters and a period. In this

case, the DO group between lines 33 and 39 is executed. On each iteration, the INDEX function is used to search for the next negative word, given by negative(i). If found, j is set to the position of the negative word and, in the assignment starting on line 36, is replaced by the positive word to which it corresponds. In this assignment,

```
substr(sent,1,j-1)
```

is the portion of the sentence which occurs before the negative word, while

```
positive(i)
```

is the replacement value for the negative word, and

```
substr(sent,j+length(negative(i)))
```

is the portion of the sentence which follows the negative word being replaced. The concatenation of these three segments produces a new sentence with the negative word replaced by the positive word. Note that since all negative words have a leading blank, the negative portion will only be found at the beginning of a word. Thus, "nevermind" is replaced by "alwaysmind" which, on occasion, can produce some interesting results. The OPTIMIST sends the resulting sentence to the console, and loops back to read another input.

Three improvements could be made to the OPTIMIST. First, the input scan will never stop if the sentence exceeds 254 characters since the period will not be found. A check should be made to ensure that the newly appended word does not exceed the maximum size. Second, an ON-unit could be included to detect an end of file so that the program terminates in a reasonable fashion. Last, you could make the OPTIMIST a bit smarter!

8.2. A Free-Field Scanner.

A second, more practical, application of string processing is given in this section. In general, it is often useful to have a "free-field scanner" which is a subroutine that reads input lines and breaks the input values into individual numbers and characters. The program shown in Figure 8-3, called FSCAN, gives an example of a free-field scanner. The function of FSCAN is to test an embedded subroutine, called GNT (Get Next Token), which reads the next input item, called a "token." In this case, the tokens are just numeric values, such as 1234.56, or individual letters and special characters. All intervening blanks between the tokens are bypassed in the token scan.

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

   1 a 0000 fscan:
   2 a 0006     proc options(main);
   3 c 0006     %replace
   4 c 0006         true by '1'b;
   5 c 0006     dcl
   6 c 0006         token char(80) var
   7 c 0006         static initial('');
   8 c 0006
   9 c 0006     gnt:
  10 c 0006         proc;
  11 e 0006         dcl
  12 e 0009             i fixed,
  13 e 0009             line char(80) var
  14 e 0009             static initial('');
  15 e 0009
  16 e 0009         line = substr(line,length(token)+1);
  17 e 0023             do while(true);
  18 e 0023             if line = '' then
  19 e 0031                 get edit(line) (a);
  20 e 004E             i = verify(line,' ');
  21 e 0063             if i = 0 then
  22 e 006C                 line = '';
  23 e 0074             else
  24 e 0074                 do;
  25 e 0074                 line = substr(line,i);
  26 e 008A                 i = verify(line,'0123456789.');
  27 e 009F                 if i = 0 then
  28 e 00A8                     token = line;
  29 e 00B6                 else
  30 e 00B6                 if i = 1 then
  31 e 00BF                     token = substr(line,1,1);
  32 e 00D5                 else
  33 e 00D5                     token = substr(line,1,i-1);
  34 e 00EE                 return;
  35 e 00F2                 end;
  36 e 00F2             end;
  37 c 00F2         end gnt;
  38 c 00F2
  39 c 00F2         do while(true);
  40 c 00F2         call gnt;
  41 c 00F5         put edit('''''!!token!!''''') (x(1),a);
  42 c 0127         end;
  43 a 0127     end fscan;

CODE SIZE = 0127
DATA AREA = 00BD
```

Figure 8-3. Listing of the FSCAN Free-Field Scanner Test.

The program shown in Figure 8-3 is broken into three logical parts. The first segment appears between lines 3 and 7, and defines the global data area called token, for use by the GNT procedure. The second portion, from line 9 through line 37, is the GNT procedure itself, while the DO group between 39 and 42 performs the GNT testing function.

An interaction with the FSCAN program is shown in Figure 8-4. Note that the program reads a line of input, then decomposes the input line into basic tokens and writes the decomposed items back to the console, with surrounding quotes. The assumption is, of course, that once the free-field scanner has been tested, the GNT procedure will be extracted from this program and placed into a production program where the scanner is required. In fact, GNT will reappear in another program later where it is used to compute values of arithmetic expressions.

The overall operation of the GNT procedure is as follows. The character variable called LINE is used to hold the input line as it is being processed. Initially, the value of LINE is empty, due to the declaration starting on line 11. On each call to GNT, the first portion of LINE is extracted and placed into TOKEN, which becomes the next input item. On each successive call, the previous TOKEN value is first removed from the beginning of LINE before the next item is scanned. As an example, suppose the operator types the line

bbb88*9.9

where "b" represents a blank character. On the first call to GNT (see line 38), both TOKEN and LINE are empty strings, so the assignment on line 16, which normally removes the previous value of TOKEN, leaves LINE as an empty string. The DO group between lines 17 and 36, however, ensures that the LINE buffer is filled. If, on line 18, an empty buffer is encountered, it is immediately refilled using a GET EDIT statement. In any case, the VERIFY call on line 20 produces the first position in LINE which is not blank. If VERIFY produces a 0 value, then the entire line is blank and must be cleared so that the refill operation will take place on the next iteration. If LINE is not entirely blank, the DO group beginning on line 24 is entered.

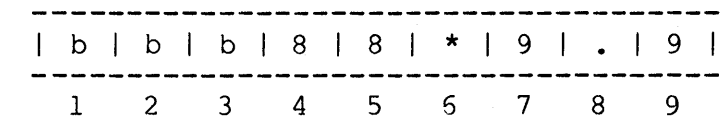
Processing within the DO group proceeds as follows. Upon entry to the group, the value of i is the first non-blank position of the LINE buffer. Thus, the statement on line 25 removes the preceding blanks from LINE, leaving the next token starting at the first position. The VERIFY function is then applied to LINE to determine if the next item is a number. The statement on line 26 sets i to 0 if the entire buffer consists of numbers and decimal points, to 1 if the first item is not a number or decimal, and to a value larger than 1 if the first item is a number which does not exend through the entire LINE buffer. The sequence of tests starting at line 27 thus extract either the entire line (i=0), the first character of the line (i=1), or the first portion of the line (i>1).

Taking the example above, LINE is immediately set to

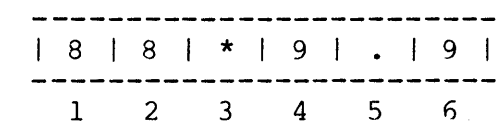
```
A>b:fscan
   88+9.9
 '88' '+' '9.9'
   1234567    89.10
 '1234567' '89.10'
  1,2,3,4,5,6,7
 '1' ',' '2' ',' '3' ',' '4' ',' '5' ',' '6' ',' '7'
 ....666... 7.7.7.
 '....666...' '7.7.7.'
^Z

END OF FILE (7), File: SYSIN=CON
Traceback: 08AE 23FF 0143 00FF # 08B8 06C6 0143 01F5
End of Execution
```

Figure 8-4. Interaction with the FSCAN Test Program.



where the index 1 through 9 into LINE is shown below each character. The initial blanks are stripped off on line 23, leaving LINE as

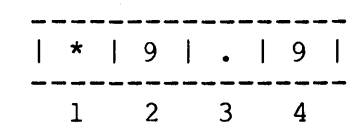


The VERIFY on line 24 locates the first position containing a non-digit or period character, resulting in the value 3 which corresponds to the "*" in position 3. As a result of a series of tests, line 33 is executed, producing the equivalent of

```
substr('88*9.9',1,2)
```

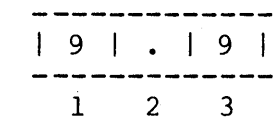
which results in a TOKEN value of '88' which is the next number in LINE.

On the next call to GNT, TOKEN is removed from LINE using the SUBSTR operation on line 16, leaving LINE as



The VERIFY function on line 26 returns the value 1, since the leading position of LINE is not a digit or a period. The first character of LINE is extracted and returned as the value of TOKEN on line 31.

The third call to GNT gets the last token in LINE by first extracting the "*" which leaves LINE as



The VERIFY on line 26 returns Ø since all characters are either digits or periods, and thus line 28 is executed, resulting in a TOKEN value of '9.9' which is the remainder of LINE.

The fourth call to GNT clears the previous value of TOKEN from LINE, leaving LINE as the empty string. This action, in turn, causes the GET EDIT statement to be executed, and refills LINE from the console. Execution proceeds in this manner until the operator aborts the program with either a control-z or control-c input.

This program is of no particular interest in itself, but, as we shall see later, it is easily incorporated into a more comprehensive

and useful function. In any case, the scanner has some drawbacks. Like our previous example, no end of file conditions are trapped. An ON-unit could be included to detect this condition, and a null TOKEN value could be returned to indicate that no more input is available. Further, the scanner makes no checks to detect multiple period characters which would cause a subsequent conversion signal (ERROR(1)) if any attempt is made to convert to a decimal value. It is a worthwhile exercise at this point to add these functions in the simplest possible form.

9. APPLICATIONS OF LIST PROCESSING.

PL/I-80 has subroutines which are included in every PL/I-80 program for dynamic storage management. When your program loads into memory for execution, the first action which takes place is to set up all remaining free storage as a linked-list structure. Your program can dynamically allocate pieces of this storage area through the ALLOCATE statement, and later release segments using the FREE statement. All storage management is done in the background, using subroutines from the PL/I-80 library. Segments of memory allocated in this manner can be logically connected to one another through "lists" which lead from one memory segment to another. The list elements are PL/I structures which contain information fields and one or more POINTER variables which provide access to other list elements.

The dynamic nature of list processing is most often used when the number and structure of data elements managed by a program varies considerably. The programs of this section illustrate the use of list processing in two cases where the data allocation is not easily predetermined. Each program is discussed in detail in order to provide a series of concrete examples of PL/I-80 list processing.

9.1. Managing a List of Words.

The first example performs a function similar to the OPTIMIST of the previous section. Recall that the length of a sentence accepted by the OPTIMIST was restricted to 254 characters, which is the maximum string length. In order to accept sentences of virtually any length, we will use a list structure instead of a single character string. For illustration, we will simplify the task somewhat by simply reversing the input sentence, rather than performing word substitution.

Before we get into the details, there are a few mechanical things to consider about list processing. First, a based variable is just a template that fits over a region of memory and has no storage directly allocated to it. The based variable template is programmatically placed over a particular piece of memory using a pointer variable in one of two ways, depending upon the form of the based variable declaration. If no implied base is included in the declaration, then any reference to the based variable must be pointer-qualified. If an implied base is included in the declaration, then references may include a pointer qualifier or simply use implied pointer given in the declaration as a base. An example should clarify the situation. Consider the following declarations:

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

The two variables i and mat are not based variables, so storage is

allocated for these data items. Similarly, the two pointer variables p and q have assigned storage locations. The three variables x, y, and z, however, are declared as based variables whose actual storage addresses will be determined during execution of the program. The variable x has no implied base, so all references to x must have a pointer qualifier, such as

```
p->x = 5;    and    q->x = 6;
```

In the first case, the fixed (two-byte) variable at the memory location given by p is assigned the value 5, while the second statement stores the value 6 at the location given by q. The variable y, on the other hand, has an implied base and can be referenced with or without a pointer qualifier. The reference

```
y = 5;  is equivalent to   p->y = 5;
```

and thus

```
y = 5;    and    q->y = 6;
```

have exactly the same effects as the two assignments to x shown above.

The variable z, like y, has an implied base. The base, in this case, is an invocation of a pointer-valued function with no arguments. The function f could, for example, take the form

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

Two valid references to z, for example, are

```
p->z = 5;     and     z = 6;
```

The first form is equivalent to those shown above, where the location is derived from the pointer variable p. The second form, however, is an abbreviation for

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

In this case, the function f is evaluated to produce the storage address for the based variable z. The advantage to using this form is twofold. First, the pointer valued expression can be a complex form, not restricted to a simple pointer variable. Second, the code for the function f occurs only once in memory, rather than being duplicated at each variable reference, thus saving a considerable amount of program space. It must also be noted in passing, that the implied base must be in the scope of the declaration for the based variable. The following non-functional program segment illustrates this notion:

```
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;
```

The declarations of p and q within the BEGIN block have no effect since the based variables x and y reference p and q which must be in the same or encompassing scope.

Now that the basics have been discussed, we can get into the list processing example. The sentence reversing program, called REV, is shown in Figure 9-1. The program is broken into three main sections. The first portion, from line 3 through 14 performs the function of reading a sentence and writing the sentence back to the console in reverse order. In order to simplify the overall program structure, the read function is performed by a separate subroutine, called READ, which starts on line 16. Similarly, the program output is performed in a third section by the subroutine named WRITE, beginning on line 34. Each input sentence consists of a sequence of words, up to 30 characters in length, sufficient to hold

floccinaucinihilipilification

which is the longest word in the English language (plus one letter because the Author is probably wrong about it being the longest word). In order to simplify the input processing, REV requires a space before the period which terminates the sentence. The program terminates when an empty sentence is typed by the operator. Figure 9-2 shows the console interaction with the REV program.

The REV program generally operates as follows. Each word is stored in a separate area of memory created using an ALLOCATE statement. Each ALLOCATE statement obtains a unique section of the free memory sufficiently large to hold the "wordnode" structure on line 5, amounting to 32 bytes for each allocation (you can verify this by examining the symbol table resulting from the $S compiler switch). The wordnode elements are linked together through the "next" field of each allocation, and the beginning of the list is given by the value of the "sentence" pointer variable.

In PL/I-80, pointer variables are really just two-byte 16-bit words which hold the address of a variable. The statement

allocate wordnode set (newnode)

```
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;
```

Figure 9-1. Listing of the REV Sentence Reverser.

```
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
```

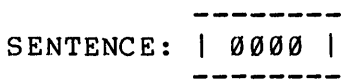
Figure 9-2. Interaction with the REV Program.

for example, finds a segment of memory to hold the 33 byte wordnode data item, and sets the value of the pointer variable newnode to the address of this memory area (wordnode is 33 bytes because the varying string "word" requires one byte to hold the current length, 30 bytes to hold the string itself, and is followed by a two byte pointer value). Given the input sentence

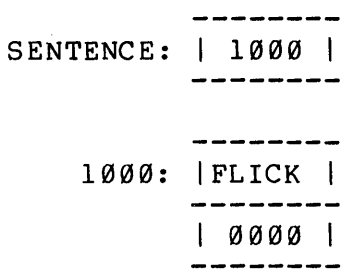
FLICK YOUR BIC

for example, the ALLOCATE statement is executed three times, once for each word in the list. For illustration, assume these three memory allocations are found at addresses 1000, 2000, and 3000. The REV program reads the sentence in the main loop within the READ procedure.

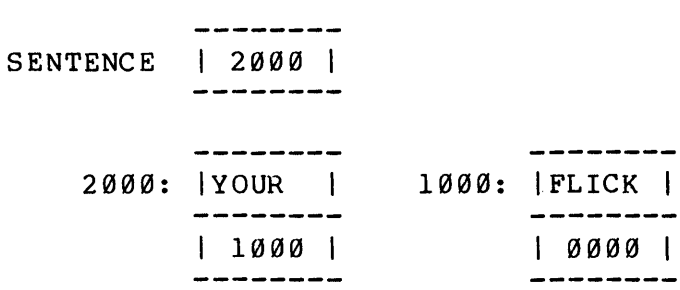
REV begins by initializing the sentence pointer to the null address which, by the way, is just address 0000. Upon entering the DO group at line 23, the value of sentence appears as shown below.



The first word, FLICK, is read by the GET statement at line 24 and, since the value is not a period, the first 33 byte area is allocated to hold the word. As the sentence is constructed, the pointer value of the sentence variable is placed into the "next" field and the input word is stored into the "word" field. The most recently read word then becomes the new head of the list. After processing the word FLICK, the list appears as shown below.

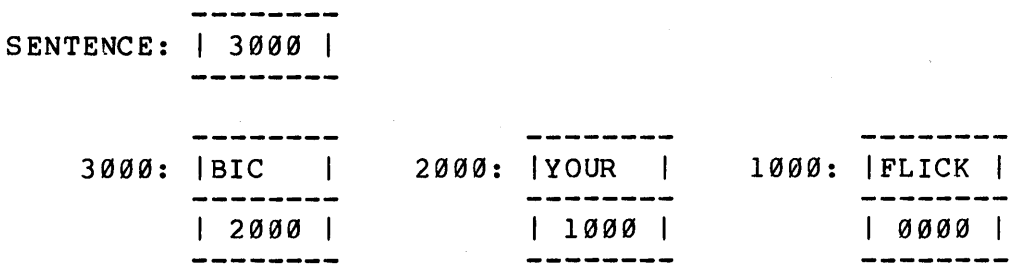


The program then proceeds through the loop again. This time, the word YOUR is read and the second 33 byte area is allocated. The newly allocated area becomes the new head of the list, with the resulting pointer structure:



The last word, BIC, is then processed. The final 33 byte

area is allocated and placed at the head of the list, with the resulting sequence of nodes:



Note that the program can follow the pointer structure from the sentence variable to the node for BIC, then to the node for YOUR, and finally to FLICK where an end of list (0000) value is encountered.

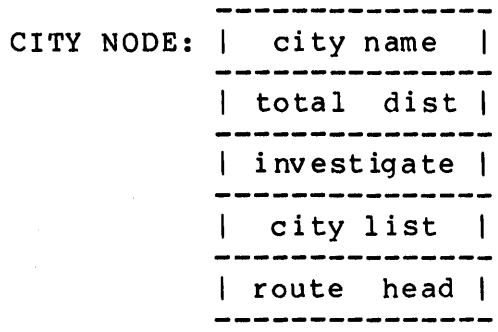
Due to the order in which the list was constructed, there is really no processing required to reverse the list. In fact, the WRITE function simply searches through the list, starting at the sentence pointer and prints each word as it is encountered within the loop between lines 39 and 44. As soon as the word has been written, the 33 byte area allocated to it is released using the FREE statement on line 43. It is important to note that the sentence pointer variable is moved to the next item in the list before FREEing the current element since, in general, you cannot expect storage to remain intact after it has been released.

In this situation, the principal advantage of the list structure is that the sentence can be arbitrarily long, limited only by the size of available memory. The disadvantage is that there is considerably more storage consumed for sentences which could be represented by a 254 character string.

9.2. A Network Analysis Program.

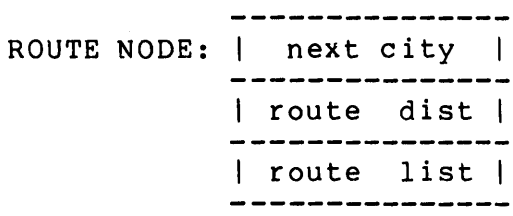
The next example is, without doubt, the most comprehensive program in this manual. The NET program, shown in Figure 9-3, performs the following function. The operator enters a network of cities and distances between these cities. The NET program constructs a connected set of network nodes, implemented using PL/I-80 list processing, which represents the graph. Upon demand from the console, the NET program computes the shortest path from all cities in the network to the designated destination, and then selectively displays particular optimal paths through the network.

NET uses two structure forms as list elements. The first corresponds to a particular city, called a "city node," and is defined on line 11 in Figure 9-3. The structure of a city node is shown graphically below.



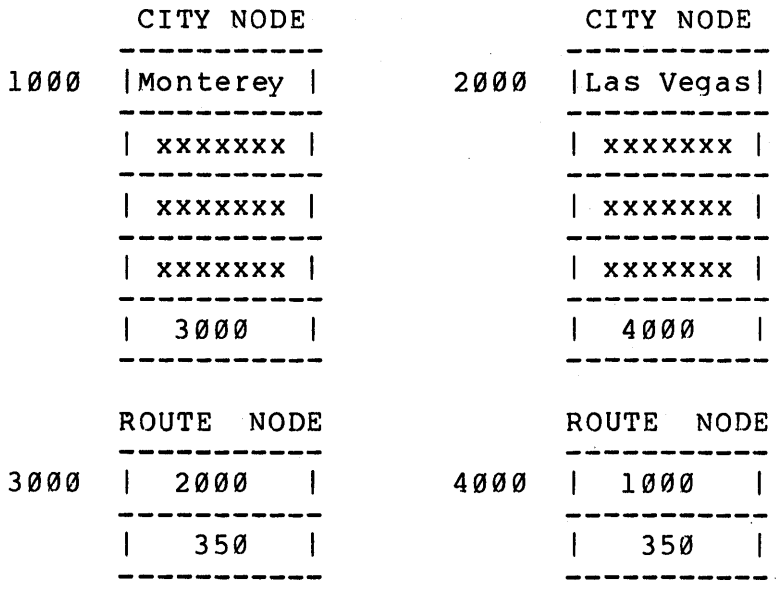
The "city name" field holds the character value of the city name itself, while the "total dist" and "investigate" fields are used during the shortest path algorithm, and are of no consequence at this point. The "city list" and "route head" pointer values are used to connect the cities in the network.

The second structure is called a "route node," and is defined on line 18. A route node establishes a single connection between one city and one of its neighbors. Normally, several route nodes are allocated for a city, corresponding to a number of connections to its neighboring cities. The structure of a single route node is

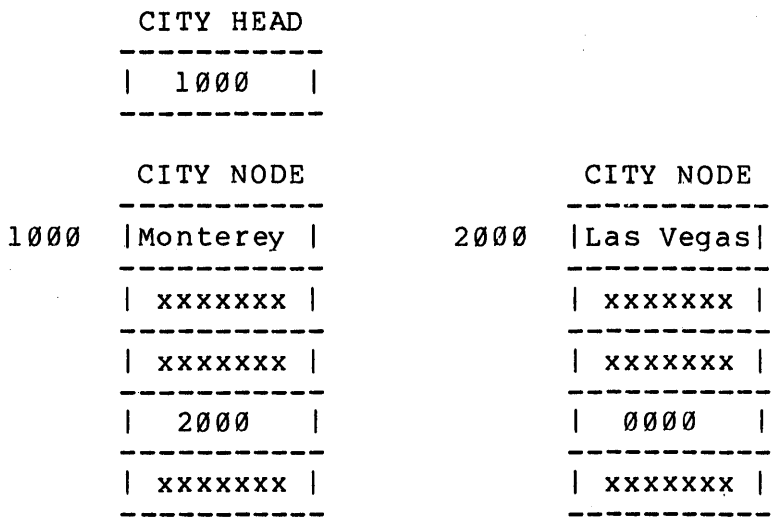


The list of route nodes associated with a particular city begins at the pointer value "route head" which is a part of the city node shown above, and continues through the "route list" pointer to additional route nodes, until a null route list is encountered. Each route node has a pointer value, denoted by "next city," which leads to a neighboring city node, along with a "route dist" field that gives the mileage to the next city.

For illustration, assume Monterey is 350 miles from Las Vegas. Two city nodes and two route nodes must be allocated to represent the graph, with the sample addresses shown to the left of each allocation:



where the "x" fields are ignored in the diagram. A linked list, starting at "city head" on line 23, leads to all cities in the network. Given the two cities shown above, the list of cities appears as



Before getting into the details of the NET program, we should note that one particular form of an iterative DO group is used throughout the program to traverse the linked lists. The program statement on line 142 is typical:

```
do p->cityhead repeat (p->citylist) while (p^=null);
```

The effect of this iterative DO group header is to successively process each element of the linked list starting at cityhead until a null (0000) link is encountered. On the first iteration, the pointer variable p is set to the value of the pointer variable cityhead, resulting in the assignment p = 1000 in the example shown above. On

the next iteration, p takes on the value of the citylist field at 1000 which addresses Las Vegas, resulting in the value p = 2000. On the last iteration, p takes on the value of the citylist field based at 2000, resulting in p = 0000. Since p is equal to null, the DO group execution is stopped.

In order to understand the operation of the NET program, take a look at the console interaction given in Figure 9-4. The operator first enters a list of cities and distances between cities, terminated by the end of file character control-z. The control-z response triggers a display of the entire network to aid in detecting input errors. The operator is then prompted for a destination city (Tijuana) and a starting city (Boise). A best route is displayed (there may be several of equal length), and then a prompt appears for another starting city. If a control-z is entered, the NET program reverts to another destination prompt, leaving the network intact. Interaction proceeds in this manner until the operator enters a control-z in response to the destination prompt. When this occurs, NET clears the network and returns to accept an entirely new network of cities and distances. The NET program terminates if an empty network is entered at this time (i.e., a single control-z is typed).

Given this background, we can now discuss the program structure. NET is logically divided into three parts: the input section which constructs and echoes the city network, the shortest path analysis, and the shortest path display operations. The input section consists of four procedures beginning on line 34, named "setup," "connect," "find," and "print all." The shortest path analysis takes place within the "shortest dist" procedure starting on line 122, while the display function is split between the two procedures "print paths" and "print route" at lines 103 and 169, respectively. The last procedure, "free all," is called to clear the old network before a new network is loaded. The top level program calls occur in the DO group between lines 25 and 32. The remainder of the program consists entirely of the nested subroutines named above.

Beginning on line 26, the main program calls "setup" to read the graph. If the city list is empty upon return, the program terminates on line 28. Otherwise, "print all" is called to display the graph, followed by "print paths" to prompt and display shortest routes. Upon return, "free all" is called to release storage. This process continues until an empty graph is entered.

The main loop within "setup" occurs between lines 43 and 47. On each iteration, a pair of cities with a connecting distance is read on line 44. The "connect" subroutine is then called twice to establish the connection in both directions between the cities. Note that the termination condition is intercepted by the ON-unit at line 39.

The "connect" subroutine, in turn, is responsible for establishing a single route node to connect the first city to the second city. The action of "connect" is to call the "find" procedure twice, once for the first city and once for the second city. The

"find" procedure will locate a city if it exists in the network, or create the city node if it does not yet exist. Upon return from "find," the route node is created and filled-in between lines 61 and 65. In the previous Monterey to Las Vegas example, the first call to "connect" would establish the city nodes for Monterey and Las Vegas (indirectly, through "find") and then produce the route node under Monterey only. The second call to "connect" establishes the route node under Las Vegas.

The "find" procedure, starting at line 68, searches the city list, beginning at "city head," until the input city is found or the city list is exhausted. If the input city is not found, it it created between lines 79 and 85. In any case, "find" returns a pointer to the requested city node.

The "print all" procedure, called after the network is created, appears between lines 88 and 101. This procedure starts at "city head" and displays all the cities in the city list. As each city is visited, the route head is also traversed and displayed. Upon completion of the "print all" procedure, all city nodes and route nodes have been visited and displayed.

The "print paths" subroutine performs the essential processing. A destination city is read on line 110, and sent to the "shortest dist" procedure. Upon return, the "total dist" field of each city node has been set to the total distance from the destination city. The operator enters the starting city on line 115, which is sent to the "print route" subroutine for the display operation.

Ignoring the shortest path analysis for now, the "print route" procedure, at line 169, is responsible for displaying the best route from the input city to the destination. The procedure essentially rediscovers the path as follows, recalling that the total distance from the input city to the destination has been computed and stored in "total dist." The first leg of the best route is discovered by finding a neighboring city whose "total dist" field differs by exactly the distance to the neighbor. The neighbor is displayed, we move to the neighboring city, and repeat the same operation. Eventually, the destination city is reached, and the display operation is completed. Mechanically, the original city node is found on line 176. The remaining distance is displayed on line 186, and the search for the first or next leg occurs between lines 188 and 197. On each iteration, line 191 tests to determine if a neighbor has been found whose total distance plus leg distance matches the current city. If so, the leg distance is displayed on line 193, and the search is terminated by setting q to null.

Now we'll go back to the "shortest dist" procedure on line 123. Basically, the function of this procedure is to take an input city, called the destination, and compute the minimum total distance from every city in the network to the destination. This total is recorded at each city node in the "total dist" field. The algorithm proceeds as follows

(a) Initially mark all total distances with infinity (32767 in PL/I-80), to indicate that the node has no connection (yet).

(b) Set the "investigate" flag to false for each city. The investigate flag marks a city node which needs to be processed further by the algorithm.

(c) Set the total distance to the destination to the value zero, all others are currently set to infinity, but will change during processing. Set the investigate flag to true for the destination only.

(d) Examine the city list for the city node which has the least total distance, and whose investigate flag is true (at first, only the destination will be found). When no city node has a true investigate flag, processing is complete and all minimum total distances have been computed.

(e) Clear the investigate flag for the city found in (d), and extract its current total distance value. Examine each of its neighbors: if the current total distance plus the leg distance is less than the total distance marked at the neighbor, then replace the neighbor's total distance by this sum, and mark the neighbor for processing by setting its investigate bit to true. After processing each neighbor, return to step (d).

Basically, the algorithm proceeds through the graph, developing the shortest path to any node. As a result, each city will be visited exactly once. Due to this fact, the algorithm is linear which means that additional nodes in the network do not significantly affect the time taken to analyze the graph.

The final procedure, "free all," starting at line 201 gives all the network storage back at the end of processing each network. Each city node is visited, the entire list of route node connections is discarded, and the city node is discarded.

The NET program can be expanded in several ways. First, it is inconvenient to type an entire network each time, so a simple addition would be to open a STREAM file and read the graph from disk. Several networks could be stored on the disk and retrieved on command from the console. Second, you could expand the basic functions of NET. If you are an aviation buff, for example, you might recast the network to model the national airway system consisting of electronic airway transmitters (VOR's and VORTAC's) which are placed throughout the

country. Directions, distances, and VOR frequencies can be stored at each node, along with additional local information. Under command from the console, the NET program would select the best route to take you from one place to another, and display the headings, distances, and VOR information for use during the trip. You could even expand further by switching between networks at the world, national, and local levels to aid in complete flight planning. If you're not an aviation fan, try some other application, such as a telephone exchange or computer program branch structure analysis.

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

   1 a 0000 graph:
   2 a 0006     proc options(main);
   3 c 0006     %replace
   4 c 0006         true     by '1'b,
   5 c 0006         false    by '0'b,
   6 c 0006         citysize by 20,
   7 c 0006         infinite by 32767;
   8 c 0006     dcl
   9 c 0006         sysin file;
  10 c 0006     dcl
  11 c 0006         1 city_node based,
  12 c 0006           2 city_name   char(citysize) var,
  13 c 0006           2 total_dist  fixed,
  14 c 0006           2 investigate bit,
  15 c 0006           2 city_list   ptr,
  16 c 0006           2 route_head  ptr;
  17 c 0006     dcl
  18 c 0006         1 route_node based,
  19 c 0006           2 next_city   ptr,
  20 c 0006           2 route_dist  fixed,
  21 c 0006           2 route_list  ptr;
  22 c 0006     dcl
  23 c 0006         city_head ptr;
  24 c 0006
  25 c 0006         do while(true);
  26 c 0006         call setup();
  27 c 0009         if city_head = null then
  28 c 0011             stop;
  29 c 0014         call print_all();
  30 c 0017         call print_paths();
  31 c 001A         call free_all();
  32 c 0020         end;
  33 c 0020
  34 c 0020     setup:
  35 c 0020         proc;
  36 e 0020         dcl
  37 e 0027             dist fixed,
  38 e 0027             (city1, city2) char(citysize) var;
  39 e 0027         on endfile(sysin) go to eof;
  40 e 0035         city_head = null;
  41 e 003B         put skip list('Type "City1, Dist, City2"');
  42 e 0057         put skip;
  43 e 0068             do while(true);
  44 e 0068             get list(city1, dist, city2);
  45 e 009C             call connect(city1, dist, city2);
  46 e 00A2             call connect(city2, dist, city1);
  47 e 00B6             end;
  48 e 00B6         eof:
  49 c 00B6         end setup;
```

Figure 9-3a. Listing of the NET Network Program Part A.

```
 51 c 00B6     connect:
 52 c 00B6         proc(source_city, dist, dest_city);
 53 e 00B6         dcl
 54 e 00C3             source_city char(citysize) var,
 55 e 00C3             dist fixed,
 56 e 00C3             dest_city   char(citysize) var;
 57 e 00C3         dcl
 58 e 00C3             (r, s, d) ptr;
 59 e 00C3         s = find(source_city);
 60 e 00D2         d = find(dest_city);
 61 e 00E1         allocate route_node set (r);
 62 e 00EA         r->route_dist = dist;
 63 e 00FB         r->next_city  = d;
 64 e 0106         r->route_list = s->route_head;
 65 e 011D         s->route_head = r;
 66 c 012D         end connect;
 67 c 012D
 68 c 012D     find:
 69 c 012D         proc(city) returns(ptr);
 70 e 012D         dcl
 71 e 0134             city char(citysize) var;
 72 e 0134         dcl
 73 e 0134             (p, q) ptr;
 74 e 0134             do p = city_head
 75 e 0142                 repeat(p->city_list) while(p^=null);
 76 e 0142             if city = p->city_name then
 77 e 0150                 return(p);
 78 e 0165             end;
 79 e 0165         allocate city_node set(p);
 80 e 016E         p->city_name  = city;
 81 e 017B         p->city_list  = city_head;
 82 e 018A         city_head     = p;
 83 e 0190         p->total_dist = infinite;
 84 e 019C         p->route_head = null;
 85 e 01A9         return(p);
 86 c 01AD         end find;
 87 c 01AD
 88 c 01AD     print_all:
 89 c 01AD         proc;
 90 e 01AD         dcl
 91 e 01AD             (p, q) ptr;
 92 e 01AD         do p = city_head
 93 e 01BB             repeat(p->city_list) while(p^=null);
 94 e 01BB         put skip list(p->city_name,':');
 95 e 01E0             do q = p->route_head
 96 e 01F6                 repeat(q->route_list) while(q^=null);
 97 e 01F6             put skip list(q->route_dist,'miles to',
 98 e 0253                           q->next_city->city_name);
 99 e 0253             end;
100 e 0253         end;
101 c 0253     end print_all;
102 c 0253
```

Figure 9-3b. Listing of the NET Network Program Part B.

```
103 c 0253     print_paths:
104 c 0253         proc;
105 e 0253         dcl
106 e 025A             city char(citysize) var;
107 e 025A         on endfile(sysin) go to eof;
108 e 0268             do while(true);
109 e 0268             put skip list('Type Destination ');
110 e 0284             get list(city);
111 e 029E             call shortest_dist(city);
112 e 02A4             on endfile(sysin) go to eol;
113 e 02B2                 do while(true);
114 e 02B2                 put skip list('Type Start ');
115 e 02CE                 get list(city);
116 e 02E8                 call print_route(city);
117 e 02F8                 end;
118 e 02F8             eol: revert endfile(sysin);
119 e 030E             end;
120 e 030E         eof:
121 c 030E         end print_paths;
122 c 030E
```

Figure 9-3c. Listing of the NET Network Program Part C.

```
123 c 030E      shortest_dist:
124 c 030E          proc(city);
125 e 030E          dcl
126 e 0315              city char(citysize) var;
127 e 0315          dcl
128 e 0315              bestp ptr,
129 e 0315              (d, bestd) fixed,
130 e 0315              (p, q, r) ptr;
131 e 0315              do p = city_head
132 e 0323                  repeat(p->city_list) while(p^=null);
133 e 0323              p->total_dist  = infinite;
134 e 032F              p->investigate = false;
135 e 0346              end;
136 e 0346          p = find(city);
137 e 0355          p->total_dist  = 0;
138 e 0360          p->investigate = true;
139 e 0368              do while(true);
140 e 0368              bestp = null;
141 e 036E              bestd = infinite;
142 e 0374                  do p = city_head
143 e 0382                      repeat(p->city_list) while(p^=null);
144 e 0382                  if p->investigate then
145 e 038E                      do;
146 e 038E                      if p->total_dist < bestd then
147 e 03A3                          do;
148 e 03A3                          bestd = p->total_dist;
149 e 03B1                          bestp = p;
150 e 03C8                          end;
151 e 03C8                      end;
152 e 03C8                  end;
153 e 03C8              if bestp = null then
154 e 03D0                  return;
155 e 03D1              bestp->investigate = false;
156 e 03DA                  do q = bestp->route_head
157 e 03EF                      repeat(q->route_list) while(q^=null);
158 e 03EF                  r = q->next_city;
159 e 03F9                  d = bestd + q->route_dist;
160 e 040B                  if d < r->total_dist then
161 e 0421                      do;
162 e 0421                      r->total_dist  = d;
163 e 0430                      r->investigate = true;
164 e 044D                      end;
165 e 044D                  end;
166 e 044D              end;
167 c 044D          end shortest_dist;
```

Figure 9-3d. Listing of the NET Network Program Part D.

```
 168 c 044D
 169 c 044D      print_route:
 170 c 044D          proc(city);
 171 e 044D          dcl
 172 e 0454              city char(citysize) var;
 173 e 0454          dcl
 174 e 0454              (p, q) ptr,
 175 e 0454              (t, d) fixed;
 176 e 0454          p = find(city);
 177 e 0463              do while(true);
 178 e 0463              t = p->total_dist;
 179 e 0471              if t = infinite then
 180 e 047D                  do;
 181 e 047D                  put skip list('(No Connection)');
 182 e 0499                  return;
 183 e 049A                  end;
 184 e 049A              if t = 0 then
 185 e 04A3                  return;
 186 e 04A4              put skip list(t,'miles remain,');
 187 e 04CB              q = p->route_head;
 188 e 04D9                  do while(q^=null);
 189 e 04E1                  p = q->next_city;
 190 e 04EB                  d = q->route_dist;
 191 e 04F7                  if t = d + p->total_dist then
 192 e 0515                      do;
 193 e 0515                      put list(d,'miles to',p->city_name);
 194 e 0540                      q = null;
 195 e 0549                      end; else
 196 e 0549                  q = q->route_list;
 197 e 055D                  end;
 198 e 055D              end;
 199 c 055D          end print_route;
 200 c 055D
 201 c 055D      free_all:
 202 c 055D          proc;
 203 e 055D          dcl
 204 e 055D              (p, q) ptr;
 205 e 055D              do p = city_head
 206 e 056B                  repeat(p->city_list) while(p^=null);
 207 e 056B                  do q = p->route_head
 208 e 0581                      repeat(q->route_list) while(q^=null);
 209 e 0581                  free q->route_node;
 210 e 0598                  end;
 211 e 0598              free p->city_node;
 212 e 05B0              end;
 213 c 05B0          end free_all;
 214 c 05B0
 215 a 05B0      end graph;

CODE SIZE = 05B0
DATA AREA = 00EB
```

Figure 9-3e. Listing of the NET Network Program Part E.

```
A>b:net

Type "City1, Dist, City2"
Seattle, 150, Boise
Boise, 300, Modesto
Seattle,400,Modesto
Modesto,150,Monterey
Modesto, 50,San-Francisco
San-Francisco,200,Las-Vegas
Las-Vegas,350,Monterey
Los-Angeles,400,Las-Vegas
Bakersfield,300,Monterey
Bakersfield,250,Las-Vegas
Los-Angeles,450,Tijuana
Tijuana, 700,Las-Vegas
Las-Vegas, 920, Boise
Pacific-Grove,5,Monterey
^Z

Pacific-Grove :
        5 miles to Monterey
Tijuana :
      700 miles to Las-Vegas
      450 miles to Los-Angeles
Bakersfield :
      250 miles to Las-Vegas
      300 miles to Monterey
Los-Angeles :
      450 miles to Tijuana
      400 miles to Las-Vegas
Las-Vegas :
      920 miles to Boise
      700 miles to Tijuana
      250 miles to Bakersfield
      400 miles to Los-Angeles
      350 miles to Monterey
      200 miles to San-Francisco
San-Francisco :
      200 miles to Las-Vegas
       50 miles to Modesto
Monterey :
        5 miles to Pacific-Grove
      300 miles to Bakersfield
      350 miles to Las-Vegas
      150 miles to Modesto
Modesto :
       50 miles to San-Francisco
      150 miles to Monterey
      400 miles to Seattle
      300 miles to Boise
Boise :
      920 miles to Las-Vegas
      300 miles to Modesto
      150 miles to Seattle
Seattle :
      400 miles to Modesto
      150 miles to Boise
```

Figure 9-4a. NET Program Network Setup.

```
Type Destination Tijuana

Type Start Boise

     1250 miles remain,        300 miles to Modesto
      950 miles remain,         50 miles to San-Francisco
      900 miles remain,        200 miles to Las-Vegas
      700 miles remain,        700 miles to Tijuana
Type Start ^Z

Type Destination Pacific-Grove

Type Start Seattle

      555 miles remain,        400 miles to Modesto
      155 miles remain,        150 miles to Monterey
        5 miles remain,          5 miles to Pacific-Grove
Type Start ^Z

Type Destination ^Z

Type "City1, Dist, City2"
^Z
```

Figure 9-4b. NET Program Network Interrogation.

10. USES OF RECURSION IN PL/I-80.

Recursion processing is a language facility often used to simplify programming problems which are partially self-defined. We will examine three such problems, where the first two illustrate the basic concepts and the last shows a more powerful use of recursion in a practical problem.

Getting into the mechanics of recursion processing for a moment, a recursive procedure is one in which an embedded call either directly or indirectly reenters the procedure before returning to the first level call. In PL/I, all such procedures must have the RECURSIVE attribute so that the local data areas are properly saved and restored at each level of recursion. In PL/I-80, there are two restrictions within recursive procedures. First, all procedure parameters are "call by value," which means that values cannot be returned from a recursive procedure by assignment to formal parameters. Instead, your program may return a functional value or assign values to global variables. In order to maintain compatibility with full PL/I, you should not use formal parameters on the left of an assignment statement within a PL/I-80 recursive procedure. Second, PL/I-80 does not allow embedded BEGIN blocks within recursive procedures. Nested procedures and DO groups are, however, allowed. The proper formulation of recursive procedures is shown in the examples which follow.

10.1. Evaluation of Factorials.

No introduction to recursion would be complete without a presentation of factorial evaluation. The factorial function, used throughout Mathematics, is a good illustration because it is easily defined through "iteration" as well as recursion. The iterative definition of the factorial function is

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

where k! is the factorial function applied to the non-negative integer k. Note that since

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

we can give the factorial definition in terms of itself, using the recursion relation

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

where we define

$$0! = 1$$

Evaluating the factorial function using either iteration or recursion

produces the values shown below

```
Ø! = 1
1! = 1
2! = (2) (1) = 2
3! = (3) (2) (1) = 6
4! = (4) (3) (2) (1) = 24
5! = (5) (4) (3) (2) (1) = 12Ø
```

Figure 1Ø-1 provides a listing of IFACT which computes values of the factorial function using iteration. The variable FACT is a fixed binary data item which accumulates the value of the factorial up to a maximum of 32767. The output from IFACT, shown in Figure 1Ø-2, gives the proper value of the factorial function up to 7! = 5Ø4Ø. At this point, the FACT variable overflows and produces improper results. Recall that PL/I-8Ø does not signal FIXEDOVERFLOW for binary computations since the overhead would significantly degrade execution time.

Figure 1Ø-3 shows the equivalent recursive evaluation of the factorial function. For comparison, the REPEAT form of the DO group is used to control the test. In this case, FACT is a procedure marked as RECURSIVE, which is called at the top level in the PUT statement on line 6, with an embedded recursive call in the RETURN statement on line 14. Note that FACT returns immediately when the input value is zero. All other cases require one or more recursive evaluations of FACT to produce the result. For example, 3! produces the sequence of computations

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

producing the final value 6. The output values for the recursive factorial evaluation are shown in Figure 1Ø-4. Note again that the values overflow beyond 5Ø4Ø due to the precision of the computations.

We can use this opportunity to examine output differences when the data item types and precisions are altered. Figure 1Ø-5 shows the recursive evaluation of factorial, where a maximum precision decimal value is used. The largest value produced by the program, as shown in Figure 1Ø-6, is

```
factorial(17) = 355,687,428,Ø96,ØØØ
```

At this point, the FIXEDOVERFLOW is signalled by PL/I-8Ø to indicate that the decimal computation has overflowed the maximum 15 digit value. Similarly, Figure 1Ø-7 shows the factorial function evaluated using floating point binary data items. The output from this program is shown in Figure 1Ø-8. Although the function can be computed beyond 17!, the number of significant digits is truncated on the right to

```
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
```

Figure 10-1. Listing of the IFACT Program.

```
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
```

Figure 10-2. Output from the IFACT Program.

```
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) recursive;
  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
```

Figure 10-3. Listing of Factorial in Fixed Binary.

```
A>b:fact

factorial(         Ø )=          1
factorial(         1 )=          1
factorial(         2 )=          2
factorial(         3 )=          6
factorial(         4 )=         24
factorial(         5 )=        12Ø
factorial(         6 )=        72Ø
factorial(         7 )=       5Ø4Ø
factorial(         8 )=     -25216
factorial(         9 )=     -3Ø336
factorial(        1Ø )=      24320
factorial(        11 )=       5376
factorial(        12 )=      -1Ø24
factorial(        13 )=     -13312
factorial(        14 )=      1Ø24Ø
factorial(        15 )=      22528
factorial(        16 )=     -32768
factorial(        17 )=     -32768
factorial(        18 )=          Ø
factorial(        19 )=          Ø
factorial(        2Ø )=          Ø
factorial(        21 )=          Ø
factorial(        22 )=          Ø
```

Figure 1Ø-4. Output from the FACT Program.

```
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
```

Figure 10-5. Listing of Factorial in Decimal.

```
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
```

Figure 10-6. Output from the DFACT Program.

```
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
```

Figure 10-7. Listing of Factorial in Float Binary.

```
A>b:ffact

factorial(          Ø )=   1.ØØØØØØE+ØØ
factorial(          1 )=   1.ØØØØØØE+ØØ
factorial(          2 )=   2.ØØØØØØE+ØØ
factorial(          3 )=   Ø.6ØØØØØE+Ø1
factorial(          4 )=   2.4ØØØØØE+Ø1
factorial(          5 )=   1.2ØØØØØE+Ø2
factorial(          6 )=   Ø.72ØØØØE+Ø3
factorial(          7 )=   Ø.5Ø4ØØØE+Ø4
factorial(          8 )=   4.Ø32ØØØE+Ø4
factorial(          9 )=   3.628799E+Ø5
factorial(         1Ø )=   3.628799E+Ø6
factorial(         11 )=   3.991679E+Ø7
factorial(         12 )=   4.79ØØ15E+Ø8
factorial(         13 )=   Ø.622702E+1Ø
factorial(         14 )=   Ø.871782E+11
factorial(         15 )=   1.3Ø7674E+12
factorial(         16 )=   2.Ø92278E+13
factorial(         17 )=   3.556874E+14
factorial(         18 )=   Ø.64Ø237E+16
factorial(         19 )=   1.216450E+17
factorial(         2Ø )=   2.432Ø1ØE+18
factorial(         21 )=   Ø.51ØØ9ØE+2Ø
factorial(         22 )=   1.124ØØØE+21
factorial(         23 )=   2.585201E+22
factorial(         24 )=   Ø.62Ø448E+24
factorial(         25 )=   1.551121E+25
factorial(         26 )=   4.Ø32914E+26
factorial(         27 )=   1.Ø88887E+28
factorial(         28 )=   3.Ø48883E+29
factorial(         29 )=   Ø.884176E+31
factorial(         3Ø )=   2.652528E+32
factorial(         31 )=   Ø.822283E+34
factorial(         32 )=   2.631ØØ8E+35
factorial(         33 )=   Ø.868331E+37
factorial(         34 )=
OVERFLOW (1)
Traceback: ØØ46 13BF 13E6 Ø19B # 86Ø8 ØB15 FB51 Ø141
End of Execution
```

Figure 1Ø-8. Output from the FFACT Program.

approximately 7-1/2 equivalent decimal digits. The floating point binary version terminates when the OVERFLOW condition is signalled by PL/I-80, produced by an exponent value which cannot be maintained in the floating point representation.

10.2. Evaluation of the Ackermann Function.

The PL/I-80 runtime system maintains a memory "stack" where subroutine return addresses and some temporary results are maintained. Under normal circumstances, the memory area allocated to hold the stack is sufficiently large for non-recursive procedure processing, as well as simple recursive procedure evaluation. The program of this section, however, illustrates a more comprehensive example of recursion using a function which is derived from Number Theory, called the Ackermann function. The Ackermann function, denoted by A(m,n), has the recursive definition:

```
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))
```

For our puposes, the Ackermann function illustrates multiple recursion using a stack depth which can exceed the default value provided by PL/I-80. The program, shown in Figure 10-9, implements the Ackermann function by reading two values for the maximum m and n for which the function is to be evaluated. The program interaction is given in Figure 10-10. It should be noted in passing that although the Ackermann function returns a fixed binary value, the DECIMAL built-in function is used to control the PUT LIST output conversion field size on lines 8, 10, and 12.

The important point in this example is that the STACK option is used on line 2 to increase the size of the memory area allocated for the runtime stack. The STACK option is only valid with the MAIN option and, in this case, increases the stack size from its default value of 512 bytes up to 2000 bytes. The value of the STACK option must be determined empirically, since the depth of recursion cannot generally be computed by the compiler. The message

```
Free Space Overwrite
```

however, occurs when the stack overflows during recursion, accompanied by program termination, as an indication that the allocated stack size is too small.

```
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 a 019F     end ack;

CODE SIZE = 019F
DATA AREA = 0048
```

Figure 10-9. Listing of the Ackermann Program.

```
A>B:ACK

Type max m,n: 4,6

          Ø     1     2     3     4     5     6
Ø :       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
```

Figure 10-10. Interaction with the Ackermann Program.

10.3. An Arithmetic Expression Evaluator.

One of the day-to-day practical uses of recursion takes place in the translation and execution of programming languages. This use is primarily due to the fact that languages are most often recursively defined. In block-structured languages like PL/I-80, for example, DO groups, and BEGIN and PROCEDURE blocks can be self-embedded, so the resulting structure is easily processed using recursion. Another example, which is the subject of this section, occurs in the evaluation of arithmetic expressions. One simple form of an expression can be recursively defined as follows.

> An expression is a simple number, or
> an expression is a pair of expressions
> separated by a +, -, *, or /, and
> enclosed in parentheses.

Using this definition, the number 3.6 is an expression since it is a simple number. Further,

```
( 3.6 + 6.4 )
```

is also an expression since it consists of a pair of expressions which are both simple numbers, separated by a +, and enclosed in parentheses. As a result,

```
( 1.2 * ( 3.6 + 6.4) )
```

is a valid expression because it also contains two valid expressions: 1.2 and (3.6+6.4), separated by a *, and enclosed in parentheses. The sequences

```
3.6 + 6.4
(1.2 + 3.6 + 6.4)
```

are not valid expressions since the first is not enclosed in parentheses, while the second is not a pair of expressions in parentheses. The definition of an expression as given above is somewhat restrictive, but once we have the foundation established it is easily expanded to include expressions of the complexity of, say, PL/I-80.

An expression evaluation program is shown in Figure 10-11. The principal processing takes place in this Figure between lines 20 and 24 where an expression is read from the console and the evaluated result is typed back to the operator. The console interaction is shown in Figure 10-12, where the operator has entered several properly and improperly formed expressions.

The heart of the expression analyzer is the recursive procedure EXP which processes input expressions according to the recursive definition given above. The EXP procedure decomposes the expression piece-by-piece as the recursion proceeds. The GNT (Get Next Token) procedure reads the next element, or "token," in the input line, which should be a left or right parenthesis, a number, or one of the

```
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('Value is:',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;
```

Figure 10-11a. Listing of an Expression Evaluator Part A.

```
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
```

Figure 10-11b. Listing of an Expression Evaluator Part B.

```
A>b:expr1

Type expression: ( 4 + 5.2 )
Value is:  Ø.92ØØØØE+Ø1

Type expression: 4.5e-1
Value is:  4.499999E-Ø1

Type expression: ( 4 & 5 )
Invalid Input at  &

Type expression: ( ( 3 + 4 ) - ( 1Ø / 8 ) )
Value is:  Ø.575ØØØE+Ø1

Type expression: ( 3 * 4 )
Value is:  1.2ØØØØØE+Ø1

Type expression: ^Z
End of Execution
```

Figure 10-12. Interaction with the Expression Evaluator.

arithmetic operators. Since GNT uses a GET LIST, each of these tokens must be separated by a blank or end of line. EXP begins by calling GNT on line 34. GNT, in turn, places the next input token into the CHAR(10) variable called token. If the first item read is a number, then the series of tests within EXP sends control to line 57 where the character value of token is automatically converted to a floating point value through the assignment to x. This converted value is returned from EXP back to line 22, where it is stored into "value" and subsequently written as the result of the expression. If the expression is non-trivial, then EXP scans the leading left parenthesis on line 35, and enters the DO group on line 36. The first subexpression is immediately evaluated, no matter how complicated, and stored into the variable x in line 37. The token is then checked for an occurrence of +, -, *, or /. Suppose, for example, that the * operator is encountered on line 45. The statement on line 46 recursively invokes the EXP procedure to evaluate the right side of the expression and, upon return, multiplies this result by the value of the left side which was previously computed. The balancing right parenthesis is checked starting on line 52, and the resulting product is returned as the value of EXP on line 58.

Exceptional conditions are handled in three places. An end of file condition on the input file is intercepted by the ON-unit at line 8, where a STOP statement is executed. A second point where an error can take place is during conversion from character to floating point at the assignment on line 57. If this occurs, the ON-unit starting at line 11 receives control. The token in error is displayed and the data is cleared to the end of line using a GET SKIP statement. Program control then recommences at the "restart" label where the operator is prompted for another input expression.

An exceptional condition is generated by the program itself when an invalid operator or unbalanced expression is encountered. If the operator is not a +, -, *, or /, then statement 51 is executed and the ON-unit at line 11 is signalled, resulting in an error report and transfer to "restart," where the current recursion levels are discarded and the program begins again. Similarly, a missing right parenthesis on line 53 triggers the error(1) ON-unit to report the error and restart the program.

The only major problem here is that the input line requires spaces between tokens, which is somewhat inconvenient. Recall, however, that we earlier tested a procedure called GNT (see Section 8.2) which reads console input lines and decomposes the line into numeric and single character tokens, without the necessity for intervening blanks. Figure 10-13 shows the expression processor of the previous figures with the GNT procedure replaced by the free-field scanner. The error recovery has also changed, since it is necessary on line 20 to discard the remainder of the current input when restarting the program. Figure 10-14 shows the console interaction with this improved expression analyzer.

There is plenty of room for expansion in this particular example. First, you could add more operators to expand upon the basic arithmetic functions. You might want to add operator precedence, and

do away with the requirement for explicit parentheses. Beyond that, you can add variable names and assignment statements and, who knows, with a bit of work you may turn the program into a Basic interpreter!

```
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)) varying
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
```

Figure 10-13a. An Expanded Expression Evaluator Part A.

```
 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 01FD              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
```

Figure 10-13b. An Expanded Expression Evaluator Part B.

```
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
```

Figure 10-14. Expanded Evaluator Console Interaction.

11. SEPARATE COMPILATION AND LINKAGE.

All of the programs presented thusfar are constructed as indivisible units, where many contain embedded local procedures. As mentioned previously, it is often useful to break larger programs into distinct modules which are subsequently linked with one another and with the PL/I-80 subroutine library. There are two reasons for separately compiling and linking programs in this manner. First, large programs take longer to compile and, in fact, may overrun the memory size available for the symbol table. Smaller segments can be independently developed, integrated, and tested, thus requiring less overall compilation time for the entire project. Second, you will soon identify particular subroutines which you find useful for your own application programming. You can build your own library of subroutines and selectively link them into your programs, as required. This section provides the basic information required to link program and data segments, and provides a complete example of separate compilation and linkage.

11.1. Data and Program Declarations.

Data areas can be shared in PL/I-80 by including the EXTERNAL attribute in the item's declaration. For example, the declaration

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

defines a variable named x occupying 10 fixed binary locations (20 contiguous bytes), which is accessible by any other module that uses this same declaration. Similarly,

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

defines a 20 byte data area named s which is accessible by other modules. There are a few basic rules which apply to the declaration of external data:

(a) An external data item automatically receives the STATIC attribute.

(b) EXTERNAL data items are accessible in any block in which they are declared, thus overriding scope rules for internal data.

(c) EXTERNAL data items must be unique in the first six (6) characters since the linkage editing format requires truncation from the seventh character on.

(d) All EXTERNAL data areas must be declared with the same length in all modules in which they appear.

(e) Avoid the use of "?" symbols in variable names, since this character is used as a prefix on PL/I-80 library names.

(f) One module, at most, can initialize an EXTERNAL data item referenced by several modules.

Similar to the label data described in Section 6, entry constants and entry variables are data items which identify procedure names and describe their parameter values. Entry constants correspond to procedures defined within the program (internal procedures) or at link-time (external procedures). Entry variables take on entry constant values during program execution, using either a direct assignment statement or an actual to formal parameter assignment implicit in a subroutine call. A procedure may be invoked through a call to an entry constant, or indirectly by calling a procedure constant value held by an entry variable. Similar to label variables, entry variables may be subscripted. The program listing shown in Figure 11-1 provides examples of entry constants and entry variables. This particular program contains four entry constants: the main program, labelled "call," the external procedure "g" declared on line 5, the "sin" function which is a part of the PL/I library, and the internal function "h" beginning on line 20. One entry variable is declared on line 4, called "f" which contains three elements. The individual subscript elements are initialized, starting on line 9, to the constants sin, g, and h. The DO-group prompts the console for a value to send to each function and, in the middle of line 16, each function is called exactly one time with the invocation

```
f(i)(x)
```

where the first parenthesis pair defines the subscript, and the second encloses the list of actual arguments. It should be noted that the declaration of entry constants and entry variables is similar to file constants and file variables: all formal parameters declared as type ENTRY are automatically assumed to be entry variables. In all other cases, an entry is constant unless it is declared with the VARIABLE keyword. Rules (b), (c), and (e) above apply to external procedure declarations. In addition, you must be careful to declare each formal parameter to exactly match the actual procedure declaration, and ensure that the RETURNS attribute exactly matches the form returned for function subroutines.

```
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
```

Figure 11-1. Use of ENTRY Variables and Constants.

11.2. An Example of Separate Compilation.

This section contains a complete example of separate compilation and linkage editing. In particular, the programs of Figures 11-2 and 11-3 together form a module that interacts with the console to produce solutions to systems of simultaneous equations. Consider the following system of three equations in three unknowns:

```
 a  -  b  +  c  =  2
 a  +  b  -  c  =  Ø
2a  -  b       =  Ø
```

The values a = 1, b = 2, and c = 3 yield a solution to this system of equations since

```
 1  -  2  +  3  =  2
 1  +  2  -  3  =  Ø
2*1 -  2       =  Ø
```

The listing shown in Figure 11-2 interacts with the console to read the coefficients and the solution vectors for the systems of equations, while the listing of Figure 11-3 shows the compilation of the separate subroutine "invert" which performs the matrix inversion that is used to solve the system of equations. The essential difference between these two programs is found in the procedure heading: the "inv" procedure is the main program, as defined by the OPTIONS(MAIN), while the "invert" program is a subroutine which is called by the main program. Referring to Figure 11-2, the declaration starting on line 16 defines the external entry constant "invert" which is called from the main program on line 46. The parameters for the invert subroutine are declared on line 18 as a matrix of "maxrow" by "maxcol" floating point numbers, where maxrow and maxcol are actually the literal constants given on lines 7 and 8. The invert subroutine is defined with two additional fixed(6) parameters, but does not return a value.

The invert procedure, shown in Figure 11-3, has three formal parameters, called a, r, and c, as defined on line 2 and declared in lines 7 and 8. It should be noted that the actual literal values of maxrow and maxcol, corresponding to the largest possible row and column value, are taken from an include file, as indicated by the "+" symbols following the line number at the left of both listings.

Following compilation of these two programs, the linking step is invoked by typing

```
link invert.com=invert1,invert2
```

which first combines these two modules, selects the necessary subroutines from the PL/I-8Ø library, and stores the resulting machine code into the "invert.com" file. Execution is started as shown in Figure 11-4.

In this sample interaction, the "identity" matrix is first

entered in order to test the basic operations. The inverse matrix produced for this input value is also the identity matrix. The system of equations shown above is then entered, along with two solution vectors. The output values for this system are shown under "Solutions:" and match the values shown above. The second set of solutions corresponds to the second solution vector input. An invalid input matrix size is then tested, followed by termination of the program as sensed by a zero row size.

This completes the examples of this applications guide. Additional information can be obtained from the accompanying PL/I-80 manuals, as well as the LINK-80 manual.

```
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;
```

Figure 11-2a. Listing of the Matrix Inversion Main Program.

```
 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
```

Figure 11-2b. Listing of the Matrix Inversion Main Program.

```
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,l) * 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
```

Figure 11-3. Listing of the Matrix Inversion Subroutine.

```
A>b:invert
Solution of Simultaneous Equations

Type rows, columns: 3,3

Type Matrix of Coefficients
Row      1 :1   Ø   Ø
Row      2 :Ø   1   Ø
Row      3 :Ø   Ø   1

Type Solution Vectors


Solutions:
a =
b =
c =

Inverse Matrix is
       1.ØØ    Ø.ØØ    Ø.ØØ
       Ø.ØØ    1.ØØ    Ø.ØØ
       Ø.ØØ    Ø.ØØ    1.ØØ

Type rows, columns: 3,5

Type Matrix of Coefficients
Row      1 :1   -1    1
Row      2 :1    1   -1
Row      3 :2   -1    Ø

Type Solution Vectors
Variable a :2    Ø    Ø
Variable b :3.5  1   -1


Solutions:
a =    1.ØØ    2.25
b =    2.ØØ    5.5Ø
c =    3.ØØ    6.75

Inverse Matrix is
       Ø.5Ø    Ø.5Ø    Ø.ØØ
       1.ØØ    1.ØØ   -1.ØØ
       1.5Ø    Ø.5Ø   -1.ØØ

Type rows, columns: 41,Ø

Matrix is Too Large

Type rows, columns: Ø

End of Execution
```

Figure 11-4. Interaction with the Matrix Inversion Program.

12. COMMERCIAL PROCESSING USING PL/I-80.

The purpose of this section is to familiarize you with some techniques used in PL/I-80 in processing commercial data. In particular, the various decimal arithmetic operations are described in some detail. Conversion between Fixed Decimal and Floating Point Binary is examined, including the use of the ftc (float to character) library function. The discussion also includes examples of picture formatted output, along with a presentation of precision and scale evaluation when using the four basic arithmetic functions with decimal operands. Four programs are presented which typify the use of decimal operations in actual applications.

12.1. A Comparison of Decimal and Binary Operations.

We have been taught from childhood to perform arithmetic operations using base ten arithmetic where the permissible digits range from 0 through 9. Further, application languages such as Basic, Fortran, Cobol, and PL/I allow us to write programs which process base ten constants and data items in simple and readable forms. Internally, however, computers generally perform the arithmetic operations using either binary or decimal numbers. Binary numbers are more "natural" for internal computer arithmetic since the 1's and 0's can be directly processed by the on-off electronic switches found in arithmetic processors. Because our programs generally process decimal values, it becomes necessary to convert into a binary form on input and back to a decimal form on output. As we shall see below, this conversion can introduce truncation errors which are unacceptable in commercial processing. Thus, decimal arithmetic is often required in order to avoid the propagation of errors throughout computations.

In most languages, the programmer has no control over the internal format used for numeric processing. In fact, two of the most popular Basic interpreters for microprocessors differ primarily in the internal number formats. One uses floating point binary, while the other performs calculations using decimal arithmetic. Pascal language translators generally use floating and fixed point binary formats with implementation-defined precision, while Fortran always performs arithmetic using floating or fixed point binary. Cobol, on the other hand, was designed for use in commercial applications where exact dollars and cents must be maintained throughout computations, and thus data items are processed using decimal arithmetic.

PL/I-80 gives the programmer the choice between representations so that each program can be tailored to the exact needs of the particular application. Fixed Decimal data items are used in PL/I-80 to perform commercial functions, while Float Binary items are used for scientific processing where computation speed is the most important factor. The two programs shown below illustrate the essential difference between the two computational forms:

```
dec_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;
```

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

The two programs perform the simple function of summing the value 3.10 a total of 10,000 times. The only difference between these programs is that "dec_comp" computes the result using a Fixed Decimal variable, while "bin_comp" performs the computation using Float Binary. Dec_comp produces the correct result 31000.00, while bin_comp produces the approximation 30997.30. The difference is due to the inherent truncation which occurs when certain decimal constants, such as 3.10, are converted to their binary approximations. Since no conversion occurs when Fixed Decimal variables are used, dec_comp produces an exact result.

These two programs can be considered simplifications of a more general situation: suppose Chase-Manhattan Bank has processed 10,000 deposits of $3.10 during a particular day. Using a program based upon Floating Binary, there would be an extra $2.70 unaccounted for at the end of the day (there have been cases where crooked systems programmers have been caught redirecting the "excess cash" produced by such errors into their own accounts!). This is due to the fact that .10 cannot be represented as a finite binary fractional expansion. That is, 3.10 is actually approximated as 3.099999E+00 in Float Binary form. Each addition propagates a small error into the sum, resulting in an incorrect total. In scientific applications, the inherent truncation errors are often insignificant and thus ignored. In commercial applications such inherent errors are unacceptable.

It should be noted that there are situations where decimal arithmetic also produces truncation errors which can propagate throughout computations. The expression 1/3, for example, cannot be represented as a finite decimal fraction, and thus is approximated as

0.3333333 ...

to the maximum possible precision. However, due to our life-long experience with decimal computations, we expect such errors to occur and adjust our programming to account for the situation. In fact, we know that such errors will only occur when explicit division operations take place. We expect that 1/10 will be represented exactly as .10, and not just a close approximation. But herein lies the difficulty with Float Binary representations: some decimal constants which can be expressed as finite fractional expansions in Fixed Decimal cannot be written as finite binary fractions and thus are necessarily truncated during conversion to Float Binary form.

With this introduction, we will now proceed to explain exactly how Fixed Decimal numbers are represented and manipulated.

12.2. Decimal Computations in PL/I-80.

Fixed Decimal arithmetic can be performed in PL/I-80 programs. There are both advantages and disadvantages in selecting Fixed Decimal arithmetic when contrasted to Floating Point formats. First, Fixed Decimal arithmetic guarantees that there will be no loss of significant digits. That is, all digits are considered significant in a computation so that multiplication, for example, will not truncate digits in the least-significant positions. Further, Fixed Decimal arithmetic precludes the necessity for exponent manipulation, and thus the operations are relatively fast when compared to alternative decimal arithmetic formats. The disadvantage, however, is that since all digits are considered significant, the programmer must keep track of the range of values that arithmetic operands can take on. The paragraphs which follow provide the necessary background to properly program using Fixed Decimal formats.

Decimal variables and constants in PL/I-80 have both "precision" and "scale." Precision denotes the number of digits in the variable or constant, while scale defines the number of digits in the fractional part. For Fixed Decimal variables and constants, the precision must not exceed 15 and the scale must not exceed the precision. The precision and scale of a PL/I-80 variable is defined in the variable's declaration:

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

while the precision and scale of a constant are derived by the compiler by counting the number of digits in the constant, and the number of digits following the decimal point. The constant

```
-324.76
```

for example, has precision 5 and scale 2. Internally, Fixed Decimal variables and constants are stored as Binary Coded Decimal (BCD) pairs, where each BCD digit occupies either the high or low order 4-bits of each byte. The most significant BCD digit defines the sign of the number or constant, where 0 denotes a positive number, and 9 defines a negative number in 10's complement form, as described below. Since numbers are always stored into 8-bit byte locations, there may be an extra "pad" digit at the end of the number to align to an even byte boundary. The number 83.62, for example, is stored as

```
-------------------
| 0 8 | 3 6 | 2 0 |
-------------------
```

where each digit represents a 4-bit "half byte" position in the 8-bit

value. The leading BCD pair is stored lowest in memory.

Negative numbers are stored in 10's complement form to simplify arithmetic operations. A 10's complement number is similar to a 2's complement binary representation, except the complement value of the digit x is 9-x. To derive the 10's complement value of a number, form the complement of each digit (by subtracting the digit from 9), and add 1 to the final result. Thus, the 10's complement of -2 is formed as follows:

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

The sign digit is attached to this number, and internally carried as the single-byte value

```
-------
| 9 8 |
-------
```

Note, for example, that you can add -2 and +3 as follows

```
98 + 03 = 101
```

The carry-out beyond the sign digit is ignored, and the correct result 01 is produced through the addition. For this reason, addition and subtraction in PL/I-80 are equivalent: in the case of subtraction, the subtrahend is first complemented and the addition operation is applied. In all cases, numeric values are sign-extended to 15 digits before arithmetic operations are applied. For convenience of notation, negative numbers will be shown with a leading "-" sign, with the assumption that the underlying representation is 10's complement form. Thus, the number shown above will be written as

```
-------
| - 2 |
-------
```

It should be noted that there is no need to explicitly store the decimal position in memory, since the precision and scale for each variable and constant is known by the compiler. Before each arithmetic operation, the compiled code causes the necessary alignment of the operands. In later examples, however, a decimal point position is often shown in order to more easily determine the effect of alignment. The number -324.76 may be shown, for example, as

```
-------------------
| - 3 | 2 4 | 7 6 |
-------------------
          ^
```

When this value is prepared for arithmetic processing, it is first loaded into an 8-byte stack frame, consisting of 15 decimal digits with a high-order sign. In this case, the -324.76 is shown as

```
- 0 0 0 0 0 0 0 0 0 0 3 2 4 7 6
                       ^
```

A convenient model for discussing the various arithmetic operations is to visualize a 15-digit mechanical or electronic calculator with a hand-movable decimal point. At the beginning of each operation, you must properly line-up the operands for the arithmetic operation and, upon completion of the operation, you must decide where the resulting decimal point appears. Actually, the compiler performs the alignment and accounts for the decimal point position, but it's useful for you to imagine what is taking place so that you can avoid overflow or underflow conditions. In some cases, you may wish to force a precision and/or scale change during the computation using the DECIMAL or DIVIDE built-in functions. Examples of such functions are given in the sample programs discussed in the sections which follow.

First, we'll examine each of the arithmetic functions in order to determine the alignment, precision, and scale which occurs in each case.

12.3. Addition and Subtraction.

As mentioned above, addition and subtraction are functionally equivalent in PL/I-80, since subtraction is accomplished by forming the 10's complement of the subtrahend and then performing an addition. Given two numbers x and y with precision and scale (p,q) and (r,s), respectively, the addition operation proceeds as follows. First, the two operands are loaded to the stack and aligned. Alignment takes place by shifting the operand with the smaller scale to the left until the decimal positions are the same. Given that the scale of x is greater than the scale of y, y is shifted q-s positions to the left, with zero values introduced in the least signficant positions. After alignment, y has precision r+(q-s) and scale q. (A Fixed Overflow condition is signalled if significant digits are shifted into the sign position during the alignment process.)

In order to provide a specific example, suppose x = 31465.2437 and y = 9343.412 so that x has precision p = 9 and scale q = 4, while y has precision r = 7 and scale s = 3. Before alignment, the numbers appear as

```
                          |<----- 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 0 9 3 4 3 4 1 2
                          |       ^       |
                          |       |<s=3>  |
                          |<--- r=7 --->  |
```

The value y is aligned with x by shifting q-s = 4-3 = 1 positions to the left, producing

```
                          |<----- 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 >|
```

Note that the number of digits in the whole part of x is p-q, while the whole part of y contains r-s digits:

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

so the sum must contain p-q=5 digits in the whole part:

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

Note, however, that sufficiently large values could produce an overflow, requiring one extra digit in the whole part:

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

Thus, the total number of digits in the sum of x and y is the number of digits in the whole part, (p-q)+1=6, plus the number of digits in the fraction, given by q, resulting in a precision of

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

Given two values x and y of arbitrary precision and scale, we can use the specific case shown above to derive the general form of the resulting precision and scale. First, the scale must be the

greater of q and s, given by

$$\max(q,s)$$

and thus, the resulting precision must have max(q,s) fractional digits. Second, the whole part x contains p-q digits, while the whole part of y contains r-s digits. The result contains the larger of p-q and r-s digits, plus the fractional digits, along with one overflow digit, or a total of

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

digit positions. Since the precision cannot exceed 15 digits in PL/I-80, the resulting precision must be

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

digits. Written as a pair, the precision and scale of the resulting addition or subtraction is

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

Using the above example,

```
                              |<------ 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 >|
          ---------------------------------
x + y =   + 0 0 0 0 0 0 0 4 0 8 0 8 6 5 5 7
                                  ^           |
                                  |<- 4 ->|   |
                              |<------ 10 ------->|
```

the precision (10,4) shown in the diagram is derived using the expression

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

or

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

12.4. Multiplication.

Evaluation of precision and scale for the result of multiplication is somewhat simpler than addition and subtraction since

no decimal point alignment is required before the multiplication is applied. The two operands x and y with precision and scale (p,q) and (r,s), respectively, are multiplied digit-by-digit to produce the result. Similar to simple hand-calculations, the number of decimal places in the result is the sum of the scale factors q and s. The number of digits in the result is the sum of the precisions of the two operands. To conform to the PL/I standard, however, one additional digit position is included in the final precision. Thus, the precision and scale of the result of multiplication is given by

```
(min(15,p+r+1),q+s)
```

Suppose, for example, that x = 924.5 and y = 862.33, yielding the precision and scale values (4,1) and (5,2):

```
x = + 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 8 6 2 3 3
                            ^
```

The product of the digits of x and y are shown below with the resulting precision and scale:

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

where the precision is computed as

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

The Fixed Overflow condition is signalled if the product contains more than fifteen significant digits. In the example of the previous section, x = 31465.2437 and y = 9343.412. The product x*y has precision (17,7) with 16 significant digits, resulting in Fixed Overflow. In this particular case, the DECIMAL function must be applied to reduce the number of significant digits in either x or y. The computation could be carried out as

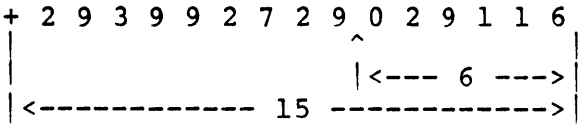
```
DECIMAL(x,9,3) * y
```

which loads the stack with the two values shown below before the multiplication takes place:

```
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
                                  ^
```

The precision and scale of the product is



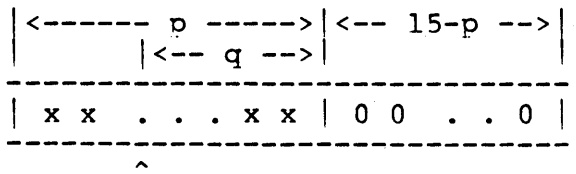
Note that the precision computation p+r+1 produces the value 16 which is then reduced to PL/I-80's maximum 15 digit precision by

```
min(15,p+r+1) = min(15,16) = 15
```

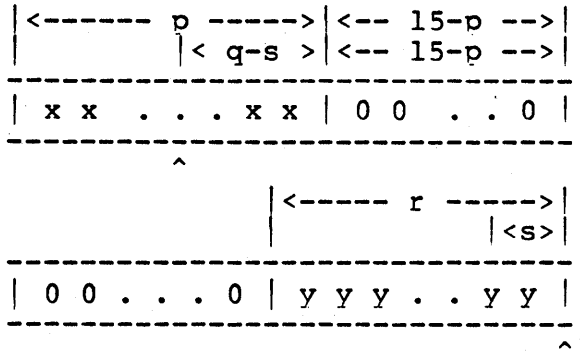
Since the precision of computations involving multiplications can grow rapidly, it is the responsibility of the programmer to ensure that the precisions of the operands involved will not produce overflow. Again, precision can be explicitly declared with the variables involved in the computation, or the DECIMAL function can be applied to reduce the precision of a temporary result.

12.5. Division.

The division operation is the only one of the four basic arithmetic operations which may produce truncation errors, as described in Section 12.1. Thus, each division operation produces a maximum precision value, consisting of 15 decimal digits, with a resulting scale which depends upon the scale values of the two operands. Assume that x and y have precision (p,q) and (r,s), and that x is to be divided by y. The division operation proceeds as follows. First, x is shifted to the extreme left by introducing 15-p zero values on the right, leaving the dividend in the stack as



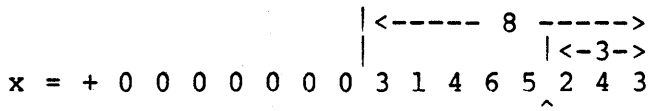
The decimal point of x is then effectively shifted right by an amount s to properly align the decimal point in the result, producing the operands



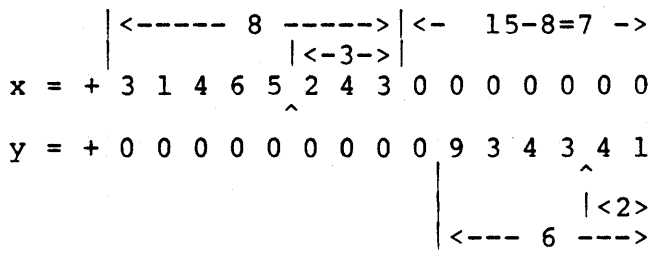
The significant digits of x are then continuously divided by the significant digits of y until 15 decimal digits are generated. Referring to the above diagram, note that the number of fractional digits produced by the division is determined by the placement of the adjusted decimal point in x. The field following the decimal point contains (q-s) plus (15-p) positions, yielding the following precision and scale for the result of the division

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

Suppose x = 31465.243, and y = 9343.41, yielding precision and scale values of (8,3) and (6,2), respectively. The value x when loaded appears as



The value of x is then shifted to the extreme left and the value of y is loaded, producing the values



The imaginary decimal points are shifted to the right by two positions in order to properly align the decimal point in the result, producing

```
          |<----- 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 --->|
```

The significant digits of x are divided by the six significant digits of y, and the result is

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

In this case, the precision and scale of the result is given by

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

The most important consideration in decimal division is to ensure that you are generating enough digits in the fractional part for the computation you are performing. Fractional digits are produced in two ways. First, the zero padding which occurs when the dividend is aligned provides 15-p fractional digits, so that dividend values with small precision generate more fractional digits. Second, if q is greater than s, then (q-s) additional fractional digits are generated as shown above. If, on the other hand, the dividend contains fewer fractional digits than the divisor then q is less than s, and (s-q) fractional digits are consumed. The simple case of q = s occurs quite often. In this particular situation, the number of fractional digits depends entirely upon the precision of the divisor, and results in 15-p fractional digits.

You may also wish to truncate or extend the result with zeroes using the DIVIDE built-in function during a particular computation (see the PL/I-80 Language Manual). The form is

DIVIDE (x,y,p,q)

where p and q are literal constants, can appear as an expression or subexpression in an arithmetic computation, and has the same effect as the statement

DECIMAL (x/y,p,q)

As above, the value x is divided by y, but the precision and scale values are forced (p,q). Note that the computation is carried out as described above, and the resulting value is then shifted by the appropriate number of digits in order to obtain the desired precision and scale.

12.6. Conversion Between Fixed Decimal and Float Binary.

It is often useful to convert Fixed Decimal values to and from a Float Binary representation. In PL/I-80, this conversion is accomplished by first converting to character format, then to either Fixed Decimal or Float Binary. Although conversion from Fixed Decimal, then to Character, and finally to Float Binary is provided directly in the language, a special library function, call "ftc," is provided for conversion from Float Binary to Character format. This particular function is useful in other applications, and is described fully in this section.

Consider the following program as an example of conversion between data formats:

```
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(c);
   f = 0.314159265e1;
   d = ftc(f);
   end conv;
```

In this example, the Fixed Decimal value d is first initialized to 123456.78. Next, the CHAR built-in function is applied to the Fixed Decimal value to produce a character string constant

```
'b-123456.78'
```

where "b" is a blank character. (Recall from the PL/I-80 Language Manual that conversion from Fixed Decimal to character produces a string of length p+3, consisting of leading blanks, a sign position, and digits of the number itself.) The store operation following the character conversion effectively converts from Fixed Decimal to Float Binary with possible truncation errors due to conversion to binary, as discussed previously. Next, the value of Pi is stored into the Float Binary value f. Normally, an assignment from f into d causes truncation of the fractional part, since the PL/I standard first requires conversion to Fixed Binary. Instead, the ftc function is applied to f to produce the character string variable of length 17:

```
'b3.14159200000000'
```

where the blank character, represented by b, is inserted if the number is positive, and "-" is included if the value is negative. The subsequent store operation into d produces a truncated value of 3.14, due to d's declared scale value of two decimal places. It should be noted that Float Binary representation allows approximately 7-1/2 significant decimal digits, and thus truncation errors may occur as

the conversions take place.

Additional examples of conversion between Fixed Decimal and Float Binary are given in the programs described below.

12.7. A Simple Loan Payment Schedule.

The first example of commercial processing is found in Figure 12-1. This program computes a loan payment schedule using three input values corresponding to the loan principal (P), the yearly interest rate (i), and monthly payment (PMT). Each month, the remaining principal is computed as

$$P + i * P$$

and is then reduced by the payment amount, producing a new principal for the next month:

$$P = (P + i * P) - PMT$$

The program iterates through the statements from line 18 through line 31 until the principal is reduced to zero, and the loan is completely paid off.

We assume in this program that the principal does not exceed $999,999,999.99, and thus the declaration on line 6 defines P as a Fixed Decimal variable with precision 11 and scale 2. Further, we shall assume that the payment does not exceed $9,999.99, so PMT is declared with precision 6 and scale 2. Finally, the interest rate is defined with the Fixed Decimal(4,2) attribute allowing numbers as large as 99.99%. The two variables "m" and "y" correspond to the month and year, beginning at the first month of the first year.

The initial values are read between lines 10 and 15. Note that for this example, no range checking is performed and thus negative values are acceptable, and payment values can be processed which would never pay off the loan. These checks must be made, of course, to be useful in an application environment.

On each monthly iteration, the month is incremented with possible overflow past the 12th month which changes the year value (lines 19 through 24). The current principal P is displayed on line 25, and the monthly interest is added on the following line. The computation on line 26 is evaluated as follows:

```
 1 a 0000 pmt:
 2 a 0006      proc options(main);
 3 c 0006      dcl
 4 c 0006          m   fixed binarv,
 5 c 0006          y   fixed binary,
 6 c 0006          P   fixed decimal(11,2),
 7 c 0006          PMT fixed decimal(6,2),
 8 c 0006          i   fixed decimal(4,2);
 9 c 0006          do while('1'b);
10 c 0006          put skip list('Principal ');
11 c 0022          get list(P);
12 c 0041          put list('Interest  ');
13 c 0058          get list(i);
14 c 0077          put list('Payment   ');
15 c 008E          get list(PMT);
16 c 00AD          m = 0;
17 c 00B3          y = 0;
18 c 00B6              do while (P > 0);
19 c 00CC              if mod(m,12) = 0 then
20 c 00DF                  do;
21 c 00DF                  y = y + 1;
22 c 00E6                  put skip list('Year',y);
23 c 010D                  end;
24 c 010D              m = m + 1;
25 c 0114              put skip list(m,P);
26 c 0142              P = P + round( i * P / 1200, 2);
27 c 0182              if P < PMT then
28 c 0198                  PMT = P;
29 c 01A8              put list(PMT);
30 c 01C6              P = P - PMT;
31 c 01E7              end;
32 c 01E7          end;
33 a 01E7      end pmt;
```

Figure 12.1. Simple Loan Payment Program Part A.

```
B>pmta

Principal 500
Interest  14
Payment   22.10

Year          1
         1              500.00      22.10
         2              483.73      22.10
         3              467.27      22.10
         4              450.62      22.10
         5              433.78      22.10
         6              416.74      22.10
         7              399.50      22.10
         8              382.06      22.10
         9              364.42      22.10
        10              346.57      22.10
        11              328.51      22.10
        12              310.24      22.10
Year          2
        13              291.76      22.10
        14              273.06      22.10
        15              254.15      22.10
        16              235.02      22.10
        17              215.66      22.10
        18              196.08      22.10
        19              176.27      22.10
        20              156.23      22.10
        21              135.95      22.10
        22              115.44      22.10
        23               94.69      22.10
        24               73.69      22.10
Year          3
        25               52.45      22.10
        26               30.96      22.10
        27                9.22       9.33
Principal ^C
```

Figure 12.1. Simple Loan Payment Program Part B.

```
    i              has precision and scale (4,2)
    P              has precision and scale (11,2)
  i * P            results in Fixed Decimal(15,4)
  1200             has precision and scale (4,0)
(i * P)/1200       has precision (15,4), since
                   precision and scale in division
                   is computed as (15,15-15+4-0)
```

The division by 1200 is required since the interest rate is expressed as a percentage (division by 100) over a one year period (division by 12). The intermediate result is ROUNDed in the second decimal place (cents position), and added to the principal. This result becomes the new principal.

In the last month of payment, it is likely that the remaining principal is less than the payment. The test on line 27 accounts for this possibility and, if so, changes the payment to equal the principal on line 28. The payment is printed on line 29 and, finally, the principal is reduced by the payment on line 30 using the assignment

```
P = P - PMT
```

The output from this program is shown following the program listing in Figure 12-1, with an initial loan of $500, interest rate 14%, and payment of $22.10 per month.

12.8. Ordinary Annuity.

Given the interest rate (i) and two of three values, the annuity program listed in Figure 12-2 computes either the present value (PV), payment (PMT), or number of periods (n). This particular program illustrates the use of several commercial processing facilities of PL/I-80, including a mix of Floating Point and Fixed Decimal arithmetic, along with picture format output.

Unlike the program of the previous section, the annuity program computes the unknown value through static formulas, rather than iteration. The static formulas are given below, assuming the interest rate is greater than zero. First, the present value is given by:

$$PV = PMT \frac{1 - \frac{1}{(1+i)^n}}{i}$$

and, by transposing the above formula, PMT can be computed as

```
 1 a 0000 annuity:
 2 a 0006     proc options(main);
 3 c 0006     %replace
 4 c 000D         clear by '^z',
 5 c 000D         true  by '1'b;
 6 c 000D     dcl
 7 c 000D         PMT fixed decimal(7,2),
 8 c 000D         PV  fixed decimal(9,2),
 9 c 000D         IP  fixed decimal(6,6),
10 c 000D         x   float binary,
11 c 000D         yi  float binary,
12 c 000D         i   float binary,
13 c 000D         n   fixed;
14 c 000D     dcl
15 c 000D         ftc entry (float binary) returns (char(17) var);
16 c 000D
17 c 000D     put list (clear,'^i^iO R D I N A R Y   A N N U I T Y');
18 c 002F     put skip (2) list
19 c 004B         ('^iEnter Known Values, or 0, on Each Iteration');
20 c 004B
21 c 004B     on error
22 d 0052         begin;
23 e 0055         put skip list('^iInvalid Data, Re-enter');
24 e 0071         go to retry;
25 d 0074         end;
26 d 0074
27 c 0074     retry:
28 c 007B         do while (true);
29 c 007B         put skip(3) list
30 c 0097             ('^iPresent Value ');
31 c 0097         get list(PV);
32 c 00B6         put list('^iPayment       ');
33 c 00CD         get list(PMT);
34 c 00EC         put list('^iInterest Rate ');
35 c 0103         get list(yi);
36 c 011E         i = yi / 1200;
37 c 012F         put list('^iPay Periods   ');
38 c 0146         get list(n);
39 c 015E
40 c 015E         if PV = 0 | PMT = 0 then
41 c 0190             x = 1 - 1/(1+i)**n;
42 c 01B3
43 c 01B3         if PV = 0 then
44 c 01C9             do;
45 c 01C9             /* compute present value */
46 c 01C9             PV = PMT * dec(ftc(x/i),15,6);
47 c 01FD             put edit('^iPresent Value is ',PV)
48 c 022C                 (a,p'$$$,$$$,$$$V.99');
49 c 022C             end;
50 c 022C
51 c 022C         if PMT = 0 then
52 c 0242             do;
53 c 0242             /* compute payment */
54 c 0242             PMT = PV * dec(ftc(i/x),15,8);
55 c 0276             put edit('^iPayment is ',PMT)
```

Figure 12.2. Ordinary Annuity Program Part A.

```
56 c 02A5                   (a,p'$$,$$$,$$$V.99');
57 c 02A5               end;
58 c 02A5
59 c 02A5           if n = 0 then
60 c 02AE               do;
61 c 02AE               /* compute number of periods */
62 c 02AE               IP = ftc(i);
63 c 02C1               x = char(PV * IP / PMT);
64 c 02EF               n = ceil ( - log(1-x)/log(1+i) );
65 c 032C               put edit('^i',n,' Pay Periods')
66 c 0362                   (a,p'ZZZ9',a);
67 c 0362               end;
68 c 0362           end;
69 a 0362       end annuity;


                O R D I N A R Y   A N N U I T Y

        Enter Known Values, or 0, on Each Iteration


        Present Value 32000
        Payment       0
        Interest Rate 8.75
        Pay Periods   360
        Payment is       $251.74


        Present Value ,
        Payment       0
        Interest Rate ,
        Pay Periods   240
        Payment is       $282.78


        Present Value 0
        Payment       ,
        Interest Rate ,
        Pay Periods   ,
        Present Value is     $31,998.87


        Present Value 32000
        Payment       ,
        Interest Rate ,
        Pay Periods   0
         240 Pay Periods


        Present Value ^C
```

Figure 12.2. Ordinary Annuity Program Part B.

$$PMT = PV \frac{i}{1 - (1 + i)^{-n}}$$

Finally, n is evaluated using:

$$n = - \frac{\text{Log}\left(1 - PV\left(\frac{i}{PMT}\right)\right)}{\text{Log}(1 + i)}$$

The program contains one main loop between lines 28 and 67 where the present value, payment, and yearly interest are read from the console. The operator must enter two non-zero values and one zero value on each iteration. The program then computes the value of the variable which was entered as zero. The values are retained on each main loop so that a comma (,) entry can be entered if the value is not to be changed. The interest rate, expressed as a yearly percentage, is reduced to a monthly period on line 36, where it is divided by 12 * 100 = 1200. Again, the program does not check for input values in the proper range. The interaction with the annuity program is shown following the program listing, with several different values used as input.

This particular program uses both Float Binary and Fixed Decimal computations since there is a mixture of simple decimal arithmetic and analytic functions. The variables used throughout the program are defined between lines 7 and 13 as follows. PMT holds the payment value, and is defined as a Fixed Decimal number as large as $99,999.99. Similarly, the present value can be as large as $99,999,999.99. The variable IP is used to hold the interest rate for a one month period, represented as a Decimal fraction with six decimal places. The variables x, yi and i are Float Binary numbers which are used during the computations to approximate decimal numbers with about 7-1/2 decimal places. Finally, the Fixed Binary variable n holds the number of payment periods, ranging from 1 to 32767.

Referring to the above formulas, the computation

```
1 - 1 / ( 1 + i ) ** n
```

occurs in both the computation of PV and PMT. Thus, line 41 stores this value into the variable x for subsequent use if either PV or PMT is to be evaluated. Again, it is important to realize that x is only an approximation to the decimal value given by this expression. If the operator enters a zero value for PV, then the statements between lines 45 and 49 are executed. In this case, PV is computed using the "ftc" external subroutine, defined on line 15, as

```
PV = PMT * dec(ftc(x/i),15,6)
```

where x/i is a Float Binary computation, and ftc converts the resulting value from float to character form. Given that x/i produces the value 3.042455E+01, for example, ftc(x/i) results in 30.42455 which is acceptable for conversion to decimal. The ERROR(1) condition is signalled by ftc, indicating a conversion error, if the floating point argument cannot be converted to a 15-digit decimal number. The "dec" function is applied to the character string to convert to a specific precision (15) and scale (6) for the subsequent multiplication. How did we decide on this particular value for precision and scale? First, consider a simpler form of this program which is shown below

```
dcl
    PMT fixed decimal(7,2),
    PV  fixed decimal(9,2),
    Q   fixed decimal(u,v);
PV = PMT * Q;
```

where we must decide upon the appropriate constant values for u and v. PV has precision and scale (9,2) and thus there must be 7 digits in the whole part and 2 digits in the fraction. We will generate the full 7 digits in the whole part if the product PMT * Q results in any of the following precision and scale values

(9,2) (10,3) (11,4) (12,5) (13,6) (14,7) (15,8)

since the assignment to PV will truncate any fractional digits beyond the second decimal place. Further, since PMT has precision and scale (7,2), we can choose (15,6) as the precision and scale of Q to produce

(min(15,7+15+1),2+6) = (15,8)

as the precision and scale resulting from the rules for multiplication stated previously. In general, given an expression with precision and scale values as shown below

```
  a     =     b     *     c
(p,q)       (r,s)       (u,v)
```

where p, q, r, and s are constants, you can set the precision and scale of c to

u = 15 v = 15 - p + q - s

which, using the values in the above statement, results in

v = 15 - 9 + 2 - 2 = 8, or (u,v) = (15,6)

as the precision and scale of Q.

Returning to the sample program in Figure 12-2, the resulting present value PV is written using a picture format with a drifting dollar sign on line 48.

Alternatively, the operator could have entered a non-zero

present value with a zero value for the payment (PMT). In this case, the group beginning at line 57 is entered, and the value of PMT is computed:

```
PMT = PV * dec (ftc(i/x),15,8);
```

using essentially the same technique as shown in the previous computation. Again, we must decide the precision and scale of the second operand in the multiplication. (We are really concerned only with the value of the scale since the precision can be taken as 15.) Using the analysis shown above, the form is

```
   a     =     b     *     c
(7,2)       (9,2)       (15,v)
```

where

```
v = 15 - p + q - s = 15 - 7 + 2 - 2 = 8
```

The computed value of PMT is written with the a picture format on line 56.

The final case occurs when the operator enters non-zero values for PV and PMT, but sets the number of periods to zero. When this occurs, the group beginning on line 60 is executed to compute n. First, the interest for a monthly period is changed from Float Binary to Fixed Decimal using the assignment on line 62. The next assignment

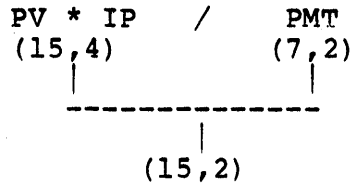
```
x = char (PV * IP / PMT)
```

first computes the partial Decimal result PV * IP / PMT, then converts the result to character, and then to Float Binary through the assignment to x. The intermediate character form is necessary since otherwise the intermediate result would first be converted to Fixed Binary, then to Float Binary, resulting in truncation of the fraction. (This sequence of conversions is necessary to maintain compatibility with the full language.)

First, we'll analyze the precision and scale of the Decimal computation. The subexpression PV * IP produces the following:

```
 PV      *      IP
(9,2)         (7,2)
  |             |
  ---------------
         |
      (15,4)
```

The computation proceeds with the division, producing the following precision and scale:



since, according to the precision and scale rules for division,

(15,15-p+q-s) = (15,15-15+4-2) = (15,2)

thus providing two decimal places in the computation. Additional fractional digits can be generated by applying the decimal function following the multiply, as shown below

x = char(dec(PV*P, 11,4) / PMT)

which would produce a quotient with precision and scale

(15,15-11+4-2) = (15,6)

The resulting value, x, is used in the expression on line 64 to compute the number of payment periods. The CEIL function is applied to the result so that any partial month becomes a full month in the payment period analysis. The number of months is written using a picture format with leading zero suppression, and the program loops for another set of input values.

12.9. Formatted Loan Payment Schedule.

The program shown in Figure 12-3 is essentially the same as that presented in Section 12.7, with a more elaborate analysis and display format. As shown starting on line 116, this program reads several data items:

PV	Present Value (Initial Principal)
yi	Yearly Interest Rate
PMV	Monthly Payment
ir	Yearly Inflation Rate
sm	Starting Month of Payment (1-12)
sy	Starting Year of Payment (0-99)
fm	Fiscal Month (End of Fiscal Year, 1-12)
dl	Display Level (0-2)

The initial principal and payment variables are declared as Fixed Decimal (10,2) on lines 16 and 19, allowing values as large as $99,999,999.99. The yearly interest rate and yearly inflation rate are expressed as percentages as large as 99.99, as defined on lines 24 and 29. The month and year variables, sm, sy, and fm are in Fixed Binary format, and are assumed to properly represent month and year values. The variable dl defines the amount of information displayed

```
 1 a 0000 pmt:
 2 a 0006     proc options(main);
 3 c 0006     %replace
 4 c 000D         true   by '1'b,
 5 c 000D         false  by '0'b,
 6 c 000D         clear  by '^z';
 7 c 000D     dcl
 8 c 000D         end bit(1),
 9 c 000D         m   fixed binary,
10 c 000D         sm  fixed binary,
11 c 000D         y   fixed binary,
12 c 000D         sy  fixed binary,
13 c 000D         fm  fixed binary,
14 c 000D         dl  fixed binary,
15 c 000D         P   fixed decimal(10,2),
16 c 000D         PV  fixed decimal(10,2),
17 c 000D         PP  fixed decimal(10,2),
18 c 000D         PL  fixed decimal(10,2),
19 c 000D         PMT fixed decimal(10,2),
20 c 000D         PMV fixed decimal(10,2),
21 c 000D         INT fixed decimal(10,2),
22 c 000D         YIN fixed decimal(10,2),
23 c 000D         IP  fixed decimal(10,2),
24 c 000D         yi  fixed decimal(4,2),
25 c 000D         i   fixed decimal(4,2),
26 c 000D         INF fixed decimal(4,3),
27 c 000D         ci  fixed decimal(15,14),
28 c 000D         fi  fixed decimal(7,5),
29 c 000D         ir  fixed decimal(4,2);
30 c 000D
31 c 000D     dcl
32 c 000D         name char(14) var static init('$con'),
33 c 000D         output file;
34 c 000D
35 c 000D     put list(clear,'^i^iS U M M A R Y   O F   P A Y M E N T S');
36 c 002F
37 c 002F     on undefinedfile(output)
38 d 0037         begin;
39 e 003A         put skip list('^i^icannot write to',name);
40 e 005F         go to open_output;
41 d 0062         end;
42 d 0062
43 c 0062     open_output:
44 c 0069         put skip(2) list('^i^iOutput File Name ');
45 c 0085         get list(name);
46 c 009F
47 c 009F     if name = '$con' then
48 c 00AD         open file(output) title('$con') print pagesize(0);
49 c 00CC     else
50 c 00CC         open file(output) title(name) print;
51 c 00E6
52 c 00E6     on error
53 d 00ED         begin;
54 e 00F0         put skip list('^i^iBad Input Data, Retry');
55 e 010C         go to retry;
```

Figure 12.3. Summary of Loan Payments Program Part A.

```
56 d 010F          end;
57 d 010F
58 c 010F      retry:
59 c 0116          do while(true);
60 c 0116          put skip(2)
61 c 0132              list('^i^iPrincipal      ');
62 c 0132          get list(PV);
63 c 0151          P = PV;
64 c 0161          put list('^i^iInterest       ');
65 c 0178          get list(yi);
66 c 0197          i = yi;
67 c 01A7          put list('^i^iPayment        ');
68 c 01BE          get list(PMV);
69 c 01DD          PMT = PMV;
70 c 01ED          put list('^i^i%Inflation     ');
71 c 0204          get list(ir);
72 c 0223          fi = 1 + ir/1200;
73 c 0253          ci = 1.00;
74 c 0263          put list('^i^iStarting Month ');
75 c 027A          get list(sm);
76 c 0292          put list('^i^iStarting Year  ');
77 c 02A9          get list(sy);
78 c 02C1          put list('^i^iFiscal Month   ');
79 c 02D8          get list(fm);
80 c 02F0          put edit('^i^iDisplay Level  ',
81 c 032E               '^i^iYr Results : 0 ',
82 c 032E               '^i^iYr Interest: 1 ',
83 c 032E               '^i^iAll Values : 2 ')
84 c 032E              (skip,a);
85 c 032E          get list(dl);
86 c 0346          if dl < 0 | dl > 2 then
87 c 0357              signal error;
88 c 035E          m = sm;
89 c 0364          y = sy;
90 c 036A          IP = 0;
91 c 037A          PP = 0;
92 c 038A          YIN = 0;
93 c 039A          if name ^= '$con' then
94 c 03A8              put file(output) page;
95 c 03BA          call header();
96 c 03BD              do while (P > 0);
97 c 03D3              end = false;
98 c 03D8              INT = round ( i * P / 1200, 2 );
99 c 0408              IP  = IP + INT;
100 c 0423             PL  = P;
101 c 0433             P   = P  + INT;
102 c 044E             if P < PMT then
103 c 0464                 PMT = P;
104 c 0474             P   = P - PMT;
105 c 048F             PP  = PP + (PL - P);
106 c 04B5             INF = ci;
107 c 04CA             ci = ci / fi;
108 c 04EA             if P = 0 | dl > 1 | m = fm then
109 c 0520                 do;
110 c 0520                 put file(output) skip
```

Figure 12.3. Summary of Loan Payments Program Part B.

```
111 c 055B                  edit('|',100*m+y) (a,p'99/99');
112 c 055B              call display(PL * INF, INT * INF,
113 c 0601              PMT * INF, PP * INF, IP * INF);
114 c 0601              end;
115 c 0601          if m = fm & dl > 0 then
116 c 061E              call summary();
117 c 0621          m = m + 1;
118 c 0628          if m > 12 then
119 c 0634              do;
120 c 0634              m = 1;
121 c 063A              y = y + 1;
122 c 0641              if y > 99 then
123 c 064D                  y = 0;
124 c 0656              end;
125 c 0656          end;
126 c 0656      if dl = 0 then
127 c 065F          call line();
128 c 0665      else
129 c 0665      if ^end then
130 c 066C          call summary();
131 c 0672      end;
132 c 0672
133 c 0672  display:
134 c 0672      proc(a,b,c,d,e);
135 e 0672      dcl
136 e 067F          (a,b,c,d,e) fixed decimal(10,2);
137 e 067F      put file (output) edit
138 e 0731          ('|',a,'|',b,'|',c,'|',d,'|',e,'|')
139 e 0731          (a,2(2(p'$zz,zzz,zz9v.99',a),
140 e 0731                  p'$zzz,zz9.v99',a));
141 c 0731      end display;
142 c 0731
143 c 0731  summary:
144 c 0731      proc;
145 e 0731      end = true;
146 e 0736      call current_year(IP-YIN);
147 e 0757      YIN = IP;
148 c 0768      end summary;
149 c 0768
150 c 0768  current_year:
151 c 0768      proc(I);
152 e 0768      dcl
153 e 076F          yp fixed binary,
154 e 076F          I fixed decimal(10,2);
155 e 076F      yp = y;
156 e 0775      if fm < 12 then
157 e 0781          yp = yp - 1;
158 e 0788      call line();
159 e 078B      put skip file(output) edit
160 e 0804      ('|','Interest Paid During ',yp,'-',y,' is ',I,'|')
161 e 0804      (a,x(15),2(a,p'99'),a,p'$$$,$$$,$$9V.99',x(16),a);
162 e 0804      call line();
163 c 0808      end current_year;
164 c 0808
165 c 0808  header:
```

Figure 12.3. Summary of Loan Payments Program Part C.

```
166 c 0808          proc;
167 e 0808          put file(output) list(clear);
168 e 0822          call line();
169 e 0825          put file(output) skip edit
170 e 0860          ('|','L O A N   P A Y M E N T   S U M M A R Y','|')
171 e 0860              (a,x(19));
172 e 0860          call line();
173 e 0863          put file(output) skip edit
174 e 08E3          ('|','Interest Rate',yi,'%','Inflation Rate',ir,'%','|')
175 e 08E3              (a,x(15),2(a,p'b99v.99',a,x(6)),x(9),a);
176 e 08E3          call line();
177 e 08E6          put file(output) skip edit
178 e 0942              ('|Date |',
179 e 0942               '  Principal   |',
180 e 0942               'Plus  Interest|',
181 e 0942               '  Payment   |',
182 e 0942               'Principal Paid|',
183 e 0942               'Interest Paid |') (a);
184 e 0942          call line();
185 c 0946          end header;
186 c 0946
187 c 0946      line:
188 c 0946          proc;
189 e 0946          dcl
190 e 0946              i fixed bin;
191 e 0946          put file(output) skip edit
192 e 099E          ('-------','------------',
193 e 099E              ('---------------' do i = 1 to 4)) (a);
194 c 099E          end line;
195 a 099E      end pmt;
```

```
                S U M M A R Y   O F   P A Y M E N T S

                Output File Name ,


                Principal       3000
                Interest        14
                Payment         144.03
                %Inflation      0
                Starting Month  11
                Starting Year   80
                Fiscal Month    12

                Display Level
                Yr Results : 0
                Yr Interest: 1
                All Values : 2 0
```

Figure 12.3. Summary of Loan Payments Program Part D.

```
|                    L O A N   P A Y M E N T   S U M M A R Y                    |
|                Interest Rate 14.00%      Inflation Rate 00.00%                |
|Date |  Principal   |Plus  Interest|  Payment  |Principal Paid|Interest Paid |
|12/80|$    2,890.97|$       33.73|$   144.03|$       219.33|$       68.73|
|12/81|$    1,479.02|$       17.26|$   144.03|$     1,647.75|$      368.67|
|11/82|$        0.25|$        0.00|$     0.25|$     3,000.00|$      456.97|

Principal      ,
Interest       ,
Payment        ,
%Inflation     ,
Starting Month ,
Starting Year  ,
Fiscal Month   ,

Display Level
Yr Results : 0
Yr Interest: 1
All Values : 2 1
```

```
|                    L O A N   P A Y M E N T   S U M M A R Y                    |
|                Interest Rate 14.00%      Inflation Rate 00.00%                |
|Date |  Principal   |Plus  Interest|  Payment  |Principal Paid|Interest Paid |
|12/80|$    2,890.97|$       33.73|$   144.03|$       219.33|$       68.73|
|              Interest Paid During '80-'80 is          $68.73                 |
|12/81|$    1,479.02|$       17.26|$   144.03|$     1,647.75|$      368.67|
|              Interest Paid During '81-'81 is         $299.94                 |
|11/82|$        0.25|$        0.00|$     0.25|$     3,000.00|$      456.97|
|              Interest Paid During '82-'82 is          $88.30                 |
```

Figure 12.3. Summary of Loan Payments Program Part E.

```
Principal       ,
Interest        ,
Payment         ,
%Inflation      ,
Starting Month  ,
Starting Year   ,
Fiscal Month    ,

Display Level
Yr Results : 0
Yr Interest: 1
All Values : 2 2
```

L O A N P A Y M E N T S U M M A R Y					
Interest Rate 14.00%			Inflation Rate 00.00%		
Date	Principal	Plus Interest	Payment	Principal Paid	Interest Paid
11/80	$ 3,000.00	$ 35.00	$ 144.03	$ 109.03	$ 35.00
12/80	$ 2,890.97	$ 33.73	$ 144.03	$ 219.33	$ 68.73
Interest Paid During '80-'80 is			$68.73		
01/81	$ 2,780.67	$ 32.44	$ 144.03	$ 330.92	$ 101.17
02/81	$ 2,669.08	$ 31.14	$ 144.03	$ 443.81	$ 132.31
03/81	$ 2,556.19	$ 29.82	$ 144.03	$ 558.02	$ 162.13
04/81	$ 2,441.98	$ 28.49	$ 144.03	$ 673.56	$ 190.62
05/81	$ 2,326.44	$ 27.14	$ 144.03	$ 790.45	$ 217.76
06/81	$ 2,209.55	$ 25.78	$ 144.03	$ 908.70	$ 243.54
07/81	$ 2,091.30	$ 24.40	$ 144.03	$ 1,028.33	$ 267.94
08/81	$ 1,971.67	$ 23.00	$ 144.03	$ 1,149.36	$ 290.94
09/81	$ 1,850.64	$ 21.59	$ 144.03	$ 1,271.80	$ 312.53
10/81	$ 1,728.20	$ 20.16	$ 144.03	$ 1,395.67	$ 332.69
11/81	$ 1,604.33	$ 18.72	$ 144.03	$ 1,520.98	$ 351.41
12/81	$ 1,479.02	$ 17.26	$ 144.03	$ 1,647.75	$ 368.67
Interest Paid During '81-'81 is			$299.94		
01/82	$ 1,352.25	$ 15.78	$ 144.03	$ 1,776.00	$ 384.45
02/82	$ 1,224.00	$ 14.28	$ 144.03	$ 1,905.75	$ 398.73
03/82	$ 1,094.25	$ 12.77	$ 144.03	$ 2,037.01	$ 411.50
04/82	$ 962.99	$ 11.23	$ 144.03	$ 2,169.81	$ 422.73
05/82	$ 830.19	$ 9.69	$ 144.03	$ 2,304.15	$ 432.42
06/82	$ 695.85	$ 8.12	$ 144.03	$ 2,440.06	$ 440.54
07/82	$ 559.94	$ 6.53	$ 144.03	$ 2,577.56	$ 447.07
08/82	$ 422.44	$ 4.93	$ 144.03	$ 2,716.66	$ 452.00
09/82	$ 283.34	$ 3.31	$ 144.03	$ 2,857.38	$ 455.31
10/82	$ 142.62	$ 1.66	$ 144.03	$ 2,999.75	$ 456.97
11/82	$ 0.25	$ 0.00	$ 0.25	$ 3,000.00	$ 456.97
Interest Paid During '82-'82 is			$88.30		

Figure 12.3. Summary of Loan Payments Program Part F.

```
Principal       ,
Interest        ,
Payment         ,
%Inflation      10
Starting Month  ,
Starting Year   ,
Fiscal Month    10

Display Level
Yr Results : 0
Yr Interest: 1
All Values : 2 2
```

L O A N P A Y M E N T S U M M A R Y					
Interest Rate 14.00%			Inflation Rate 10.00%		
Date	Principal	Plus Interest	Payment	Principal Paid	Interest Paid
11/80	$ 3,000.00	$ 35.00	$ 144.03	$ 109.03	$ 35.00
12/80	$ 2,864.95	$ 33.42	$ 142.73	$ 217.35	$ 68.11
01/81	$ 2,733.39	$ 31.88	$ 141.58	$ 325.29	$ 99.45
02/81	$ 2,602.35	$ 30.36	$ 140.42	$ 432.71	$ 129.00
03/81	$ 2,471.83	$ 28.83	$ 139.27	$ 539.60	$ 156.77
04/81	$ 2,341.85	$ 27.32	$ 138.12	$ 645.94	$ 182.80
05/81	$ 2,212.44	$ 25.81	$ 136.97	$ 751.71	$ 207.08
06/81	$ 2,083.60	$ 24.31	$ 135.82	$ 856.90	$ 229.65
07/81	$ 1,955.36	$ 22.81	$ 134.66	$ 961.48	$ 250.52
08/81	$ 1,829.70	$ 21.34	$ 133.65	$ 1,066.60	$ 269.99
09/81	$ 1,702.58	$ 19.86	$ 132.50	$ 1,170.05	$ 287.52
10/81	$ 1,576.11	$ 18.38	$ 131.35	$ 1,272.85	$ 303.41
Interest Paid During '80-'81 is				$332.69	
11/81	$ 1,451.91	$ 16.94	$ 130.34	$ 1,376.48	$ 318.02
12/81	$ 1,326.68	$ 15.48	$ 129.19	$ 1,478.03	$ 330.69
01/82	$ 1,203.50	$ 14.04	$ 128.18	$ 1,580.64	$ 342.16
02/82	$ 1,079.56	$ 12.59	$ 127.03	$ 1,680.87	$ 351.67
03/82	$ 957.46	$ 11.17	$ 126.02	$ 1,782.38	$ 360.06
04/82	$ 835.87	$ 9.74	$ 125.01	$ 1,883.39	$ 366.92
05/82	$ 714.79	$ 8.34	$ 124.00	$ 1,983.87	$ 372.31
06/82	$ 594.25	$ 6.93	$ 123.00	$ 2,083.81	$ 376.22
07/82	$ 474.26	$ 5.53	$ 121.99	$ 2,183.19	$ 378.66
08/82	$ 354.84	$ 4.14	$ 120.98	$ 2,281.99	$ 379.68
09/82	$ 236.02	$ 2.75	$ 119.97	$ 2,380.19	$ 379.27
10/82	$ 117.80	$ 1.37	$ 118.96	$ 2,477.79	$ 377.45
Interest Paid During '81-'82 is				$124.28	
11/82	$ 0.20	$ 0.00	$ 0.20	$ 2,457.00	$ 374.25
Interest Paid During '81-'82 is				$0.00	

Figure 12.3. Summary of Loan Payments Program Part G.

during a particular iteration of the program, where 0 provides an abbreviated display, 1 provides additional information, and 2 gives the full trace.

Using an algorithm similar to that described in Section 12.7, the primary loop occurs between lines 96 and 131, where the initial principal is increased by the monthly interest and reduced by the monthly payment until the principal becomes zero. Several examples of program interaction are shown following the listing of Figure 12-3. The first output listing shows a minimal display corresponding to a loan of $3000 at 14% interest rate with a payment of $144.03. In this case, an inflation rate of 0% is assumed with a starting payment on 11/80, and end-of-year taxes due in December of each year. The display shows the principal, interest in December, monthly payment, amount paid toward principal in December, and amount of interest paid in the last month of the fiscal year.

The second output listing shows an execution of the main loop using the same values shown above, with display level 1. In this case, the output also contains the yearly interest paid on the loan for each fiscal year which would, presumably, be deducted from the taxable income.

The third output listing again uses the same initial values used in the previous examples, but provides a full display of the monthly principal, interest, monthly payment, payment applied to the principal, and interest payment.

The last display shows the same loan and interest rate with an adjustment in dollar value due to inflation. The (rather conservative) inflation rate of 10% is assumed in this example, so that all amounts are scaled to the value of the dollar at the time the loan was issued. For tax reporting purposes, the display showing the total interest paid at the end of each year is not scaled, and thus does not match the sum of the interest paid during the year. It is interesting to note that if we assume a 0% inflation rate, the total loan payment is 3,456.97, taken from the previous output. Assuming an inflation rate of 10%, however, the total cost of the loan in today's dollars is

```
  2,457.00
+   374.25
 --------
  2,831.25
```

resulting in a net gain of 68.75 over a two year period!

Several operational details must be presented in order to properly understand the operation of this program. First, there are several additional variables declared between lines 15 and 29 which are used throughout the program:

P	initially set to PV, but changes during execution (see lines 63, 101, and 104)
PP	total principal paid (see line 105)
PL	principal for current line, holds P for display purposes (see lines 100 and 112)
PMT	payment initially set to PMV, but changes during execution (see lines 69 and 103)
INT	computed interest during current month (see lines 98, 101, and 112)
YIN	interest at beginning of current year (see lines 92, 146, and 147)
IP	total interest paid (see lines 90, 99, and 146)
i	interest rate, initialized to yi (see line 66)
INF	percent of devaluation of the original dollar due to inflation (see lines 106, 112, and 113)
ci	current devaluation due to inflation (see lines 73, 106, and 107)
fi	factor for computing current inflation (see lines 72 and 107)

It should be noted that P and PMT are "working" variables for principal and payment so that the original variables PV and PMV are not destroyed during the computations. As a result, the operator can simply enter a comma (,) for subsequent input requests to indicate that the previously entered value is to be retained.

The program execution actually begins on line 35 with a "clear screen" character for the Lear-Siegler ADM-3A CRT. This control character is defined in the replace statement on line 6. If you are not using an ADM-3A, you can substitute the proper character in the replace statement and recompile the program.

In preparation for the subsequent OPEN, an ON-condition is set to trap possible OPEN errors (see lines 37 through 41). The operator is then prompted for the report output file name on line 44. The character variable "name" is initialized to the value "$con" on line 32: if the operator enters a comma rather than a file or device name, the console is assumed as the output device. If either a comma or the name $con is entered as the output file name, the console is OPENed with a zero page size so that no form-feeds are issued at the end of each logical page (see lines 47 and 48). Otherwise, the output file or device is OPENed as a normal PRINT device so that form-feeds are placed into the output file or sent to the physical output device

(usually the printer, $lst).

The ON-condition set at line 52 traps any occurrence of the ERROR condition, including ERROR(1) which indicates a data conversion error (a complete list of the ERROR subcodes is given in the "Recoverable Errors" section of the PL/I-80 Command Summary). Invalid data is also programmatically SIGNALed on line 87 if the value of dl is out-of-range. To make this particular program commercially palatable, it would be necessary to SIGNAL errors for all other invalid input data items, such as a negative interest rate. Further, the Fixed Overflow condition (FOFL) should also be set to intercept out-of-bound computations.

Program variable initialization for each set of input values begins on line 88. A page-eject is executed if the output file is not the console, followed by a page header printed by the "header" subroutine on line 165. It is instructional to compare the formatting statements in the header subroutine with the output values shown following the program listing.

The main processing loop, beginning at line 96, is executed repetitively until the principal reduces to zero. The variable "end" indicates whether or not an end-of-year summary has been printed (see line 145), and is used at the end of processing to avoid a possible duplicate summary (see line 129). The monthly interest (INT) for the current principal (P) is then computed and summed in IP on lines 98 and 99. The current principal is saved for later display in PL, and the monthly interest is added to the principal. If the payment exceeds the remaining principal on line 102, then the payment is reduced to cover this remainder. The principal is then reduced by the payment amount, which will eventually produce a zero value (if the original payment is sufficiently large to pay off the loan!). The total principal paid is summed on line 105, and the inflation rate is computed on line 106.

Since we have three display formats, the decision to display the current computation is somewhat complicated: if this is the last iteration (the principal P is zero), or if the full display format is selected (dl > 1), or if the current month is the end of the fiscal year (m = fm) then the current computation is written between lines 109 and 114. The picture format p'99/99' displays the month and year, where 100*m+y produces a four-digit number to match this format. If, for example, m = 11 and y = 64, then

```
100 * m + y   =   100 * 11 + 64   =   1164
```

which appears as 11/64 when printed using this picture. The "display" subroutine actually performs the output function, based upon the six actual parameters listed on lines 112 and 113. Each argument is adjusted by the current inflation rate INF and passed to the display subroutine. If the inflation rate has been set to 0%, the value of INF is 1.00 at this point in the computation. The body of the display subroutine, listed between lines 134 and 141 could, of course, be inserted in-line since there is only one call to display. However, the display subroutine does illustrate Fixed Decimal parameter passing

mechanisms and serves to break the program into smaller, more readable, segments. Again, it may be worthwhile comparing the formatting operations in the display subroutine with the actual program output.

The statement on line 115 then checks for the end of fiscal year (m = fm) and, if the display mode is either 1 or 2, a yearly interest summary is printed using the "summary" subroutine. The summary subroutine, listed between lines 144 and 148, in turn, calls the "current_year" subroutine to write the yearly interest paid (IP-YIN). The base value for next year's display is retained in YIN through the assignment on line 147. The current_year subroutine is listed between lines 151 and 163. If the fiscal year does not end in December (fm<12), the interest rate payment is split between two calendar years (yp = y - 1). Again, the current_year subroutine could be combined with the summary subroutine without changing the program logic.

The end of the main loop, between lines 126 and 130, contains statements which finalize the report. If the abbreviated display format was selected (dl = 0), a simple line of dashes completes the display. Otherwise, a check is made to ensure there have been intervening output lines (^end) and, if so, an interest summary is printed on line 130. The program then returns to the top of the loop and reads additional input parameters for production of another report.

12.10. Computation of Depreciation Schedules.

The final example illustrates a number of commercial processing concepts in PL/I-80 using evaluation of Depreciation Schedules as an example. The sample program listing is shown in Figure 12-4 followed by several examples of program interaction.

The Depreciation program reads several input values and prints a table based upon these values according to one of three different depreciation schedules: Straight-Line, Sum of the Years, or Double Declining. The program also accounts for bonus depreciation during the first year, reduction in taxable income due to sales tax, and investment tax credit on new or used equipment. The following general algorithms are used in this program:

> Investment Tax Credit (ITC) is assumed to be 10% of the selling price (see the replace statment, line 7), applied to the full price of new equipment, or up to $100,000 in the case of used equipment.
>
> Bonus Depreciation is assumed to be 10% of the selling price, up to a maximum of $2,000 (see the replace statement, lines 8 and 9).
>
> Under all three depreciation schedules, the amount to

depreciate is taken as the difference between the selling price minus the bonus depreciation, and the residual value of the equipment.

Under all schedules, the depreciation value computed for the first year is prorated by month through the remainder of the fiscal year (not including bonus depreciation).

In the case of Straight-Line depreciation, the amount to depreciate is spread uniformly over the number of years in which the depreciation occurs.

For the Sum of the Years, the year values are summed starting at 1, through the number of years in which depreciation takes place:

ys = 1 + 2 + 3 + . . . + years

The depreciation is distributed over the total number of years by computing years/ys times the depreciation value for the first year, (years-1)/ys times the remainder for the second year, and so-forth until the last year in which 1/ys times the remaining depreciation value is taken.

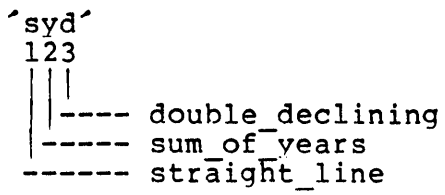
For the Double Declining case, each year's depreciation is computed as the book value divided by the number of years, which is then multiplied by 2 for new equipment, or 1.5 if the equipment is used.

The program reads the selling price, residual value, percentage sales tax, the percentage income tax bracket, the number of months remaining in the current fiscal year, and the number of years in which to depreciate the equipment. The program then asks whether the equipment is new or used, and then reads the depreciation schedule code for the subsequent report. A sample input sequence is given in Figure 12-4, immediately following the program listing. Although the exact details of program organization and flow is left to the reader as an exercise, there are a number of constructs in this program worthy of discussion.

First, this particular program uses an entry variable array to "dispatch" the calls to compute one of three schedules. The entry array is defined on line 40, with a subscript range of 0 through 3. The individual elements of this vector are initialized between lines 42 and 45, allowing an indirect call to either the "error" subroutine or one of the depreciation schedule handling subroutines. The actual call to one of these subroutines occurs later in the program. The schedule selection takes place on line 71, where one of the characters s, y, or d is read from the console into the character variable "select_sched." After variable initialization has occurred, the "display" subroutine is invoked from line 89. The display subroutine, listed between lines 97 and 101, performs the actual dispatch to the schedule handler through the statement

call schedule (index (schedules,select_sched))

This particular statement can be decomposed as follows. The "schedules" variable is defined on line 39 and initialized to the character string 'syd', where each letter corresponds to one of the valid schedule handlers, as shown below



The evaluation of

```
index (schedules,select_sched)
```

is the same as

```
index('syd',select_sched)
```

which, for valid inputs s, y, or d, produces 1, 2, or 3. If the value of select_sched is not one of s, y, or d, then the index function returns a zero value. Thus, if select_sched is s, the call statement evaluates to

```
call schedule(1)
```

which, due to the assignment on line 43, calls the subroutine "straight_line." Similarly, an input of y or d produces

```
call schedule(2)  or  call schedule(3)
```

producing a call to "sum_of_years" or "double_declining," respectively. Since the index function returns zero if select_sched is not one of s, y, or d, all invalid character input values produce

```
call schedule(0)
```

which calls the "error" subroutine where the error condition is reported to the operator.

The second construct of interest in this program is the use of the "output" file variable, defined on line 35. During the parameter input phase, the operator is prompted with

```
List? (yes/no)
```

If the operator reponds with "yes" then the program writes the depreciation report to both the console and the listing device. The manner in which the program performs this function is presented below.

Two file constants, sysprint and list, are declared on line 36 to address the console and the list device. The console file is OPENed first, on line 47, using an infinite page length to avoid form-

feed characters. If, on any iteration of the main loop, the operator responds in the affirmative on line 73, the list device is subsequently OPENed on line 75. It should be noted that this statement may be executed several times on any particular execution of this program, but only the first OPEN has any effect. The "display" subroutine is called on line 89 to compute and display the output report for a specific set of input values. Display has a single actual parameter which is the file constant "sysprint" passed to the subroutine as the formal parameter "f" on line 99. The formal parameter, in turn, is assigned to the global variable "output" on line 100. Subsequent PUT statements of the form

```
put file(output) ...
```

write data to the console, producing the first report.

Referring to line 90 of Figure 12-4, if "copy_to_list" has the character value 'yes' then display is called once again. This time, however, the actual parameter is "list" which corresponds to the system listing device. Similar to the actions given above, the output file variable is indirectly assigned the value "list" and all PUT statements which reference file "output" write their data to the printer, resulting in both a soft and hard copy of the report.

Again, it is worthwhile examining the various components of this program while cross-checking output formats with the displayed results, since there are several different forms of decimal arithmetic and formatting which occur throughout.

```
 1 a 0000 depreciate:
 2 a 0006     procedure options(main);
 3 a 0006
 4 c 0006     %replace
 5 c 0006         clear_screen by '^z',
 6 c 0006         indent   by 15,
 7 c 0006         ITC_rate by .1,
 8 c 0006         bonus_rate by .1,
 9 c 0006         bonus_max  by 2000;
10 c 0006
11 c 0006     declare
12 c 0006         selling_price  decimal(8,2),
13 c 0006         adj_price decimal(8,2),
14 c 0006         residual_value decimal(8,2),
15 c 0006         year_value decimal(8,2),
16 c 0006         depreciation_value decimal(8,2),
17 c 0006         total_depreciation decimal(8,2),
18 c 0006         book_value decimal(8,2),
19 c 0006         tax_rate decimal(3,2),
20 c 0006         sales_tax decimal(8,2),
21 c 0006         tax_bracket decimal(2),
22 c 0006         FYD decimal(8,2),
23 c 0006         ITC decimal(8,2),
24 c 0006         bonus_dep decimal(8,2),
25 c 0006         months_remaining decimal(2),
26 c 0006         new char(4),
27 c 0006         factor decimal(2,1),
28 c 0006         years decimal(2),
29 c 0006         year_sum decimal(3),
30 c 0006         current_year decimal(2),
31 c 0006         select_sched char(1);
32 c 0006
33 c 0006     declare
34 c 0006         copy_to_list char(4),
35 c 0006         output file variable,
36 c 0006         (sysprint, list) file;
37 c 0006
38 c 0006     declare
39 c 0006         schedules char(3) static initial ('syd'),
40 c 0006         schedule (0:3) entry variable;
41 c 0006
42 c 0006     schedule (0) = error;
43 c 000C     schedule (1) = straight_line;
44 c 0015     schedule (2) = sum_of_years;
45 c 001E     schedule (3) = double_declining;
46 c 0027
47 c 0027     open file (sysprint) stream print pagesize(0)
48 c 0043         title ('$con');
49 c 0043
50 c 0043         do while('1'b);
51 c 0043         put list(clear_screen,'^i^i^iDepreciation Schedule');
52 c 0065         put skip(3) list('^i^iSelling  Price? ');
53 c 0081         get list(selling_price);
54 c 00A0         put list('^i^iResidual Value? ');
55 c 00B7         get list(residual_value);
```

Figure 12.4. Depreciation Schedule Program Part A.

```
 56 c 00D6          put list('^i^iSales Tax (%)?  ');
 57 c 00ED          get list(tax_rate);
 58 c 010C          put list('^i^iTax Bracket(%)? ');
 59 c 0123          get list(tax_bracket);
 60 c 0142          put list('^i^iProRate Months? ');
 61 c 0159          get list(months_remaining);
 62 c 0178          put list('^i^iHow Many Years? ');
 63 c 018F          get list(years);
 64 c 01AE          put list('^i^iNew? (yes/no)   ');
 65 c 01C5          get list(new);
 66 c 01DF          put edit('^i^iSchedule:',
 67 c 021D               '^i^iStraight   (s)',
 68 c 021D               '^i^iSum-of-Yrs (y)',
 69 c 021D               '^i^iDouble Dec (d)? ')
 70 c 021D              (a,skip);
 71 c 021D          get list(select_sched);
 72 c 0237          put list('^i^iList? (yes/no)  ');
 73 c 024E          get list(copy_to_list);
 74 c 0268          if copy_to_list = 'yes' then
 75 c 0278              open file(list) stream print title('$lst');
 76 c 0294          factor = 1.5;
 77 c 02A4          if new = 'yes' then
 78 c 02B4              factor = 2.0;
 79 c 02C4          sales_tax =
 80 c 0304              decimal(selling_price*tax_rate,12,2)/100+.005;
 81 c 0304          if new = 'yes' | selling_price <= 100000.00 then
 82 c 032E              ITC = selling_price * ITC_rate;
 83 c 0351          else
 84 c 0351              ITC = 100000 * ITC_rate;
 85 c 0371          bonus_dep = selling_price * bonus_rate;
 86 c 0391          if bonus_dep > bonus_max then
 87 c 03A7              bonus_dep = bonus_max;
 88 c 03B7          put list(clear_screen);
 89 c 03CE          call display(sysprint);
 90 c 03DA          if copy_to_list = 'yes' then
 91 c 03EA              call display(list);
 92 c 03F6          put skip list('^i^i^i   Type RETURN to Continue');
 93 c 0412          get skip(2);
 94 c 0426          end;
 95 c 0426
 96 c 0426 display:
 97 c 0426      procedure(f);
 98 e 0426      declare
 99 e 042D          f file;
100 e 042D      output = f;
101 e 0437      call schedule (index (schedules,select_sched));
102 c 0453      end display;
103 c 0453
104 c 0453 error:
105 c 0453      procedure;
106 c 0453      /* bad entry for schedule */
107 e 0453      put file (output) edit('Invalid Schedule - Enter s, y, or d')
108 e 0473          (page,column(indent),x(8),a);
109 e 0473      call line();
110 c 0477      end error;
```

Figure 12.4. Depreciation Schedule Program Part B.

```
111 c 0477
112 c 0477 straight_line:
113 c 0477     procedure;
114 e 0477     adj_price = selling_price - bonus_dep;
115 e 0492     put file (output) edit('S T R A I G H T   L I N E')
116 e 04B2         (page,column(indent),x(14),a);
117 e 04B2     call header();
118 e 04B5     depreciation_value = adj_price - residual_value;
119 e 04D0     book_value = adj_price;
120 e 04E0     total_depreciation = 0;
121 e 04F0         do current_year = 1 to years;
122 e 0526         year_value =
123 e 0560             decimal(depreciation_value/years,8,2) + .005;
124 e 0560         if current_year = 1 then
125 e 0576             do;
126 e 0576             year_value =
127 e 05A6                 year_value * months_remaining / 12;
128 e 05A6             FYD = year_value;
129 e 05B6             end;
130 e 05B6         depreciation_value = depreciation_value - year_value;
131 e 05D1         total_depreciation = total_depreciation + year_value;
132 e 05EC         book_value = adj_price - total_depreciation;
133 e 0607         call print_line();
134 e 0624         end;
135 e 0624     call summary();
136 c 0628     end straight_line;
137 c 0628
138 c 0628 sum_of_years:
139 c 0628     procedure;
140 e 0628     adj_price = selling_price - bonus_dep;
141 e 0643     put file (output) edit('S U M   O F   T H E   Y E A R S')
142 e 0663         (page,column(indent),x(11),a);
143 e 0663     call header();
144 e 0666     depreciation_value = adj_price - residual_value;
145 e 0681     book_value = adj_price;
146 e 0691     total_depreciation = 0;
147 e 06A1     year_sum = 0;
148 e 06B1         do current_year = 1 to years;
149 e 06E7         year_sum = year_sum + current_year;
150 e 071C         end;
151 e 071C
152 e 071C         do current_year = 1 to years;
153 e 0752         year_value =
154 e 07A3             decimal(depreciation_value *
155 e 07A3                 (years - current_year + 1),12,2)
156 e 07A3                 / year_sum + .005;
157 e 07A3         if current_year = 1 then
158 e 07B9             do;
159 e 07B9             year_value =
160 e 07E9                 year_value * months_remaining / 12;
161 e 07E9             FYD = year_value;
162 e 07F9             end;
163 e 07F9         depreciation_value = depreciation_value - year_value;
164 e 0814         total_depreciation = total_depreciation + year_value;
165 e 082F         book_value = adj_price - total_depreciation;
```

Figure 12.4. Depreciation Schedule Program Part C.

```
166 e 084A          call print_line();
167 e 0867          end;
168 e 0867      call summary();
169 c 086B      end sum_of_years;
170 c 086B
171 c 086B double_declining:
172 c 086B      procedure;
173 e 086B      adj_price = selling_price - bonus_dep;
174 e 0886      put file (output) edit('D O U B L E    D E C L I N I N G')
175 e 08A6          (page,column(indent),x(10),a);
176 e 08A6      call header();
177 e 08A9      depreciation_value = adj_price - residual_value;
178 e 08C4      book_value = adj_price;
179 e 08D4      total_depreciation = 0;
180 e 08E4          do current_year = 1 to years
181 e 0931              while (depreciation_value > 0);
182 e 0931          year_value =
183 e 0971              decimal(book_value/years,8,2) * factor+.005;
184 e 0971          if current_year = 1 then
185 e 0987              do;
186 e 0987              year_value =
187 e 09B7                  year_value * months_remaining / 12;
188 e 09B7              FYD = year_value;
189 e 09C7              end;
190 e 09C7          if year_value > depreciation_value then
191 e 09DD              year_value = depreciation_value;
192 e 09ED          depreciation_value = depreciation_value - year_value;
193 e 0A08          total_depreciation = total_depreciation + year_value;
194 e 0A23          book_value = adj_price - total_depreciation;
195 e 0A3E          call print_line();
196 e 0A5B          end;
197 e 0A5B      call summary();
198 c 0A5F      end double_declining;
199 c 0A5F
200 c 0A5F header:
201 c 0A5F      procedure;
202 c 0A5F      /* print header record */
203 e 0A5F      dcl
204 e 0A5F          new_or_used char(5);
205 e 0A5F      if new = 'yes' then
206 e 0A6F          new_or_used = ' New';
207 e 0A7F      else
208 e 0A7F          new_or_used = ' Used';
209 e 0A8B      put file (output) edit(
210 e 0B5D          '-------------------------------------------------',
211 e 0B5D              '|',selling_price+sales_tax,new_or_used,
212 e 0B5D              residual_value,' Residual Value|',
213 e 0B5D          '|',months_remaining,' Months Left ',
214 e 0B5D              tax_rate,'% Tax',tax_bracket,'% Tax Bracket|')
215 e 0B5D          (2(skip,column(indent),a),
216 e 0B5D          2(p'B$$,$$$,$$9.V99',a),
217 e 0B5D          skip,column(indent),a,x(5),f(2),a,2(x(2),p'B99',a));
218 e 0B5D
219 e 0B5D          put file (output) edit(
220 e 0B9F          '-------------------------------------------------',
```

Figure 12.4. Depreciation Schedule Program Part D.

```
221 e 0B9F      '| Y | Depreciation | Depreciation |  Book Value  |',
222 e 0B9F      '| r |   For Year   |  Remaining   |              |',
223 e 0B9F      '-------------------------------------------------')
224 e 0B9F      (skip,column(indent),a);
225 c 0B9F      end header;
226 c 0B9F
227 c 0B9F print_line:
228 c 0B9F      procedure;
229 c 0B9F      /* print current line */
230 e 0B9F      put file (output) edit(
231 e 0C34      '|',current_year,
232 e 0C34      ' |',year_value,
233 e 0C34      ' |',depreciation_value,
234 e 0C34      ' |',book_value,' |')
235 e 0C34      (skip,column(indent),
236 e 0C34      a,f(2),4(a,p'$z,zzz,zz9v.99'));
237 c 0C34      end print_line;
238 c 0C34
239 c 0C34 summary:
240 c 0C34      procedure;
241 e 0C34      declare
242 e 0C34          adj_ITC decimal(8,2),
243 e 0C34          total decimal(8,2),
244 e 0C34          direct decimal(8,2);
245 e 0C34      call line();
246 e 0C37      adj_ITC = ITC * 100 / tax_bracket;
247 e 0C67      total = FYD + sales_tax + adj_ITC + bonus_dep;
248 e 0C98      direct = total * tax_bracket / 100;
249 e 0CC8      put file (output) edit(
250 e 0DEE      '|       First Year Reduction in Taxable Income    |',
251 e 0DEE      '-------------------------------------------------',
252 e 0DEE      '|       Depreciation                ',FYD,'  |',
253 e 0DEE      '|       Sales Tax                   ',sales_tax,'  |',
254 e 0DEE      '|       ITC (Adjusted)              ',adj_ITC,'  |',
255 e 0DEE      '|       Bonus Depreciation          ',bonus_dep,'  |',
256 e 0DEE      '|                                   -------------  |',
257 e 0DEE      '|       Total for First Year        ',total,'  |',
258 e 0DEE      '|       Direct Reduction in Tax     ',direct,'  |')
259 e 0DEE      (2(skip,column(indent),a),
260 e 0DEE          2(4(skip,column(indent),a,
261 e 0DEE          p'$z,zzz,zz9v.99',x(3),a),
262 e 0DEE              skip,column(indent),a));
263 e 0DEE      call line();
264 c 0DF2      end summary;
265 c 0DF2
266 c 0DF2 line:
267 c 0DF2      procedure;
268 c 0DF2      /* print line of "-" */
269 e 0DF2      put file (output) edit(
270 e 0E13      '-------------------------------------------------')
271 e 0E13      (skip,column(indent),a);
272 c 0E13      end line;
273 a 0E13 end depreciate;
```

Figure 12.4. Depreciation Schedule Program Part E.

Depreciation Schedule

```
Selling  Price? 200000
Residual Value? 40000
Sales Tax (%)?  6
Tax Bracket(%)? 50
ProRate Months? 10
How Many Years? 7
New? (yes/no)   no
Schedule:
Straight   (s)
Sum-of-Yrs (y)
Double Dec (d)? d
List? (yes/no)  no
```

D O U B L E D E C L I N I N G

$212,000.00 Used 10 Months Left		$40,000.00 Residual Value 06% Tax 50% Tax Bracket	
Yr	Depreciation For Year	Depreciation Remaining	Book Value
1	$ 35,357.14	$ 122,642.86	$ 162,642.86
2	$ 34,852.04	$ 87,790.82	$ 127,790.82
3	$ 27,383.75	$ 60,407.07	$ 100,407.07
4	$ 21,515.79	$ 38,891.28	$ 78,891.28
5	$ 16,905.27	$ 21,986.01	$ 61,986.01
6	$ 13,282.71	$ 8,703.30	$ 48,703.30
7	$ 8,703.30	$ 0.00	$ 40,000.00

First Year Reduction in Taxable Income	
Depreciation	$ 35,357.14
Sales Tax	$ 12,000.00
ITC (Adjusted)	$ 20,000.00
Bonus Depreciation	$ 2,000.00
Total for First Year	$ 69,357.14
Direct Reduction in Tax	$ 34,678.57

Type RETURN to Continue

Figure 12.4. Depreciation Schedule Program Part F.

Depreciation Schedule

```
Selling  Price? ,
Residual Value? ,
Sales Tax (%)?  ,
Tax Bracket(%)? ,
ProRate Months? 8
How Many Years? ,
New? (yes/no)   yes
Schedule:
Straight   (s)
Sum-of-Yrs (y)
Double Dec (d)? y
List? (yes/no)  no
```

S U M O F T H E Y E A R S

$212,000.00 New	$40,000.00 Residual Value
8 Months Left	06% Tax	50% Tax Bracket

Yr	Depreciation For Year	Depreciation Remaining	Book Value
1	$ 26,333.33	$ 131,666.67	$ 171,666.67
2	$ 28,214.29	$ 103,452.38	$ 143,452.38
3	$ 18,473.64	$ 84,978.74	$ 124,978.74
4	$ 12,139.82	$ 72,838.92	$ 112,838.92
5	$ 7,804.17	$ 65,034.75	$ 105,034.75
6	$ 4,645.34	$ 60,389.41	$ 100,389.41
7	$ 2,156.76	$ 58,232.65	$ 98,232.65

First Year Reduction in Taxable Income

Depreciation	$ 26,333.33
Sales Tax	$ 12,000.00
ITC (Adjusted)	$ 40,000.00
Bonus Depreciation	$ 2,000.00
Total for First Year	$ 80,333.33
Direct Reduction in Tax	$ 40,166.66

Type RETURN to Continue

Figure 12.4. Depreciation Schedule Program Part G.

```
                    Depreciation Schedule

Selling  Price? 310000
Residual Value? 30000
Sales Tax (%)?  ,
Tax Bracket(%)? ,
ProRate Months? 12
How Many Years? 5
New? (yes/no)   yes
Schedule:
Straight    (s)
Sum-of-Yrs  (y)
Double Dec (d)? d
List? (yes/no)  no
```

D O U B L E D E C L I N I N G

$328,600.00 New	$30,000.00 Residual Value
12 Months Left	06% Tax 50% Tax Bracket

Yr	Depreciation For Year	Depreciation Remaining	Book Value
1	$ 123,200.00	$ 154,800.00	$ 184,800.00
2	$ 73,920.00	$ 80,880.00	$ 110,880.00
3	$ 44,352.00	$ 36,528.00	$ 66,528.00
4	$ 26,611.20	$ 9,916.80	$ 39,916.80
5	$ 9,916.80	$ 0.00	$ 30,000.00

First Year Reduction in Taxable Income

Depreciation	$ 123,200.00
Sales Tax	$ 18,600.00
ITC (Adjusted)	$ 62,000.00
Bonus Depreciation	$ 2,000.00
Total for First Year	$ 205,800.00
Direct Reduction in Tax	$ 102,900.00

Type RETURN to Continue

Figure 12.4. Depreciation Schedule Program Part H.

```
                    Depreciation Schedule

          Selling  Price? ,
          Residual Value? ,
          Sales Tax (%)?  ,
          Tax Bracket(%)? ,
          ProRate Months? ,
          How Many Years? ,
          New? (yes/no)   ,
          Schedule:
          Straight   (s)
          Sum-of-Yrs (y)
          Double Dec (d)? s
          List? (yes/no)  ,
```

S T R A I G H T L I N E

$328,600.00 New		$30,000.00 Residual Value
12 Months Left	06% Tax	50% Tax Bracket

Yr	Depreciation For Year	Depreciation Remaining	Book Value
1	$ 55,600.00	$ 222,400.00	$ 252,400.00
2	$ 44,480.00	$ 177,920.00	$ 207,920.00
3	$ 35,584.00	$ 142,336.00	$ 172,336.00
4	$ 28,467.20	$ 113,868.80	$ 143,868.80
5	$ 22,773.76	$ 91,095.04	$ 121,095.04

First Year Reduction in Taxable Income

Depreciation	$ 55,600.00
Sales Tax	$ 18,600.00
ITC (Adjusted)	$ 62,000.00
Bonus Depreciation	$ 2,000.00
Total for First Year	$ 138,200.00
Direct Reduction in Tax	$ 69,100.00

```
          Type RETURN to Continue^C
```

Figure 12.4. Depreciation Schedule Program Part I.