ANL-76-70

ANL-76-70

INL Technical Library
403160

PLEASE RETURN TO
MFC BRANCH LIBRARY

# CLASS NOTES FOR A PL/I COURSE

by

Kenneth W. Dritz

PROPERTY OF
ANL-W Technical Library

UWC-AUA-USERDA

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS

Prepared for the U. S. ENERGY RESEARCH
AND DEVELOPMENT ADMINISTRATION
under Contract W-31-109-Eng-38

The facilities of Argonne National Laboratory are owned by the United States Government. Under the terms of a contract (W-31-109-Eng-38) between the U. S. Energy Research and Development Administration, Argonne Universities Association and The University of Chicago, the University employs the staff and operates the Laboratory in accordance with policies and programs formulated, approved and reviewed by the Association.

MEMBERS OF ARGONNE UNIVERSITIES ASSOCIATION

The University of Arizona
Carnegie-Mellon University
Case Western Reserve University
The University of Chicago
University of Cincinnati
Illinois Institute of Technology
University of Illinois
Indiana University
Iowa State University
The University of Iowa
Kansas State University
The University of Kansas
Loyola University
Marquette University
Michigan State University
The University of Michigan
University of Minnesota
University of Missouri
Northwestern University
University of Notre Dame
The Ohio State University
Ohio University
The Pennsylvania State University
Purdue University
Saint Louis University
Southern Illinois University
The University of Texas at Austin
Washington University
Wayne State University
The University of Wisconsin

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research and Development Administration, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights. Mention of commercial products, their manufacturers, or their suppliers in this publication does not imply or connote approval or disapproval of the product by Argonne National Laboratory or the U. S. Energy Research and Development Administration.

Printed in the United States of America
Available from
National Technical Information Service
U. S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22161
Price: Printed Copy $10.00; Microfiche $2.25

Distribution Category:
Mathematics and Computers
(UC-32)

ANL-76-70

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

# CLASS NOTES FOR A PL/I COURSE

by

Kenneth W. Dritz

Applied Mathematics Division

November 1975

# PREFACE

These notes were written for use as a supplement to a three-week PL/I course taught by the author from October 20, 1975 to November 7, 1975 at the Applied Mathematics Division of Argonne National Laboratory. The course was intended to attract scientists and engineers from other Laboratory divisions who contemplated using PL/I in their future programming. No special emphasis was placed on features useful in business applications.

In the preparation of these notes (and of the classes themselves), use was made of the fact that the scientists for whom they were intended could be assumed to have had prior experience in programming with high level languages (probably FORTRAN). This assumption is reflected in the absence of frequent demonstrations of the practical application of language elements to the solution of complete and realistic problems. The notes (and the course) thus do not address the problem of teaching the non-programmer how to program in PL/I; rather, they supply the practicing programmer with the information needed to begin using PL/I to solve problems he is already accustomed to solving in other languages.

That is not to say that the experienced FORTRAN programmer will necessarily find the road to conversion to be free of holes and bumps. Certain traps are lurking. Specifically, certain techniques and concepts of FORTRAN, if translated in the obvious way to PL/I, result in incorrect programs. Special emphasis has been devoted to this problem. It is apparent, for instance, in the discussions of the differences between fixed-point data (in PL/I) and integer data (in FORTRAN); the differences between the respective roles of defining (in PL/I) and equivalencing (in FORTRAN); and the proper, and very different, ways to pass and use variable dimension information in the two languages.

These notes were written over the short period of five weeks. Because of that rush, they are inevitably less polished than they could have been. This is hopefully compensated by the very careful attention given to the ordering of topics for effective learning. The chosen order of introduction of topics, which was worked out over a three-week period before writing commenced, is intended to help the students avoid mental overload even when classes (corresponding to chapters) are taught on successive days.

The very frequent references to passages in IBM manuals (which are keyed indirectly through the reference list following Chapter 15) are an essential factor in keeping these notes as short as they are. For instance, detailed syntax of statements is usually omitted from the notes, as are certain tables of information easily found in the manuals. The notes emphasize concepts more than details. Unfortunately, the utility of the references will be diminished in the future unless the page numbers can be successfully updated to reflect such revisions as may have been incorporated in the manuals by then.

The author has pointed out some differences between the "current" language and the proposed ANSI Standard for PL/I. The reader must be cautioned, however, that not all of the differences have been documented. (For instance, Chapter 1 does not mention the dropping of the I-to-N rule for default arithmetic attributes, which is certainly very important.) The absence of a complete comparison is due to the fact that lists of known differences were not constantly reviewed during the preparation of these notes; differences were cited when they just happened to come to mind.

Structured programming advocates may be disappointed by the almost total absence of orientation toward structured coding and development practices. The GO TO statement is taught. The reason is that this course is about the PL/I language and its concepts; it is not a course in programming methodology. Structured programming is a separate topic and can be (and in the author's opinion should be) taught independently of any particular language. The author has not, however, entirely ignored the question of program correctness. His contribution has been to emphasize language purity and to enhance transportability by carefully distinguishing between the formal language definition and implementation-defined features. Illegal language is never demonstrated. No concessions are made to convenience.

Finally, the author wishes to acknowledge the help of Matt Prastein and April Heiberger in preparing Chapters 1 and 6 for text editing in TSO; of the following secretaries in the Applied Mathematics Division for their many weeks spent typing the copy:

Marge Visser
April Heiberger
Judy Beumer
Grace Krause
Nancy Piazza;

of Linda Clark and Sue Katilavas for handling all aspects of the class notes after typing; of Graphic Arts for typing two chapters and printing all of them; and of Paul Messina, Lou Just, and Dean Davis for general administrative support.

Kenneth W. Dritz
Applied Mathematics Division
Argonne National Laboratory
November, 1975

TABLE OF CONTENTS


| | | PAGE |
|---|---|---|
| 0. | INTRODUCTION TO PL/I COURSE | 0-1 |
| 1. | VARIABLES, ATTRIBUTES, AND DECLARATIONS; ARITHMETIC DATA TYPES, ARITHMETIC EXPRESSIONS, PRECISION RULES | 1-1 |
| 2. | STRING DATA TYPES; STRING AND LOGICAL EXPRESSIONS | 2-1 |
| 3. | AGGREGATES | 3-1 |
| 4. | BLOCK STRUCTURE AND SCOPE OF NAMES | 4-1 |
| 5. | STORAGE CLASS AND BLOCK INVOCATIONS | 5-1 |
| 6. | (a) CONTROL CONSTRUCTS | 6-1 |
| | (b) CONDITIONS | 6-1 |
| 7. | INTRODUCTION TO I/O; STREAM I/O | 7-1 |
| 8. | INTRODUCTION TO RECORD I/O; CONSECUTIVE DATASETS | 8-1 |
| 9. | INDEXED AND REGIONAL DATASETS | 9-1 |
| 10. | (a) BUILTIN FUNCTIONS AND PSEUDO-VARIABLES | 10-1 |
| | (b) INTERLANGUAGE COMMUNICATION | 10-1 |
| 11. | LIST PROCESSING AND LOCATE-MODE I/O | 11-1 |
| 12. | (a) MISCELLANEOUS FEATURES | 12-1 |
| | (b) PREPROCESSOR | 12-1 |
| 13. | (a) ADVANCED JCL AND COMPILER OPTIONS | 13-1 |
| | (b) PROGRAM DEVELOPMENT AND DEBUGGING | 13-1 |
| 14. | MULTITASKING AND ASYNCHRONOUS I/O | 14-1 |
| 15. | THE CHECKOUT COMPILER IN TSO | 15-1 |
| 16. | REFERENCES | 16-1 |
| 17. | INDEX | 17-1 |

## ABSTRACT

Presented here are notes for a course in PL/I. They might serve as a guide to others who are developing a course, and indeed as class notes for that course. They might be useful as a textbook independent of any course; as such a textbook, however, they are not self-contained because of the built-in assumption that they will supplement lectures and be accompanied by manuals.

Very nearly the full language is taught here, with the emphasis on concepts rather than practical details. The unorthodox order in which concepts are introduced is the deliberate invention of the author. One effect of this is the complete avoidance of any discussion of I/O until roughly the midpoint of the course. The hoped-for consequence for students is an enhanced perception and understanding of the many concepts and their logical relationships.

The dawning of the age of transportability for PL/I programs gives the user a reason, for the first time, to avoid convenient but illegal language. In their attention to this issue, these notes should help the user appreciate the value of sound coding practices and their negligible incremental cost at the most important time — when he is first starting out.

# 0. Introduction to PL/I course.

## 0.1. Welcome!

Welcome to the PL/I course!

It is hoped that over the next three weeks you will realize your goal of learning to write effective programs in PL/I.

Why so many class sessions? PL/I is a "massive" language. Even if much of the bewildering detail is stripped away, leaving the major concepts, there is a lot to be taught and a lot to be learned. We have, in fact, left out many of the subtleties and a lot of the detail (rules, conventions, restrictions, interactions, etc.). No one can remember all that, anyway. That's what we have reference manuals for.

Although they are improving, reference manuals are still not very good for teaching the broad concepts of a programming language! That's why we have developed this course. In its planning we have devoted particular emphasis to the choice of a logical order for the introduction of successive concepts. We believe this is the recipe for successful learning. A consequence of this is the deferring of any discussion of I/O until about the midpoint of the course; since we don't wish to "jump the gun," examples and homework problems are necessarily and unrealistically I/O-free until then. But even when we finally get to I/O, we don't take an overdose. Progressively more advanced aspects of I/O are assigned to Lessons 7, 8, 9, 11, and 14.

Still, this is an ambitious undertaking. A college semester is being crammed into three weeks! To receive full value from this course, you will need to attend every lesson. Beyond that, you will need to read the class notes and selected passages in the manuals, and you are strongly urged to attempt the homework problems.

## 0.2. Goals of the course.

Sophisticated engineering applications in programming today are characterized by the combination of properties and features they are required to exhibit. For instance, a single, coherent application program may need to combine scientific calculations, non-numerical calculations (such as logical calculations or text manipulations), large-scale auxiliary data management, and internal resource management. And certain kinds of programs, particularly those modeling physical systems, can benefit from more "natural" ways of representing information, such as by time-varying "structural" or hierarchical relationships between items of data. Because PL/I can satisfy all these needs in a smooth and consistent way, the primary goal of the PL/I course is to teach nearly all the major concepts of the language.

(The only significant one omitted is "teleprocessing," which is not available in our system anyway and which is not in the ANSI standard.)

Experience has shown that PL/I programmers who have an incomplete knowledge of the language are likely to use inappropriate, i.e., less than natural, language features to accomplish a particular task. The result of this is frequently inefficiency in the object program and, as a consequence, dissatisfaction with the language.

For many years people believed that PL/I was the sole province of IBM. PL/I code interchange with non-IBM installations was out of the question. Well, in 1975 PL/I has come a long way. A proposed international ANSI-ECMA standard for PL/I is on "final approach" and likely to be accepted in 1976. Honeywell, Univac, and Burroughs have viable PL/I compilers which have been aimed at the proposed standard (a moving target) during their development. Even Control Data, which abandoned its early efforts in PL/I years ago, appears to be reviving its interest in the language (perhaps they thought it wouldn't catch on--and guessed wrong).

Thus, a second goal of the course is to prepare you for the day in the not too distant future when you may be writing programs that have portability requirements extending to other PL/I systems and other hardware. This is done in two ways. First, we will point out some significant differences between the IBM implementations of PL/I and the proposed standard. Second, we will make a clear distinction between <u>official language</u> and <u>current implementation</u>. Unfortunately, many programmers believe <u>PL/I is</u> whatever our compilers let them do; that is, they write technically illegal language which happens to give them a convenient and useful effect <u>on our system</u>. They may feel justified in doing this because they have no intention of exporting their programs to other installations with different compilers. The hidden danger, though, is that they may have to export their programs to themselves someday. There is no guarantee that <u>we</u> will always have IBM equipment! We have already experienced some <u>of</u> the problems that can be encountered with illegal language because IBM has changed the implementation of certain language features within their own progression of compilers over the years. (They have also changed--improved--the language itself several times. Unfortunately, this has made trouble even for honest programmers. One more round of "incompatible changes" must be expected with the introduction of the ANSI standard, after which we should enter an era of relative stability of the language.) So, on the theory that it is preferable to learn how to do it legally from the beginning and avoid possible problems later, we will emphasize language purity.

## 0.3. Topics to be covered.

The following is a broad outline for the fifteen lessons (class sessions).

1. Variables, attributes, and declarations:
   arithmetic data types, arithmetic expressions, precision rules.
2. String data types; string and logical expressions.
3. Aggregates.
4. Block structure and scope of names.
5. Storage classes and block invocations.
6. (a) Control constructs.
   (b) Conditions.
7. Introduction to I/O; stream I/O.
8. Introduction to record I/O; consecutive datasets.
9. Indexed and regional datasets.
10. (a) Builtin functions and pseudo-variables.
    (b) Interlanguage communication.
11. List processing and locate mode I/O.
12. (a) Miscellaneous features.
    (b) Preprocessor.
13. (a) Advanced JCL and compiler options.
    (b) Program development and debugging.
14. Multitasking and asynchronous I/O (optional).
15. Checker/TSO demonstration.

## 0.4. Class notes.

You are reading Chapter 0 of the class notes. A set of fairly extensive notes will be handed out in each class. The notes will make it generally unnecessary to take notes in class, and they will make it easy to review the material later. The notes, however, are not a substitute for the lectures. The lectures will provide more motivation than the notes and different examples, though perhaps less detail. Some blank space is provided for you to take extra notes, doodle, etc.

## 0.5. Manuals and outside reading.

Five manuals are being distributed with this introduction for your use during and after the course. There are frequent references in the class notes to passages in the manuals. Each manual is codified by an abbreviation in the reference, as follows:

LRM - Language Reference Manual
CPG - Checkout Compiler Programmer's Guide
OPG - Optimizing Compiler Programmer's Guide
CTUG - Checkout Compiler TSO User's Guide
OTUG - Optimizing Compiler TSO User's Guide

The number that follows a manual code, as in LRM 57, is not a page number but rather an entry number in a reference list which is being supplied separately. The entries in the reference list give page numbers and text to identify the beginning and end of each passage.

Manual references are made for two reasons: in a few cases, to point you to well documented details that it would be silly to copy in the class notes; more often, to point you to material you can use for review, and for a different perspective, after it is covered in the notes. You are urged to read all the references, though time may not permit you to read the longer passages during the course.

Unfortunately, you will find that the passages do not always correspond in scope to the material treated in the notes; they will frequently reference related topics that we won't cover until later, and they may mention details that we don't cover at all. Be alert for terms we haven't covered; try to skip, on the first reading, anything that looks foreign.

If you do pursue most of the references, you will acquire a great deal of familiarity with the manuals and with their organization. You won't be afraid later to look something up, because you will have a pretty good idea of where to look. Actually, that is another goal of the course.

It would be a good idea for you to browse through the tables of contents of the manuals now. You will notice a great deal of duplication in the two programmer's guides, and in the two TSO user's guides.

There is also a Messages Manual available for each compiler, though these aren't essential to own. And if, somehow, this course leaves you gasping for more, go out and get the Execution Logic Manuals for each compiler.

There are a few reasons why we don't generally recommend books on PL/I. We haven't evaluated many. Those we have seen have been disappointingly incomplete, erroneous, or obsolete more often than not. Several books are in preparation by authors known personally by the instructor; it is expected that these will be commendable.

## 0.6. Homework.

Each set of notes has several homework problems based on the material taught in that class. The purposes of the homework are to allow you to test your understanding of the material, to give you some experience with the language concepts, and to lead you through a discovery of some revealing insights that will, hopefully, influence your program design and coding style. For this reason you are strongly urged to attempt the homework problems in a timely manner. You will not be required to turn in completed homework; however, if you do, the instructor will go over your work, make comments, and return it to you.

## 0.7. Running programs.

You are not generally asked to run programs as part of the homework; for about half the course you are not even asked to write whole programs. However, once you have learned enough to write a program, you may find it instructive to run it. Follow these guidelines.

Use the Checkout Compiler in batch. It will be preferable for you to punch your program and data on cards, for now.

Source programs can be free form. You can start a statement in any column within the source margins (see below). You can put as many blanks as you want between language keywords, identifiers, constants, and special symbols. A semicolon marks the end of a statement, which may continue over as many cards as necessary. A comment, which is any text surrounded by /* and */, can be written wherever arbitrary blanks are permitted, as described above.

Standard (default) source margins are columns 2 and 72. Leave column 1 blank, and do not write source text beyond column 72.

Use the following JCL if your program consists of a single "external procedure," i.e., a main program that doesn't need to be link-edited with any subroutines.

```
// job card -- express limits are adequate
   account card
// EXEC PLCCG  (Note:  not PLCCLG)
//SYSCIN DD *  (Note:  not SYSIN)
      :
      :
      source program (column 1 blank)
      :
      :
      :
/*
//GO.SYSIN DD *
      :
      :
      data
      :
      :
/*
```

Can be omitted if no data.

External procedures aren't mentioned until Lesson 4.

If your program has several external procedures, i.e., a main program and subroutines to be link-edited together, use the following JCL.

```
// job card
   account card
// EXEC PLCCLG      (different cataloged procedure)
//SYSCIN DD *
      :
      :
      main program
      :
      :
*PROCESS;           This starts in column 1!
      :
      :
      subroutine
      :
      :
*PROCESS;
      :
      :
      subroutine
      :
      :
      :
/*
//GO.SYSIN DD *
      :
      :
      data                  Optional
      :
      :
/*
```

0.8. After the course.

Let us hear from you! We want to know how you are doing. The consultant can provide help with particular problems.

For FORTRAN-type computations, PL/I can be about as efficient at run time as FORTRAN. Certain features, because of their power and generality, are inevitably and inherently less efficient, but then many have no direct counterparts in FORTRAN. If you are unsatisfied with the performance of your programs, the consultant might be able to help you find some simple adjustments to make to tune it. There are a variety of optimization features you have to ask for explicitly. The compilers themselves can be made more efficient if future use warrants it.

If you encounter bugs in the compiler, report them to the consultant! IBM wants to find them and fix them, because it strongly supports PL/I. Anyway, chances are we have a later version of the compiler around which we are checking before releasing it. We can tell you how to STEPLIB to it.

Finally, after half a year or a year of experience, you may find it useful to reread the notes. You will be in a better position to appreciate and utilize some of the advanced features of the language.

# 1. Variables, attributes, and declarations; arithmetic data types, arithmetic expressions, precision rules.

## 1.1. Variables

A variable has a name, which is an identifier. It is located somewhere in storage, and it has a value.

Basically, identifiers may be up to 31 characters long. Examples are:

```
TIME_OF_FLIGHT
CHANNEL#
COEFFICIENT_OF_EXPANSION
X21
```

Rules for identifiers are given at LRM 1. A language keyword (such as DO) may be used as an identifier; language keywords are not "reserved."

Much more will be said about variables later. We will look again at variable names in Lesson 4 and at their locations in Lesson 5.

## 1.2. Attributes

In addition to a name, a location, and a value, every variable has some attributes, which are characteristics that tell the system exactly how the bits stored in its location represent its value.

See LRM 2 and LRM 3.

## 1.3. Declarations

Names and attributes are associated with variables by the process of declaration. The DECLARE statement may be used to "declare" one or more variables. Simple forms of the DECLARE statement useful for

present purposes are as follows:

```
DECLARE id attributes;
DECLARE id1 attributes1,
        ...,
        idn attributesn;
DECLARE (id1,...,idn) attributes;
```

In the above, *id* is the name of the variable and *attributes* is a list of attribute keywords. The first form declares a single variable. The second declares several, with potentially different attributes, in one DECLARE statement. The third declares several with a common set of attributes. DECLARE may be abbreviated DCL, as in the following examples:

```
DCL X FLOAT BINARY;
DCL Y FIXED DECIMAL,
    Z FLOAT BINARY;
DCL (U,V,W) COMPLEX FLOAT BINARY;
```

The definitive rules for the DECLARE statement, which go far beyond what we need now, are at LRM 4.

## 1.4. Types of declarations

<u>Explicit</u>: By use of DECLARE statement.
<u>Contextual</u>: Certain uses of identifiers, in the absence of an explicit declaration, result in a contextual declaration of a variable with that name and attributes deduced from context. The contexts for which this is possible are those that require particular attributes and cannot tolerate other alternatives. Arithmetic attributes are never deduced from context; there are many alternative arithmetic attributes, any of which can be used in any arithmetic context.
<u>Implicit</u>: An identifier which is neither explicitly declared nor used in a context resulting in a contextual declaration is implicitly declared as the name of a variable, which is given certain default attributes. The language specifies a set of defaults which are, in fact, particular arithmetic attributes. The programmer can change the defaults with the DEFAULT statement, which is considered in Lesson 4.

See LRM 5 and LRM 6. The latter text uses many terms and concepts that we will not consider until Lesson 4; try to ignore them for now.

## 1.5. Arithmetic data types

As in FORTRAN, there are many arithmetic data types and corresponding attributes (many more, in fact). All arithmetic variables have four characteristics, chosen from four sets of alternatives. The sets are as follows:

Mode: The choices are REAL and COMPLEX. Note that, in PL/I, REAL only means not COMPLEX; it does not mean floating-point, as it does in FORTRAN.
Scale: The choices are FLOAT and FIXED. FIXED means the decimal point is assumed to be in a fixed position relative to the internal representatition of the variable's value. However, that position need not be the right-hand edge; it can be, in which case you have roughly the equivalent of FORTRAN's INTEGER, but it may be specified to be elsewhere. FLOAT means the assumed position of the decimal point is not in a fixed place; it floats from place to place with the gross magnitude of the variable's value (floating-point hardware is used).
Base: The choices are BINARY and DECIMAL. Any reference to digits refers to either bits, if binary, or decimal digits, if decimal.
Precision: This is a number specifying the number of digits to be used for the internal representation of the variable's value. For fixed-point variables it specifies the exact number of digits that participate in operations on the variable according to the rules of the language. For floating-point variables it specifies the minimum number of digits that participate in operations on the variable according to the rules of the language. For fixed-point variables (only), precision includes, in addition to the number of digits, another number called the scale factor. This essentially denotes how many of the digits are to the right of the assumed decimal point. A scale factor of 0 means the value of the variable is always an integer and that the smallest difference in two different values that the variable can have is 1. A positive scale factor means the decimal point is assumed to be so many digits left of the least significant digit position. For instance, a scale factor of 1 means the value of the variable always has one fractional digit; the "resolution" of such a variable is thus one-half, if the base is binary, or one-tenth, if decimal. A positive scale factor may even exceed the number of digits specified for the variable, in which case all of the digit positions between the high-order one (leftmost) and the assumed position of the decimal point, which is even farther to the left, are assumed always to contain zeroes. A negative scale factor means the decimal point is assumed to be so many digits to the right of the least significant digit position, with the intervening digits assumed always to contain zeroes. Thus, with a scale factor of -1, the resolution is two, for binary base, or ten, for decimal; the value represented is always an integer. A better way of thinking about the scale factor is as follows. Suppose the precision is (p,q), i.e., the number of digits is p and the scale factor is q. Then first consider those p digits to represent a p-digit integer, say U. The value of the variable is then actually $U \cdot b^{-q}$, where b is either 2 or 10, according to the base.

Beware of the following differences from FORTRAN:

(a) In FORTRAN, REAL means floating-point and not complex. In PL/I it only means not complex; the variable may be either fixed-point or floating-point.

(b) In FORTRAN, COMPLEX means floating-point and in the complex, as opposed to real, domain. In PL/I it does not imply floating-point.

(c) In FORTRAN, INTEGER means fixed-point integer in the real domain. In PL/I you can have fixed-point integers in the complex domain.

References will be given later.

## 1.6. Attributes and declarations for arithmetic data.

By example:

DCL X REAL FIXED BINARY(15,0);

The value of X is a real binary integer. The number of digits is 15; the scale factor, 0. The range of the variable is $-2^{15}$ to $+2^{15}-1$, with a resolution of 1.

DCL X REAL FIXED BINARY(15);

Same as above. If omitted, the scale factor is assumed to be 0.

DCL Y COMPLEX FIXED BINARY(15);

Y has both a real and an imaginary part, each with the properties of X, above.

DCL Z FIXED DECIMAL(5,2) REAL;

The value of Z is a real decimal number with two fractional decimal digits and three in the integral part. The range of the variable is $-(10^5-1)\cdot 10^{-2}$, i.e., -999.99, to $+(10^5-1)\cdot 10^{-2}$, i.e., +999.99, with a resolution of $10^{-2}$.

DCL U FIXED DECIMAL(2,5) REAL;

U has a range of $-(10^2-1)\cdot 10^{-5}$ to $+(10^2-1)\cdot 10^{-5}$, i.e., -.00099 to +.00099, with a resolution of $10^{-5}$.

DCL T REAL DECIMAL FIXED(2,-5);

T has a range of $-(10^2-1)\cdot 10^5$ to $+(10^2-1)\cdot 10^5$, i.e., -9900000 to +9900000, with a resolution of $10^5$.

DCL R REAL FLOAT BINARY(21):

The value of R is a real number represented in floating-point. The range of the representable values is not a property of this declaration; it is a property of the implementation, i.e., the underlying hardware. For IBM 360/370 hardware this is approximately $-2^{252}$ to $+2^{252}$. The resolution is not uniform over this range. The absolute value of the smallest non-zero number that can be represented is approximately $2^{-260}$. The precision specification of 21 digits (bits) means that the most significant 21 bits (and maybe more) of the value are retained; where the decimal point is in relation to these is carried in the information contained in the exponent field in the hardware realization of the value.

DCL R1 REAL FLOAT BINARY(31);

R1 cannot be less precise than R since the 31 most significant bits (and maybe more) are retained.

DCL S REAL FLOAT DECIMAL(6);

The value of S is also a real number represented in floating-point. On IBM 360/370, the range, expressed in decimal terms, is approximately $-10^{75}$ to $+10^{75}$. The absolute value of the smallest non-zero number that can be represented is approximately $10^{-78}$. At least the 6 most significant decimal digits are retained.

DCL C COMPLEX DECIMAL FLOAT(6);

The value of C is a complex number represented in floating-point. The real and imaginary parts each have the properties of S, above.

## 1.7. Hardware implementation of arithmetic data

The intent of PL/I is to free the programmer from the need to consider the hardware representations of data. Ideally, precisions should be chosen based on the requirements of the problem. The precisions specified will then have the same implications on the behavior of the data on all implementations (providing no maximum precisions are violated). Often, however, the programmer is interested in economy (storage or time) with respect to one implementation, and precisions are chosen based on knowledge of the amount of storage which that implies for that hardware. Such programs are still portable, of course, but the efficiency considerations may not match the "other" hardware very well.

For machine equivalents between FORTRAN and PL/I arithmetic data types, see LRM 7. For a summary of storage requirements, see LRM 20.

## 1.8. Language defaults for arithmetic attributes

If a variable is not declared explicitly or contextually, it acquires the following attributes implicitly.

| First letter of Identifier | Default Attributes |
|---|---|
| I-N | REAL FIXED BINARY(15) |
| Other | REAL FLOAT DECIMAL(6) |

If some, but not all four, of the arithmetic attributes (mode, scale, base, precision) are explicitly declared, the remainder are chosen from complicated defaults. The only one that may <u>safely</u> be omitted

is mode: REAL is always assumed.

Default precisions are defined by the implementation, not the language; they may differ amongst implementations.

For all the gory details, see LRM 8 - LRM 12.

## 1.9. Implementation maximum precisions

See LRM 12.

## 1.10. Use of arithmetic data

New arithmetic values are "generated" by:
- (a) Reference to arithmetic constants.
- (b) Input operations.
- (c) Arithmetic operations on other arithmetic values.
- (d) Certain operations on other things.

They are propagated by assignment.
They may be used in diverse ways, some of which are:
- (a) Arithmetic operations.
- (b) Comparison operations (Lesson 2).
- (c) Output operations (Lessons 7-9).
- (d) Subscripting (Lesson 3).

## 1.11. Arithmetic constants

Arithmetic constants denote, by the way they are written, objects that have (always) the indicated arithmetic value <u>as well as the indicated attributes</u>. It is important to realize that all arithmetic constants have attributes of mode, base, scale, and precision which are determined by how the constants are written.

REAL FIXED DECIMAL constants are comprised of the decimal digits, an optional sign, and an optional decimal point. The number of digits of precision is the number of decimal digits written; the scale factor is the number of them which are to the right of the decimal point. Examples:

| Constant | Precision |
|---|---|
| 1 | (1,0) |

| | |
|---|---|
| 3.14159 | (6,5) |
| -.008 | (3,3) |

REAL FIXED BINARY constants are similar, except that one uses only the binary digits and follows them with a B. Examples:

| Constant | Precision |
|---|---|
| 1B | (1,0) |
| 101.11B | (5,2) |
| -.0101B | (4,4) |

REAL FLOAT DECIMAL constants are written as REAL FIXED DECIMAL constants followed by an E and an optionally signed exponent. The number of digits of precision is the number of digits written. Examples:

| Constant | Precision |
|---|---|
| 1E0 | (1) |
| 1.648E+24 | (4) |
| -.0031E-37 | (4) |

REAL FLOAT BINARY constants are similar, except that only the binary digits are used to the left of the exponent and the exponent is followed by a B. The exponent is written with decimal digits but is interpreted as a power of 2. Examples:

| Constant | Precision |
|---|---|
| 1E0B | (1) |
| 1110.0010E-3B | (8) |
| -.10001E+06B | (5) |

There are no complex constants in PL/I, but there are imaginary constants. An imaginary constant is any real constant followed by an I. Examples:

1I
-.0101BI
3.48E+51I
-11.01E-22BI

Constant complex values can be written as expressions, as in the following:

```
3+4I
-.25F0+.75E0I
```

A review of this material can be found at LRM 13. This reference, as well as LRM 12, covers default and maximum precisions.

Be aware of several differences from FORTRAN:

(a) A constant such as 5 denotes a binary integer in FORTRAN and a decimal integer in PL/I. However, it is not necessary to write this constant as 101B in PL/I if the compiler can tell that a binary integer is needed (which it almost always can); it will substitute the equivalent binary integer.

(b) A decimal point is sufficient to denote floating-point in FORTRAN. 5.0 is a fixed-point constant in PL/I; it has a scale factor of 1 (remember, fixed-point data can have fractional parts). Again, if the compiler can tell that a floating-point constant is required, it will substitute the equivalent floating-point constant.

(c) To get a double-precision floating-point constant you merely write the required number of digits; there is no D exponent character as in FORTRAN. If you have written a single-precision floating-point constant where double-precision is required, the compiler substitutes a double-precision constant obtained by supplying low-order zeroes. Thus, the nearest equivalent to FORTRAN's 0.1D0 is 0.100000E0 (on our implementation). (The fact that you only need 7 digits, and not 16, the maximum for double-precision, is a consequence of our implementation and not the language rules.)

Do not become paranoid over this! If you initially do what seems natural, you will most often be right. Some knowledge of, and experience with, the precision and conversion rules, as well as our compilers, will prepare you for the few cases where what seems natural is not right.

## 1.12. Scalar arithmetic assignments

Assignment of an arithmetic value to an arithmetic variable may require conversion of that value to an "equivalent" one having the attributes of the target. The conversion occurs automatically and is determined by conversion rules.

Forms of the <u>assignment statement</u> are:

*variable* = *expression;*

*variable*$_1$,...,*variable*$_n$ = *expression;*

The letter form denotes <u>multiple assignment</u> of the value of the expression (which is only evaluated once) to each of the variables (which may have different attributes). See LRM 14.

## 1.13. Conversion rules for arithmetic assignments

In converting REAL to COMPLEX, a zero imaginary part is supplied. Going the other way, the imaginary part is just dropped.

Other conversions are more or less obvious: they try to preserve the value being assigned, if possible. If change of base is required, low-order accuracy may be lost in going from decimal to binary because some decimal fractions with finite representations do not have finite binary representations. In general, with change of base, one decimal digit corresponds to about 3.32 binary digits (bits). The consequences of insufficient precision in the target depend on whether the target is floating-point or fixed-point. If it is floating-point, low-order accuracy may be lost. Examples of this situation are:

```
FLOAT DECIMAL(16) to FLOAT DECIMAL(6).
FLOAT DECIMAL(16) to FLOAT BINARY(21).
FIXED DECIMAL(8, x) to FLOAT DECIMAL(6)
     for any scale factor x.
FIXED BINARY(31) to FLOAT DECIMAL(6).
```

That is, as long as the target is floating-point, the consequences of insufficient precision in the target are not influenced by base or scale conversion. If the target is fixed-point, there are two possible consequences (which also are not influenced by base or scale conversion). Loss of low-order accuracy is possible due to the limited resolution implied by the scale factor of the target. For instance, in assigning to an integer, i.e., a fixed-point variable with a scale factor of 0, any fractional part is lost; this is common in FLOAT to FIXED (integer) conversions. A worse situation occurs when the target does not have enough high-order digit positions to accommodate all the non-zero high-order digits of the value being assigned. Exactly what happens in this case will be covered later (Lesson 6). This situation could easily occur in FLOAT to FIXED conversions, regardless of the precisions involved, because of the very large values that can be represented in floating-point.

The assignment of a constant to a variable is one case in which the

compiler "converts" constants at compile time.

## 1.14. Arithmetic operations

As in FORTRAN, they are prefix and infix <u>addition</u> (+) and <u>subtraction</u> (-), <u>multiplication</u> (*), <u>division</u> (/), and <u>exponentiation</u> (**).

A few differences from FORTRAN:

(a) Prefix + and - have the highest priority (equal to that of **) instead of the lowest (equal, in FORTRAN, to that of infix + and -).

| Example | Interpretation in: FORTRAN | PL/I |
|---|---|---|
| A=-B*C | A=-(B*C) | A=(-B)*C |
| A=-B-C | A=(-B)-C | A=(-B)-C |

(b) A prefix operator may follow another operator, e.g., the following are allowed in PL/I but not FORTRAN:

| | |
|---|---|
| A+-B | A**-B |
| A/-B | --B |

(c) In exponentiation of a complex value, the exponent (second operand) may be complex. In FORTRAN it must be not only not complex but also an integer.

(d) Division of fixed-point integers (i.e., values with a scale factor of 0) does <u>not necessarily</u> yield an integer, as it does in FORTRAN. (See below.) This often causes people trouble.

## 1.15. Conversion rules for arithmetic operations

The two operands of an infix operator (except exponentiation) must have the same mode, scale, and base. If mode, base, or scale differs, conversion occurs as follows:

(a) If the modes of the operands differ, the REAL operand is converted to COMPLEX by supplying a zero imaginary part.

(b) If the bases differ, the DECIMAL operand is converted to BINARY (its precision being increased by a factor of 3.32, approximately, because it will have to represent bits instead of decimal digits). Caution: if B is FIXED BINARY in .1*B, the FIXED DECIMAL constant .1 will be converted (at compile time) to a FIXED BINARY constant with a value of one-sixteenth, not one-tenth, since its precision will be (5,4). A value closer to one-tenth is obtained if you write .10 or .100, etc.

(c) If the scales differ, the FIXED operand is converted to FLOAT having the same number of digits.

By the above rules we will have obtained operands that (may) differ

in precision only. The result will have the same mode, base, and scale, and a precision defined by the precision rules for arithmetic operations (see below).

For exponentiation, see LRM 15.

The use of a constant as an arithmetic operand is another case in which the compiler "converts" constants at compile time.

## 1.16. Precision rules for arithmetic operations

These rules are concerned with the precision of the result of addition, subtraction, multiplication, or division, when the operands have the same mode, base, and scale.

FLOAT scale is easy: the precision of the result is the larger of those of the operands (for all the operations). Promotion of the "shorter" operand from single to double precision, or double to extended, is done by supplying low-order zeroes.

The formulas for the fixed-point precision rules seem complicated, but they derive from simple principles. Basically, the goal is to retain as much precision as possible, both high-order and low-order, without excess precision.

In what follows, let the operands have precisions (p1, q1) and (p2,q2) respectively, and denote the precision of the result by (p,q). For all sets of indices, let r=p-q (the number of digits to the <u>left</u> of the decimal point).

Addition and subtraction: If you were to write a pair of operands one above the other, with decimal points aligned, you would see that no precision is lost if the number of fractional digits of the result is the greater of the numbers of fractional digits of the operands (i.e., q=max(q1,q2)) and if the number of integral digits of the result is one more than the greater of the numbers of integral digits of the operands (the additional digit allows for a carry) (i.e., r=1+max(r1,r2)).

Example:

```
     xx.xxxx
± xxx.xx
  xxxx.xxxx
```

Substituting for the r's, we get $p=1+\max(p_1-q_1,p_2-q_2)+\max(q_1,q_2)$. If this formula yields a value of p in excess of the maximum permitted by the implementation, for the given base, that maximum is used instead.

Multiplication: Playing the same game, you see that the number of fractional digits of the result needs to be the sum of the numbers of fractional digits of the operands, and likewise for the integral digits. Example:

```
    xx.xxxx
* xxx.xx
xxxxx.xxxxxx
```

However, when you consider what happens in complex multiplication, you will see that one more digit is needed on the high-order end. Thus, $q=q_1+q_2$, $r=1+r_1+r_2$. Therefore, $p=1+p_1+p_2$, subject to the limitation on the implementation maximum number of digits.

Division: This is the weird one. Clearly, the fractional part of the quotient could go on forever. So, to retain as much precision as possible the result must have the maximum number of digits. As many as necessary for the worst case are used for the integral digits with the rest assigned to the fractional digits (thus determining the scale factor). The worst case occurs with a maximum dividend and minimum non-zero divisor, yielding $r=r_1+q_2$. The final result is p=N (the maximum for the given base) and $q=N-((p_1-q_1)+q_2)$. Notice the consequences of this. A/2 in PL/I may have a fractional part, unlike FORTRAN. (It will if A is FIXED BINARY(15), for example.) Furthermore, the fractional part will be exactly represented so that A/2*2 will equal A and not A-1 (as it does in FORTRAN when A is odd). Clearly, you can see that the PL/I rule gives a more accurate result than the FORTRAN rule.

Note that the precision rule for division introduces a weak implementation dependence into the actual numerical results that may be obtained, in fixed-point division, although most realistic programs will not be affected by it.

The resultant precision of exponentiation is given at LRM 15.

For additional information, see LRM 16, OTHER 1, and OTHER 2.

## 1.17. Arithmetic builtin functions

PL/I has a large set of builtin functions which are akin, generally, to the FORTRAN "intrinsic functions." The general treatment of builtin functions is in Lesson 10; however, those applicable to arithmetic and mathematics are initially covered now.

The arithmetic builtin functions perform certain basic operations or conversions on arithmetic values. They are "generic" in the sense that a wide variety of attributes are permitted for the arguments. The attributes of the result are, in many cases, derived from the attributes of the arguments.

Detailed information can be found at LRM 19 and relevant portions of LRM 18. The functions are listed below, with brief indications of their use. See also LRM 29.

| | |
|---|---|
| ABS | Absolute value of real quantity; modulus of complex quantity. |
| MAX | Maximum of several real quantities. |
| MIN | Minimum of several real quantities. |
| REAL | Real part of complex quantity. |
| IMAG | Imaginary part of a complex quantity (the result is real). |
| MOD | Remainder on division of real quantities. |
| SIGN | Sign of a real quantity (as +1,0, or -1). |
| COMPLEX | $a+bi$ for real quantities $a$ and $b$. |
| CONJG | Complex conjugate of a complex quantity. |
| FLOOR | Largest integer less than or equal to a real quantity (result has same scale as argument). |
| CEIL | Smallest integer greater than or equal to a real quantity. |
| TRUNC | Truncation of a real quantity to an integer. Truncation is towards zero, so TRUNC=FLOOR for positive arguments and TRUNC=CEIL for negative ones. |
| ROUND | A real value rounded in the specified digit position (not useful for floating-point). |
| BINARY, DECIMAL, FIXED, FLOAT | Conversion to the indicated base or scale with an optionally specified precision. If not specified, the conversion rules determine the precision. Other attributes remain unchanged. |

| | |
|---|---|
| PRECISION | Conversion to the given precision. Other attributes remain unchanged. |
| ADD, MULTIPLY, DIVIDE | Operations carried out in the given precision instead of that determined by the precision rules. See LRM 28. |

Note that the DIVIDE builtin function can be used to overcome the (weak) dependency of fixed-point division on the implementation maximum precision.

## 1.18. Arithmetic pseudo-variables

Some builtin functions can be used, with suitably restricted arguments, on the left-hand side of an assignment statement. In that form they are known as pseudo-variables. The restrictions on the argument (or arguments, in some cases) guarantee that some portion of the storage belonging to a variable is being addressed.

Three of the arithmetic builtin functions can also be used as pseudo-variables:

| | |
|---|---|
| REAL | For assignment to the real part (only) of a complex variable, e.g., REAL(Z)=1E0; |
| IMAG | As for REAL, but the imaginary part, e.g., IMAG(Z)=5E-01; |
| COMPLEX | For assignment of the real part of a complex value to one real variable and the imaginary part to another real variable, e.g., COMPLEX(X,Y)=Z. Note: the proposed ANSI standard does not include the COMPLEX pseudo-variable. |

## 1.19. Guidelines on choice of arithmetic attributes

Use FLOAT when a variable has a very wide range of values, and "enough" precision. There are no significant differences between FLOAT BINARY and FLOAT DECIMAL in our implementation since both are implemented with the 360/70 "float hexadecimal" hardware.

There is both binary and decimal fixed-point hardware, but binary is generally more "efficient" and is to be preferred. Certain uses of arithmetic values, such as for subscripting, require binary base (conversion is performed, if necessary). Operations involving powers of ten may indicate the use of decimal base.

## 1.20. Mathematical builtin functions

The following mathematical builtin functions, some of which have counterparts among the intrinsic functions of FORTRAN, are available in PL/I:

| | | |
|---|---|---|
| ACOS | ERF | SINH |
| ASIN | ERFC | SQRT |
| ATAN | EXP | TAN |
| ATAND | LOG | TAND |
| ATANH | LOG2 | TANH |
| COS | LOG10 | |
| COSD | SIN | |
| COSH | SIND | |

All operate on floating-point arguments (conversion is performed, if necessary) and yield floating-point results. These functions are generic in the sense that either base or mode is allowed for the argument, the result having the same base and mode; likewise, any precision is allowed. (Certain of these require REAL arguments; example: ERF.)

Caution: As of September 19, 1975, the following mathematical builtin functions are not in the proposed ANSI standard for PL/I: ACOS, ASIN, ATANH, COSH, ERF, ERFC, SINH, and TANH. There has been some effort to restore them. If they are not restored, they will be available in a particular implementation, as an extension, only if the vendor sees fit to provide them.

See LRM 17 and relevant parts of LRM 18.

## 1.21. Unanswered questions

We have already posed the question "What happens when a fixed-point assignment target has insufficient precision to receive the high-order non-zero digits of a value being assigned?" Other questions to be answered in Lesson 6 are:

What happens when you try to compute a fixed-point value that is too "big" for the hardware?

Similarly, for a floating-point value.

Similarly, for a too-"small" floating-point value.

What happens when the argument of a mathematical builtin function is "bad"? Example: a real (not complex) -1 for SQRT.

## 1.22. Homework problems

(#1A) What are the attributes (including precision) of the following arithmetic constants?

```
629          6E-1
3478.        0.05E0
.1B          11.E0B
```

(#1B) What are the ranges and resolutions of variables having the following attributes?

```
REAL FIXED DECIMAL (3)
REAL FIXED DECIMAL (3,2)
REAL FIXED DECIMAL (3,4)
REAL FIXED DECIMAL (3,-1)
REAL FIXED BINARY  (4)
REAL FIXED BINARY  (4,3)
REAL FIXED BINARY  (4,7)
REAL FIXED BINARY  (4,-2)
```

(#1C) In the following, what are the attributes of the constants, as written, and to what attributes will they be converted according to the conversion rules? What are the values of the converted constants?

```
N+1       (N is FIXED BINARY(15))
X+1       (X is FLOAT BINARY(21))
.5*X
.5*N
1.1*N
5E-1*Y    (Y is FLOAT DECIMAL(6))
5E-1*Z    (Z is FLOAT DECIMAL(16))
```

(#1D) What arithmetic builtin functions could you use in a modification of J/2*2, for J FIXED BINARY(15), that would give the same results as FORTRAN (i.e., how can you force the division to behave like FORTRAN's integer division)? Write the modified expression. Note that there are several possibilities.

## 2. String data types; string and logical expressions.

### 2.1. Character string values.

Character string values are elementary values like arithmetic values (i.e., they can be the operands or results of certain operations). A character string value is a sequence of characters. In addition to its identity (the sequence itself), a character string value has another property: the length of a character string value is the number of characters in the sequence. ABC is a character string value of length 3.

### 2.2. Bit string values.

Like character string values except that the sequence is a sequence of 0 or 1 bits. 1010 is a bit string value of length 4.

### 2.3. String variables.

Character (bit) string variables are variables that can acquire character (bit) string values.

When string variables are declared, with the CHARACTER (abbreviation: CHAR) or BIT attribute, the length of the string values to be stored in the variables must be specified. Examples:

```
DCL  C  CHAR (20);
DCL  J  CHAR (5);
DCL  Q  BIT (1);
DCL  F  BIT (33);
```

The maximum length of a string value in our implementations is 32767.

Another attribute applicable to string variables will be given later.

### 2.4. Use of character string data.

New character string values are "generated" by:

(a) Reference to character string constants.
(b) Input operations.
(c) String operations on other character string values.
(d) Certain operations on other things.

They are propagated by assignment.
They may be used in diverse ways, some of which are:

(a) String operations.
(b) Comparison operations.
(c) Output operations. (Lessons 7-9)

## 2.5. Character string constants.

The string value is enclosed in single quotes. If a single quote is to be a character in the sequence constituting the string value, it must be written twice. Examples:

| Constant | Character string value |
|---|---|
| 'ABC' | ABC |
| 'b' | b (b is a blank) |
| 'IT''S' | IT'S |
| '''A''' | 'A' |

A long constant which is the repetition of a shorter constant may be written with a <u>repetition factor</u>, as in the following examples:

| Constant | Character string value |
|---|---|
| (3)'XY' | XYXYXY |
| (8)'b' | bbbbbbbb |
| (4)'''' | '''' |

## 2.6. Fixed-length scalar character string assignments.

The kinds of string variables described above are termed <u>fixed-length</u> string variables, because their values always have exactly the length specified. In assignment of a character string value to a fixed-length character string variable, using the assignment statement, the character string value being assigned is either truncated on the right, or extended on the right with blanks, if necessary, to make it conform to the length of the target. Example:

```
DCL  C  CHAR (6);
C = 'ABCD';  The value of C after assignment is the
             6-character sequence ABCDbb.
C = 'ABCDEFG';  Here, it is ABCDEF.
```

To review, see LRM 21.

2.7. Use of bit string data.

New bit string values are "generated" by:

(a) Reference to bit string constants.
(b) Input operations.
(c) String operations on other bit string values.
(d) Logical operations on other bit string values.
(e) Comparison operations.
(f) Certain operations on other things.

They are propagated by assignment.
They may be used in diverse ways, some of which are:

(a) String operations.
(b) Logical operations.
(c) Comparison operations.
(d) Output operations. (Lessons 7-9)

2.8. Bit string constants.

The string value, written with 0's and 1's, is enclosed in quotes and followed by a B. Repetition factors are allowed. Examples:

| Constant | Bit string value |
|---|---|
| '1'B | 1 |
| '00110'B | 00110 |
| (2)'111'B | 111111 |

2.9. Fixed-length scalar bit string assignments.

By analogy with fixed-length character string assignments, a bit string value being assigned is either truncated or extended on the right, if necessary, to make it conform to the length of the target. Extension is with 0-bits. Example:

```
DCL  B  BIT (5);
B = (2)'1'B;  The value of B after assignment is
              the 5-bit sequence 11000.
B = (2)'1100'B;  Here, it is 11001.
```

To review, see LRM 22 and LRM 23.

2.10. Conversions between bit string and character string.

This conversion is required, for example, when assignment of a bit string value is made to a character string variable, or when assigning a character string value to a bit string variable.

Bit-to-character conversion results in a string of the same length with 0-bits becoming the character 0 and 1-bits becoming the character 1.

Character-to-bit conversion proceeds as above, but in reverse. Only the characters 0 and 1 are permitted in the character string value being converted. In Lesson 6 we will see what happens when this rule is violated, and what the program can do about it.

2.11. Conversions between string and arithmetic data.

If conversion from string to arithmetic is required, it proceeds as follows:

Bit string values are interpreted as unsigned binary integers.

Character string values must represent valid arithmetic constants (possibly surrounded by blanks). The arithmetic constant represented by the character string value will have a self-denoting mode, base, scale, and precision. In a context where the target arithmetic attributes are independent (such as in assignment), the conversion occurs (interpretively) according to the rules of arithmetic conversions for the specific source arithmetic type represented by the character string value. However, in a context where any arithmetic attributes would be permissible (such as the operand of an arithmetic operation), the arithmetic constant represented in the character string value is first converted to DECIMAL FIXED (15,0), interpretively; that intermediate target may require further conversion, depending on the operation and its other operand.

When arithmetic values are to be converted to string, the context may or may not determine whether character strings or bit strings result (some contexts permit either). In this case conversion is to bit string if the base is binary and character string if it is decimal.

Conversion from arithmetic to bit string proceeds by obtaining first a binary integer from the arithmetic value (ignoring both the sign and any fractional part). The precision of the binary integer depends on the precision of the source. That integer is then considered to be a bit string value.

Conversion from arithmetic to character string proceeds by obtaining an equivalent decimal value (with the same mode and scale and the derived precision). The decimal value is then expressed in the form of a decimal arithmetic constant of the mode, scale, and derived precision, with leading zeroes replaced by blanks. This constant is then embedded in a character string of a length determined from the precision.

These rules, especially those for arithmetic to character string, are very complicated (see LRM 16 for all the details). A common case is conversion of a fixed-point value with zero scale factor (i.e., a binary or decimal integer value) to character. If the decimal precision of the arithmetic value is (p,0), the resulting character string will have a length of p+3. The arithmetic value will be assembled as the equivalent decimal constant in the low-order (rightmost) p digits (with leading zeroes replaced by blanks). The next character to their left will either be a minus sign or a blank, and the remaining characters will be blanks.

The important thing to realize is that there are defined conversions between all types of arithmetic and string data. (This generality can be a convenience or the cause of unexpected results.) Both types are often lumped together under a category called problem data because these are the only kinds of data that can be manipulated, or operated upon, in expressions; sometimes it seems the name is due to the problems the conversion rules cause.

## 2.12. String operations.

There is only one, concatenation. The infix operator is || (two vertical bars). Concatenation may be applied to either bit strings or character strings, yielding a result of the same string type. (If one operand is bit string, and the other character string, the bit string is converted to character string.)

Concatenation juxtaposes the two string values, yielding a string value whose length is the sum of the lengths of the operands. Examples:

```
DCL  A  CHAR (3),  B  CHAR (4);
DCL  C  BIT (6),   D  BIT  (2);
A = 'ABC';
B = 'DEFG';
C = '011011'B;
D = '00'B;
DCL  AB  CHAR (7),  CD  BIT (8);
AB = A || B;  The value of AB after the assignment is the
              7-character string ABCDEFG.
CD = D || C;  The value of CD is the 8-bit string 00011011.
```

See LRM 26.

## 2.13. Logical operations.

The logical operations and (&), or (|), and not (¬) operate on bit string values and produce a bit string result. Strings of any length may be used, and the operation proceeds bit-wise on the operands. If the operands of & or | are of unequal length, the shorter is extended on the right to the length of the longer, with 0-bits. Examples:

| Value of first operand | Operation | Value of second operand | Value of result |
|---|---|---|---|
| 011001 | & | 111100 | 011000 |
| 011001 | \| | 111100 | 111101 |
| 10 | & | 1111 | 1000 |
| 10 | \| | 1111 | 1111 |
| 101 | ¬ | | 010 |

Note: ¬ is a prefix operator. See LRM 27.

## 2.14. Comparison operations

The comparison operations =, ¬=, <, >, <=, >=, ¬<, ¬> may be applied to any pair of operands of "compatible" data type. If both operands are arithmetic, the comparison is algebraic (only = and ¬= are allowed for complex operands). If both operands are string, the shorter is extended on the right to the length of the longer, if necessary, using 0-bits if they are bit strings and blanks if they are character strings; the comparison then proceeds left to right in the strings using the character collating sequence of the hardware for character strings and the obvious comparison rules for bit strings. If the operands are not immediately compatible, conversion occurs according to the rules given at LRM 24 and LRM 25.

The result of a comparison operation (for any type of operand) is a one-bit string whose value is the single bit 1 if the comparison is true and 0 if it is false. (See LRM 24.) This definition permits comparison operations to be intermixed with other logical operations in an assignment statement. The most common use of comparison operations, however, is in the IF statement, as we shall see in Lesson 6. In that case, the one-bit bit string may not actually be generated in storage but may be represented in the state of the "condition code" of the hardware as it executes comparison instructions and conditional branches.

We have now seen all of the operations that may be used in operational expressions, i.e., computational expressions involving problem data. As in FORTRAN, any of the operations and any data types may be intermixed in any expression. For a discussion of the priority, or precedence, of operations in such mixtures, see LRM 30 and LRM 31.

One difference from FORTRAN should be noted: the "not" operator (¬) in PL/I has a different position in the hierarchy than its counterpart (.NOT.). In PL/I it has the highest precedence (equal to that of ** and prefix + and -) placing it, in particular, above the comparison operators. In FORTRAN it is below the comparison operators. The PL/I equivalent of .NOT. A .LT. B is not ¬A<B, which means (¬A) < B, but ¬(A<B). Of course, this may be written instead as A ¬< B or A >= B (in FORTRAN it could have been written as A .GE. B).

## 2.15. Varying-length string variables.

An additional attribute, VARYING (abbreviation: VAR), may be specified in declarations of character string and bit string variables. String variables which have been declared with the VARYING attribute are called varying-length string variables because the string values they may acquire are not restricted to have always the length specified in their declaration. They may acquire any string values of the declared length or less (hence, the declared length of a varying-length string variable is called its maximum length).

## 2.16. Varying-length scalar character and bit string assignments.

On assignment of a string value to a varying-length string variable, padding (with blanks or 0-bits) does not occur (as it does in fixed-length string assignments) if the string value being assigned is of shorter length than that declared for the target variable; the target variable receives the string value unpadded, and that is the value that will be used on any subsequent reference to the variable. Note, however, that if a string value longer than the declared (i.e., maximum) length of the target variable is assigned, truncation to that length occurs on the right, as in fixed-length string assignments. Examples:

```
DCL  A  CHAR (5),  B  CHAR (8)  VAR;
A = 'STR5A';
B = A;  The value of B is now the 5-character string STR5A.
B = B || 'ND';  Now it is the 7-character string STR5AND.
B = B || A;  Now it is the 8-character string STR5ANDS.
```

## 2.17. The null string value and null string constant.

Remember that string values are sequences of characters or bits. The sequence of length 0 is allowed; it is called the null string (note that the null character string value and the null bit string value have different data types).

The character string constant representing the null character string value is written as''. The bit string constant representing the null bit string value is written as''B. Examples:

```
DCL  A  CHAR (5),  B  CHAR (8)  VAR;
B ='';  B now has the null character string value.
A = B;  The value of A is the 5-character string bbbbb.  Why?
B = A ||'' || A;  B's value is now the 8-character string bbbbbbbb.
```

It is important to note that VARYING is a property of string variables and not string values (i.e., not expressions). A string expression involving string variables, some of which may be VARYING, has, for any particular evaluation, a value which has a particular length. VARYING addresses the fact that variables may take on values of different lengths at different times.

Whereas fixed-length string variables with declared length n require n bytes (or bits) of storage, varying-length string variables with declared (maximum) length n require n bytes (or bits) plus two more bytes. Storage is always reserved for the maximum length of the variable's value, and the additional halfword is used to record the length of the variable's current value. There is no legal way in PL/I to get access to bytes reserved for the value of a varying-length string variable, but not actually part of (i.e., needed for) its current value (there are lots of illegal ways!). It is entirely imaginable that some other implementation of PL/I may use an entirely different representation for varying-length string variables.

For additional information, see LRM 32 - LRM 36 (ignoring parts of LRM 36 involving things we haven't covered yet).

## 2.18. String-handling builtin functions.

One large group of builtin functions is concerned with string handling. Certain of these extend, in an essential way, the rather meager capabilities afforded by assignment and concatenation. Others could be programmed by the user (using loops and other things we haven't seen), so their existence is properly viewed as a matter of convenience and efficiency (the latter because of the tight in-line code usually generated by the compiler).

Full details are given at LRM 37 and LRM 18, but essential features are described here.

The LENGTH builtin function returns the length of the value of the string-valued expression which is its argument. When its argument is a fixed-length string variable, the result is the variable's declared length. In the case of a varying-length string variable, LENGTH returns the length of its current value. Examples:

```
DCL  U  CHAR (10)  VAR,  B  BIT (6)  VAR;
DCL  I  FIXED  BINARY;
U = 'ABCDE';
I = LENGTH(U);  Value of I is 5.
I = LENGTH(U || '.');  Value is 6.
B = '101'B;
I = LENGTH(B);  Value is 3.
B = ''B;
I = LENGTH(B);  Value is 0.
```

The SUBSTR builtin function is one of the most essential. It allows you to select a contiguous portion ("substring") of a larger string. One form of SUBSTR is

SUBSTR(*string-expr, arith-expr-1, arith-expr-2*).

Let *string-expr* have a value of length n. Let the values of *arith-expr-1* and *arith-expr-2* be i and j respectively. Then the result is the string of length j starting at the i-th character (or bit) of *string-expr*. (The first character or bit of a string value has position 1.) Constraints on i and j are as follows:

i must be >= 1 and <= n+1.
j must be >= 0 and <= n.
i+j must be <= n+1, in addition.

These constraints guarantee that the substring lies within the bounds of the string itself (the case i = n+1, j = 0 is a degenerate, limiting case). It is illegal to reference outside the bounds of a string using SUBSTR.

Note the following identities:

SUBSTR(*x*, 1, LENGTH(*x*)) = *x* for any *x*.
SUBSTR(*x*, *i*, 0) = the null string for any *x* and any *i* between 1 and LENGTH(*x*) + 1.
SUBSTR(*x*, 1, 1) is the first character (or bit) of any *x* whose length is not 0.
SUBSTR(*x*, LENGTH(*x*), 1) is the last character (or bit) of any such *x*.

Examples:

```
DCL  U  CHAR (10)  VAR,  T  CHAR (4)  VAR;
U = 'ABCDEF';
T = SUBSTR(U, LENGTH(U), 1);  Value of T is F.
T = SUBSTR(U, 1, 4);  Value of T is ABCD.
T = SUBSTR(U || T, LENGTH(T) - 1, LENGTH(U) - 1);
The above statement has the same effect as:
T = SUBSTR('ABCDEFABCD', 3, 5);  which assigns the 5-character
                                 string CDEFA to T.
```

Another form of SUBSTR is

SUBSTR(*string-expr*, *arith-expr*).

The substring starts at the position given by the second argument but in this case extends to the end of the string. Therefore,

SUBSTR($x$, $p$) = SUBSTR($x$, $p$, LENGTH($x$) - $p$ + 1).

Thus, while SUBSTR($x$, 1, 1) picks off the first character (or bit) of a string, SUBSTR($x$, 2) returns everything after that.

The remaining functions are:

| | |
|---|---|
| INDEX | Find the location of a pattern in a string. |
| VERIFY | Find the location of the first character (or bit) in a string which is not among a set of "acceptable" characters (or bits). |
| TRANSLATE | Map the characters (or bits) of a string as specified. Useful in terminal-oriented programs to translate input from lower to upper case. |
| REPEAT | Concatenate a string with itself a given number of times. |
| HIGH | Return a string of the specified length consisting of repetitions of the highest character in the collating sequence. |
| LOW | Same for lowest character. |
| CHAR | Convert to character string. |
| BIT | Convert to bit string. |
| BOOL | Used to obtain any of the 16 boolean functions of two bit strings (e.g., "implies," "exclusive or," etc.). |
| STRING | See Lesson 10. |
| UNSPEC | See Lesson 10. |

In the proposed ANSI standard, the function of REPEAT is taken over by a new builtin function, COPY. Other new functions are:

| | |
|---|---|
| BEFORE | Return the portion of a string before the first occurrence of a specified pattern. |
| AFTER | Same, but the portion after its occurrence. |
| DECAT | Sort of generalized BEFORE and AFTER. |
| REVERSE | Return the reverse of a string. |

## 2.19. String-handling pseudo-variables.

SUBSTR, UNSPEC, and STRING can be used as pseudo-variables. UNSPEC and STRING will be described in Lesson 10.

SUBSTR(*string-variable, arith-expr-1, arith-expr-2*), when used as a target in assignment, allows a string value (the right-hand side of the assignment statement) to be assigned to the substring of *string-variable* beginning at the position given by the value of *arith-expr-1* and extending for a number of characters (or bits) given by the value of *arith-expr-2*. The designated substring must be within the bounds of the *string-variable* (and if that is a varying-length string variable, within the bounds implied by its current length). The SUBSTR pseudo-variable may also be used in the two-argument form. Examples:

```
DCL  S  CHAR (10)  VAR;
S = 'ABCDEF';
SUBSTR(S, 3, 2) = 'XY';  Value of S is now the 6-character
                         string ABXYEF.
SUBSTR(S, 5) = 'Z';  Now it is ABXYZb.  Why?
```

Note that the SUBSTR pseudo-variable cannot change the length of its first argument, even when that is a varying-length string variable.

## 2.20. Pictured data.

Pictured data is a special form of character string data. There are two varieties, character pictured data and numeric pictured data. Which of these two is specified depends on details and contents of the PICTURE attribute used to declare pictured data. See LRM 38.

## 2.21. Character pictured data.

Character pictured data is specified when the picture specification given with the PICTURE attribute contains at least one A or X and no other picture characters except A, X, and 9. All of this is explained by an example, which will also serve to show the use and meaning of character pictured data.

```
DCL  CP  PICTURE  'AXXX9';
```

In this declaration of the variable CP, the PICTURE attribute is used. The keyword PICTURE (abbreviated PIC) is always followed by a picture specification, which looks like a character string constant. The picture specification here is 'AXXX9'. It uses the picture characters A, X, and 9.

This declaration says:

(a) CP is stored as a fixed-length character string of length 5.
(b) It may be used in the same ways as any character string variable. Its value is indeed a 5-character string.
(c) The picture character A says that the first character of the value of CP will always be an alphabetic character.
(d) The picture character 9 says that the last character of the value of CP will always be a numeric character or a blank.
(e) The three picture characters X say that the middle three characters of the value of CP will be any characters (no restrictions).
(f) Whenever a value is assigned to CP, it is converted, if necessary, to a character string of length 5. The individual characters are then checked for conformance to the picture as specified above. It is an error to violate the conformance requirements.

See LRM 39 - LRM 42.

2.22. Numeric pictured data.

Numeric pictured data is specified when the picture specification given with the PICTURE attribute does not contain the picture characters A or X. There are an incredible number of picture characters that may be specified, and we will not go into them here. The important things to note for numeric pictured data are as follows:

(a) The data is stored as a fixed-length character string whose length is a function of the picture specification (same as character pictured data so far).
(b) When a reference is made to a numeric pictured variable in a context where a character string value is required, the character string value (exactly as stored) is used.
(c) The character string value stored will always be capable of being interpreted as a numeric (i.e., arithmetic) value, the interpretation (i.e., the mapping from character representation to arithmetic value) being carried out according to the picture specification.
(d) When a reference is made to a numeric pictured variable in a context where an arithmetic value is required, the arithmetic value is obtained from the stored character string value by a conversion that proceeds, as implied above, according to the picture specification.

(e) In addition to directing the mapping from character form to arithmetic form, the picture specification always implies certain arithmetic attributes. These attributes are the attributes used for the arithmetic value obtained by the above conversion. The attributes implied by the picture specification include scale and precision; the base is always decimal.

(f) What guarantees that the character string value stored will always be capable of being interpreted as a numeric value is the following: on assignment of a value to a numeric pictured variable, the value (whether arithmetic or string) is converted, if necessary, to an arithmetic value having the attributes implied by the picture specification. The arithmetic value is then converted to character form and "edited" (mapped) according to the picture specification (the mapping it implies is thus bidirectional).

As you can see, the picture specification is used in quite a few ways.

One simple example will illustrate the above rules. The numeric picture specification '9999', as in DCL NP PIC '9999', means the following:

(a) NP is stored as a fixed-length string of length 4.

(b) The arithmetic attributes implied by PIC '9999' are REAL FIXED DECIMAL (4, 0).

(c) On assignment of any value to NP, the value is converted to REAL FIXED DECIMAL (4, 0) if it does not already have those attributes. It is an error if this conversion cannot occur. That would be the case, for instance, if the character string value ABC were being assigned.

(d) The REAL FIXED DECIMAL (4, 0) value is then converted to a 4-character string and "edited" as follows: The character representing the least significant digit will be aligned on the right-hand edge. Any leading blanks are changed to the character 0. (The picture character 9 allows, during this editing process, only the decimal numeric characters 0 through 9, and not a blank.) If the arithmetic value is negative, the minus sign will not appear in the edited character representation (other numeric picture characters can be used for that). For example, if the arithmetic value is 12, it is stored in NP as 0012. Note that the usual conversion rules for REAL FIXED DECIMAL (4, 0) to CHARACTER would yield a string of length 7 containing, in this case, ƀƀƀƀƀ12.

(e) If NP is referenced in character context, the value used is the 4-character string 0012.
(f) If NP is referenced in arithmetic context, its stored character value is converted to, and interpreted as, REAL FIXED DECIMAL (4, 0) having value 12.

Some of the numeric picture characters specify the insertion of particular characters, like commas, periods, dollar signs, etc., into the character form during the editing that occurs on assignment to a numeric pictured variable. These characters are part of the character value used in character context, but they are "edited out," or ignored, when the arithmetic value is obtained for use in arithmetic context.

Relevant references are LRM 43 - LRM 46 and LRM 16.

2.23. Guidelines on choice of string attributes.

Use bit strings for logical (i.e., boolean-valued) data. This includes program switches, binary state ("true" or "false", "on" or "off") variables, etc. A length of one is most common. Bit strings of greater length can conveniently represent finite ordered sets of boolean objects.

Use character strings to spruce up your output (page headers, all sorts of explanatory or descriptive fields, etc.). Of course, character strings (usually varying-length) are most useful in non-numeric applications such as text processors, compilers, symbolic formula manipulators, etc.

Because of the editing behavior of numeric pictured data, that is most useful in commercial applications. Simple forms, such as the editing of leading blanks into leading zeroes shown earlier, are useful elsewhere.

2.24. Unanswered questions.

In Lesson 6 we will answer these questions:

What happens when an illegal conversion is attempted (i.e., character to arithmetic, where the character value is not the image of an arithmetic constant; character to bit, where the characters are other than 0's and 1's)?

What happens when a character string value being assigned to a character pictured item does not conform to the picture specification?

What happens when the arguments of SUBSTR define a substring outside the bounds of the string?

2.25. Homework problems.

(#2A) What values are assigned to B in the two assignments to B, below?

```
DCL  B  BIT (1);
DCL  S  CHAR (5)  VAR;
S = 'b';
B = LENGTH(S) = 0;
B = S = '';
```

Note that the second = is a comparison operator.

What can you conclude about the "proper" (i.e., "safe") way to determine whether or not the value of a varying-length string variable is the null string?

(#2B) What is the value of each of the following?

```
INDEX('ABCDE',  'CD')
INDEX('ABCDE',  'CF')
VERIFY('CD',  'ABCDE')
VERIFY('CF',  'ABCDE')
TRANSLATE('ABCDE',  '24',  'BD')
REPEAT('b', 5)
```

Read about these builtin functions at LRM 18.

(#2C) Assume you have entered a section of code in which a variable S has already been declared as CHAR (100) VAR and has already been given a value. Write a section of program that will "squeeze out" all the blanks in S, leaving the result in S. Declare as many additional variables as necessary. You will need to code a loop. Code your loop in the following way:

```
DO WHILE (expr-1 ? expr-2);
   .
   .
   body of loop
   .
   .
END;
```

where "?" is a comparison operator, such as =, ¬=, >, etc. On arriving at the DO statement, the indicated comparison is performed. If the comparison holds, or is true, the body of the loop is executed; on arriving at the END statement, control is sent back to the DO statement where the process repeats by doing the comparison again. When the

comparison is false, or doesn't hold, the body of the loop is skipped and control passes to the statement after the END statement. If you think you need IF or GO TO statements, look them up; however, by employing the proper builtin functions, you should need only DECLARE statements, assignment statements, and the DO loop construction shown above.

# 3. Aggregates

## 3.1. Element variables and aggregate variables.

All of the variables we have seen so far have been element variables, i.e., scalar variables. An aggregate is a collection (aggregation) of related element variables. An aggregate variable has identity as a whole; in addition, one may focus on the constituent elements.

There are two kinds of aggregate variables in PL/I: arrays and structures.

## 3.2. Arrays.

Arrays are multidimensional ordered collections of elements all having the same attributes. The collection as a whole has a name. The whole may be referred to by that name or an element may be designated by giving its order in each dimension. For this purpose, a list of subscript expressions enclosed in parentheses is written after the variable name just as in FORTRAN. For example, if A is a two-dimensional array having 5 elements in the first dimension (numbered, say, 1 through 5) and 3 in the second (numbered 1 through 3), then we may refer to the whole 5x3 array by the name A; the element at the intersection of the 4th "row" and 2nd "column" is designated A(4,2).

There are no restrictions on subscript expressions in PL/I. They may be of arbitrary complexity. The value of a subscript expression is converted, if necessary, to a binary integer of default precision.

In PL/I, it is illegal to reference outside the bounds of an array. For example, a reference to A(4,4) is illegal. What happens when this is attempted is deferred to Lesson 6.

## 3.3. The dimension attribute and declarations of arrays.

The dimension attribute is used in a declaration to specify an array. The attribute must immediately follow the variable name, i.e., it must precede other attributes. By "other attributes" is meant the data type attributes specifying the common data type of the elements.

The dimension attribute is written as

$(bounds_1,\ldots,bounds_n)$

where each *bounds* is either *bound* or *bound*:*bound*. In the first form, *bound* is taken as the upper bound of the dimension, with 1 being assumed for the lower bound. In the second form, the two *bound*'s are respectively the lower and upper bounds for the dimension. For now, a bound must be specified as a decimal integer constant.

Examples:

DCL A (5) FIXED BIN (20);

A is a one-dimensional array of elements having the attributes FIXED BIN (20). The lower and upper bounds of the first (and only) dimension are 1 and 5.

DCL B (-1:1, 3, 0:2) BIT (2);

B is a three-dimensional array of 2-bit bit strings. The lower and upper bounds of the three dimensions are, respectively, -1 and 1, 1 and 3, and 0 and 2 for a total of 27 elements.

Caution: The current implementation limits the bounds and the values of subscript expressions to the range -32768 to 32767. This may be a serious restriction to some.

There is generally no need to be concerned with how arrays are mapped in storage. However, in our implementation, two-dimensional arrays are stored "by row," i.e., in general the right-most subscript is the one that varies most rapidly as we proceed to successive elements in storage. (This is just the opposite of FORTRAN.)

In our implementation, the maximum number of dimensions is 15.

For review, see LRM 47 and LRM 48 (skipping parts of the latter that don't look familiar).

## 3.4. Array assignments.

One array can be assigned to another. An assignment statement is an <u>array assignment</u> if the target variable is an array. The right-hand side need not be merely an array variable; as we will see shortly, it may be an expression.

In array assignment, the dimensions and bounds of the array value on the right must exactly match those of the target variable. The assignment is carried out by iterating over the range of subscript values (in some cases the compiler may generate a loop, in others it may generate a "bulk move," but the effect is the same in either case).

## 3.5. Arrays as operands in expressions.

The right-hand side of an array assignment statement may be an <u>array expression</u>. Essentially, any of the operands may be arrays (having the same dimensions and bounds as the target variable). The array assignment is interpreted as an iteration over the (common) bounds of

all the arrays. Scalar operands are also permitted, the value of a scalar operand being used in each implied iteration. (In fact, the entire right-hand side may be a scalar expression, in which case its value is assigned to all elements of the array target.)

See LRM 52.

Examples:

```
DCL A (3) FLOAT,
    B (3) FLOAT;
A(1) = 1; A(2) = 0; A(3) = 1;
B(1) = 3; B(2) = 4; B(3) = 5;
DCL C (3) FLOAT;
C = A + B;  The elements of C have values 4, 4, and 6.
C = 0;      All the elements of C have value 0.
C = B + 1;  The elements of C have values 4, 5, and 6.
C = C * A;  C is now 4, 0, 6.  Observe that corresponding
            elements are multiplied, i.e., matrix multiplication
            is not used.
C = C/C(1); C is now 1, 0, 6.  This statement is equivalent to:
                C(1) = C(1)/C(1); Sets C(1) to 1.
                C(2) = C(2)/C(1); Divides by 1!
                C(3) = C(3)/C(1); Ditto!
            The ANSI standard will make this behave as
                TEMP = C(1);
                C(1) = C(1)/TEMP;
                C(2) = C(2)/TEMP;
                C(3) = C(3)/TEMP;
```

By the way, the declarations of A and B may be written in either of the following ways:

```
    DCL (A (3), B (3)) FLOAT;
and DCL (A, B) (3) FLOAT;
```

(See LRM 49.)

Array expressions can appear in contexts other than assignment statements. In a subroutine call, an actual argument may be an array expression, as we will see in Lesson 5. Certain builtin functions take only array arguments (Lesson 10).

The builtin functions and pseudo-variables shown so far can be given array arguments; they return similarly structured array results, the operation being performed on an element-by-element basis. Their use in more complicated array expressions and assignments is consistent with this. For instance, if A and B are congruent arrays, A = SIN(B) assigns the sine of each element of B to the corresponding element of A, and B = SIN(A)**2 + COS(A)**2 is an <u>expensive</u> way of assigning 1 to each element of B (the individual elements of A are sined and squared, then added to the squares of their cosines). Similarly, if Z is an array of COMPLEX elements, IMAG(Z) = 0 sets all of their imaginary parts to 0. See LRM 50.

## 3.6. Array cross sections.

A special notation can be used to denote a generalized "slice" through an array. The cross section notation A(*,I) means the Ith column of A. This is a one-dimensional array with bounds equal to those of A in the first dimension. Another example: B(*,2,*) means the plane coincident with the 2nd column of B. This is a two-dimensional array with bounds in the first dimension equal to those of B in the first dimension and bounds in the second equal to those of B in the third.

Note that A(*,...,*) denotes the array itself. Whenever a reference to an entire array is written, it is usually good documentary practice to write it as an identity cross section, i.e., the whole array. This practice will be followed subsequently in these notes.

The following statement assigns the Ith row of A to the Jth column of B:

```
B(*,J) = A(I,*);
```

For this statement to be legal, the bounds of A in its second dimension must be identical to the bounds of B in its first.

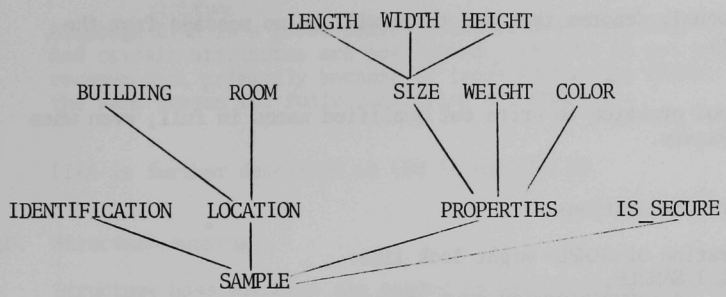
Since arrays are stored by row in PL/I, A(I,*) occupies contiguous storage locations. A(I,*) is said to be a connected reference. B(*,J), on the other hand, does not occupy contiguous storage locations. It is said to be an unconnected reference. Only connected references are permitted in certain contexts, as we shall see later. See LRM 51.

## 3.7. Structures.

A structure, like an array, is a collection of related data items which is assigned a name. Unlike an array, each constituent item also has a name, and the constituent items may all have different attributes.

In fact, a structure is, in general, a hierarchical collection of "things." The things may be thought of as organized in a "tree." The elements at the ends of the "branches" have names and data type attributes. Other "nodes" in the tree represent intermediate levels of the hierarchy; they have names, but not data types.

Consider the following pictorial representation of a structure:




The base elements of this structure, and typical attributes they may have, are as follows:

| | |
|---|---|
| IDENTIFICATION | CHAR (50) VAR |
| BUILDING | CHAR (3) |
| ROOM | CHAR (4) |
| LENGTH | FLOAT DECIMAL (3) |
| WIDTH | FLOAT DECIMAL (3) |
| HEIGHT | FLOAT DECIMAL (3) |
| WEIGHT | FLOAT DECIMAL (5) |
| COLOR | CHAR (10) VAR |
| IS_SECURE | BIT (1) |

This entire collection may be referred to with the name SAMPLE; SAMPLE is called a major structure name. Subsets of the collection forming subtrees may also be referred to by their names, viz. LOCATION, SIZE, and PROPERTIES. These are names of minor structures. Minor structures are indeed structures, but they are not independent; they belong to a major structure.

Suppose we have another structure, called EXPERIMENT. An experiment can have a location (i.e., a building and a room), too, so we might like to have a substructure (minor structure) of EXPERIMENT called LOCATION having, in turn, the same constituents as the LOCATION in SAMPLE. How do we distinguish between references to SAMPLE's LOCATION and EXPERIMENT's LOCATION, if we should need to? By writing a qualified name. The name SAMPLE.LOCATION refers to the LOCATION in SAMPLE, while EXPERIMENT.LOCATION refers to that in EXPERIMENT. Similarly, SAMPLE.LOCATION.ROOM and EXPERIMENT.LOCATION.ROOM distinguish between the two element variables called ROOM.

One need not always write all levels of structure qualification in a qualified name. The only requirement is to avoid ambiguity. Thus, SAMPLE.ROOM and EXPERIMENT.ROOM are sufficient, but ROOM alone is not. If the above two uses of ROOM were the only ones appearing in a program, the compiler would tell you that ROOM (unqualified) is ambiguous. However, if you declared a scalar variable ROOM as well, then ROOM

unambiguously denotes that and there will be no message from the compiler.

It is good practice to write out qualified names in full, even when not necessary.

3.8. Structure declarations.

A declaration of SAMPLE might look like:

```
DCL 1 SAMPLE,
      2 IDENTIFICATION CHAR (50) VAR,
      2 LOCATION,
        3 BUILDING CHAR (3),
        3 ROOM CHAR (4),
      2 PROPERTIES,
        3 SIZE,
          4 LENGTH FLOAT DEC (3),
          4 WIDTH FLOAT DEC (3),
          4 HEIGHT FLOAT DEC (3),
        3 WEIGHT FLOAT DEC (5),
        3 COLOR CHAR (10) VAR,
      2 IS_SECURE BIT (1);
```

The numbers in front of the names are called level numbers. The indentation is purely documentary; what is subordinate to what else is uniquely determined by the sequence of level numbers.

Factoring of attributes can be used here. A part of this declaration could have been written

```
        3 SIZE,
          4 (LENGTH, WIDTH, HEIGHT)
                FLOAT DEC (3),
```

as described at LRM 49.

For a review of structures so far, see LRM 53.

3.9. The LIKE attribute.

A convenience feature that saves writing when similar structures are declared is the LIKE attribute. In the declaration of EXPERIMENT, one need not write out the details of the minor structure LOCATION. If it is just like the one in SAMPLE, one could write

```
DCL  1  EXPERIMENT,
        :
        :
        2 LOCATION LIKE SAMPLE.LOCATION,
        :
        :  ;
```

The structuring and attributes are copied from the declaration of SAMPLE.

Although LIKE is a great convenience, it does have many restrictions. And certain attributes are not copied. Its use is not generally recommended, primarily because it tends to obscure facts. (That's the same reason for fully qualifying all names.)

LIKE is further described at LRM 54 and LRM 55.

## 3.10. Structure mapping.

Structure base elements are mapped consecutively in storage. However, since consecutive elements may have differing alignment requirements (due to having different attributes), a small amount of padding, which is unused space, may be allocated between consecutive base elements. The padding is not accessible to the program, and its existence does not cause a structure reference to be an unconnected reference.

Since alignment requirements are a property of the hardware (i.e., the implementation), the amount of padding may vary from one implementation to another. But so does the amount of storage allocated to element variables, as we have seen. The only time this is likely to be of concern to the programmer is when he is trying to figure out record lengths for certain kinds of I/O (Lessons 8-9). A compiler option, AG, which is "on" by default in our batch compilers and "off" in our conversational ones, can be used to show how each aggregate is mapped. The listing is part of the compilation listing. See CPG 1 and CPG 2, CTUG 1, OPG 1 and OPG 2, and OTUG 1.

The algorithm our compilers use for structure mapping is described at LRM 56.

## 3.11. ALIGNED and UNALIGNED attributes.

Reference has been made above to alignment of data. It is possible to tell the compiler not to worry about alignment requirements during mapping or allocation of data. When so told, it assigns most things to the next available byte boundary (bit boundary in the case of bit strings). The main purpose of this is to achieve greater data packing in aggregates; it may also be of use in certain I/O situations. To avoid machine errors in addressing data which is not known to be on a "natural" boundary, the compiler generates extra code to move it to or from a properly aligned boundary. This can increase program size and execution time, so the feature shouldn't be used indiscriminately.

Two mutually exclusive attributes, ALIGNED and UNALIGNED, select these options. These attributes may be specified for any variable. They apply to every variable, and, when not specified, language defaults are taken. All of the variables we have talked about so far are subject to this default, though we have had no reason to concern ourselves with it yet.

Basically, the default is UNALIGNED for string data and ALIGNED for everything else. Alignment of character string variables is a moot point; they begin on the next available byte boundary in either case (fixed-length strings do, at least). UNALIGNED bit strings begin on the next available bit boundary, while ALIGNED bit strings begin on the next available byte boundary. Because of this, arrays of, say, BIT (1) variables will occupy only one-eighth of the storage under the default (UNALIGNED) as they would were ALIGNED specified, but addressing elements of the array will be much slower (in general, most unaligned bit references or operations are performed by library routines, while aligned references and operations are done by in-line code).

The alignment attributes may be specified at any level in a structure declaration. They apply to all of the constituent element variables subordinate to that level except those which are subordinate to an intermediate level which also specifies alignment, in which case the latter specification is used. For example, in

```
DCL 1 STRUCTURE ALIGNED,
      2 A UNALIGNED,
        3 B ALIGNED,
        3 C,
      2 D,
        3 E UNALIGNED,
        3 F;
```

the base elements are B, C, E, and F. B and E are clearly ALIGNED and UNALIGNED, respectively. C is UNALIGNED (inherited from A). F is ALIGNED (inherited from D, which inherited ALIGNED from STRUCTURE).

For reference, see LRM 57 and LRM 58.

## 3.12. Structure assignments.

One structure may be assigned to another. The hierarchical structuring of the two structures must match at all levels. (It is not sufficient to have just the same number and types of base elements.) However, the names of matching levels of the hierarchy need not match, nor need the attributes of corresponding base elements match. The assignment statement is "expanded" into a sequence of scalar assignment statements.

Example:

```
DCL 1 S1,
      2 A FIXED BIN,
      2 B,
        3 C FLOAT DEC,
        3 D CHAR (5),
      2 E BIT (1),
    1 S2,
      2 V FLOAT BIN,
      2 W,
        3 X CHAR (8),
        3 Y BIT (20),
      2 Z FIXED DEC;
S1 = S2;  This is equivalent to
            S1.A = S2.V;
            S1.B.C = S2.W.X;
            S1.B.D = S2.W.Y;
            S1.E = S2.Z;
          In each of these scalar assignments,
          different conversions will occur.
DCL 1 S3,
      2 A FIXED BIN,
      2 C FLOAT DEC,
      2 D CHAR (5),
      2 E BIT (1);
S1 = S3;  This is illegal.
DCL 1 S4,
      2 M FLOAT BIN,
      2 N CHAR (6) VAR;
S1.B = S4;  This structure assignment is equivalent to
              S1.B.C = S4.M;
              S1.B.D = S4.N;
            In other words, a substructure (minor structure)
            is a structure in its own right.
```

3.13. Structures as operands in expressions.

By analogy with array expressions, structure expressions are expressions whose operands are congruent structures (congruent in the sense of structure assignments). Using the declarations of the previous section, one could write S1 = S1 - S2, for instance. This is equivalent to

```
S1.A = S1.A - S2.V;
S1.B.C = S1.B.C - S2.W.X;
S1.B.D = S1.B.D - S2.W.Y;
S1.E = S1.E - S2.Z;
```

Also, S4 = 0 is equivalent to

```
S4.M = 0;
S4.N = 0;
```

and S4 = 3 * S2.W is equivalent to

```
S4.M = 3 * S2.W.X;
S4.N = 3 * S2.W.Y;
```

Structure expressions may also be actual arguments in subroutine calls (Lesson 5). The builtin functions described so far cannot, however, take structure arguments.

See LRM 59.

3.14. Structures of arrays and arrays of structures.

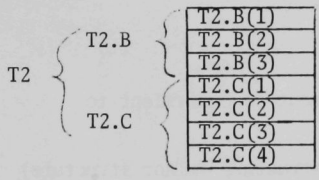
The two kinds of aggregation can be compounded. The following is an example of a structure of arrays, i.e., a structure with some arrays at the deepest level.

```
DCL 1 T1,
      2 A (10) FLOAT,
      2 B,
        3 C (-1:3) CHAR (6) VAR,
        3 D (2,4) BIT (7) ALIGNED;
```

A structure of arrays such as

```
DCL 1 T2,
      2 B (3),
      2 C (4);
```

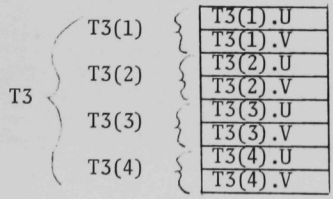
is mapped in storage as follows:




An example of an array of structures is:

```
DCL 1 T3 (4),
      2 U,
      2 V;
```

An array of structures can be thought of as a structure with the dimension attribute or (what is, of course, the same thing) an array whose components are not element variables but structures. T3 is mapped, and its components named, as shown below.



A reference to T3.U (i.e., T3(*).U) is allowed. This designates the one-dimensional array whose four elements are T3(1).U, T3(2).U, T3(3).U, and T3(4).U. Note that this is another example of unconnected storage. There are, however, some apparently poorly documented restrictions on the use of cross sections of arrays of structures.

Since T3.U is an array, as described above, you might ask whether it is possible to write T3.U(1), T3.U(2), T3.U(3), T3.U(4) instead of T3(1).U, etc. The answer is yes, and they mean the same thing. This seems to be an ill-advised flexibility because it tends to obscure the real structure of T3 (again: when things "ain't what they seem," it's bad).

Compounding of aggregation can be carried to ridiculous, seldom needed, extremes, as in

```
DCL 1 T5 (5),
      2 A (3),
        3 B,
        3 C,
        3 D (3),
      2 E,
        3 F,
        3 G (8);
```

for which T5(3).A(1).D(2) is a legal reference, and the same as T5.A.D(3,1,2).

See LRM 60 and LRM 61.

## 3.15. BY NAME assignment.

Another type of structure assignment, BY NAME assignment, is obtained by adding the BY NAME option to an assignment statement, as in *variable* = *expression*, BY NAME; The structure operands in a BY NAME assignment statement need not be congruent, as in a regular structure assignment. Basically, the statement is expanded into a sequence (ultimately) of scalar assignment statements, with the expansion proceeding deeper and deeper in the structure only as long as all structure operands have items with the same names at the level being considered. For example:

```
DCL 1 A,          | DCL 1 M,          DCL 1 U,
      2 B,        |       2 I,              2 E,
        3 C,      |         3 K,              3 F,
        3 D,      |         3 J,                4 G,
      2 E,        |       2 F,              2 T,
        3 F,      |         3 G,              3 J,
          4 G,    |         3 H,            2 B,
          4 H,    |       2 B,                3 C,
      2 I,        |         3 Q,              3 Z;
        3 J,      |         3 C,
        3 K;      |       2 E,
                  |         3 G;
```

A = M * U, BY NAME; is expanded as follows:
A.B = M.B * U.B, BY NAME; (1)
A.E = M.E * U.E, BY NAME; (2)
(1) is further expanded to
A.B.C = M.B.C * U.B.C; (3)
(2) is further expanded into nothing, since while both A.E and U.E have a component called F, M.E does not. Thus, the original statement is equivalent to (3).

See LRM 62 and those parts of LRM 63 that look familiar.

3.16. Equivalencing of data.

PL/I provides facilities comparable to FORTRAN's EQUIVALENCE statement for equivalencing data. Before proceeding to specifics, we should take a good look at some very important fundamental differences in the concept between the two languages.

The FORTRAN EQUIVALENCE statement is provided to allow storage to be shared amongst several variables. In standard FORTRAN the user is not supposed to rely on two equivalenced variables always having the same values by virtue of occupying the same storage. Some optimizing compilers, in fact, may omit store instructions in certain circumstances, actually destroying value-equivalence between storage-equivalenced variables. Because it need not guarantee value-equivalence, FORTRAN permits equivalenced variables to have different data types.

The PL/I DEFINED attribute allows several variables to share the same storage. In this case, the language guarantees value-equivalence, i.e., the equivalenced variables become fully interchangeable. Because of this, PL/I does not permit variables having different data types to be equivalenced. This is an important point to understand because it makes the PL/I analogs of several common FORTRAN constructions illegal. Other facilities are provided in PL/I for looking at storage in different ways--legally, less conveniently, and by rules that are inevitably implementation-dependent (Lesson 10).

Because PL/I guarantees value-equivalence as well as storage-equivalence, the use of the DEFINED attribute can inhibit certain optimizations that might otherwise occur.

There are three different types of defining (i.e., equivalencing) in PL/I, depending on what else is written with the DEFINED attribute. Each serves a unique purpose. You will see that defining is actually much more powerful than FORTRAN equivalencing.

3.17. Simple defining.

Simple defining merely allows storage belonging to one variable to be referred to by another name. However, several flexibilities are permitted. Simple defining is illustrated by several examples.

```
DCL A FLOAT BIN (21);
DCL B FLOAT BIN (21) DEFINED (A);
```

B is "defined on" A. Note that the data type attributes of A are repeated in the declaration of B. A and B are variables with the same location and value (recall Lesson 1) but different names.

```
DCL C(0:9) FIXED DEC (5);
DCL D FIXED DEC (5) DEFINED (C(I+3*J));
```

The defined variable is D. The base variable, i.e., the variable on which it is defined, is an element of the array C. Both defined variable and base variable are thus scalars. The element of C to which D corresponds is determined dynamically; on each reference to D, I+3*J is evaluated to determine the proper element of C.

```
DCL E (10,10) FLOAT;
DCL F (10,10) FLOAT DEFINED (E);
```

This needs no comment. Note, however, that the dimension attribute for F may not have been written as (100), because defined arrays must have the same dimensionality as their base array. One of the other kinds of defining permits "remapping" of arrays.

```
DCL G (2:6, 3:8) FLOAT DEFINED (E);
```

Though the dimensionality of defined and base item must be identical, the extent of a dimension of the defined variable may be less than the extent of the corresponding dimension of the base array (it cannot be greater). A reference to G($i$,$j$) is identical to a reference to E($i$,$j$). A reference to G(1,5) is illegal, even though E has a (1,5) element, because G doesn't. Note that G is an unconnected array, although E is connected.

```
DCL H (10) FLOAT DEFINED (E(*,I));
```

The base array is the Ith column of E, which is an array of one dimension with bounds (1:10). H has identical structuring and is, in fact, synonymous with the Ith column of E. A reference to H(*i*) is the same as a reference to E(*i*,I).

## 3.18. ISUB defining.

ISUB defining allows an array (the base array), or part of an array, to be addressed through another array (the defined array). The dimensions may differ because, in fact, an arbitrary mapping from elements of the defined array to elements of the base array is specified. ISUB defining is also best explained with examples.

```
DCL A (4) CHAR (1);
DCL B (3) CHAR (1) DEFINED (A(1SUB+1));
```

In the subscript list for the base array, the "1SUB" is a funny kind of variable; it stands for the value of the 1st subscript expression in any reference to the defined array. That is, B(K) is the same as A(K+1). Pictorially,

| | | | |
|---|---|---|---|
| A(1) | A(2)<br>B(1) | A(3)<br>B(2) | A(4)<br>B(3) |

```
DCL C (2,3) BIT (3) UNALIGNED;
DCL D (6) BIT (3) UNAL
          DEF (C((1SUB+2)/3,MOD(1SUB-1,3)+1));
```

Note the abbreviations. D is mapped into C as shown below.

| | | |
|---|---|---|
| C(1,1)<br>D(1) | C(1,2)<br>D(2) | C(1,3)<br>D(3) |
| C(2,1)<br>D(4) | C(2,2)<br>D(5) | C(2,3)<br>D(6) |

```
DCL E (10,10) FLOAT;
DCL F (2,2) FLOAT DEF  E(I+1SUB-1,I+2SUB-1);
```

Note that the parentheses surrounding the base variable may be omitted. F is a 2 x 2 submatrix of E, whose upper left-hand element (F(1,1)) is coincident with E(I,I). I must have a value between 1 and 9 for a reference to F(*,*), i.e., the whole array F, to be legal.

```
DCL G (6) BIT (1) ALIGNED;
DCL H (2,3) BIT (1) ALIGNED
          DEF G(10-3*1SUB-2SUB);
```

| | | | | | |
|---|---|---|---|---|---|
| G(1)<br>H(2,3) | G(2)<br>H(2,2) | G(3)<br>H(2,1) | G(4)<br>H(1,3) | G(5)<br>H(1,2) | G(6)<br>H(1,1) |

DCL I (2,3) BIT (1) ALIGNED DEF (G(1SUB));
Note that although I has two dimensions, the subscript list for the base variable does not use 2SUB. Thus, I(1,1), I(1,2), and I(1,3) are <u>all</u> synonymous with G(1), and I(2,1), I(2,2), I(2,3) are <u>all</u> synonymous with G(2). Is I connected? Actually, because isub-defined variables can have non-linear subscripting functions, the concept is inapplicable. Since they cannot always be determined to be connected, they may not be used where unconnected variables are prohibited (as we shall see later).

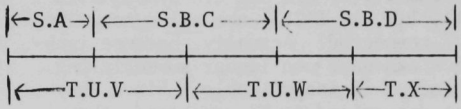
## 3.19. String overlay defining.

String overlay defining allows strings, or aggregates of strings, to be overlayed on other element or aggregate string variables of the same string type (i.e., character or bit). If the base variable is an aggregate, it must be connected and unaligned. This guarantees that consecutive elements of the base variable will be mapped consecutively. Therefore, the defined variable, which will also have contiguous elements because it, too, may not be aligned, need not have the same structuring as the base variable. Examples follow.

DCL A CHAR (10);
DCL B (10) CHAR (1) DEFINED (A);
The ith element of B, which is a CHAR (1) item, is synonymous with the ith character of A, i.e.,
B($i$) = SUBSTR(A,$i$,1).

DCL C CHAR (10);
DCL D (5) CHAR (2) DEF C;
D($i$) is equivalent to SUBSTR(C,2*$i$-1,2).

```
DCL 1 S,
      2 A BIT (1),
      2 B,
        3 C BIT (2),
        3 D BIT (2);
DCL 1 T DEF S,
      2 U,
        3 V BIT (2),
        3 W BIT (2),
      2 X BIT (1);
```




```
DCL E CHAR (8);
DCL F CHAR (6) DEF (E) POSITION (3);
```

The POSITION attribute may be used in string overlay defining to denote the offset of the first character (or bit) of the defined item from the first character (or bit) of the base item. If it is omitted, POSITION (1) is assumed (no offset). F is the last six characters of E.

```
DCL F CHAR (6) DEF (E) POS (I+1);
```

The POSITION attribute (note the abbreviation) may contain an expression. I must have a value between 0 and 2; otherwise a reference to F would yield an equivalent reference to E outside of its bounds, i.e., F is the same as SUBSTR(E,I+1,6).

3.20. Determination of type of defining.

If isub variables are used, isub defining is in effect. If the POSITION attribute is used, string overlay defining is in effect. Otherwise, either simple defining or string overlay defining is in effect, depending on whether or not the attributes of the defined item match those of the base item (if they don't, they must satisfy the constraints for string overlay defining, of course). These rules are summarized at LRM 64 and LRM 65. Defining in general is summarized and completely described at LRM 66 and LRM 67, respectively.

3.21. Homework problems.

(#3A) Consider the following declarations, which may legally appear together.

```
DCL 1 A,
      2 B,
        3 C,
        3 A,
      2 D,
        3 C,
          4 A,
        3 E,
      2 A;
DCL 1 D,
      2 C,
      2 F,
        3 G,
          4 A;
```

To what does each of the following references refer? Which are ambiguous? For those that are ambiguous, which items could they refer to, and how would you write unambiguous references to those items?

```
A
A.C
A.C.A
A.A
D
D.C
D.A      (tricky)
```

Try to state a rule for determining whether a reference is ambiguous or not (difficult).

(#3B) Consider the declarations

```
DCL 1 S (3),
      2  U,
      2  V;
DCL 1 T,
      2 W (3),
      2 X (3);
```

Is S.U = T.W legal? If so, what does it mean? If not, why not? Answer the same questions for the assignment S = T.

(#3C) Consider the declarations

```
DCL 1 A,
      2 B,
      2 C,
        3 D,
        3 E,
          4 F,
            5 G;
DCL 1 M,
      2 C,
        3 N,
          4 D,
        3 E,
          4 P,
            5 G,
      2 X;
```

Determine the expansion of

```
A = M, BY NAME;
```

(#3D) Let A be a 10 x 10 array. Write a single assignment statement that will assign 0 to all the elements of A. Write a single assignment statement that will assign 1 to the diagonal elements (only) of A. Hint: figure out how to use isub defining to declare a one-dimensional array synonymous with the diagonal of A. Show the declaration.

(#3E) Let U be a 3 x 3 array. Show how you can use isub defining to declare an array V which is synonymous with the transpose of U.

(#3F) Let A have the attributes CHAR (10). Show how you can reverse the value of A (leaving the result in A) using only assignment statements. You will need to declare some auxiliary variables using isub defining and string overlay defining. Note that the base variable in a DEFINED attribute may not be declared with the DEFINED attribute, i.e., you can't define X on Y and Y on Z.

# 4. Block structure and scope of names.

## 4.1. External procedures.

A PL/I external procedure is a segment of a program that may be separately compiled. It is entirely analogous to a FORTRAN "program unit." A FORTRAN program consists of one program unit which is a "main program" and possibly other program units which are "subprograms." In the same way, a PL/I program consists of one external procedure which is a "main procedure" and possibly other external procedures. In FORTRAN, subprograms (other than BLOCK DATA subprograms) represent common sequences of code that need to be executed logically at several different points in the overall program. By packaging them separately, they need only be written once. Control can be transferred to them from each point at which they are needed. The external procedures of a PL/I program, other than the main procedure, serve the same purpose.

As in their FORTRAN analogs, external procedures can either be executed for their effect or for their returned value. This use corresponds to the two kinds of executable FORTRAN subprogram, subroutine subprogram and function subprogram. As in FORTRAN, when they are executed for their effect they are invoked by a CALL statement, and when they are executed for their returned value they are invoked by a "function reference" in an expression. The dynamic aspects of PL/I procedures will be covered in Lesson 5.

A PL/I external procedure starts with a PROCEDURE statement and ends with an END statement. In between comes the body of the procedure, i.e., executable and declarative statements. The minimum content of a PROCEDURE statement is an entry label (i.e., a procedure name), a colon, the keyword PROCEDURE (abbreviation: PROC), and, of course, a semicolon. Example:

```
MYPROG: PROC;
```

Lots of other things can be hung onto a PROCEDURE statement. If the procedure is to be invoked with some arguments, a parameter list must immediately follow the PROCEDURE keyword. (We will save arguments and parameters for Lesson 5.) Several other options may follow it (or the PROCEDURE keyword, if there is no parameter list). The RETURNS option indicates

that the procedure will return a value and must be invoked as a function reference; thus, a PROCEDURE statement with the RETURNS option is the equivalent of a FORTRAN FUNCTION statement. If the RETURNS option is omitted, the procedure will not return a value back to the point of invocation and must therefore be invoked by a CALL statement. Thus, a PROCEDURE statement without the RETURNS option is akin to a FORTRAN SUBROUTINE statement.

Another option is the OPTIONS option. This is the keyword OPTIONS followed by a parenthesized list of keywords for options. The function of the OPTIONS option is to provide a language-defined (i.e., standardizable) way of supplying implementation-defined options to your particular system. (Thus, exactly what can appear inside the parentheses, and the meaning of what appears there, is implementation-defined, not language-defined.) One of the options that can be used in our system is MAIN. It designates that the external procedure is a main procedure. Example:

```
MYPROG: PROC OPTIONS(MAIN);
```

Note that a FORTRAN main program does not start with any particular kind of statement; the absence of a FUNCTION or SUBROUTINE statment as the first statement implies the program unit is a main program. In PL/I, one and only one of the external procedures of a program can have, and must have, OPTIONS(MAIN).

Other items that can appear on a PROCEDURE statement will be introduced at relevant places.

We will catch up with references for the above material a little later.

## 4.2. Internal procedures.

An internal procedure is a procedure nested inside another procedure. Internal procedures may be nested inside external procedures or other internal procedures. A procedure nested inside another procedure (the "containing procedure") has its matching PROCEDURE and END statements, and the body of code between them, contained within the body of code delimited by the containing procedure's matching PROCEDURE

and END statements. Example:

```
MAIN: PROC OPTIONS(MAIN);
   .
   .
   .
   SUBR: PROC;
      .
      .
      .
   END; /* OF PROCEDURE 'SUBR'*/
   .
   .
   .
END; /* OF PROCEDURE 'MAIN'*/
```

Like an external procedure, an internal procedure is used to package common code that needs to be executed at many places (within the containing procedure). Like an external procedure, it may be invoked for its effect, with a CALL statement, or invoked for its returned value, via a function reference in an expression. (The one shown above, because it does not use the RETURNS option, presumably is invoked by CALL.) An internal procedure may not be a main procedure.

Internal procedures can be used in simple ways analogous to FORTRAN "arithmetic statement functions." However, they are far more general and their generality has no counterpart in FORTRAN. Differences between internal procedures and arithmetic statement functions may be summarized (at least partly) as follows:

(a) Internal procedures may be invoked by a CALL statement or a function reference. An ASF is only invoked by a function reference.
(b) In either case, they may or may not take arguments. An ASF (like all FORTRAN functions) must take at least one argument.
(c) They may embody arbitrary code, using arbitrary logic. An ASF is restricted to a single expression.
(d) They may invoke themselves recursively.
(e) They need not be placed, in their containing procedure, ahead of executable statements or after declarations.

An overview of procedures (going a little beyond the above material) is at LRM 68 and LRM 69.

## 4.3. Scope of a declaration.

We saw in Lesson 1 that a declaration associates a name and some attributes with a variable. We will soon see that declarations can associate names and attributes with certain kinds of constants, too, called named constants. So we will just be general and say that declarations associate names and attributes with objects. And when we say "declare a name..." we mean "declare an object named...".

A DECLARE statement, i.e., an explicit declaration, is said to belong to the procedure in whose body it appears (or "to which it is internal"). Note that if a procedure named INNER is nested inside a procedure named OUTER, and a DECLARE statement is written between the PROCEDURE and END statements of INNER, then the declaration belongs to INNER and not to OUTER. That is, an explicit declaration belongs to the "nearest" containing procedure.

The scope of such a declaration is the procedure to which it belongs, including any contained (i.e., nested or internal) procedures, excluding any nested procedures (no matter how deeply nested) containing another explicit declaration for an object with the same name. The object declared is known (by its name) in the scope of its declaration, that is, any reference to the name in that scope is a reference to the object. As we will see soon, a reference to the same name in the scope of a different explicit declaration may or may not be a reference to the same object.

See LRM 70.

Contextual or implicit declarations (recall Lesson 1), i.e., those resulting from uses of names not explicitly declared, belong to the containing external procedure. In other words, the scope of a contextual or implicit declaration is the whole external procedure in which the name is used, excluding any internal procedure (and its descendants) where the name is explicitly declared. See LRM 71 and LRM 72.

Although we will not be considering arguments and parameters of a procedure in detail until the next lesson, there are some things to be noted with respect to the scope of a

parameter declaration. (A parameter in PL/I is what is called a "dummy argument" in FORTRAN. The names appearing in the parameter list of a PROCEDURE statement are the names of parameters.)

A parameter name may or may not appear in a DECLARE statement in the procedure of which it is parameter (that is, it is not required to appear in a DECLARE statement). If it does appear in a DECLARE statement there, it is explicitly declared with the given attributes. If it does not, it is as if it had appeared there in a DECLARE statement with no attributes. This is sufficient to establish an explicit declaration, with all of the attributes taken from the applicable defaults. Thus, parameters can never be contextually declared, that is, they never acquire attributes based on the context of their use. See LRM 73 and LRM 74.

## 4.4. INTERNAL and EXTERNAL attributes; scope of a name.

There is another pair of alternative attributes which may be given to any variable. Like the ALIGNED and UNALIGNED attributes, they apply to every variable and if they are not given to it explicitly one or the other will be acquired by default. These are the INTERNAL attribute and EXTERNAL attribute, collectively called scope attributes. Their abbreviations are INT and EXT. Unlike the alignment attributes, the scope attributes apply to named constants as well as variables.

An object declared with the INTERNAL attribute (explicitly or by default) is associated with its name in the scope of the declaration and nowhere else. Thus, two different declarations of the same internal name, in different scopes, establish different objects which happen to be known by the same name.

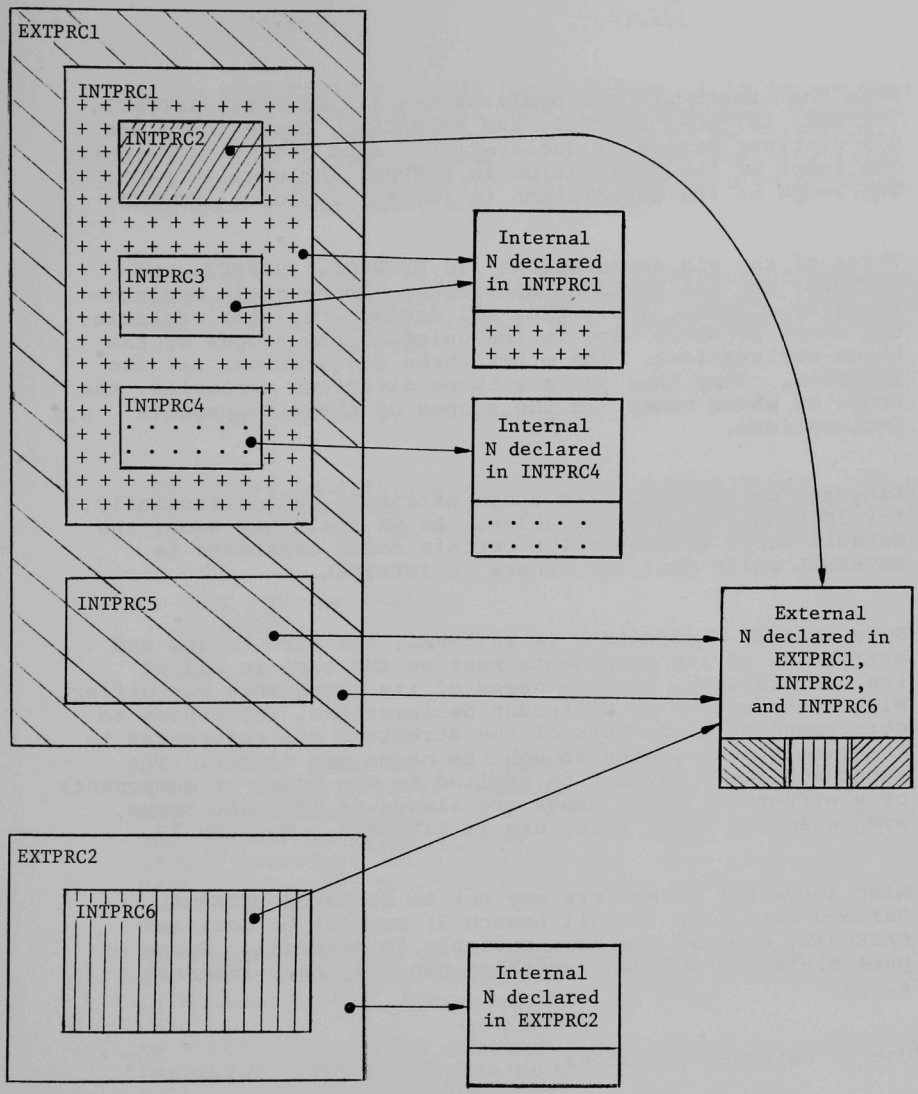
The effect of the EXTERNAL attribute is as follows. All declarations of a given name (say E) having the EXTERNAL attribute, i.e., of an external name, are linked together so that they refer to the same object, rather than to different objects. It is then required that all such declarations (of E in this case) specify identical attributes. The linking occurs at link-edit time.

The scope of a name (say N) can now be defined, as follows. If a declaration for N includes INTERNAL, the scope of the name is the scope of the declaration. The scope of an EXTERNAL name N is the union of the scopes of its declarations (all of which must be identical). See LRM 75 and LRM 76.

Consider the following example. The nesting of procedures and the occurrences of declarations is shown first. We then show nested areas representing the nested procedures, using different shadings to show the different scopes of declarations. Off to the side we show the distinct variables, each one shaded to show the scope of its name.

```
EXTPRC1: PROC;
  DCL N FLOAT EXT;
  INTPRC1: PROC;
    DCL N FLOAT INT;
    INTPRC2: PROC;
      DCL N FLOAT EXT;
    END;
    INTPRC3: PROC;
    END;
    INTPRC4: PROC;
      DCL N FLOAT INT;
    END;
  END;
  INTPRC5: PROC;
  END;
END;

EXTPRC2: PROC;
  DCL N FLOAT INT;
  INTPRC6: PROC;
    DCL N FLOAT EXT;
  END;
END;
```


EXTPRC1
INTPRC1
INTPRC2
INTPRC3
INTPRC4
INTPRC5
Internal
N declared
in INTPRC1
Internal
N declared
in INTPRC4
External
N declared in
EXTPRC1,
INTPRC2,
and INTPRC6
EXTPRC2
INTPRC6
Internal
N declared
in EXTPRC2

Note that there are six declarations (in EXTPRC1, INTPRC1, INTPRC2, INTPRC4, EXTPRC2, and INTPRC6). Hence there are six distinct scopes of declarations, each shaded differently. The scope of the declaration in EXTPRC1 includes INTPRC5. The scope of the declaration in INTPRC1 includes INTPRC3.

Three of the six declarations (in EXTPRC1, INTPRC2, and INTPRC6) use the EXTERNAL attribute. Hence their declarations are linked, i.e., they all declare the same variable, the scope of whose name is the union of the scopes of the three declarations. The other three declarations all use INTERNAL. They thus declare three different variables, the scope of whose names are the scopes of their respective declarations.

Language defaults for the scope attribute call essentially for INTERNAL for all variables. As we shall see soon, the default scope attribute for certain named constants is EXTERNAL while that for others is INTERNAL.

Note that if a structure is EXTERNAL, the structuring and attributes of its components must be the same in all of its declarations, but the names of its components may differ. Within the scopes of different declarations, references to corresponding components of the structure are references to the same storage, even though the names may differ. The scope attribute may not be applied to the names of components of a structure; their names are always of INTERNAL scope, even when the major structure is EXTERNAL. See LRM 77.

Also note that parameters may not be declared EXTERNAL. Defined variables (recall Lesson 3) may not be declared EXTERNAL, even if the base variable is external. Names of parameters and defined variables can only have internal scope.

## 4.5. Use of external variables.

External variables permit communication via "global variables" amongst several separately compiled procedures. The same communication can always be achieved by passing arguments, but external variables are cleaner in many situations. For instance, one procedure may initiate a chain of calls say ten levels deep. It may need to pass a particular argument all the way through this chain to the procedure at its end. Even if the intermediate procedures had no use for the data

item, they would at least have to pass it on to the next procedure. They thus all have to be aware of its existence; all the declarations would have to be just right, and so on. By using external variables, only the first and last procedures in the chain would have to be aware of the data's existence, and declare it.

It should be noted that each external structure declaration behaves like a complete FORTRAN "named common" specification. Even external scalars behave like "named common" blocks--with just one item in them.

## 4.6. Procedure names; entry constants.

Let us now look at procedure names in more detail. We have not yet acknowledged the fact that they constitute the first use of names that we have seen other than for variables.

```
P: PROC OPTIONS (MAIN);
   Q: PROC;
   END;
   CALL Q;
END;
```

Let us look first at the procedure name Q. It appears as a label on a PROCEDURE statement, and as the name of a procedure to be called on a CALL statement. Q is said to be the name of an <u>entry constant</u>. The value of this constant is the procedure which it names (or, more precisely, the particular entry point into it, since there may be others).

So that we may talk about that constant, we gave it a name, Q (just as the FIXED DECIMAL(1) constant with value "one" is denoted by 1). The appearance of Q in

```
Q: PROC;
```

is a reference of the constant which serves to establish its value. The appearance of Q in

```
CALL Q;
```

is a reference of the constant which does something with its value, i.e., it invokes the procedure which is the constant's value.

## 4.7. Declaration of entry constants.

We have talked about entry values (values which represent entry points) and entry constants (objects whose permanent value is an entry value). Indeed, we will see later that there are entry variables (objects whose changeable value is an entry value). Thus, "entry" is a legitimate data type. In fact, the attribute used for declaring entry variables is the ENTRY attribute, but more about that later. Unlike, say, character strings, we cannot manufacture new values of type "entry" by operating on old ones. Thus, the number of different entry values that can exist at any moment during the execution of a program is exactly the number of different entry points of procedures that there are in it--each one named by an entry constant. Recall that we earlier called data that can be operated upon in expressions "problem data." In contrast to that, entry values constitute the first of many types of program control data that we will see.

We have seen how declarations associate a name with an object (and also associate some attributes with it). We have at hand a kind of constant that has a name, which is an identifier like a variable name. In fact, the association of that name with the constant named is also an act of declaration. In this regard, Q: PROC; constitutes an explicit declaration of an entry constant named Q. The scope of the declaration is the procedure containing the declaration, i.e., P. Attributes furnished by this declaration are ENTRY (the data type of the value) and INTERNAL (the scope of the name)--the latter because the procedure Q is an internal procedure. By the same logic, P: PROC OPTIONS (MAIN); explicitly declares P to be an EXTERNAL ENTRY constant, EXTERNAL because P is an external procedure.

Carrying this discussion a little further, we may ask how we may know that CALL P; appearing in some other external procedure (say Z) refers to this external procedure P. For that to be the case, P must be known, in Z, to be an external entry constant. How is that achieved? Answer: by the declaration, in Z, DCL P ENTRY EXTERNAL;. Note that this establishes P as an external entry constant; looking ahead, we may have external entry variables, but you have to do something extra to declare them. The scope of this declaration is the procedure Z (and, of course, descendants in which the name is not redeclared). The scope of the name is the union of the scopes of all its declarations, including the

one resulting from its use as a label on a PROCEDURE statement. All these declarations associate the name with the same object, an entry constant.

## 4.8. Begin blocks.

Procedures are one kind of block. Anther kind is the begin block. A begin block is delimited by a matching BEGIN statement and END statement, as in

```
BEGIN;
  :
  :
  body of begin block
  :
  :
END;
```

A begin block is sort of an unnamed procedure that takes no arguments and doesn't return a value. Its body is thus executed for its effect. If it is "like a procedure" then presumably it gets executed by being invoked. If it doesn't have a name, by what do we call it to invoke it? The answer is we don't need to call it, because we don't do anything special to invoke it. It is "invoked" (let us just say executed) when control reaches it in the normal way, for example, after executing the preceding statement. Thus, the BEGIN statement is executable, unlike a PROCEDURE statement (if control should reach a PROCEDURE statement from above, i.e., after executing the preceding statement, the procedure is not invoked; control skips to the first executable statement after the procedure).

Why then have begin blocks? Wouldn't the effect be exactly the same if we deleted a BEGIN statement and its matching END statement?

One reason for begin blocks is that they delimit scopes just like procedure blocks do. In fact, everywhere we have used the word "procedure" in terms of the concept of scope of a declaration, we should have used "block." Note that begin blocks may be nested inside begin blocks or procedure blocks, and internal procedure blocks may be nested inside begin blocks as well as other procedure blocks. At the outermost

level we still have external procedure blocks; there are no "external begin blocks." We will see another significant use of begin blocks in Lesson 5.

See LRM 78 and LRM 79.

## 4.9. The DEFAULT statement.

In Lesson 1 we said that the programmer can change the standard system defaults used to furnish attributes in implicit declarations or to complete partial declarations. The DEFAULT statement provides this facility. We will illustrate it by examples. (The abbreviation for DEFAULT is DFT.)

```
DFT RANGE (*) FIXED BINARY;
```

RANGE (*) says this DEFAULT statement applies to all variables. If they have no data type attributes they get FIXED BINARY (irrespective of the first letter of their name). If they already have a scale or a base attribute, but not both, the other is FIXED or BINARY as needed. This default is inapplicable to any variable that already has both scale and base attributes.

```
DFT RANGE (B:D) BINARY VARYING;
```

This specification is only applicable to variables whose names begin with B, C, or D. The attributes shown may seem to be in conflict with each other. They are just a list from which is taken, in order, any attributes that don't conflict with what the variable already has. If BINARY is taken, VARYING won't be. If the variable already has CHARACTER, BINARY won't be taken but VARYING will be.

```
DFT RANGE (XYZ) VALUE (BIT(8));
```

This specification only applies to variables whose names begin with XYZ. If the variable has BIT but no length specification, the length specification acquired is the value 8. Though we didn't say so in Lesson 2, one can write DCL XYZAB BIT;. The system default for string length is 1.

The order in which DEFAULT statements are processed is significant. If a variable belonging to a particular block

needs more attributes to complete its description, the DEFAULT statements of that block are examined, in top to bottom order, first. If its description is still incomplete, the block, if any, that contains that block has its DEFAULT statements examined, and so on out to the external procedure block. Thus, we may say that DEFAULT statements have a scope of applicability related to the block structure, i.e., the nesting properties of blocks.

Considerably more can be done with DEFAULT statements. See LRM 80 through LRM 82.

In the ANSI standard, the syntax and capabilities of the DEFAULT statement are almost totally changed--for the better. The applicability of DEFAULT statements may depend on the attributes a name already has or doesn't have. Additional attributes, such as DIMENSION, NONVARYING, STRUCTURE, CONSTANT, etc., are also provided for use in defining the universe of applicability of a given DEFAULT statement. It is also possible to default attributes of NONE, which will make it necessary to explicitly declare all required attributes, thus eliminating the danger of misspelling a name. And there are other useful and exotic things that can be done with it.

4.10. Unanswered questions.

How do we declare entry variables? How may they be used (other than in assignments)? (We know how entry constants are used.) See Lesson 5.

What are the requirements for argument/parameter matching? Also in Lesson 5.

4.11. Homework problems.

(#4A) Multiple declarations are not allowed. (For a definition of multiple declaration, see LRM 83.) If there are multiple declarations in any of the following, identify them.

(a)
```
P: PROC;
   DCL X FIXED BIN;
   Q: PROC;
      DCL X FLOAT BIN;
   END;
END;
```
(b)
```
P: PROC;
   DCL X FIXED BIN;
   Q: PROC;
      DCL X FLOAT BIN EXT;
   END;
END;
```
(c)
```
P: PROC;
   DCL X FIXED BIN EXT;
   Q: PROC;
      DCL X FLOAT BIN EXT;
   END;
END;
```
(d) Same as (a), but with the addition of DFT RANGE (*) EXT; just after the PROCEDURE statement for P.

(e) Same as (d), but with the DEFAULT statement added just after the PROCEDURE statement for Q.

(f)
```
S: PROC;
   T: PROC;
      T: PROC;
      END;
   END;
END;
```
(g)
```
S: PROC;
   T: PROC;
   END;
   T: PROC;
   END;
END;
```

(#4B) Suppose two different external procedures, E1 and E2, needed to call a third external procedure, E3. They would each, of course, contain a declaration such as

DCL E3 ENTRY EXT;

What do you think would happen if you forgot to write E3 and link-edit it in with E1 and E2? If you have linkage editor experience, describe what you think the linkage editor would have to say. Also see if you can give an answer purely in PL/I terms (hint: What kind of object is E3? What is its value?).

(#4C) Write a DEFAULT statement that will cause all variables not explicitly declared with a scale attribute, and all variables declared with FLOAT but no precision attribute, to default to double precision floating-point. Make sure that double precision will be used, even for variables explicitly declared with one of the base attributes. In the case where neither base attribute is explicitly declared, make BINARY the default. What is the effect of your DEFAULT statement on the following?

```
DCL J;
DCL X;
DCL U BINARY;
DCL V DECIMAL;
DCL F FLOAT;
```

# 5. Storage class and block invocations.

## 5.1. Storage allocation and initialization.

Storage allocation means the process of acquiring storage for a variable. There are several ways this may be carried out in PL/I, depending on choices made by the programmer. The choices range from having the compiler "assign" storage essentially at compile time (like in FORTRAN) to taking on full responsibility for saying when, during execution, storage should be acquired for a variable (and when it should be released). The latter extreme is an example of dynamic storage allocation. See LRM 84.

So far we have not been concerned with the process of storage allocation. It is sufficient to have thought in FORTRAN terms up to now.

Initialization is the process of assigning initial values to variables. In FORTRAN this is carried out with the DATA statement and BLOCK DATA subprograms. There are facilities for initialization in PL/I which are a little more general. To handle the requirements for initialization when storage is allocated dynamically, initialization occurs when (and each time) storage is allocated.

## 5.2. Storage class attributes.

The storage allocation technique to be used for a specific variable is selected by declaring one of four alternative storage class attributes for it. Storage class is a property of all (or essentially all) variables. With its study we will complete the analysis of properties (data type, aggregate type, alignment, scope, storage class) that all variables have.

The four storage class attributes are STATIC, AUTOMATIC (abbreviation: AUTO), CONTROLLED (abbreviation: CTL), and BASED. The last three designate different types of dynamic storage allocation. BASED will not be considered until Lesson 11. Static, automatic, and controlled storage are described separately below.

## 5.3. INITIAL attribute.

First we will consider the common aspects of initialization, since it will be appropriate to consider certain aspects of it which differ with the storage class as the individual storage classes are studied.

Initial values are specified by the INITIAL attribute. The attribute may be used for scalars, arrays, and structure base elements. Its abbreviation is INIT.

For a scalar or structure base element, the form is INIT (*initial-value*). *initial-value* may be any constant, and in some cases it may be a variable reference or function reference or even an arbitrary expression (if it is an expression it must be surrounded by parentheses).

Examples:

```
DCL N FIXED BIN (31) INIT (0);
DCL X FLOAT INIT (1);
DCL 1 STRUC,
      2 PART1 CHAR (3) INIT ('ABC'),
      2 PART2,
        3 PART2A BIT (2) INIT ('01'B),
        3 PART3B CHAR (4) VAR INIT ('');
DCL Y FIXED DEC (7,2) INIT (X);
DCL Z PIC '9999' INIT ((N**2-14));
```

For an array one form is

INIT (*initial-value*, ..., *initial-value*)

i.e., a list of initial values, one for each array element. The order corresponds to successive elements of the array "by row," i.e., with the right-most subscript varying most rapidly. For example, to initialize a 3x2 array A to

```
1 0
-3 3
8 -1
```

we would write

```
DCL A (3,2) INIT (1,0,-3,3,8,-1);
```

The number of initial values given may be less than the number of elements in the array (in which case elements at the end remain unitialized), but it may never be greater

(excess values are ignored). To denote that a particular element is not to be initialized, an asterisk may be used instead of an initial value. For instance, if we did not need to, or care to, initialize the second row of A we could have written

```
DCL A (3,2) INIT (1,0,*,*,8,-1);
```

A sequence of similar initial values may be "factored out" and preceded by a parenthesized <u>iteration factor</u>, which denotes how many times the following item or list of items is to be iterated. Examples:

```
DCL A (10) INIT (3,(9)0);
```

A(1) is initialized to 3 and the remaining elements are initialized to 0.

```
DCL B (3,3) INIT ((3)(0,1,2));
```

Each row of B is initialized to 0,1,2.

```
DCL C (3,3) INIT ((3)(0,(2)1));
```

Each row of C is initialized to 0,1,1.

```
DCL D (10) INIT (0,(8)*,0);
```

The first and last values of D are initialized to 0; the middle eight values are uninitialized.

The INITIAL attribute may be specified in a DEFAULT statement. Note that standard system defaults do not cause initialization of any variables. It is illegal to use a variable in a context where its value is required before it is assigned a value (either by initialization, by assignment, or by an input operation). Under the Optimizing compiler, reference to an uninitialized variable will access garbage, and unpredictable errors may result. The Checkout compiler, however, is able to detect and report use of uninitialized variables (which is a very common error).

See LRM 85 and LRM 86.

Note that if A and B are similar arrays, it is not legal to write, say,

```
DCL A (3,2) INIT (B);
```

even though it may seem intuitively clear. Any references in the INITIAL attribute must be references to element variables (scalars), and expressions must be element expressions (those that evaluate to scalar quantities).

## 5.4. Adjustable extents.

All of the array bounds and string lengths we have shown so far have been expressed as unsigned decimal integer constants. Syntactically, they may, in general, be expressions (element expressions), but this is permitted only with certain storage classes, as we will see below. An array bound or string length which is not constant is called an adjustable extent. In Lesson 11 we will see another type of variable which can have an adjustable extent.

## 5.5. Static variables.

Variables declared with the STATIC storage class attribute are fully mapped and logically allocated a place in storage at compile time. In fact, this storage is just a part of the "load module" containing the program itself. Initial values are assembled right into this storage.

When a program is loaded, static storage--already initialized, if required--is brought in with it. Static variables retain their assigned locations throughout execution.

In order to permit full mapping and initialization at compile time, static variables cannot have adjustable extents, and initial values and iteration factors in any INITIAL attribute must all be constants. See LRM 87.

Example:

```
     P: PROC;
        DCL #CALLS FIXED BIN STATIC INIT (0);
        #CALLS = #CALLS + 1;
        :
        :
     END;
```

In this example, the static internal variable #CALLS is used to record the number of times P is invoked. Because #CALLS has internal scope (by default), it is not accessible to the program outside of the procedure P. However, it continues to occupy its storage location, and its value, even when control leaves P. It still has the same location and value when control re-enters P at a later time. Thus, static variables may be used to maintain a "history" across procedure calls.

## 5.6. Automatic variables.

Variables declared with the AUTOMATIC storage class attribute are allocated, and initialized, whenever control enters the block that declares them. The storage is freed when that block terminates.

This is one of the types of dynamic storage allocation. Since storage for an automatic variable is not allocated and initialized until a certain point during execution, it may have adjustable extents as well as expressions in the INITIAL attribute.

Example:

```
P: PROC;
   DCL (L,M,N) FIXED BIN;
   L = 3;
   M = 8;
   N = 6;
   BEGIN;
      DCL C CHAR (L) AUTO;
      DCL B BIT (L+1) VARYING AUTO;
      DCL A (M,N) BIT (L**2) AUTO;
      DCL X (N) INIT ((N)0) AUTO;
      DCL Y (L,M)
            INIT ((L)(1,(M-1)0)) AUTO;
   END;
END;
```

When the begin block is entered, C is established as a character string variable of length 3 (the value of L). B is established as a varying-length bit string of maximum length 4. A is established as an 8x6 array of bit strings of length 9. X is a 6-element array all of whose elements are initialized to 0. (Note that if we had written

```
DCL X (N) INIT (0) AUTO;
```

only the first element would have been initialized.) Y is a 3x8 array whose first column is initialized to 1 and whose remaining elements are initialized to 0.

Note that the determination of adjustable extents and initial values is determined exactly at block entry time, before any statements are executed in the block. Also, even though the variables used in extent expressions may have new values assigned to them in the block, the bounds and string lengths won't change.

Note that, since storage for automatic variables is freed when their containing block terminates, they may not be used to retain a history across block invocations. The next time their declaring block is entered they will be assigned fresh storage, which may be in a different location. See LRM 88.

Automatic storage is primarily used for local (i.e., internal) variables with adjustable extents. It is also essential in recursive procedures, as we shall see later in this lesson, and re-entrant procedures (Lesson 14).

Initialization of automatic variables is carried out by generated code. If they have adjustable extents, storage allocation is also carried out by generated code. However, if they have fixed extents they come essentially for free: since the compiler knows their extents, it assigns them consecutive locations in one contiguous area which is not allocated until the declaring block is entered. The allocations are "free" since each block will need such an area anyway, for housekeeping, even if it has no automatic variables.

## 5.7. Controlled variables.

Variables declared with the CONTROLLED storage class attribute are allocated, and initialized, upon execution of an <u>ALLOCATE statement</u> naming them, and they are released upon execution of a <u>FREE statement</u> naming them. The allocation and freeing need not occur in the same block.

Controlled variables can have several simultaneous <u>generations</u> of storage. If a controlled variable being allocated already has an allocation (called a generation), that former allocation is placed on a <u>stack</u>. All subsequent references to the variable are references to the newly allocated generation of it, until a FREE statement is executed. At that time the "current" generation is released and the one on top of the stack replaces it.

Example:

```
DCL X CTL;
ALLOCATE X;
X = 1;
ALLOCATE X; Stacks previous X (having value 1).
X = 2;
Y = X; Stores the value 2.
FREE X; Unstacks previous X.
Y = X; Stores the value 1.
FREE X; There are now no allocations of X.
```

It is an error to refer to a controlled variable for which no allocations exist.

Controlled variables are the thing to use, obviously, whenever you need a real "pushdown" stack, or LIFO (last-in-first-out) stack.

Since the controlled storage class is one of the dynamic storage classes, controlled variables can have adjustable extents and variable initializations. An ALLOCATE statement for a controlled variable may well appear in a block different from the one containing its declaration. There may also appear in that block declarations of variables having the same names as ones used, for instance, in extent expressions in the declaration of the controlled variable. Upon allocation, the variables accessed during the evaluation of extent expressions are the ones "known" in the block containing the controlled declaration; the values used, however, are their current values, i.e., not necessarily the ones in effect when the declaring block was entered. A homework problem will illustrate this.

In reading LRM 89, you will see that it is possible to override extent expressions, etc., given in the declaration, by using different ones in the ALLOCATE statement (for this purpose you have to write out the attributes in the ALLOCATE statement). When extents are given in the ALLOCATE statement they may be omitted (replaced by asterisks) in the declaration. Use of the features described in this paragraph is not recommended because they are not carried over to the ANSI standard.

## 5.8. Combinations of storage class and scope attributes.

Static variables may have either internal or external scope.

Automatic variables can have only internal scope. Since automatic variables only "exist" while the declaring block is active, it is not meaningful to link the scopes of different declarations so that they refer to the same automatic variable. Of course, automatic variables may be referenced in blocks contained within the declaring block (because the scope of the declaration contains the nested block). There is no way for the declaring block not to be active when such a reference is made.

Controlled variables can have either internal or external scope. With controlled external, the whole stack of allocations is "shared" amongst the scopes of the various external declarations of the variable.

In Lesson 4 we stated that external variables can conveniently be used for communication amongst several external procedures. Now consider that external variables can have either static or controlled storage class, but not automatic. Since static variables can not have adjustable extents, if a variable communicated amongst external procedures by giving it external scope (as opposed to passing it as an argument) needs to have extents determined during execution, it will obviously have to be controlled. Note that there may be no need for the general stacking capability in this case, i.e., only one generation of the controlled variable is ever allocated. This, in addition to LIFO stacks, is an "appropriate" use of controlled variables.

If the storage class attribute is omitted from a declaration, standard defaults will supply AUTO for internal variables and STATIC for external ones. Since INTERNAL is the standard default when the scope attribute is omitted, most variables will probably end up being automatic. Since additional execution time is incurred for certain uses of automatic variables, it may well be worthwhile to say DFT RANGE (*) STATIC; to change the default.

For a review, see LRM 90 ignoring (for now) all discussion of the BASED attribute.

## 5.9. Parameters.

Names appearing in a parameter list in a procedure statement are names of formal <u>parameters</u> ("dummy arguments" in FORTRAN).

The process of invoking a procedure makes the formal parameters synonymous with the actual arguments in a CALL statement or function reference. By synonymous is meant that they designate the same storage and the same value, as with defined variables (Lesson 3). Hence, an assignment to a formal parameter may be instantly perceived as a change in the value of the actual argument, assuming it is a variable. And there are no restrictions on that variable (the actual argument) like those of FORTRAN; specifically, the variable may be another argument, as in

```
CALL F(A,A,B);
```

or it may be an external variable to which the invoked procedure has direct access. The price of this flexibility is inhibited optimization. For instance, suppose in F an assignment is made to the first formal parameter. The compiler must be aware that the second parameter, which is a different variable in F, can have its value changed by that assignment.

Note that formal parameters do not denote local variables which are assigned the value of the actual argument on entry and which are assigned back to the argument on return, as in FORTRAN (for scalar arguments). This has consequences that will be seen when we consider multiple entry points, later.

There is also no restriction against assigning to a formal parameter whose actual argument is a constant. In this case the constant is protected because the calling procedure makes a copy of it just before the call and passes the copy instead.

Parameters generally cannot be declared with a storage class attribute. They don't have storage of their own; they share the storage of the actual argument. In this sense, "parameter" may be considered an alternative to the other storage classes. An exception is discussed immediately below.

When a controlled variable is passed as an argument, either the current generation of the variable or the whole stack of generations may be considered passed, depending on whether the formal parameter does not, or does, have the CONTROLLED attribute, respectively. This is the one exception to the above prohibition of storage class for parameters. It is an error to pass a non-controlled variable to a controlled parameter. Note that controlled parameters are not permitted in the ANSI standard.

## 5.10. Review and extension of DEFINED attribute.

Before proceeding with the study of parameters we shall look again at defined variables, first introduced in Lesson 3.

The first point to be made is occasioned by the comment above that parameters don't have storage class. Neither do defined variables. They share the storage of their base variable. DEFINED, like "parameter," may be thought of as an alternative to storage class.

The second point to be made is that defined variables, like variables of any of the dynamic storage classes, can have adjustable extents. The extent expressions, like those for automatic variables, are evaluated on entry to the declaring block. Consider the following example:

```
J = 3;
K = 5;
L = 7;
BEGIN;
   DCL A (J,K,L) FLOAT;
   DCL B (K,J) FLOAT
         DEF A(2SUB,1SUB,I);
   :
   :
END;
```

Note: In the ANSI standard, declarations of defined and auto variables may not reference other defined or auto variables <u>declared in the same block</u>. Hence, the declaration of B is in error. It is corrected by enclosing it in another begin block.

In the begin block, A is a 3x5x7 array. B is a 5x3 array made coincident with the transpose of the I-th plane of A. While the values of K and J are determined for purposes of ascertaining B's extents once, on entry to the begin block, the extents not subsequently tracking any changes in the values of K and J, I is <u>not</u> evaluated at block entry but rather on every reference to B. See LRM 91.

## 5.11. Argument/parameter matching requirements.

As you might expect by now, arguments and parameters must have the same data type, i.e., it is illegal to pass a floating-point argument to a fixed-point parameter, illegal to pass a CHAR (4) argument to a BIT (32) parameter, and so forth. You should expect this because of the matching requirements we have seen for defined variables and external variables. In all cases, the reason is to guarantee identical semantics for all implementations of PL/I; it just cannot be done when one is allowed to relax these rules.

Suppose a parameter is declared FIXED BIN (15). If one wants to pass the constant "one" to this parameter, can one write "1" for the actual argument? After all, "1" as written is FIXED DEC (1). The answer is yes, if. If you tell the compiler what kind of value the invoked procedure expects. If you don't, it will just pass a FIXED DEC (1) constant and errors surely will result.

Actually, it is necessary to provide the compiler with information about the invoked procedure's parameters, in the calling procedure, only when the procedure being called is an external procedure. An entry declaration is used for this purpose. The reason it is not necessary (in fact, not allowed) for internal procedures is because in this case the compiler can look inside the procedure to be invoked while it is compiling the calling procedure, and it can thus find out what attributes are required.

One essential freedom permitted in these otherwise stringent matching requirements in that array bounds and string lengths of parameters need not be specified as unsigned decimal integer constants. (They may be, however, and then they must agree exactly with the array bounds or string lengths of their actual arguments.) These extents can be expressed as asterisks, which means that the extent of the formal parameter is inherited from the actual argument. This permits arrays with different bounds (but the same number of dimensions), or strings with different lengths, to be passed as arguments, at different times, to the same formal parameter.

For example:

```
DCL S1 CHAR (5) INIT ('AAAAA');
DCL S2 CHAR (3) INIT ('BBB');
CALL INTPROC)S1);
CALL INTPROC(S2);
INTPROC: PROC (S);
   DCL S CHAR (*);
   I = LENGTH(S);
END;
```

The first time INTPROC is called, its parameter, S, behaves like a CHAR (5) variable; in particular, 5 is assigned to I. On the second invocation, S behaves like a CHAR (3) variable and 3 is assigned to I.

Suppose we pass arrays with different extents to an array parameter with asterisk extents. How are we to ascertain

the bounds of the parameter (i.e., those of the actual argument), if we should need to (for instance, to iterate over all elements of the array)? Certain builtin functions, in the category called "array-handling builtin functions," serve this need.

If A is an array, HBOUND(A,*i*) is the upper bound ("high bound") of A in the *i*-th dimension. *i* may, in general, be an expression, but it is usually a constant like 1 or 2. Similarly, LBOUND(A,*i*) is the lower bound of A in the *i*-th dimension. DIM(A,*i*) is equal to HBOUND(A,*i*)-LBOUND(A,*i*) + 1, i.e., it is the number of elements in the *i*-th dimension of A.

Example:

```
     P: PROC (A);
        DCL A (*,*) FLOAT;
        DCL B (LBOUND(A,2): HBOUND(A,2))
              FLOAT DEF A(I,*);
        .
        .
        .
     END;
```

A is a two-dimensional array with bounds in both dimensions inherited from the actual argument. B is defined on the I-th row of A; in its one and only dimension, it has bounds equal to those of the second dimension of A.

Note that "asterisk extents" are a type of adjustable extent. It is the only type permitted in parameter declarations, i.e., it is illegal to write

```
     P: PROC (A,I,J);
        DCL A (I,J);
```

The FORTRAN programmer converting to PL/I must make a conscious effort not to think about array parameters in terms of the address of the first element. Array parameters can only be associated with array arguments; they must have the same number of dimensions and the same bounds in each dimension. It is never necessary to pass the bounds separately. It is just as illegal to refer outside the bounds of a parameter array as it is to reference outside the bounds of any array.

## 5.12. Entry declarations.

In Lesson 4 we saw that the ENTRY attribute can be used in a declaration to declare a name as that of an external

procedure (i.e., an external entry constant). The declaration may also describe the attributes of the formal parameters of the external procedure.

Example:

```
     DCL F ENTRY (FIXED BIN (15)) EXT;
```

This says that F is an external entry constant, and that the procedure F has one parameter, which in F is declared as FIXED BIN (15). Having written the above declaration, you can now write CALL F(1); without fear of having the wrong data type for the actual argument. The compiler has the information it needs to substitute a FIXED BIN (15) constant with value "one."

The conversion of argument type to parameter type occurs whenever it is necessary. For instance, in

```
DCL J FLOAT BIN (10);
CALL F(J);
```

J is converted from FLOAT BIN (10) to FIXED BIN (15). The result is placed in a "temporary," sometimes called a "dummy" in PL/I, and it is the temporary which is passed as an argument. In this case, assignment of a value to the parameter of F will <u>not</u> cause the value of J to change, because the parameter is not associated with J but rather with an auxiliary variable containing the converted value of J. The compiler tells you whenever it creates a "dummy" for an argument in order to get the matching required.

As you read LRM 92 and LRM 93, you will see that the descriptions for individual parameters may be omitted (replaced by asterisks), in which case it is assumed that the argument as passed is correct for the parameter (it is an error if it isn't). Indeed, the whole list of parameter descriptions, and their enclosing parentheses, may be omitted (with the same assumptions). However, it is good practice to declare the parameter attributes of external procedures <u>always</u>, and the ANSI version requires this.

External entry constants <u>must</u> be declared in an entry declaration, even if there are no parameters to describe. You might well ask why. If a name appears in a CALL statement, as in CALL SUBR, or in function reference context, as in A=B+SIZE(C), why is not that name <u>assumed</u> to be an external entry, as in FORTRAN, when no array declaration (in the latter case) or internal procedure (in either case) (in FORTRAN this would be an arithmetic statement function) were observed by the compiler? The answer is: to permit growth

of the language in the area of builtin procedures. (In Lesson 12 we will see that there are some implementation-defined builtin procedures that are subroutines, i.e., to be invoked by CALL statements.) What would happen if SIZE (as in the above example) were to be added to the language as a builtin function tomorrow? If SIZE could be an external entry without declaration, then the meaning of the program would change after SIZE is added as a builtin function. (Though it has not been emphasized, builtin functions generally do not have to be declared. Exceptions to this rule are treated in Lesson 10.) By declaring SIZE as an external entry, you are protected even if SIZE is added as a builtin function tomorrow.

## 5.13. The CONNECTED attribute.

The CONNECTED attribute may be specified for aggregate parameters. In general, the compiler may not assume that a parameter which is an aggregate is connected. For example, since arrays are stored by row in PL/I, passing a column, such as A(*,I) to a one-dimensional array results in the parameter being associated with unconnected storage. Even if the parameter is a structure, it can refer to unconnected storage! A case in point is the passing of an element of an array of structures. The CONNECTED attribute tells the compiler that the associated aggregate argument actually is in connected storage. Besides leading to certain efficiencies, this information confirms a condition which is a prerequisite for certain kinds of I/O involving aggregate parameters (Lessons 8-9) and for string overlay defining (Lesson 3) on a parameter base.

When the CONNECTED attribute is specified in a parameter description in an ENTRY attribute, for instance

```
DCL P ENTRY ((*) FLOAT CONNECTED);
```

which says that P expects a one-dimensional connected array of FLOAT elements, a copy of the argument is made in connected temporary storage if the argument, as supplied, is not connected. See LRM 94.

CONNECTED is not a part of the ANSI standard. If you use the features cited above as requiring connected references, it is assumed that the connected condition is met; otherwise, the program is in error.

## 5.14. Function references and the RETURN statement.

When a procedure is invoked as a subroutine reference, it may return to the point of invocation either by executing a RETURN statement that does not include an expression for the returned value, or by executing (i.e., reaching) the END statement of the procedure.

When a procedure is invoked as a function reference, the latter mechanism is not available to it. It must execute a RETURN statement that includes an expression giving the returned value, as in

```
RETURN (B**2-4*A*C);
```

Note that the mechanism for specifying a returned value is rather different from FORTRAN. Instead of assigning to a variable which has the name of the function, then executing a RETURN statement later, we carry out both functions in a single statement.

Returned values have data types. Both the calling and the invoked procedure must agree on the data type of the returned value. The rule is that the data type is inferred from the first letter of the procedure's name (more precisely, the name of the entry point), in the same way as for undeclared variables and using the same defaults, unless specified otherwise. There are two places where other attributes may be specified.

The first place is on the PROCEDURE statement, in the RETURNS option.

```
P: PROC (X) RETURNS (CHAR (40));
```

specifies, for example, that P returns a value of type CHAR (40). If you happen to write RETURN ('NONE') the given value will be converted from CHAR (4) to CHAR (40), in the invoked procedure, to conform to the CHAR (40) that you have said must be returned.

The second place is in an entry declaration (for an external entry) on the calling side. The difference between

```
DCL P ENTRY (FIXED) EXT:
```

and

```
DCL P ENTRY (FIXED) RETURNS (CHAR (40)) EXT;
```

when P is invoked in function reference context, as in

```
S = T || P(5);
```

is that in the former case the attributes assumed for the value returned depend on the first letter of the name (and will be FLOAT DEC (6) in this case), whereas in the latter case they are known to be CHAR (40).

In the ANSI version, the RETURNS option and RETURNS attribute can be used if and only if the procedure is invoked in function reference context, and they must be used then.

In the current language a returned value must be a scalar. Furthermore, if it is a string it must have a non-adjustable length (or maximum length, in the case of varying-length strings). In the ANSI language, a returned value may be an array or a structure and it can be specified to have adjustable extents (using the asterisk notation only).

See LRM 95.

## 5.15. Recursive procedures.

Recursive procedures are allowed. They must be identified as recursive by the RECURSIVE option on the PROCEDURE statement. The familiar example of FACTORIAL is given below. (It uses an IF statement, which we will encounter in Lesson 6.)

```
FACTORIAL: PROC (N) RETURNS (FIXED BIN (31)) RECURSIVE;
   DCL N FIXED BIN (31);
   IF N > 1 THEN RETURN (FACTORIAL(N-1));
   ELSE RETURN (1);
END;
```

If a recursive procedure needs any local variables, it is essential that the automatic storage class be used for them. The essential feature of a recursive procedure is that several invocations of it are active simultaneously. If STATIC is used for local variables, all invocations would share the one "generation" of the static variable. With AUTO, each active invocation has its own "generation" of the local variable.

## 5.16. Multiple entry points and the ENTRY statement.

Like FORTRAN, PL/I provides for multiple entry points into a procedure. The ENTRY statement is used to designate a secondary entry point. The ENTRY statement looks basically just like a PROCEDURE statement except that the ENTRY keyword replaces the PROCEDURE keyword and certain options are not allowed.

The different entry points of a procedure can have different parameter lists. It is incorrect to refer, in the body of a procedure, to a parameter appearing in some parameter list but not the one at the entry point through which entry was made. Example:

```
P: PROC (A,B,C);
   :
   :
Q: ENTRY (B,C,D);
   DCL (A,B,C,D)...;
   body of procedure
END;
```

If entry is made at P, references to A, B, and C are legal; references to D are illegal. If entry is made at Q, references to B, C, and D are legal; references to A are illegal. Note that this is in contrast to the FORTRAN technique of establishing various values in parameters of the procedure by entering once through an "initialization" entry point with a long parameter list, and then making subsequent "high-speed" entries through a different entry point having a much shorter parameter list, with subsequent references to the earlier parameters.

The different entry points may return values with different attributes. When a RETURN statement is executed, a "switch" is tested by the compiled code to determine which entry point was used; the code may need to branch on the outcome of this test to different sections of code that convert the returned value to the appropriate attributes. Example:

```
P: PROC (X) RETURNS (FIXED);
Q: ENTRY (X) RETURNS (FLOAT);
   :
   :
   RETURN (X/3+Y);
END;
```

The value of the expression X/3+Y, which has certain attributes, will be converted to FIXED or FLOAT depending on whether entry was made at P or at Q.

See LRM 96.

## 5.17. Generic procedures.

Recall in Lesson 1 we said that the mathematical builtin functions were "generic" in the sense that they could accept, under one name, arguments with a wide range of different attributes.

It is possible to give the appearance of calling a user-defined procedure with different types of arguments (maybe even different numbers of arguments) in the different calls. The name called is not itself an entry constant, that is, a label on some procedure. It will be replaced by an entry constant selected from a list, based on the numbers and types of the arguments. The GENERIC attribute is used for this.

Example:

```
DCL E GENERIC
      (E1 WHEN (*),
       E2 WHEN (*,*));
DCL E1 ENTRY (FIXED) EXT;
DCL E2 ENTRY (FIXED, FLOAT) EXT;
```

A reference to E with one argument, as in CALL E(A+B); resolves to E1, i.e., the statement is the same as CALL E1(A+B). A reference to E with two arguments, as in CALL E(A,B); resolves to E2,i.e., the statement is the same as CALL E2(A,B).

```
DCL F GENERIC
      (F1 WHEN (FIXED BIN),
       F2 WHEN (FLOAT BIN),
       F2 WHEN (FLOAT DEC));
DCL F1 ENTRY (FIXED BIN (15)) EXT;
DCL F2 ENTRY (FLOAT DEC (6)) EXT;
```

CALL F(N+1) resolves to F1 (if N is FIXED BIN).
CALL F(X+1) resolves to F2 if X is either FLOAT BIN or FLOAT DEC; conversion of the argument from FLOAT BIN to FLOAT DEC occurs in the former case.

Note that generic selection is carried out statically, i.e., the resolution occurs at compile time. See LRM 97.

5.18. Review of procedure invocations.

For a complete review of the dynamic aspects of procedures, see LRM 98 (which covers some material we will see later) and LRM 99.

5.19. Homework problems.

(#5A) Assume S is a square array of CHAR (1) elements with N rows and columns (N > 1). Write a declaration for S that initializes the elements on the perimeter of the array to '*' and those in the interior to 'Ø'.

(#5B) What value is assigned to I?

```
DCL (I,N) FIXED BIN;
N = 3;
BEGIN;
   DCL A (N) FLOAT AUTO;
   N = 7;
   I = HBOUND(A,1);
END;
```

Would the result be the same if the first two statements after BEGIN were interchanged?

(#5C) What values are assigned to I?

```
DCL (I,N) FIXED BIN;
DCL A (N) FLOAT CTL;
N = 3;
BEGIN;
   DCL N FIXED BIN;
   N = 4;
   ALLOCATE A;
   I = HBOUND(A,1);
   N = 5;
   I = HBOUND(A,1);
END;
N = 6;
I = HBOUND(A,1);
ALLOCATE A;
I = HBOUND(A,1);
N = 7;
FREE A;
I = HBOUND(A,1);
```

(#5D) Write a procedure, to be called as a subroutine, which accepts a square array of any size and sets all the diagonal elements to 0. You won't need to code any loops.

(#5E) Suppose you are designing a procedure to carry out some transformation on an array. Suppose this trans-

formation requires "workspace" which is a function of the size of the array. Discuss how you would solve this problem in FORTRAN (if you have FORTRAN experience) and in PL/I.

(#5F) Write a procedure, to be called as a subroutine, which accepts a square array of any size and assigns to that array its own transpose. Do it without coding any loops.

(#5G) Can you guess why the expression for the returned value in a RETURN statement must be surrounded by parentheses? That is, why is RETURN (A+B) required? Why not just RETURN A+B? Hint: Suppose the outer parentheses <u>could</u> be omitted in RETURN ((A+B)=1). What problems would be encountered?

6. (a) Control constructs
   (b) Conditions

## 6.1. IF Statement

The IF statement may be used to achieve conditional execution of a statement or group of statements.

There are two forms:

(1) IF *expression* THEN *true-part;*
(2) IF *expression* THEN *true-part;*
ELSE *false-part;*

*true-part* and *false-part* are either single statements or groups of statements, as we shall see below. They may be other IF statements, begin blocks, etc.

*expression* is evaluated and converted, if necessary, to a bit string value. If any bit in the bit string is a 1, the *true-part* is executed, after which control goes to the next statement (case 1) or the statement after the *false-part* (case 2). If no bit is a 1, the *true-part* is not executed. In case 1, control arrives at the next statement without executing the *true-part*. In case 2, the *false-part* is executed, then control goes to the statement after that.

The most common form for *expression* is a comparison operation, which yields a BIT(1) result. Example:

```
IF A < B THEN A = A + 1;
```

Often, *expression* is a logical expression representing logical operations on bit strings (usually obtained from comparisons). Example:

```
IF I < 10 | J = I THEN CALL FOUL;
ELSE RETURN (J+2);
```

Another useful form is illustrated in

```
IF L THEN ...;
```

where L is a bit string variable (BIT(1) probably) given a value in a previous assignment.

In Lesson 2 we saw bit string expressions in the context of assignment statements. Although the same kinds of expressions are employed in an IF statement, the code generated may be quite

different since here it has as its goal a conditional branch. An optimizing compiler may not in fact need to evaluate the whole expression in order to determine the end result. However, that is not something you should count on, because the language definition does not insist that the code stop evaluating an expression as soon as the result is known; it merely permits it. Hence, the statement

```
IF I <= HBOUND(X,1) & X(I) = Y THEN ...;
```

is at best risky; the proper way to code this is

```
IF I <= HBOUND(X,1) THEN
      IF X(I) = Y THEN...;
```

It should be noted that the expression in the IF clause must be an element expression (i.e., a scalar-valued expression). That means that if A and B are congruent arrays, it is not possible to write IF A = B THEN ...; (Recall the discussion of aggregate expressions from Lesson 3.) The result of A = B is a congruent array of BIT (1) elements, each element having the bit value 1 or 0 depending on whether or not the corresponding elements of A and B are equal.) Certain builtin functions, which we shall see in Lesson 10, can be employed to achieve what is probably desired here.

When IF statements are nested, an ELSE clause is assumed to belong to the <u>nearest</u> "unmatched" THEN clause. That is, in

```
IF B THEN
   IF C THEN action-1;
   ELSE action-2;
```

*action-2* is executed when B is "true" and C is "false". (Neither action is executed if B is "false".) If it is intended that the ELSE clause match the other THEN clause in this example, one solution is to match the inner THEN clause with a <u>null statement</u>, which is just a semicolon. (You wouldn't believe how fast the generated code for a null statement is!) Example:

```
IF B THEN
  IF C THEN action-1;
  ELSE;
ELSE action-2;
```

Now *action-2* is executed if B is "false". If B is "true" and C is "false", nothing is executed.

See LRM 100 and LRM 101.

## 6.2. Non-iterative DO groups.

If either the *true-part* or *false-part* of an IF statement must be more than a single statement, a <u>non-iterative DO group</u> may be employed, as

in

```
IF A > B THEN DO;
   TEMP = A;
   A = B;
   B = TEMP;
END;
```

The list of statements bracketed by DO...END becomes a single syntactical unit that may be used wherever a single statement is allowed.

The problem solved earlier with the null statement may equally well have been solved with a non-iterative DO group as follows:

```
IF B THEN DO;
   IF C THEN action-1;
END;
ELSE action-2;
```

The difference between a non-iterative DO group and a begin block (which could also have been used to achieve the desired statement grouping) is that a DO group does not alter the "block structure," i.e., does not introduce a nested block inside which declarations may have their own local scope. The limited purpose it serves is implemented much more efficiently than would be the case with a begin block, even one devoid of local declarations and other things that require special housekeeping actions during execution.

See LRM 102.

## 6.3. Iterative DO groups.

There are two kinds of DO groups that provide for repetitive execution of a group of statements, the WHILE-only DO group and the controlled (or indexed) DO group.

## 6.4. WHILE-only DO groups.

This form of DO group is as follows:

```
DO WHILE (expression);
   body of group
END;
```

The body of the group (a statement list) is executed as long as the

*expression* evaluates to "true". The *expression* is evaluated at the top of the loop, so that if it is initially "false" the body of the loop is not executed at all. The *expression* is converted, if necessary, to a bit string value and interpreted to mean true or false exactly as in IF statements. That is, if any bit has the bit value 1, it means true; otherwise, it means false.

Examples:

```
DO WHILE(A < B & ¬DONE);
  .
  .
END;
DO WHILE('1'B);
  .
  .
END;
```

The second DO group will be executed forever. Presumably, provision is made to break the loop by executing a RETURN statement somewhere inside the loop which will immediately return control from the containing procedure to its point of invocation.

See LRM 103.

## 6.5. Controlled (indexed) DO groups.

In its simplest form this is analogous to the FORTRAN DO loop. For example,

```
DO I = 1 TO K;
  .
  .
END;
```

says that the body of the loop is repeated K times with I having the values 1,...,K. Note that if K is 0 or negative, the body of the loop is not executed at all since the test is performed at the top of the loop.

A BY clause may be added to the above form to permit increments other than 1. The increment may be negative, in which case the loop terminates when the control variable (I in the example) exceeds the final value in the negative direction; a simple, useful example is

```
DO I = K TO 1 BY -1;
   .
   .
END;
```

The initial and final values, and the increment, may be specified by arbitrary element expressions; they need not be restricted to constants or variables. The expressions are evaluated once and the saved values are used in the test each time through the loop.

Another useful form is to employ the BY clause but not the TO clause. This designates an infinite loop which must be broken by a RETURN statement or a branch to a point outside the loop (as in DO WHILE('1'B);).

To any of the above forms may be added a <u>WHILE clause</u> (which has the same meaning as in a WHILE-only DO group). The while-test is performed after the comparison of the control variable with the final value, and of course only if the final value has not been exceeded. If the while-test fails, the loop is terminated. Example:

```
DO I = 1 TO HBOUND(X,1) WHILE (X(I)=Y);
END;
```

This loop, which has an empty body, terminates either when I exceeds the upper bound of X (with all elements of X equal to Y) or when an element of X not equal to Y is found. By the way, the control variable may be used below the loop, after its termination; it has the value it had when the loop terminated (e.g., in this case either HBOUND(X,1)+1 or the smallest value *i* between 1 and HBOUND(X,1) such that X( *i* ) is not equal to Y).

The different forms shown above for what can come after the assignment symbol in the DO statement are all referred to as forms of a single <u>DO specification</u>. In general, any number of separate DO specifications may be written. When one is "exhausted," the next one is begun. For example:

```
DO I = 1 TO J-1, J+1 TO K;
   .
   .
END;
```

Here we have two specifications, each of the form *a* to *b*. The effect of the above is to execute the body of the loop for all values of I from 1 to K, except for the single value J.

One final form for a DO specification is permitted. It is the form without a TO clause <u>or</u> a BY clause ( <u>or</u> a WHILE clause). This says

execute the body exactly once, namely with the control variable taking on the initial value. This form is of use when several such DO specifications are written. For example,

```
DO I = 1,10,2;
  .
  .
END;
```

The body of the loop is executed exactly three times, with I taking on the three values shown during successive iterations. Do not confuse this with the FORTRAN DO loop!

Note that the control variable can be any kind of element variable; it is not restricted to being an "integer variable" (and unsubscripted) as in FORTRAN.

See LRM 104.

An additional form

DO *variable* = *initial-value* REPEAT (*expression*);

is provided in the ANSI language. *expression* is evaluated each time through the loop, after the first, and assigned to variable. Termination would be controlled by a WHILE clause (not shown). An example is DO I = A(1) REPEAT (A(I)) WHILE (I ¬= 0);

## 6.6. GO TO statement and statement labels.

A statement label is an identifier prefixed to a statement (other than a PROCEDURE or ENTRY statement) with a colon, as in

```
LAB3:  A = B-2;
```

A statement label may be used in a GO TO statement to effect an unconditional branch, as in GO TO LAB3; Statement labels and GO TO statements should be avoided in preference to the other control constructs since their undisciplined use results in programs that are harder to understand, harder to prove correct, and harder to modify.

## 6.7. label values; the LABEL attribute.

In Lesson 4 we casually hinted at a data type called "entry", explaining that procedure labels were entry constants, i.e., constants of that data type. We will explore that more fully below.

We have at hand another kind of program control (as opposed to problem, or computational) data type: "label". A statement label is actually a label constant. (Like an entry constant, a label constant is a kind of "named constant.") Label values originate with label constants and may be propagated by assignment to label variables. Label variables are variables declared with the LABEL data type attribute. This information is essentially repeated in the next paragraph in the form used in Lessons 1 and 2 to introduce various computational data types.

New label values are "generated" by:

(a) Reference to a label constant.

They may be propagated by assignment.

They may be used in the following ways:

(a) In GO TO statements.

(b) In comparison operations.

(c) In remote format items (Lesson 7).

The appearance of a statement label constitutes an explicit declaration of the name as a label constant, with scope rules that should be familiar by now. Consider three examples:

```
P: PROC;
   .
   .
   BEGIN;
      .
      .
      GO TO L1;
      .
      .
   END;
   .
   .
L1:  ...
   .
   .
END;
```

Here, the scope of the name L1 is all of P, including the begin block (assuming L1 is not redeclared therein). The GO TO transfers control outside the begin block to the statement labeled L1 (what happens in detail is described later).

```
P: PROC;
   .
   .
L2:  ...
   .
   .
   BEGIN;
      .
      .
      GO TO L2;
      .
      .
   L2:  ...
      .
      .
   END;
   .
   .
   GO TO L2;
   .
   .
END;
```

Here, there are two different label constants denoting different statements. The scope of the first is all of P except the begin block. The scope of the second is the begin block. The first GO TO is within the scope of the second and transfers to the statement labeled by it. The second is within the scope of the first and transfers control to the statement labeled by <u>it</u>.

```
P: PROC;
   .
   .
   BEGIN;
      .
      .
   L3:  ...
      .
      .
   END;
   .
   .
   GO TO L3;
   .
   .
END;
```

The scope of the label constant L3 is the begin block. The GO TO statement is not within that scope, so the name L3 is unknown there. The program is in error.

It goes without saying that everything that has been said about variables in general applies to label variables, too. They have

alignment, scope, storage class; you can have arrays of label variables; they may be base elements of structures; they can be initialized. In comparison operations, only = and ¬= are allowed for label data. (This is true of all types of program control data, i.e., algebraic comparisons are not defined for them.) The control variable of a controlled DO group may be a label variable, but the TO and BY clauses may not be used (because no algebraic comparisons are defined). An example where this is useful is:

```
DCL L LABEL;   Declares a label variable.
DO L = L1,L2,L3;  These are label constants.
   GO TO L;   Goes to either L1 or L2 or L3.
 L1:  ...
      .
      .
      GO TO COMMON;
 L2:  ...
      .
      .
      GO TO COMMON;
 L3:  ...
      .
      .
 COMMON:  ...
      .
      .    This code executed all three times.
END;
```

Label values may be arguments, and obviously parameters can be label variables. Procedures can return values of type "label", so that what follows GO TO may be a function reference.

Care must be exercised to ensure that a label variable, when used in a GO TO statement, does not designate a statement in an <u>inactive block</u>. It is illegal to transfer control into an inactive block. For example, the GO TO statement in the following, if executed, would be illegal:

```
DCL L LABEL;
BEGIN;
   .
   .
L1:  ...
   .
   .
   L = L1;
   .
   .
END;  The begin block becomes inactive here.
.
.
GO TO L; The value of L, i.e., the statement labeled by L1, is in
         an active block.
```

Actually, the semantics of label values are a little more subtle than they appear. They are composed of <u>two parts</u>: one is the statement labeled (represented by its address), and the other is an indication of the <u>activation</u> (or invocation) of the block containing the statement labeled. Consider the following.

```
DCL L LABEL STATIC;
DO I = 1 TO 2;
   BEGIN;
      .
      .
      IF I = 1 THEN L = L1;
      ELSE GO TO L;
      .
      .
   L1:  ...
      .
      .
   END;
END;
```

The begin block is invoked twice. The <u>first</u> time through, the label constant L1 is assigned to the label variable L. The value of the label variable L now represents the statement labeled by L1 <u>and</u> the <u>first</u> invocation of the begin block. The <u>second</u> time through a new value is not assigned to L. Its former value is used in the GO TO statement. Because that designates a statement in an inactive block, it is illegal. This may not seem intuitively necessary, but hopefully the reason why will become clear shortly. (We will later recall this example as "Example Z".)

Consider the following:

```
P: PROC;
   .
   .
   BEGIN;
      .
      .
      GO TO L;
      .
      .
   END;
   .
   .
L:  ...
   .
   .
END;
```

The label constant L in the GO TO statement represents a label value designating the statement labeled by L together with the current invocation of P. When the GO TO is executed, two things actually happen. All block invocations from the current one (the begin block) up to, but not including, the one contained in the label value (i.e., the current invocation of P) are terminated. There is no possibility of re-entering the terminated blocks without re-invoking them. Note

that if the begin block had instead been a procedure invoked from a function reference, control does not go back to the expression containing the function reference (as it would on a normal return); evaluation of that expression is discontinued, and control is transferred to the labeled statement instead.

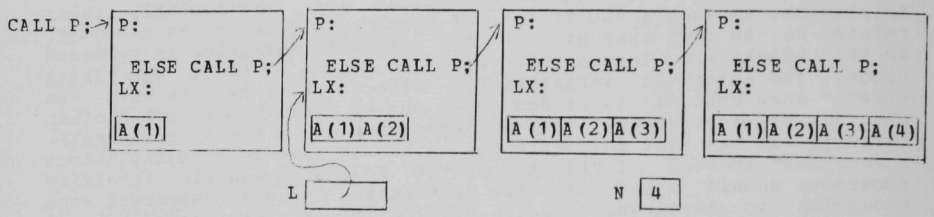
The significance of block invocations and particularly their termination by such a "GO TO out of block" (referred to as a "GOOB") relates to the fact that storage for automatic variables is released as the blocks are terminated. When we arrive back at the target block, the automatic variables "in effect" will be the ones "in effect" when control first descended out of that block into another one (as by a procedure call or execution of a BEGIN statement). Actually, in the last few sentences we should have been saying "block invocation" instead of "block" as the following example involving recursion should point out (the distinction is only apparent when recursion is involved, i.e., when a block can have several simultaneously active invocations).

```
P: PROC RECURSIVE;
   DCL N STATIC INIT (0);
   DCL A (N+1) FLOAT AUTO;
   DCL L LABEL STATIC;
   N = N + 1;
   IF N = 2 THEN L = LX;
   IF N = 4 THEN GO TO L;
   ELSE CALL P;
LX: I = HBOUND(A,1);
   Y = A(I);
   RETURN;
END;
```

Let's trace through what happens. Initially P is invoked from outside. On entry, N has initial value 0. An automatic array A with one element is allocated. N is increased to 1. Since N does not equal 2, LX is <u>not</u> assigned to L. Since N does not equal 4, we skip the GO TO. P is then called recursively.

As P is entered the second time, N (which, significantly, is a static variable) has the value 1. A new generation of A is allocated with upper bound 2. Throughout this second invocation of P, it is <u>this</u> generation of A which is addressed when A is referred to. Next, N is increased to 2. As a result, LX is assigned to the static label variable L. The value assigned represents the statement labeled LX <u>and</u> this current (i.e., second) invocation of P. Since N does not equal 4, the GO TO is again skipped and P is called recursively.

We go through yet another invocation of P, eventually (the fourth), whereupon when we arrive at the statement IF N = 4... things look like the following. Each large box represents an invocation of P. Boxes inside these represent generations of automatic variables belonging to the respective invocations. The small boxes at the bottom represent the static variables.




The statement GO TO L, which is executed next, causes the third and fourth invocations of P to be discarded, since the <u>environment</u> part of the value of L indicates the second invocation of P. Control is transferred to LX. The current environment is now that of the second invocation of P, no longer that of the fourth, so I is assigned the value 2 and Y is assigned the value of A(2). The RETURN statement returns control to the point of the second invocation of P, i.e., the CALL statement in the first. The next statement executed there is the one labeled by LX (as a result of normal statement sequence, not because of any GO TO). I is assigned the value 1 and Y is assigned the value of A(1). Control then returns to the outside, original, point of call of P.

Though the above example is contrived and not realistic, it does illustrate the meaning of the environment part of a label value.

Now recall "Example Z". The reason execution of the GO TO statement there is illegal is because it would require us to retrieve, or make current, an environment containing possibly some automatic variables that have long since been released. When they're gone, they're gone!

As you read LRM 105, you will see that static label variables cannot be initialized. This is because static variables are initialized at compile time, while label values, because they carry an indication of an environment, don't exist until run time. As you read that and LRM 106, you will see that statement labels can be subscripted with constants, as in

```
L(4,7): Y=0;
```

In the current implementation, this does not represent a subscripted label constant; it denotes an alternate form of initialization of an element of an array of label variables (in this case, the (4,7) element of the array L). The element being initialized may of course have its value changed subsequently by assignment, so that in this example L(4,7) may denote a different statement later! The ANSI language treats L as an array of label constants, which is different.

See LRM 108.

## 6.8. Entry variables.

We have seen in Lesson 4 how the ENTRY attribute can be used in a declaration of an external entry constant, and, in Lesson 5, how parameter descriptions and returned value descriptions can also be given in such a declaration. We will now consider entry values in general, and entry variables. "Entry" is a legitimate data type, like "label".

New entry values are "generated" by:

(a) Reference to an entry constant.

They are propagated by assignment.

They may be used in the following ways:

(a) In a CALL statement or function reference, to denote the procedure to be invoked.

(b) In comparison operations.

An entry variable is declared by adding the <u>VARIABLE attribute</u> to the types of entry declarations already demonstrated (without this attribute the declaration is that of an entry constant). Example:

```
DCL E ENTRY (CHAR(*)) RETURNS (BIT(1))
          VARIABLE EXT;
```

E is an entry <u>variable</u> whose name has external scope. Any entry value which it may have must designate a procedure that accepts a fixed-length character argument of any length and returns a one-bit bit string. Entry variables may have any of the properties (storage class, etc.) attributed to variables in general.

Entry values, like label values, consist of two parts: an entry point (represented by its address), and an environment. The environment is an indication of the activation (invocation) of the block containing the entry constant whose reference gave rise to the entry value; this applies, of course, only to internal entry

constants, since external entry constants have no containing block, i.e., no environment.

Consider the following:

```
P: PROC(J);
   DCL A(J) AUTO;
   .
   .
   Q: PROC;
      .
      .
      Y = A(I);
      .
      .
   END;
   .
   .
   CALL Q;
   .
   .
END;
```

No recursion is involved. When the internal procedure Q is invoked, the entry constant Q is referenced. That entry constant Q inherits the environment of its containing block, P. Thus, a reference inside Q to A(I) is a reference to an element of the automatic array A belonging to the one invocation of P in question (which is obviously the one that was "current" when Q was referenced in the CALL statement).

Observe in the following example the role of the environment of an entry value when recursion is involved.

```
P:  PROC (J) RECURSIVE;
    DCL A (J) AUTO;
    DCL N FIXED BIN STATIC INIT (0);
    DCL E ENTRY VARIABLE STATIC;
    N = N + 1;
    IF N = 2 THEN E = Q;
    IF N = 4 THEN CALL E;
    ELSE CALL Q;
    IF N < 5 THEN CALL P(N);
    RETURN;
    Q:  PROC;
        .
        .
        Y = A(I);
        .
        .
    END;
END;
```

Notice that P calls itself recursively until five invocations of it are active. Then N will equal 5 and the chain of calls will start

returning. Each invocation of P has a generation of an automatic array with a different bound. In the second invocation of P (N=2), the static entry variable E is assigned the value of the internal entry constant Q. The environment which is part of this value is that of the second invocation of P. In all five invocations of P, Q is called; it references an element of A and returns to the point of call. In all invocations of P except the fourth, Q is called by referring directly to the entry constant Q, and the environment of Q used in the reference to A inside Q is thus the current invocation of Q's containing block, P. However, in the fourth invocation of P, Q is called by referencing the entry variable E. Because the environment part of the entry value denotes the second invocation of P, the reference to A inside Q is a reference to the generation of A allocated at the time of the second invocation of P.

See LRM 107.

There are somewhat messy rules for determining when (except in obvious cases) a reference to an entry constant or an entry variable denotes the procedure itself and when it denotes the value returned by invoking the procedure. See LRM 109 and LRM 110. The ANSI standard uses different, but much simpler, rules for this determination.

For a complete review of the ENTRY attribute, see LRM 111.

## 6.9. Program termination.

A program ends by executing a RETURN statement in the main procedure or by reaching the END statement of the main procedure. It may also end by executing a STOP statement in any procedure. The latter mechanism is considered to be an abnormal termination of the program; in our system it causes a step condition code, which may be tested in JCL, of 1000 to be set. Information going beyond the above is in two places: LRM 112 and LRM 113.

## 6.10. Exceptional conditions.

In several of the earlier lessons we left for later consideration an examination of what happens when an exceptional condition occurs. An exceptional condition is a possible, though not usually likely

(in the sense of being frequent), unusual outcome of some operation or requested action. PL/I does not require the programmer to test constantly for unusual outcomes of operations. It provides you a way of being informed, in the program, when one occurs in such a way that you are not bothered when it doesn't. See LRM 114.

## 6.11. "Occurrence" of a condition.

PL/I defines and names a whole set of possible conditions, i.e., unusual outcomes of operations. It also defines what constitutes an occurrence of each condition. The list of conditions is given in LRM 115, and individual conditions are described in LRM 116. Certain of the conditions will be saved for later. A brief definition of what constitutes an occurrence of those considered here follows.

Computational conditions

FIXEDOVERFLOW (abbrev. FOFL). This occurs when a fixed-point operation produces a result that cannot be expressed in the maximum number of digits of the implementation. For example, note that the precision rule for addition (Lesson 1) of two FIXED BINARY (31,0) values would specify FIXED BINARY (32,0) for the result, were it not for the implementation maximum number of digits of 31, for binary base. The substitution of 31 for 32 is a hint that FOFL can occur on addition of two FIXED BINARY (31,0) numbers; indeed, it will occur when $2^{30}$ is added to $2^{30}$ (for example). The result, $2^{31}$, requires a non-zero digit in the 32nd position from the right end. Observe that FOFL cannot occur on the addition of two FIXED BINARY (15,0) values because the result precision, (16,0), is well within the implementation maximum precision.

OVERFLOW (abbrev. OFL). This occurs when a floating-point operation produces a result with a magnitude in excess of what the hardware can represent.

UNDERFLOW (abbrev. UFL). UFL occurs when a floating-point operation produces a result with a magnitude too small for the hardware to represent.

ZERODIVIDE (abbrev. ZDIV). This occurs on an attempt to divide by zero.

SIZE. This occurs when an attempt is made to assign a value to a fixed-point target variable that does not have enough high-order digit positions to accommodate non-zero high-order digits of the value being assigned.

CONVERSION (abbrev. CONV). CONV is raised if a character string value, which is the source value in a conversion operation, contains an illegal character. CONV also occurs on assignment to a character pictured variable (Lesson 2) if the source value does not conform to the picture specification, and it may occur on certain kinds of input operations (Lesson 7).

Program checkout conditions

SUBSCRIPTRANGE (abbrev. SUBRG). This occurs when a reference is made to an element of an array outside the bounds of any of its dimensions.

STRINGRANGE (abbrev. STRG). This occurs whenever a reference to the SUBSTR builtin function or pseudo-variable describes a substring which does not lie entirely within the bounds of the string value which is its first argument. See Lesson 2.

STRINGSIZE (abbrev. STRZ). This occurs whenever a string value having a length in excess of the length (or maximum length) of a string variable is about to be assigned to that variable.

System action conditions

FINISH. This condition occurs as the result of any action that would terminate the program. Examples are: execution of STOP statement; execution of RETURN or END statement of main procedure. Others will follow.

ERROR. ERROR occurs in many circumstances. One category of circumstances is detection of an illegal argument to a mathematical builtin function (e.g., the real value -1 to SQRT). Another is any error that an implementation may care to detect for which no specific condition is provided. Others will follow.

6.12. Enablement/disablement of conditions.

Not all occurrences of conditions need be detected and reported. For certain conditions, the programmer may choose to ignore an occurrence. In such a case it is important to note that the condition has occurred (because that may have consequences on the meaning of the program's execution as defined by PL/I) even if the programmer elects not to be notified.

Occurrences of certain conditions are detected by the hardware; others, by compiled code.

Whether the occurrence of a condition is detected or not depends on whether the condition is enabled or disabled at the point in the program where it occurs. This property of a condition is called its status.

Certain conditions are enabled by default. Others are disabled by default. A programmer may specify a particular status for a

condition to hold during the execution of a statement or of a whole block, thus overriding the default. There are a few conditions whose default status may not be overridden.

An explicit status may be specified by a <u>condition prefix</u>. Examples follow.

| | |
|---|---|
| (SIZE): I = 3*J; | SIZE is enabled during the execution of this statement. |
| (NOSIZE): B = C; | SIZE is disabled for this one. |
| (OFL,NOUFL): X = Y*Z; | OFL is enabled, UFL disabled. |
| (OFL): (NOUFL): X = Y*Z; | Same as above. |
| (OFL): L: Y = 2**X; | This statement has a label, too. (It must follow any condition prefixes.) |

When a condition prefix is attached to a BEGIN or PROCEDURE statement, it applies to all statements in the block except those to which a complementary condition prefix is attached. It applies to (i.e., is inherited by) any nested blocks.

Note that status of a condition is a static property of a statement that can be determined (like scope of a declaration) by the compiler. The status of a condition in an external procedure Q called by procedure P, for example, has nothing to do with its status in P.

The following table indicates the default status for the conditions considered so far, and whether they can be disabled.

| Condition | Default status | Can it be disabled? |
|---|---|---|
| FOFL | Enabled | Yes |
| OFL | Enabled | Yes |
| UFL | Enabled | Yes |
| ZDIV | Enabled | Yes |
| SIZE | Disabled | |
| CONV | Enabled | Yes |
| SUBRG | Disabled | |
| STRG | Disabled | |
| STRZ | Disabled | |
| FINISH | Enabled | No |
| ERROR | Enabled | No |

See LRM 117 through LRM 120.

## 6.13. Establishment of conditions.

What happens when a condition occurs depends first of all on whether it is enabled or disabled.

When any of the above conditions occurs while disabled, the result of the operation that caused the condition to occur is undefined, with two exceptions. The exceptions are as follows. When UFL is disabled, the result of an operation that causes it to occur is taken to be zero. When STRZ is disabled, the source string is truncated on the right to make it fit the target variable, as we saw in Lesson 2.

When we say that the result is undefined, we mean that the language does not define a result. The result is entirely determined by the implementation; it may be useless (garbage) or useful, but it is not guaranteed to be the same in another implementation. Note that simple, useful random number generators are frequently designed around the occurrence of a disabled FOFL condition.

When a condition occurs while enabled, the condition is said to be raised. The programmer can specify an action to be taken when a condition is raised or he can rely on system default actions (called standard system action).

The programmer specifies an action to be taken when a condition is raised by establishing an on unit for the condition. This is accomplished by executing an ON statement prior to the raising of the condition.

An ON statement has the typical form

```
     ON condition on-unit;
```

*condition* is the keyword naming the condition. *on-unit* is either a single statement or a begin block. Examples:

```
     ON FOFL GO TO L;
     ON UFL N = N + 1;
     ON SIZE BEGIN;
        S = 'OOPS';
        GO TO DONE;
     END;
```

Once an on unit has been established for a condition, in a block, i.e., once an ON statement for that condition has been executed in the block, subsequent raising of the condition in that block, or any block invoked from it in which another on unit for the same condition has not been established, causes the on unit to be executed. "Subsequent" is in the sense of later in time.

Another way of describing which on unit gets control when a condition is raised is as follows. If an on unit for the condition has been established in the current block, it is executed. If none has been established there, the block that invoked the current block is examined for an established on unit. The search for an on unit proceeds in this way all the way out to the main procedure.

Suppose that a procedure P has an established on unit; P calls Q; and Q establishes an on unit for the same condition. The on unit established by P is "stacked". If the condition occurs subsequently in Q, the on unit established in Q is executed. Once Q returns to P, however, the on unit in Q is no longer in effect. If the condition subsequently occurs in P, P's established on unit gets control.

If another ON statement is executed in the same block in which an on unit (for the same condition) is already in effect, the on unit specified in the new ON statement supplants that specified earlier, i.e., <u>it</u> becomes the established on unit in the block. That is, the new on unit is not stacked.

The on unit itself may be thought of as a parameterless internal procedure. When a condition is raised, the current operation is identified as the <u>point of interrupt</u>, and it is just as if the internal procedure represented by the on unit were <u>invoked</u>, the point of invocation being the point of interrupt. The on unit may or may not reach its normal end. If it does, control returns to the point of interrupt and the program (usually) continues from there. This is called <u>normal return</u> of the on unit. The other choice is to execute a GO TO out of block, transferring control from the on unit to some labeled statement outside the on unit. As in all GOOB's, there is no possibility of going back to the point of invocation of the block (i.e., the point of interrupt).

The view of on units as internal procedures "invoked" from the point of interrupt is completed by noting that the environment part of the entry value representing such a procedure is that denoting the invocation of the block containing the ON statement when it was

executed. Thus, references to automatic variables of the block containing the ON statement, from within the on unit, are references to the generations corresponding to the invocation of the containing block which executed the ON statement.

In some cases the interrupted operation does not continue from the point of interrupt on normal return from the on unit. The exceptions are as follows:

STRINGRANGE: The SUBSTR reference is amended to yield a valid substring, then the program continues.

CONVERSION: It is assumed the on unit has made an attempt to correct the condition using facilities described in Lesson 10. If the attempt has been made, the conversion operation is retried (this could raise CONV again if the attempt was not successful). If no attempt has been made, ERROR is raised.

SUBSCRIPTRANGE: ERROR is raised.

ERROR: FINISH is raised. Note that when ERROR is raised, there is no way the program can be made to continue from the point of interrupt.

FINISH: The program terminates.

Of the remaining cases, only two (UNDERFLOW and STRINGSIZE) continue from the point of interrupt with a defined result. The other four (FIXEDOVERFLOW, OVERFLOW, ZERODIVIDE, and SIZE) continue with an undefined result.

## 6.14. Standard system action.

When the search for an established on unit doesn't turn up any, standard system action is taken. Standard system action is as follows:

STRG: Issue a message, then continue with amended SUBSTR reference as described for normal return from a STRG on unit.

STRZ and UFL: Issue a message and continue with the defined result.

CONV, FOFL, OFL, SIZE, SUBRG, and ZDIV: Issue a message and raise ERROR.

ERROR: Issue a message and raise FINISH.

FINISH: Terminate the program.

Suppose you are writing an external procedure as part of a program which is a team effort. How do you arrange for standard system action to be taken (if that is what you want) when a condition is raised in your procedure, not knowing whether some other block above yours in the chain of active blocks has established an on unit for it? You may establish a "system action on unit" by executing an ON

statement with the keyword SYSTEM in place of an on unit. Example:
ON FOFL SYSTEM;

## 6.15. The REVERT statement.

Another problem you may have in designing an external procedure as part of a team effort is the following. You may have established an on unit in order to intercede when a condition is raised in a certain part of your procedure. Having passed the point at which you are no longer interested in interceding, how do you "cancel" the established on unit so that subsequent action, if the condition should occur later in your procedure, will be governed entirely by any on units that may be established in the blocks above yours on the chain of active blocks? By executing a REVERT statement for the condition. Example:

```
REVERT ZERODIVIDE;
```

The effect of this is to cancel, or nullify, any ZDIV on unit previously established in the current block. There will then be no ZDIV on unit established in the current block, i.e., the situation is the same as it was just after the block was entered and before any ON ZDIV...; statement was executed.

It is legal to revert a condition which hasn't been established in the current block. This has no effect. See LRM 121.

## 6.16. The SIGNAL statement.

You can cause a simulated occurrence of a condition (useful in testing) by executing a SIGNAL statement naming the condition. After normal return from an on unit entered as a result of raising a condition in this way, execution continues with the next statement (this is true even for the FINISH condition). The one exception is ERROR; upon normal return from an ERROR on unit raised by signaling ERROR, FINISH is raised as usual. See LRM 122.

## 6.17. Programmer-named conditions.

You can define and name your own conditions. A programmer-named condition is an identifier; its use (demonstrated below) constitutes an explicit declaration of the name as a condition name having internal scope. The name may also be given external scope (so that

the same name in different external procedures denotes the same programmer-named condition, as opposed to different programmer-named conditions that happen to have the same name) by declaring it with the CONDITION attribute and EXTERNAL (see LRM 123). Note that there are condition names, but not condition constants, values, or variables.

The only way to raise a programmer-named condition is to signal it.

A programmer-named condition, *name*, is used in the following way in ON, SIGNAL, and REVERT statements

```
ON CONDITION (name)...;
SIGNAL CONDITION (name);
REVERT CONDITION (name);
```

i.e., the programmer-named condition masquerades as the CONDITION condition. We can then talk about enablement status, standard system action, etc., for programmer-named conditions by describing these properties for the CONDITION condition. Specifically, the CONDITION condition is enabled by default and cannot be disabled. Standard system action is to issue a message and continue.

## 6.18. Review

See LRM 124, skipping anything we haven't covered yet, and LRM 125.

In the ANSI standard there are a few highly technical differences in some actions on normal return from on units and in some standard system actions. In addition, attempt to continue with an undefined result is in violation of the standard. A brief handout is available from the instructor for those who are interested.

## 6.19. Effect of optimization on conditions.

If you had the job of hand-optimizing a program, you would discover ways to common expressions, move invariant expressions out of loops, etc. The final program, hopefully, will produce the same answer as the original one, at least when you do not rely on the raising of conditions and the entering of on units to implement your logic. Clearly, moving expressions around might change the order and number of interrupts and thus condition raisings. The same is true when you request the compiler to optimize your program.

A more subtle problem occurs with certain kinds of optimizations. The compiler might find it advantageous to keep a variable in a register inside a loop. Even if you assign to that variable in the loop, the compiler might not generate code to store the contents of the register into the assigned storage location for the variable (it would do so only at the conclusion of the loop, if the value of the variable is needed subsequently). Thus, if an on unit is entered as the result of a condition raised in the loop, and the on unit references such a register-held variable, it would not retrieve the current value of the variable.

Two options, which may appear on a BEGIN or PROCEDURE statement, can be used to tell the compiler whether your program can be safely optimized in the way described above. The options affect the code generated in the block, and are inherited by contained blocks on which they are not respecified. ORDER (which is the default if neither is stated) says that the compiler is not allowed to perform the optimizations described above because the order in which variables are assigned and referenced <u>must</u> be observed, even across on unit boundaries. REORDER essentially says that such on units will not be executed, or, if they are, they won't reference variables that may not have had their most recent value stored. This permits greater optimization. See LRM 126 through LRM 128.

ORDER and REORDER are not available in the ANSI standard. The standard essentially permits implementations to behave as the current one does under REORDER, i.e., it always permits maximum optimization. At the same time it places restrictions on which variables can be referenced in on units. These restrictions are necessary to guarantee the same behavior of the program in all standard implementations, even though the extents to which they carry out certain optimizations may differ.

Do not let all of the above scare you! You will probably discover that you will have very little need for on units for computational conditions in most realistic programs.

The amount of optimization attempted by the compiler is also governed by the OPTIMIZE compiler option. See OPG 4 and OTUG 2. A complete discussion of efficiency considerations, with regard to all areas of the language, is in LRM 129.

## 6.20. Unanswered questions.

In an ERROR on unit, how can one obtain information about what caused ERROR to be raised?

In any on unit, how can one determine whether the condition occurred naturally or by being signaled?

These questions will be answered in Lesson 10. Other conditions will be considered in appropriate lessons.

6.21. Homework problems.

(#6A) When is

```
IF expression THEN true-part;
ELSE false-part;
```

not the same as

```
IF ¬ (expression)  THEN false-part;
ELSE true-part;  ?
```

Hint: Explain what may happen when, for instance, *expression* is a BIT(10) variable.

(#6B) Suppose you have an array of 100 elements (bounds 1 to 100) that is to be filled with unique values in the order in which they are presented. A variable records the index (i.e., subscript value) of the last position filled. Write a procedure to accept a value, as an argument, and insert it in the next position in the array if it is not already in the array. The procedure is to be invoked by a CALL statement. Concern yourself with the following:

(a) After the array has been completely filled, another procedure will probably retrieve its entries. Make sure both procedures have access to the necessary variables. Be careful with initial values.

(b) Unless special precautions are taken, your program will be in error if more than 100 unique values are presented to the procedure. What will happen if no special precautions are taken? Discuss several ways of detecting the situation and of preventing errors. Also discuss several methods of informing the calling procedure about the occurrence of the situation, and discuss their implications on its design. Hint: consider the following methods:

(i) A returned value to indicate success or failure.

(ii) An additional parameter through which success or failure is conveyed out.

(iii) Use of appropriate PL/I conditions.
(iv) Use of a programmer-named condition.

(#6C) Simulate by hand the execution of the following code to determine the value assigned to I. If you survive the tedium and get the answer 2501, you understand entry variables, label variables, and their behavior in recursive environments.

```
I = P(1);
P:  PROC (X) RETURNS (FIXED BIN) RECURSIVE;
    DCL X FIXED BIN;
    DCL A FIXED BIN AUTO;
    DCL (N,S) FIXED BIN STATIC INIT (0);
    DCL L LABEL STATIC;
    DCL Q ENTRY (FIXED BIN)
          RETURNS (FIXED BIN) VARIABLE STATIC;
    A,S = X+S;
    N = N + 1;
    IF N = 2 THEN L = LX;
    IF N = 4 THEN Q = QX;
    IF N = 5 THEN S = S + Q(A);
    ELSE S = S + QX(A);
    IF N = 6 THEN GO TO L;
    S = P(A);
LX: RETURN (A + S);
    QX: PROC (Y) RETURNS (FIXED BIN);
        DCL Y FIXED BIN;
        RETURN (Y + A);
    END;
END;
```

(#6D) Precisely what happens when a FIXED DECIMAL (8,0) variable with value 12345678 is added to a FIXED DECIMAL (8,8) variable with value <u>zero</u> in our implementation?

(#6E) Can FIXEDOVERFLOW occur during a division of two fixed-point variables? Can you explain your answer?

(#6F) What is the difference, for all practical purposes, between

```
ON UFL SYSTEM;
```

and

```
ON UFL ;
```

?

(#6G) (Difficult) Suppose you have a program which you have developed to satisfy the ANSI standard and which you will be

shipping to other installations that have different machines and different ANSI standard compilers. Though you may have enabled the SIZE condition during testing, why is it generally not necessary or useful to leave it enabled in the export version once you are satisfied that SIZE cannot occur? Why is it, however, desireable to leave OFL, UFL, and FOFL enabled, even though you are satisfied that they are not occurring?

(#6H) What action is taken if ZDIV is raised in each of the eight places marked "*" in the following program?

```
P:  PROC OPTIONS (MAIN);
    *
    ON ZDIV X = 1;
    *
    BEGIN;
       *
     ON ZDIV X = 2;
       *
     ON ZDIV X = 3;
       *
     ON ZDIV SYSTEM;
       *
     REVERT ZDIV;
       *
    END;
    *
END;
```

(#6I) What can happen in the following program segment?

```
ON ERROR BEGIN;
   T = SQRT(Y);
   GO TO RESUME;
END;
X = some value, possibly negative;
Y = some value, also possibly negative;
T = SQRT(X);
RESUME:  ...
```

How does this improve when the program is changed, as follows?

```
ON ERROR BEGIN;
   ON ERROR SYSTEM;
   T = SQRT(Y);
   GO TO RESUME;
END;
etc.
```

(#6J) What PL/I facilities serve the function of the FORTRAN "computed GO TO"? The "assigned GO TO"?

(#6K) Occasionally, one wants to take some action when a condition, such as ERROR, occurs, then let the next higher level block that has an established on unit for the condition take its action, and so on. A technique frequently tried is

```
ON ERROR BEGIN;
   take some action
   REVERT ERROR;
   SIGNAL ERROR;
END;
```

Why does this not achieve the desired result, and what does it really do? How can the desired result be achieved?

# 7. Introduction to I/O; stream I/O.

## 7.1. Datasets vs. files.

In PL/I, I/O is performed by doing certain things to abstract objects called "files." Files can be associated with datasets so that the operations on files have useful effects on the associated datasets. Several different files can be simultaneously associated with the same dataset. A particular file can be associated with different datasets at different times. See LRM 130.

## 7.2. File constants, values, and variables.

A file value is an object referred to above as a file. We are now embarking on a discussion of our third program-control data type: file.

New file values are "generated" by:
(a) Reference to a file constant.
They are propagated by assignment.
They may be used in the following ways:
(a) In I/O statements.
(b) In ON, REVERT, and SIGNAL statements dealing with certain conditions pertinent to I/O.
(c) In comparison operations.

Recall that DECLARE statements can be used to declare names as entry constants or entry variables, and that entry constants were also capable of being contextually declared by their appearance as a label prefix on a PROCEDURE or ENTRY statement. Similarly, DECLARE statements can be used to declare names as file constants or file variables, and file constants may be contextually declared by their appearance in I/O statements or I/O condition names. The data type attribute, not surprisingly, is FILE. File constants, like entry and label constants, are "named constants." Examples:

```
DCL F FILE;  F is a file constant. The default scope is external.
DCL G FILE INTERNAL;  G is an internal file constant.
DCL H FILE VARIABLE EXT;  H is an external file variable.
```

## 7.3. File description attributes.

There is a very large set of attributes that describe certain properties of files. These file description attributes (FDA's), as they are called, may be declared for file constants but not file variables. If a file constant is assigned to a file variable, any FDA's declared for the file constant are inherited by the file variable in the sense that they are properties of the current file value assigned to it. If, later, a different file constant is assigned to the file variable, the file variable will reflect possibly different properties represented by the FDA's which were declared for this second file constant. More on this later.

## 7.4. Opening a file.

In order to do I/O on a dataset, it is first necessary to associate the dataset with a file. One way of accomplishing this is by executing an OPEN statement. (This is called explicit opening.) The typical form is

OPEN FILE (*file*) TITLE (*ddname*);

Here, *file* is a file constant, or a file variable, or a function reference returning a file value; in any case it denotes a file value originally obtained by reference to some file constant. Note that it is as much an error to reference, in an OPEN statement, a file variable which has not been assigned a value as it is to reference any variable that has not been assigned a value. *ddname* is a character-string valued expression whose value (truncated to 8 characters, if necessary) is taken to be the "ddname" of the dataset. The actual dataset denoted is the one associated with that "ddname" in the JCL.

The TITLE option may be omitted from the OPEN statement, in which case the ddname used is the first 8 characters of the identifier naming the file constant from whose reference the value of *file* was derived. Examples:

```
OPEN FILE (X) TITLE ('ABC');
     The ddname is ABC.
DCL DEF FILE;
OPEN FILE (DEF);
     The ddname is DEF.
DCL U FILE VARIABLE;
U = DEF;
OPEN FILE (U);
     The ddname is DEF.
```

If a file is already "open," an attempt to explicitly open it again is treated as a "no-op." E.g.,

```
DCL H FILE VARIABLE;
DCL FF FILE;
H = FF;
OPEN FILE (FF);
OPEN FILE (H) TITLE ('XYZ');
```

The last OPEN statement has no effect, since the file denoted equally well by the file constant FF or the file variable H is already "open."

Several files can be opened in one OPEN statement. Example:

```
OPEN FILE (F1),
     FILE (F2) TITLE ('HUH'),
     FILE (F3);
```

The second way a dataset can be associated with a file is by implicit opening. Implicit opening occurs when a file which is not open is referenced in an I/O transmission statement. The ddname of the dataset to be associated with the file is derived in exactly the same way as for explicit opening when the TITLE option is omitted.

## 7.5. The UNDEFINEDFILE condition.

If an attempt to open a file fails, the UNDEFINEDFILE condition (abbreviation: UNDF) occurs for that file. An on unit for UNDF may be established for that file by executing an ON statement as in

```
ON UNDF (file) on-unit;
```

Because the UNDF condition is a qualified condition (like the CONDITION condition), separate UNDF on units may be established for each file in a program.

An attempt to open a file may fail for several reasons, including: no DD statement in the JCL for the ddname used; conflicting DCB attributes; etc. In Lesson 10 we will see how one may tell why an attempted opening was unsuccessful.

The UNDF condition, like the ERROR condition, is enabled by default and cannot be disabled. Standard system action, which applies when the condition is raised and no on unit has been established, is to issue a message and raise ERROR. If, on the other hand, an on unit is entered and the on unit returns normally, subsequent action depends on whether the attempted opening was explicit or implicit. In the former case, execution continues from the point of interrupt. In the latter case, execution continues if the file was (somehow) successfully opened in the on unit, e.g., by trying a different ddname); otherwise, the ERROR condition is raised.

See the description of UNDF in LRM 116.

## 7.6. Closing a file.

The association between a dataset and a file is broken by executing a CLOSE statement for the file:

```
CLOSE FILE (file);
```

Several files can be closed simultaneously:

```
CLOSE FILE (A),
      FILE (B),
      FILE (C);
```

Closing an already closed file, like opening an already opened file, has no effect.

Files left open when a program terminates are closed by a PL/I termination routine. Any output data left in a buffer is transmitted to the dataset before the file is closed. After a file has been closed, either the same dataset or a different dataset may be associated with it by subsequently executing another OPEN statement for the file. See LRM 131.

## 7.7. Overview of transmission statements.

The I/O statements that cause data transmission that we will examine in this lesson are GET (input) and PUT (output). In the next two lessons we will study READ (input), and three output statements: WRITE, REWRITE and DELETE. In Lesson 11 we will add LOCATE (output). In Lesson 9, and again in Lesson 14, we will look at the UNLOCK statement.

## 7.8. Overview of file description attributes.

As stated earlier, FDA's may be used in a declaration of a file constant. It is not necessary, however, to declare any FDA's for a file constant, even though a set of properties for the file must have been provided by the time it is opened.

We will be looking at the many different FDA's gradually. Suffice it to say that some are alternatives to others; i.e., a conflict arises if two mutually exclusive alternatives are provided.

If the file properties described by FDA's are not complete when a file is opened, additional properties are supplied during the opening process. This proceeds as follows.

If the opening is explicit, additional FDA's may be written as options on the OPEN statement. These must not conflict with any declared for the file in a DECLARE statement. Examples:

```
OPEN FILE (F) INPUT;
OPEN FILE (G) OUTPUT TITLE ('SYSPUNCH'),
     FILE (H) INPUT TITLE ('SYSIN');
```

If the opening is implicit, additional FDA's are deduced from the statement causing the opening. For example, INPUT will be deduced from GET and OUTPUT from PUT.

If the "merging" of FDA's that occurs during explicit or implicit openings produces any conflicts, the UNDEFINEDFILE condition is raised. If the merging still leaves the set of file properties incomplete, others may be supplied by implication (i.e., those that a file has may imply others that it must also have) and finally by default.

When a file is closed, any FDA's supplied during the opening process are divorced from the file. It continues to have only those with which it was declared (which may be none). If the file is again opened, it may acquire a different complete set of properties.

File properties are used, among other things, to determine which operations may legally be carried out for a file. For instance, it is illegal to WRITE to an INPUT file. An attempt to do so will raise the ERROR condition.

The different FDA's are briefly described, and the defaults listed, in LRM 132. Other, detailed, descriptions are scattered throughout LRM 133. The opening and closing of files may be reviewed at LRM 134; that reference also shows the FDA's deduced on implicit opening and those that may be implied. The OPEN statement is further detailed at LRM 135. Finally, the whole subject of datasets vs. files is also treated in OPG 5 and CPG 4, with emphasis on device and dataset characteristics.

## 7.9. Stream vs. record I/O.

Two alternative FDA's which describe properties of all files are STREAM and RECORD.

The dataset associated with a stream file is viewed as a continuous stream of characters, rather than as a sequence of records. Its processing is inherently sequential. Stream output, which is accomplished with the PUT statement, consists of the issuing of a stream of characters to be written to the dataset. Stream input, which is accomplished with the GET statement, consists of the acceptance of a stream of characters read from the dataset. Although all datasets are actually organized as records, stream transmission may be oblivious to record boundaries; it may, however, also be made cognizant of them.

The dataset associated with a record file on the other hand, is viewed as a sequence or set of discrete records. Each transmission statement transmits exactly one record. The data in a record need not be in character form; it can be in any of the forms capable of being represented internally in PL/I.

For the remainder of this lesson, we will be concerned with stream I/O only. Hence, we assume that the STREAM FDA applies to any file we are talking about. The STREAM attribute may be acquired:

(a) By declaration of the file constant with STREAM.
(b) By specification of the STREAM option on an OPEN statement.
(c) By deduction on implicit opening of a file by a GET or PUT statement.
(d) By implication from the PRINT attribute on an explicit opening.
(e) By default on explicit opening.

See LRM 136.

7.10. File description attributes applicable to stream files.

The other FDA's applicable to stream files are INPUT, OUTPUT, PRINT, and ENVIRONMENT (abbreviation: ENV).

INPUT and OUTPUT are two alternatives that any file (whether stream or record) may have. A third alternative, applicable only to record files, will be given in Lesson 8. The meaning of INPUT and OUTPUT should be obvious. Only GET statements may be used for stream input files, and only PUT statements for stream output files. See LRM 137.

PRINT is an additive attribute that may be specified only for stream output files. It says that the output dataset is ultimately to be printed. See LRM 138.

The ENV attribute is much like the OPTIONS option (Lesson 4) in that it encloses a list of implementation-defined options. It is important to note that the contents and meaning of environment options is not specified by the language, but by each implementation. The basic function of environment options is to provide the implementation with extra information it may require, such as the physical organization of records in a dataset. See LRM 139.

ENV is the only FDA that may not appear on an OPEN statement (except in the ANSI version). We will have very little to say about the individual environment options, although they are important, so you should read LRM 140, OPG 6 and CPG 5. The ENV attribute will be in conflict with other FDA's if it contains options in conflict with other FDA's. See LRM 141 for a table of conflicts.

The PRINT attribute, being additive, is never deduced, implied, or defaulted. It must be specified (either in a DECLARE statement or OPEN statement).

If an implicit opening occurs and neither INPUT nor OUTPUT was declared for the file, GET implies INPUT and PUT implies OUTPUT. If explicit opening occurs without specifying either, the default used is INPUT.

7.11. Further OPEN statement options for stream files.

The LINESIZE option can be used on an OPEN statement for any stream output file to establish a record length for the dataset. (This information can also be conveyed in the ENVIRONMENT attribute or in JCL; and there is a standard default value if none of these sources supplies the information.)

The PAGESIZE option can be used on an OPEN statement for any print file (i.e., stream output file which has the PRINT attribute). It can be used to establish the maximum number of lines to appear on each page when it is printed.

7.12. Overview of stream transmission statements.

The PUT statement specifies one or more expressions of computational data type whose values are to be converted to character representations which are then inserted in the output dataset. Generally, successive characters go into successive positions of the current output line (record). When an output line is filled, characters continue on the next line. Successive PUT statements do not automatically start new lines; the characters transmitted continue where the last PUT statement left off, which may be in the middle of a line. Facilities are also provided for starting a new line or, in the case of a print file, a new page.

The GET statement specifies one or more variables of computational data type to be assigned values from an input dataset. The values are assumed to be represented in character form on the dataset and are converted to the appropriate internal form. This process consumes a number of characters from the dataset starting at the place where the last GET statement left off (which may be in the middle of a line). If a line is exhausted, remaining characters come from the next line. Successive GET statements do not automatically start new lines. Facilities are provided, however, for skipping to the start of the next line.

7.13. Data lists.

The part of a PUT statement that specifies the expressions whose values are to be disposed of, and the part of a GET statement that specifies the variables whose values are to be acquired, is called a data list ("I/O list" in FORTRAN). It is surrounded by parentheses. The list is a list of data list items separated by commas. A data list item is one of the following:

(a) An expression. This may be just a constant or variable.
(b) A repetitive specification. This is a parenthesized list of data list items ending with what looks like a controlled DO statement without the semicolon.

Examples of data lists, including their surrounding parentheses, follow.

```
(X)
(X,Y)
(A+B, 'THIS', 'THAT' || V, 1)
(U, (V(I), W(I) DO I = 1 TO N))
```

Notice the syntax of the repetitive specification in the above example. If N has the value 3, say, the effect of the data list is the same as would be obtained by the following one.

```
(U, V(1), W(1), V(2), W(2), V(3), W(3))
(((A(I,J) DO I = 1 TO N) DO J = 1 TO M))
((A(I), (B(I,J) DO J = 1 BY 2 TO 5), C(I) DO I = 1, N))
```

The above is equivalent to:

```
(A(1), B(1,1), B(1,3), B(1,5), C(1),
 A(N), B(N,1), B(N,3), B(N,5), C(N))
```

If a data list item is a structure, it is equivalent to a sequence of scalar items, namely, those which are (in order) the base elements of the structure. If a data list item is an array, it is equivalent to a sequence of scalar items, namely, all the array elements in the order having the rightmost subscript varying most rapidly. Thus, the item

```
A(*,*)
```

is equivalent to the item

```
((A(I,J) DO J = LBOUND(A,2) TO HBOUND(A,2))
         DO I = LBOUND(A,1) TO HBOUND(A,1))
```

The elementary data items in data lists in GET statements cannot be arbitrary expressions; they can only be variables (although they may, of course, be subscripted by expressions) because the context is one of assigning a value to them. See LRM 142.

## 7.14. Modes of stream transmission.

There are three modes of stream transmission: list-directed, data-directed, and edit-directed, as determined by the form of the GET or PUT statement. The different modes may be intermixed on the same file.

## 7.15. List-directed transmission.

In list-directed transmission, which is the simplest, the keyword LIST precedes the parenthesized data list. Together they constitute a LIST option. If the option immediately follows the keyword GET or PUT, the keyword LIST may be omitted. List-directed transmission provides simple, "free-form" stream I/O. Examples will be given later.

On input, character representations of values in the input stream must be separated by one or more blanks, or by a comma and any number of surrounding blanks. Each input stream item must be written as a valid computational constant, i.e., arithmetic constant, character string constant, or bit string constant. The "attributes" of the input stream item, deduced from the form in which it is written in the same way that attributes are deduced for a constant written in the program, need not match the attributes of the corresponding variable in the data list; conversion between the source and target attributes occurs as necessary. The CONVERSION condition can occur in this process (a homework problem will deal with this).

It is possible to omit values from a list-directed input stream. Consecutive commas, or commas separated only by blanks, indicate that no value is to be assigned to the variable in the input data list with which a value in that position would be matched; the variable thus retains its current value. Finally, a semicolon may be used in the input stream to indicate that all the remaining variables in the data list are to be skipped over. All these features are demonstrated in the following example.

```
DCL N FIXED BIN;
DCL X FLOAT BIN (21),
    A (3) CHAR (10) VAR,
    B (30) FIXED BIN (15);
GET FILE(F) LIST (X,A,N, (B(I) DO I = 1 TO N BY 2));
Input stream:
6.4  'VAL1'  , ,  ''
15
6  1E2,, 5.1;
```

The first input stream item, 6.4, is associated with X. The value, expressed as FIXED DECIMAL (2,1), is converted to FLOAT BIN (21) for assignment to X. The next input stream item is a character string constant and is associated with A(1); A(1) thus acquires the 4-character character string value VAL1. The next input stream item is missing, so A(2) retains its current value. The next one results in A(3) being assigned the value of the null character string. The next one results in N being assigned the value 15; during that assignment, the value is converted from FIXED DECIMAL (2,0) to FIXED BINARY of default precision. The repetitive specification appearing next in the data list would cause successive input stream items to be assigned to B(1), B(3), B(5), ... , B(15). The contents of the input stream result in the following assignments (only), however:

```
  6  to  B(1)
100  to  B(3)
  5  to  B(7).
```

On output, the values of the data list items, which may be arbitrary expressions, are converted to character form according to the conversion rules. Thus, the converted character form will reflect the attributes of the variables or expressions from whose values they were obtained. Note that the conversion rules for binary arithmetic data to character string call for an intermediate conversion to decimal, so that the value "three" of a FIXED BINARY variable, for instance, will be printed as 3 instead of 11B.

Placement of the character representations of the values in the output file depends on whether that file is a print file or not. If it is not, they are separated by one blank. If it is, successive values are aligned on predefined "tab" columns. (The tab columns can be changed as described in OPG 7 and CPG 6. In the ANSI language, a TAB option is provided on the OPEN statement, which will simplify the specification of user-defined tab positions for print files.)

Also, for non-print files the values of character string variables or expressions in the data list are surrounded by quotes in the external representation. (If the data being written out with list-directed output were to be read back in later with list-directed input, these quotes will be needed to identify the input stream item as a character string constant.) For print files they are not surrounded by quotes (remember what the PRINT attribute says: the file is to be printed, i.e., not read back in).

See LRM 143 and LRM 144.

## 7.16. Data-directed transmission.

Data-directed transmission also permits simple, free-form stream transmission. The essential difference from list-directed transmission is that values on the external medium are accompanied by the names of the variables in the program from which they were obtained or to which they are to be assigned. Because of this, the elementary data list items in a data-directed PUT statement must be variables (possibly subscripted by expressions); they cannot be arbitrary expressions. The keyword DATA precedes the parenthesized data list, forming the DATA option.

On input, since each item in the input stream has its name associated with it (the form being essentially that of a scalar assignment statement without a semicolon, and written with constant subscripts and full structure qualification), the items in the input stream need not appear in the same order as the items in the data list. In fact, the order of items in the data list is totally irrelevant. Not all of the variables appearing in the data list need appear in the input stream, but names appearing in the input stream must appear in the

data list. Transmission for a single data-directed GET statement is stopped only when a semicolon is encountered in the input stream. A data-directed input data list item may not be subscripted or a repetitive specification; when array elements are to be received from the input stream, it is sufficient to have the whole array as a data list item.

Example (using the variables declared in the previous example for list-directed input):

```
GET FILE (F) DATA (B, A, X, N);
Input stream causing the same assignments as in the
previous example:
X=6.4 A(1)='VAL1' A(3)='' N=15
B(1)=6 B(3)=1E2 B(7)=5.1;
```

Note that items in the input stream are separated by a comma and/or one or more blanks.

On output, repetitive specifications, subscripted variables, etc., are allowed. The values are accompanied by their variable names with subscript expressions evaluated to a constant value. Items are separated as in list-directed output. A semicolon is written following the last item.

In a data-directed transmission statement, the data list following the keyword DATA may be entirely omitted. This is equivalent to specifying a data list containing all variables known at that point in the program which are legal in a data-directed data list.

See LRM 145 and LRM 146.

## 7.17. Edit-directed transmission.

Edit-directed transmission gives the programmer full control over the format of data on the external medium. Edit-directed transmission statements include not only data lists but format lists as well. During their execution, the two lists are matched so that the value being written out (or read in) is assembled (or decoded, respectively) according to the format item in the format list. Values on the external medium are not self-delimiting with blanks or commas as in list-directed or data-directed transmission; the format item for a particular value specifies the number of characters to be used on the external medium as well as the format of the contents of that field.

In edit-directed GET or PUT statements the parenthesized data list is preceded by the keyword EDIT. The format list is also parenthesized and immediately follows the data list (i.e., no keyword is used). All of this constitutes the EDIT option.

7.18. Format lists.

A format list is a list of format items separated by commas. Each format item is one of the following:

(a) A data format item, control format item, or remote format item (described below).

(b) One of those preceded by either an unsigned decimal integer constant or a parenthesized expression, representing an iteration factor.

(c) A parenthesized format list preceded by an iteration factor.

An iteration factor effectively replicates the elementary format item or list of items that follows it.

Data format items describe the format of a field on the external medium corresponding to an item from the data list. Control format items do not correspond to items in the data list and thus do not describe the format of a value; they indicate control actions such as skipping to a new line or page, as well as others. Remote format items are described later.

Matching of items between data lists and format lists proceeds as follows. The process is "driven" by the data list. Once the next scalar item is obtained from the data list (remember that a structure item is equivalent to a list of its scalar base items, in order, and an array item is equivalent to a list of its subscripted elements in row-major order), control advances in the format list until a data format item is encountered, and it is that data format item which is paired with the scalar data list item. Any actions specified by control format items encountered while looking for the next data format item are taken. An iteration factor is evaluated when it is encountered and causes repetition of the following item or list the indicated number of times (which may be zero). When the data list is exhausted, any remaining format items (even if the next one is a control format item) are ignored. However, if the format list is exhausted first, it is rescanned from the beginning (note: from the beginning of the whole list).

It should be remarked that pairs of data lists and their corresponding format lists may be repeated in an edit-directed transmission statement. When one data list is exhausted, the second is begun; the second format list is used for subsequent matching, even if the first one was not exhausted. If a format list is exhausted before its corresponding data list, that format list is rescanned from the beginning.

See LRM 147 and LRM 148.

7.19. Data format items.

Detailed descriptions of the six data format items would take many pages and will not be attempted here. The flavor of three of them will be given. More information is in LRM 149 and LRM 150.

F format item. On output, the value is converted to FIXED DECIMAL (the data list item may have attributes of any computational data type). The format item specifies a total field width, an optional number of fractional positions (taken as 0 if not specified), and an optional scale factor. Examples:

F(5) might produce ƀƀ123, ƀƀƀƀ0, or -1003.

F(6,3) might produce ƀ1.000, -3.012, or 10.640.

On input, the contents of the field width specified must be a decimal integer constant, positioned anywhere in the field. If a decimal point is used, it overrides the fractional-part field width in the format item; if it isn't, it is assumed to appear in the position specified by the format item.

E format item. On output, the value is converted to the form of a decimal floating-point constant having the specified total field width and number of fractional digits. On input, the field must contain a valid decimal floating-point or fixed-point constant.

A format item. On output, the value is converted to character and disposed of in the field width specified. The field width may be omitted, in which case the field width is the length of the character value. On input, the field width specified (it cannot be omitted) is assumed to contain a character string value (all characters are legal).

The remaining data format items are B (bit), C (complex), and P (picture).

Field widths, etc., may be given by the values of expressions; they need not be constants.

Note that there is no correspondence of data types required for data items and their matching format items. Conversions are performed as necessary. E.g., suppose a data item were a CHAR (50) VAR variable, and suppose the format item were F(5). On output, the character string value will be converted to fixed decimal, which may cause the CONVERSION condition to occur. On input, the 5-character field must contain a decimal fixed-point constant. If it doesn't, the CONVERSION condition will occur. If it does, its value will be converted to CHAR (8) for assignment to the target variable.

7.20. Control format items.

X format item. X($n$) causes the next $n$ positions to be filled with blanks, on output, or skipped, on input.

SKIP format item. SKIP($n$) causes the current line to be terminated and the next $n$-1 lines to be skipped. SKIP is equivalent to SKIP(1). SKIP(0) is allowed only for print files; it suppresses spacing and causes the next line to be overprinted on the current one. This is useful for underscoring.

COLUMN format item. Abbreviation is COL.
COL($n$) causes the cursor to be repositioned forward to the given position in the line. Intervening positions are filled with blanks on output and are skipped on input. If the current line is already past the designated column, SKIP(1) is assumed; i.e., the next line is positioned to the designated column.

PAGE format item. Used for print files only. Succeeding output will continue on the next page.

LINE format item. For print files only. Succeeding output will continue on the designated line. If the current page is already past that line, a new page is begun.

See LRM 151.

7.21. Remote format item.

The remote format item has the form R(*label*) where *label* is a label-valued expression. When one is encountered, the FORMAT statement whose statement label is the value of *label* is scanned. A FORMAT statement merely contains a format list; it can be used to provide several different edit-directed transmission statements with the same format list. Example:

```
GET FILE (IN) EDIT (N,X) (R(LAB));
PUT FILE (OUT) EDIT (N+2,X-1) (R(LAB));
LAB:  FORMAT (F(8),X(1), E(15,5));
```

A FORMAT statement is not executable in the normal sense. In the ANSI standard, the label on a FORMAT statement is of a new data type, "format", and there are format variables and a FORMAT attribute. I.e., there is a clear distinction between format values and label values, and they serve different functions. The current language is a little cloudy in this area. See LRM 152.

7.22. Other stream transmission statement options.

Any stream transmission statement may contain a SKIP option. The syntax and meaning are the same as for the SKIP format item. The skipping takes place before the data list is processed, i.e., first.

A PUT statement for a print file may contain a PAGE option or LINE option, or both. The syntax and meaning are as for the same format items, and the action is taken before data transmission.

A statement with one of the above options may omit the LIST, DATA, or EDIT option. For example, PUT FILE (SYSPRINT) PAGE; causes a new page to be positioned on the file SYSPRINT without data transmission.

The COPY option in a GET statement says that the input stream read is to be copied, exactly as read, to the stream output file specified in the COPY option.

The FILE option, which designates the stream input or output file, may be replaced by a STRING option. In a GET statement, the STRING option provides a character string expression which serves as the source of input stream data instead of a file. In a PUT statement, it specifies a character string variable that serves as a sink of output stream data instead of a file. The STRING option extends the facilities of stream I/O to operations on strings (for instance, formatting) performed in core as string manipulations (see LRM 153).

Now review LRM 154 through LRM 157. Certain options of PUT statements intended for debugging and implemented only by the Checkout compiler (and which are not part of the ANSI standard) are described in Lesson 13. More review: LRM 159.

7.23. Standard files.

The language recognizes SYSPRINT as a standard print file and SYSIN as a standard stream input file. A GET or PUT statement not containing either a FILE option or a STRING option is equivalent to one containing FILE (SYSIN) or FILE (SYSPRINT). Thus:

| | |
|---|---|
| GET (A,B,C); | is an easy way to get input. |
| GET DATA; | allows any variables known to be "assigned" a value from SYSIN. |
| PUT DATA; | is an easy way to print all known computational variables and their values on SYSPRINT. |
| PUT (A,B,C); | is a carefree way to provide output. |
| PUT SKIP; | conditions SYSPRINT to start receiving future output on a new line. |

See LRM 158, OPG 8, and CPG 7.

7.24. Conditions applicable to stream I/O.

The UNDEFINEDFILE condition, which is applicable to all I/O, has already been mentioned; so has CONVERSION, which can occur during stream input or output (as well as the situations mentioned in Lesson 6). Another condition from Lesson 6, the SIZE condition, occurs in edit-directed output if the field width specified in an E or F format item is not large enough to contain non-zero high-order significant digits or a leading minus sign.

Four new conditions are applicable. The TRANSMIT condition (which is a qualified condition, like UNDEFINEDFILE, i.e., it is qualified by a file value) occurs if a real, live I/O error occurs on any input or output statement. Its default status is enabled and it cannot be disabled. Standard system action is to issue a message and raise ERROR. If normal return from a TRANSMIT on unit occurs, execution continues from the point of interrupt, but the effect of the I/O operation that raised TRANSMIT is unpredictable.

The ENDFILE condition (also qualified) occurs on any input operation when no more data is available. In the case of a GET statement, it occurs if the physical end of file is reached before data transmission or between two data transmissions associated with data format items. If the physical end of file is encountered during the processing of a data format item or X format item, ERROR is raised instead. The default status of ENDFILE is enabled; it cannot be disabled. Standard system action is to issue a message and raise ERROR; thus, even if you don't like on units, you pretty much need an ENDFILE on unit. On normal return from an ENDFILE on unit, execution continues with the statement following the input statement.

The ENDPAGE condition (also qualified) occurs when an attempt is made to transmit data to a line on a page of a print file having a line number in excess of the value of PAGESIZE (as specified in an OPEN statement or defaulted). Status is as for the above conditions. Standard system action is to start a new page; this useful action occurs without any specific request! Note, however, that if an ENDPAGE on unit is entered, any further output that it does to the same file will continue to appear on the same page, on lines with even higher line numbers. This is useful for printing page footings. After printing a footing, if it desires, the on unit could execute PUT FILE(...) PAGE; to skip to the next page. It may execute, then, further PUT statements to print a page heading (column headings, etc.). When normal return from the on unit is finally made, execution continues from the point of interrupt in the PUT statement that raised the condition. Note that

execution of a LINE or SKIP format item or statement option can cause ENDPAGE to be raised; on normal return, the action specified by LINE or SKIP is ignored.

The final condition, NAME (also qualified), occurs on data-directed input if a name in the input stream does not appear in the data list or, if no data list is provided, is not known in the current block. It also occurs in various cases of ill-formed input. Default status is as for the above. Standard system action is to ignore the incorrect input stream item, issue a message, and continue. On normal return, the GET statement continues with the next input stream item.

See LRM 116 for further details on the above.

7.25. Stream I/O to a terminal.

The Optimizing and Checkout compilers modify certain aspects of stream I/O when a file is associated with a terminal instead of a dataset, the goal being better human engineering.

Normally, successive PUT statements merely place successive values into a line buffer; data transmission does not actually occur unless a line is completed. In TSO, each PUT statement transmits its data to a terminal immediately so that you may see all output generated logically before you are required to supply input. Nevertheless, successive PUT statements without intervening GET statements continue to write in the same line.

SKIP(0) is implemented by backspacing! This is only useful on an IBM 2741 terminal.

When a GET statement is executed, the carriage is returned and you are prompted with a colon and another carriage return! However, if the last PUT statement directed to the terminal transmitted a colon as the last character, that is taken to be a prompt issued by the program and the prompting action described above is not taken. End-of-line is taken as a delimiter between items, unlike the usual behavior, so that you may type one item per line without blanks. If a data list isn't exhausted at end-of-line, you are prompted for more with a plus sign followed by a colon. Finally, if end-of-line is encountered inside a data format item, i.e., before the whole field width is exhausted, sufficient trailing blanks are assumed to match the field width.

These features and others are described in CTUG 2 and 3 and OTUG 3 and 4.

7.26. Comparison to FORTRAN.

Edit-directed I/O corresponds to FORTRAN "formatted I/O' (but not the "direct access" kind). The format list may accompany the GET statement or it may be remote (which is more like FORTRAN). Each transmission statement does not automatically start a new line, as in FORTRAN. For a print file, you do not provide a carriage control character as the first character of data for each line; that is taken care of automatically by PAGE, SKIP, or LINE options or format items and if data just overflows a line. (For a non-print file, however, SKIP merely causes the output line to be finished. The system does not provide carriage control characters, and if you intend to print a dataset created via a non-print file, and you tell ASP via the RECFM DCB operand that the dataset has carriage control characters, you will have to generate them in the output data. Use PRINT for datasets to be printed! If not declared, SYSPRINT is a PRINT file.)

Items in an edit-directed output data list can be expressions, while in FORTRAN formatted I/O they cannot be.

The repetitive data list item is like FORTRAN's "implied DO," but a little more general.

Formatted I/O in FORTRAN is driven by the format list, while edit-directed I/O in PL/I is driven by the data list.

If the format list is exhausted before the data list, it is rescanned from the beginning in PL/I, even if it contains a nested (and iterated) format list (what is called a "group format specification" in FORTRAN).

There is nothing to correspond to FORTRAN's H format item or literal format item; data can only come from the data list.

A given data format item can be matched with any type of data item, while a specific correspondence of types is required in FORTRAN.

Leading or trailing blanks in F or E-format input fields are ignored rather than treated as zeroes. Embedded blanks will cause the CONVERSION condition to occur.

There is no equivalent to FORTRAN's format arrays and object-time formats. However, much of the flexibility that it provides is available in the fact that iteration factors and field widths can be expressions whose values are obtained by reference to input variables.

List-directed I/O is roughly equivalent to the same feature of FORTRAN, though the contents of list-directed input data streams are different.

Data-directed I/O is roughly equivalent to "NAMELIST I/O" in FORTRAN, though the details are different.

7.27. Unanswered questions.

The question "How do we correct a conversion condition?" first asked in Lesson 6 is relevant here, too. It is answered in Lesson 10.

We will learn in Lesson 10 not only how we can tell what raised ERROR, but if we find it was caused by standard system action for one of the I/O conditions that can do that when no on unit is established, how we can determine what file is involved.

We will also see how we can tell what garbage caused the NAME condition to occur, and which of the many possibilities was the cause of an UNDEFINEDFILE condition.

7.28. Homework problems.

(#7A) Suppose F is declared as FILE. What file description attributes will it have if it is opened implicitly by a PUT statement? Suppose the PUT statement says

```
PUT FILE (F) LINE (10) LIST ('BEGINNING');
```

Why is this illegal?

(#7B) A 613 ABEND occurs if you try to open a particular SYSOUT <u>dataset</u> when it is already open. The error messages that are written out when the ERROR condition is raised are written on <u>file</u> SYSPRINT (one of the standard files). If the file is not already open, it is opened by this action. Suppose the DD statement for SYSPRINT says SYSOUT=A (as it does in our cataloged procedures). Recall that an explicit OPEN for an already opened file is ignored in PL/I. Suppose an error message has already been produced on SYSPRINT. What happens if your program subsequently executes

```
OPEN FILE (F) TITLE ('SYSPRINT') OUTPUT;
```

(The same can happen if the error message is produced <u>after</u> F is opened.)

(#7C) What happens if an attempted opening of file SYSPRINT raises UNDEFINEDFILE and there is no UNDF on unit for SYSPRINT? An intelligent recovery from this situation will be described in Lesson 12.

(#7D) Write some code that will associate file F with the dataset identified by the DD statement with ddname DD01. If the OPEN fails, assume no DD statement was provided and try DD02. Continue on to DD99 until you succeed on one or fail on all. (Though not the main point of this problem, you should see how a numeric pictured variable can be useful in generating those ddnames.)

(#7E) Show some code which initializes the elements of an array of file variables to different file constants. Assume the program will access all the files "in parallel" instead of one after another, so that they must all be open simultaneously. Open them in a DO group. Be prepared to write a message for each one whose open fails, giving its index. Establish the on units in the same DO group. If you were to establish the on units in a separate DO group, executed before the one that opens the files, would you have to do anything different to make sure the proper index is printed out when a particular file can't be opened?

(#7F) Is it possible to write a utility program in PL/I which is capable of manipulating any number of datasets simultaneously? Assume the processing required is methodical enough to access the files through an array (allocated dynamically with adjustable extents once the program determines how many files it will have to deal with).

(#7G) If the input stream

```
3  5  10B  35J  WORD  'AGAIN'  '16S'  '5.1'
```

were read by the statement

```
GET (N, S, M, X, T, U, V, J);
```

with the variables declared as follows:

```
N  FIXED BIN
S  CHAR (20) VAR
M  FIXED DEC
X  FLOAT DEC
T  CHAR (20)
U  CHAR (3)
V  FIXED DEC
J  FIXED DEC
```

for which input stream items would the CONVERSION condition occur? (Assume it is corrected whenever it occurs, so that the whole list is processed.) In which cases is the raising of the condition dependent on the attributes of the variable in the data list, and in which cases not?

(#7H) If you have FORTRAN experience, compare the PL/I and FORTRAN format items.

(#7I) Write a portion of a program that reads two input values which are taken to be the row and column dimensions of an array, allocates an array of that size (use an automatic variable), then reads in values for the array elements under format control. Demonstrate several alternatives:

(a) A DO group containing a GET statement that reads a single value into the next element of the array.

(b) A single GET statement (for the array elements) that uses a repetitive specification.

(c) As in (b), but without a repetitive specification. This is only possible if the values are presented in row-major order.

(#7J) There is no REWIND statement in PL/I. How would you accomplish that function?

(#7K) There is no BACKSPACE statement. Suppose you had to read a line of input twice, under different format controls. How would you do that?

(#7L) Suppose you want to print the elements of an array using F(8,3) format. What happens if an element has the value 10000? Or -1000? Suppose you want the field to be filled with eight asterisks when the value won't fit, as in FORTRAN. Show a way of doing this which involves the statement

```
PUT FILE (OUT) EDIT ((TEST(A(I)) DO I = 1 TO 100))
                    (100 A(8));
```

TEST is a function procedure. Your job is to show what is in TEST.

(#7M) S is declared as a varying length character string variable. If either of the following statements is legal, what does it mean? If not, why not?

```
PUT EDIT (S) (A);
GET EDIT (S) (A);
```

(#7N) What is the difference between the following three statements, in terms of their effects?
X is an array.

```
PUT EDIT (X) (E(20,8), SKIP);
PUT EDIT (X) (SKIP, E(20,8))
PUT EDIT (X) (E(20,8)) SKIP;
```

What would be the difference if X were a scalar variable and the PUT statement were executed inside a loop?

(#7O) Read a list of values into an array using list-directed input. You do not know in advance how many values you will get. The input file contains only the array values. Be prepared to receive up to 1000 values, and set a variable to indicate how many were read into the array. Print a message if more than 1000 values are received.

(#7P) In the above problem you may be left with an incomplete array, i.e., one some of whose elements are unused. What could you do subsequently to take advantage of array assignments, array expressions, etc., which operate on all the elements of an array?

(#7Q) Suppose you are using an ENDPAGE on unit to print a page footing at the bottom of every page of output on a print file. How do you get the footing printed at the bottom of the last page, which may be incomplete?

(#7R) Demonstrate the use of an ENDPAGE on unit for the production of page headings.

# 8. Introduction to record I/O; consecutive datasets.

## 8.1. Record I/O.

In record I/O, transmission of data occurs in units of discrete records, which correspond to logical records in a dataset. Each record transmission statement transmits exactly one record into or out of a variable, called a record variable. (Some record transmission statements don't cause any data transmission and don't use record variables, but they still do something to a record in a dataset.)

Transmission consists of the mass transfer of so many contiguous bytes of storage between the record on the external medium and the record variable in core. It should be obvious that the record variable must represent connected storage (Lesson 3). Beyond that, record variables may be just about anything -- scalars, arrays, or structures. The data in the record is a byte-for-byte image of the data in core, regardless of the data type. Transmission occurs without conversion of any kind. Record variables can contain program control data, but values read from such records may not be valid, particularly if the reading occurs in a different execution of the program from its writing.

Record files are used for various purposes. Because data transmission takes place without conversion between internal machine form and external character form, record files (and the datasets associated with them) are particularly appropriate for intermediate storage, i.e., data created by the program for reading back in later (may be much later). By the same token, record files (and their datasets) are not suited for human consumption except in the special case that the record variables are character string variables.

A record file is one which has the RECORD file description attribute (FDA) instead of STREAM. See LRM 136 and LRM 160.

## 8.2. Records and keys.

All datasets are composed of a set of records. Even datasets associated with stream files are composed of a sequence of records, but that is not always of much consequence.

The records of some types of datasets are accompanied by identification fields called recorded keys. A recorded key contains a character string value, called a key, which identifies the record with which it is associated. A recorded key may be physically separate from the record or embedded within it. A dataset containing keyed records is called a keyed dataset.

When a program wishes to designate a particular record in a dataset, it does so by computing and presenting a key value. Key values in the program are called source keys. Their meaning is defined by the implementation. Usually they correspond to the values in recorded keys, but this is not always necessary.

## 8.3. Language vs. implementation: history of record I/O.

The language features for record I/O seem, more than any other parts of the language, to have been strongly influenced by the kinds of datasets and processing techniques available in IBM's OS operating system. Perhaps what happened went something like this: The language designers used the capabilities of IBM hardware and the OS operating system to set goals and target capabilities. From a huge array of possibilities they distilled out some common features and called this language. The common features constituted a minimal set of capabilities but could be combined in diverse ways to provide many variations in behavior. Some of these variations corresponded to the hardware capabilities they had in mind while others didn't.

Other manufacturers have selected combinations that corresponded to their hardware and operating system capabilities. Certain combinations of language features that IBM has found a use for may not have a use in another system. Even within the IBM implementations, the introduction of the VS operating system has led to the assigning of meaning to certain combinations of features that were previously meaningless.

Our study of record I/O will thus proceed as follows. We will look first at some of the individual language features and what they mean. We will then turn our attention to one of the kinds of datasets IBM supports and will discuss the kinds of processing that may be done with it and the language features used to accomplish it. In Lesson 9 we will study additional language features and apply them to other kinds of IBM datasets and processing techniques.

## 8.4. Types of access.

PL/I provides two kinds of access to datasets associated with record files, sequential and direct. Direct access, denoted by the DIRECT FDA, means that records will be accessed in an arbitrary sequence. Each record to be processed must be identified by a source key. Sequential access, denoted by the SEQUENTIAL (abbreviation: SEQL) FDA, means that records will be accessed in some kind of sequential order. What is meant by sequential order is up to the implementation: it may be the physical order of records in a dataset or the order defined by ascending or descending key values. (Though we haven't said so explicitly, keyed records may have a logical order defined by their keys which is not identical to their

physical order in the dataset.) An implementation may provide a choice between physical sequence and key sequence. When key sequence is being used, the program may use source keys.

All record files will have either the SEQL or the DIRECT attribute. See LRM 161. As for other FDA's, these may be specified explicitly or they may be acquired at open time by deduction, implication, or default.

8.5. The KEYED attribute.

Certain options of transmission statements provide for the communication of key values. For certain types of operations their use is mandatory; in other cases they are not used. If they are to be used, the file must have the KEYED FDA. Such a file is called a keyed file. It is not required that keyed files and keyed datasets be associated only with each other; they may also be associated with their non-keyed alternatives. We will see how a keyed file can be used with a non-keyed dataset, and how a non-keyed file can be used with a keyed dataset, when we look in Lesson 9 at certain types of processing provided for in IBM implementations.

Keyed files may be accessed sequentially or by direct access. Sequential files may be accessed using keys or not, hence a sequential file may be keyed or non-keyed. The arbitrary order in which direct files are accessed requires the use of keys, hence direct files may not be non-keyed. Thus, the DIRECT attribute implies the KEYED attribute. See LRM 162. The three valid combinations are SEQL, KEYED SEQL, and KEYED DIRECT.

8.6. Record transmission statements.

In this lesson we will study three record transmission statements.

The READ statement obtains an existing record from a file.

The REWRITE statement replaces an existing record with new, or updated, data.

The WRITE statement adds a new record to a file.

In Lesson 9 we will study the DELETE statement, which deletes an existing record from a file. In Lesson 11 we will study an alternative to the WRITE statement that can be used in certain cases.

## 8.7. Common options of record transmission statements.

All record transmission statements contain the FILE option which, as in stream I/O, designates the file.

The READ statement uses the INTO option to name the variable into which a record is to be read. A general requirement is that the amount of storage occupied by the variable, i.e., its size taking into account any array extents, etc., must be equal to the length of the record read. Note, however, that if you read into a scalar varying-length string the current length of the string variable is set by the reading of the record, as on assignment. If you intend to read into different variables having different sizes, then the dataset must have V or U format records (not F format records) -- independent of any blocking. See LRM 165. An alternative to the INTO option will be discussed in Lesson 11.

The REWRITE and WRITE statements use the FROM option to name the variable from which a record is to be written. The same requirements for matching the size of the record and record variable exist. That is, if different variables, having different sizes, are to be written, the dataset will have to have V or U format records. When a scalar varying-length string is written, the length of the record is determined by the string's current length. See LRM 166.

It is frequently useful to use a structure (Lesson 3) for a record variable. This serves to group related data items having potentially different attributes (the structure base elements) together as fields within one record.

## 8.8. Data movement direction attributes.

In Lesson 7 we saw the INPUT and OUTPUT FDA's. A third alternative, UPDATE, can be used with record files. It indicates that records may be both read from and written to the file. These attributes hold significance for the types of record transmission statements that can be used with files having those attributes as follows.

| | INPUT | OUTPUT | UPDATE |
|---|---|---|---|
| READ | √ | | √ |
| REWRITE | | | √ |
| DELETE | | | √ |
| WRITE | | √ | √ |

See LRM 137. At this point you should review the various different ways FDA's may be acquired. See LRM 132, LRM 134, and LRM 141.

## 8.9. Minor attributes and options.

The BUFFERED or UNBUFFERED (BUF or UNBUF) attributes may be used in certain cases. See their description at LRM 167. For the I/O we will be dealing with in Lessons 8 and 9 we need not be concerned with this; it is sufficient to let it default. These attributes are not in the ANSI standard. In Lesson 11 we will consider a certain type of sequential I/O that requires the BUFFERED attribute (in the current language). In Lesson 14 we will look at another type of I/O that requires the UNBUFFERED attribute, but that type of I/O isn't in the ANSI standard.

The BACKWARDS attribute may be used for sequential input record files associated with datasets on magnetic tape. It specifies that the file is to be read in the reverse sequential order of its records. This permits greater efficiency when making multiple passes over a tape dataset. This attribute is also not in the ANSI standard. See LRM 168.

The IGNORE option of the READ statement may be used instead of the INTO option when the statement addresses a sequential file (either input or update). IGNORE($n$) causes $n$ records to be skipped, i.e., read but not assigned to any record variable. See LRM 169.

## 8.10. ENVIRONMENT attribute for record files.

A vast number of different options can be specified in the ENV FDA for record files. Full details are given in LRM 163. Although this implementation-defined material is very important, and you should read it sometime, we will discuss here only certain essential ENV options applicable to our implementation.

## 8.11. Dataset organizations.

In Lesson 7 we pointed out that the function of the ENVIRONMENT FDA is to provide a system with implementation-dependent information it may need to relate your standard PL/I I/O statements to the facilities available in the system. Some of this information is optional; for instance, a good number of the ENV options are essentially JCL DCB parameters moved into the program itself. Others may be mandatory.

Earlier we said that an implementation has certain "native" I/O capabilities around which particular combinations of transmission statements and options are centered. In the IBM systems, you must designate one of the native types of record I/O processing using particular ENV options. The alternatives we will consider in this course are called consecutive, indexed, and regional dataset organizations. The ENV options, and their basic meaning, are as follows.

CONSECUTIVE. Consecutive datasets are non-keyed. Records are stored consecutively in the dataset. The "order" implied by sequential access is physical order.

INDEXED. Indexed datasets are keyed datasets. Records are stored, with their keys, in an order which is usually not material. The "order" implied by sequential access is logical order by increasing key value. This may not be the physical order of records in the dataset.

REGIONAL. Regional datasets come in three subvarieties, as we shall see in Lesson 9. One is non-keyed and two are keyed datasets. Sequential order is a peculiar mixture of both physical and logical.

To reiterate a point made earlier, combinations of FDA's, the transmission statements and their options have validity only with respect to particular dataset organizations. Combinations valid for some organizations may be invalid for others. Valid combinations are summarized at LRM 141.

Indexed and regional organizations will be the subject of the next lesson. The remainder of this one will be devoted to consecutive organization.

## 8.12. Consecutive datasets.

The CONSECUTIVE option of the ENVIRONMENT attribute is used to identify a dataset as being organized consecutively. If the ENV attribute is not declared for a file, or if it is but doesn't contain any of the dataset organization options, then consecutive organization is assumed. Though we did not say so in Lesson 7, consecutive organization applies also to datasets associated with stream files; it is the <u>only</u> organization applicable to them.

Consecutive datasets can only be accessed through sequential files. The KEYED attribute is not used because there is no use for the various statement options that have to do with keys since there are no keys in the dataset. Do not, however, confuse the meanings of consecutive and sequential. "Consecutive" is an IBM implementation concept to describe a particular type of organization and representation of records in a dataset in IBM systems. "Sequential" is a standardized PL/I concept to describe the fact that records will be accessed in some kind of order. Sequential access applies to datasets with other organizations as well, but consecutive organization demands sequential access.

The kinds of processing permitted on consecutive datasets are as follows. You may create one by writing its records sequentially; these are placed in the dataset physically in the order in which they are written. You may read the records of an existing one sequentially, obtaining them in their physical order (hence, in the order in which they were written). Or, you

may read the records sequentially, making changes to some and replacing them in the dataset (in the same place from which they came) before going on to read further records. See LRM 170.

## 8.13. Creating a consecutive dataset.

The applicable FDA's are SEQL OUTPUT. The WRITE statement with the FROM option is used. If the file hasn't been opened explicitly, execution of the first WRITE statement causes implicit opening with the attributes RECORD and OUTPUT being deduced; if an access attribute wasn't declared, SEQL is assumed by default. See LRM 132 and LRM 134. An example follows.

```
DCL F FILE, S CHAR (80);
OPEN FILE (F) SEQL OUTPUT;
DO WHILE (more to write);
   .
   .
   .
   WRITE FILE (F) FROM (S);
   .
   .
   .
END;
CLOSE FILE (F);
```

In the above example, the dataset written is identified in the JCL by the DD statement with ddname F (we didn't use the TITLE option on the OPEN statement). The DD statement might look like

```
//F DD DISP=(NEW,CATLG),DSN=whatever,
// UNIT=whatever,SPACE=whatever,
// DCB=(RECFM=f,LRECL=r,BLKSIZE=b)
```

The DCB parameters f, r, and b could be just about anything. If f is F (or FB or FBS), r <u>must</u> be 80 (since the program, as shown, writes records of length 80) and b must be 80 (or a multiple if f is FB or FBS). If f is V (or VB or VBS), r must be at least 84 and b must be equal to or greater than r. If f is U, b must be at least 80 (r is not used). You can intermix records of other lengths (written from other record variables) if and only if f is <u>not</u> F, FB, or FBS, and r and b are large enough.

Most DCB parameters can be provided via the ENV attribute in the program, which, if done, takes precedence to JCL. A useful feature is that the LRECL and/or BLKSIZE may be computed by the program and specified using variables, e.g.,

```
DCL F FILE ENV(FB RECSIZE(R) BLKSIZE(B));
DCL (R, B) FIXED BIN (31) STATIC;
R = some value;
B = 5*R;
OPEN FILE (F) SEQL OUTPUT;
BEGIN;
   DCL S CHAR (R) AUTO;
   etc.
```

Generally, we will not go into so much detail with JCL considerations. The programmer's guides have extensive sections on JCL with examples. For creation of a consecutive dataset, see CPG 8 and OPG 9.

It should be pointed out that when a DISP of OLD is used with an existing dataset, it is "recreated", i.e., totally overwritten. If DISP=MOD is used, the records written will be appended to the end of the dataset, after any existing records.

8.14. Retrieving a consecutive dataset.

A consecutive dataset may be read by opening a sequential input file and reading successive records with the READ statement using the INTO option. If at any point you can anticipate that you are not interested in the contents of the following n records, you may skip over them by substituting the IGNORE option for the INTO option.

8.15. Altering a consecutive dataset.

By opening a file associated with a consecutive dataset for sequential update (i.e., using FDA's SEQL and UPDATE) you can read the records sequentially and, for any that you choose, alter their contents and write them back out (in place). That is, you use the READ statement with INTO option and then, after altering the record by assignments to the record variable, you use the REWRITE statement with the FROM option. If you do not wish to alter a record, just skip the REWRITE statement. Note that processing is strictly sequential: you cannot rewrite the n-th record after reading the n+1st record, and you obviously cannot rewrite a record before reading it. See LRM 171. Since an existing record is being replaced, its length must not be changed.

Note that when a member of a partitioned dataset (as denoted entirely by the DSN JCL parameter) is <u>written</u> (or created), using SEQL OUTPUT, unused space at the end of the dataset is used; the newly written member will then replace any existing member with the same name. (The space occupied by the replaced member is not accessible and not available for re-use unless the dataset is "compressed" with a utility.) However, a member of a PDS may be <u>rewritten</u> in place, using SEQL UPDATE. It is possible to do an update in place because there is no way of changing the size of a record or of adding extra records (the WRITE statement is illegal for a sequential update file).

JCL details for accessing (reading or altering) consecutive datasets are given in CPG 9 and OPG 10. Complete examples appear in CPG 10 and OPG 11 (however, they use certain I/O facilities we won't see until Lesson 11).

8.16. The TOTAL option.

Normally, record I/O is accomplished by a call to a library routine. Under certain conditions, however, in-line code may be generated leading to substantial efficiencies. This is possible only for sequential output or input of consecutive datasets, and then only when certain other conditions are met. You have to specifically request in-line code because the compiler cannot always detect, for a given READ or WRITE statement, whether or not the conditions will be met when it is executed. By using the ENV option TOTAL you are promising to meet the conditions. See LRM 172 and LRM 173.

8.17. Conditions applicable to record I/O (consecutive datasets).

UNDEFINEDFILE, being applicable to all I/O, is applicable here. Likewise TRANSMIT. ENDFILE is applicable to sequential input or update files (as well as stream input files) and is raised when a READ statement attempts to read a record beyond the last one in the dataset. Note that normal return from an on unit entered from a READ statement results in the next statement being executed without anything having been read into the record variable.

One new condition, the RECORD condition (a qualified condition) is applicable. This occurs whenever the size of a record does not match the size of the record variable. The condition cannot occur in most situations with varying-length scalar string record variables. The status is enabled, and it cannot be disabled. Standard system action is to issue a message and raise ERROR. Normal return from an on unit continues execution with the next statement; in this case, the effect on the record (on output) or the record variable (on input) is not defined by the language. What happens in our implementation is described in LRM 116.

8.18. Comparision to FORTRAN.

Sequential input and output to consecutive datasets is comparable to FORTRAN unformatted input and output. FORTRAN has no equivalent to sequential update.

8.19. Homework problems.

(#8A) Suppose you employ a WHILE-only DO group to read a sequential input file and process its records. How many different coding techniques for breaking the loop when ENDFILE occurs can you demonstrate? Which do you like best? Least?

(#8B) Suppose you have a "data base" for an experiment containing data on the number of occurrences of various responses to different stimuli. Suppose records are organized in "groups" with each group consisting of two kinds of records.

(a) One header record containing a stimulus type, a stimulus subtype, and a count, n, of the number of possible response types (n may be zero). The record is described by the record variable declared below.

```
DCL 1 HEADER,
      2 STIMULUS_TYPE CHAR (20),
      2 STIMULUS_SUBTYPE CHAR (10),
      2 #RESPONSES FIXED BIN (15);
```

(b) Following the header record, a detail record for each of the possible responses (there may be none of these) containing a response type and a count of occurrences (historical data). Each detail record is described by the record variable declared below.

```
DCL 1 DETAIL,
      2 RESPONSE_TYPE CHAR (50),
      2 #OCCURRENCES FIXED BIN (15);
```

Groups, and detail records within groups, are in no particular order. Multiple subtypes for a given stimulus type are possible, and different stimulus types might have the same subtype.

You have just performed an experiment characterized by a certain stimulus type, and you have observed a particular type of response. You consider the stimulus subtype to be irrelevant in this experiment. You would like to update your records to show one more occurrence for the particular response and stimulus without regard to the stimulus subtype.

Write a program to update the existing data base in the desired way. Make it as "efficient" as you can. Use a sequential update file.

Suppose either the stimulus type or combination of stimulus type and response type are not to be found in the data base. What are you going to do about that? What happens to your data base if the system crashes in the middle of a run? Is it hard to recover from that? (You bet!)

(#8C) Discuss an alternative design to the program you wrote for #8B which can accommodate new stimulus or response types and which is not subject to severe recovery problems if the system crashes in the middle of a run. What is the "cost" of this extra flexibility and protection?

# 9. Indexed and regional datasets.

9.1. One of the alternatives to consecutive dataset organization is indexed organization, specified by the INDEXED option of the ENVIRONMENT attribute.

Indexed datasets are keyed datasets. The records and their recorded keys are maintained in logical order by ascending key values; their physical order is not material, as far as the program is concerned, because there is no way it can be known. The sequential order defined for indexed datasets is key-sequence order.

Whereas the records in a consecutive dataset can only be accessed sequentially, those in an indexed dataset can be accessed sequentially (in key sequence) or in arbitrary order; that is, either a sequential or a direct file may be used with an indexed dataset. An index is maintained in the dataset, by the operating system, to aid in the location of a record having a particular recorded key value. See LRM 164, LRM 174, CPG 11, and OPG 12.

9.2. Statement options dealing with keys.

When a WRITE statement needs to identify the particular record to be written, it uses the KEYFROM option. The option contains a character-string valued expression whose value is used as the source key (i.e., the program's designation of the record's key value). Example:

```
WRITE FILE (F) FROM (V) KEYFROM (K || P);
```

This causes the contents of the record variable V to be written in the dataset associated with the file F at a place identified by the value of K || P. Usually the value of this source key becomes the value of the recorded key; however, we will see later that the associated dataset need not be a keyed dataset, in which case the source key is used for something else. KEYFROM means "take the source key from the expression..."

There are two ways keys enter the picture in read operations. You may identify the record to be read by giving a particular source key value, using, the KEY option. Example:

```
READ FILE (G) INTO (S) KEY ('REC' || K);
```

This causes the record identified by the value of the expression 'REC' || K (the source key value) to be read from the dataset associated with file G into the record variable S. KEY means "the record whose key is..." Alternatively, if you are reading sequentially you may read the next record in sequence and have its key value handed to you. The KEYTO option is used for this. It names a scalar character string variable to which the key value is assigned. Example:

```
READ FILE (H) INTO (R) KEYTO (KEYVAR);
```

This causes the next sequential record to be read into the record variable R from file H; its key value is assigned to the key variable KEYVAR. As with the WRITE statement, these source key values usually correspond to recorded key values, but here too the dataset need not be keyed and the source key may have a different meaning. KEYTO means "assign the key to...".

When the sequential order defined for a dataset is key sequence, the meaning of the IGNORE option, which can be used -- as we saw in Lesson 8 -- in place of the INTO option, is "read the given number of records in key sequence and ignore them."

A REWRITE statement that replaces an existing record may or may not need to identify the record to be replaced. If it does, it uses the KEY option in the same way as for the READ statement.

See LRM 175 and LRM 176, and review LRM 169.

9.3. ENVIRONMENT options for indexed datasets.

There are a number of ENV options applicable to indexed datasets. Some of these have to do with specifying the length of the recorded key field or its relative position in the record (if it happens to be of the embedded kind). This and other information can also be specified in JCL. We will not cover JCL for indexed datasets. The programmer's guides (references given later) do a good job in this area. For now, see LRM 177. Other ENV options particularly applicable to indexed datasets are scattered throughout LRM 163.

9.4. Creating an indexed dataset.

An indexed dataset must be created sequentially. The records are presented with their keys in ascending key sequence, using WRITE...FROM...KEYFROM. Applicable FDA's are KEYED SEQL OUTPUT. The source key values become the recorded key values. See LRM 178, CPG 12, and OPG 13.

9.5. Retrieving an indexed dataset.

The records of an existing indexed dataset may be read sequentially (in ascending key sequence) or in arbitrary order.

If, when you are reading them sequentially, you don't care to know what the key values are, you can use the same techniques as for consecutive datasets. I.e., you can open the file for sequential input and use READ...INTO or READ...IGNORE. This is an example of a non-keyed file being associated with a keyed dataset.

If you want to know the key values, open the file for keyed sequential input and use READ...INTO...KEYTO. The recorded key values become the source key values. You can also use READ...IGNORE. Another thing you can do is skip ahead in the key sequence to a record having a particular key, by using

READ...INTO...KEY. Having positioned ahead to the desired record, you can then continue reading sequentially with READ...INTO...KEYTO.

To read records in arbitrary order, use the KEYED DIRECT INPUT FDA's and READ...INTO...KEY statements. The key values can be presented in any order.

Normally, the source key value specified in a KEY option of the READ statement translates directly into a recorded key value. The translation has interestingly different properties when the GENKEY option of the ENV attribute is used; read about that at LRM 179.

## 9.6. Altering an indexed dataset.

There are several ways you can update an indexed dataset. Sequential updating is like the kind of updating we showed for consecutive datasets. You first read a record (using any of the forms of READ statement described above), alter the record variable, then write the updated data back out by executing a REWRITE...FROM statement. The KEY option is not used because the record being replaced is the last one read. Applicable FDA's are SEQL UPDATE or KEYED SEQL UPDATE.

You can update records in random order if the file is opened for keyed direct update. In this case a record need not be read before it is rewritten, so you must use the KEY option on the REWRITE statement to designate the record to be rewritten. However, the designated record must exist. If in fact the one you designate for rewriting was not the last one read, the REWRITE causes an implicit READ just to check that the record exists.

An existing indexed dataset opened for keyed direct update can also have new records added to it. Use WRITE...FROM...KEYFROM. Keys can be presented in any order, but they must designate non-existent records (conformance is checked!). Be sure you see the difference between REWRITE and WRITE for a direct update file.

Another way in which an existing indexed dataset can be altered is described immediately below.

## 9.7. The DELETE statement and dummy records.

We come now to the first use of the DELETE statement.

Execution of a DELETE statement causes the specified existing record to be marked as deleted. Subsequently, it is as if the record had never been in the dataset in the first place.

The statement is permitted for direct update files. The KEY option is used to identify the existing record to be deleted. Example:

```
DELETE FILE (F) KEY (K || SUBSTR (S, 2));
```

The record identified by the key whose value is given by K || SUBSTR(S, 2) is deleted from file F.

For indexed datasets, the DELETE statement is also permitted with sequential update files. In this case the KEY option is not used. As with rewriting records, only the last record read may be deleted.

Note that no record variable is used. Data transfer in the usual sense does not occur. However, in the case of indexed datasets accessed through direct update files an implicit READ is performed under the same circumstances as with REWRITE to check that the record exists. The record to be deleted is physically replaced with a special mark that indicates "deleted record." Such a record is called a dummy record. With indexed datasets you actually have your choice (expressed through JCL) as to whether dummy records are or are not to be invisible during a READ operation. See LRM 180, CPG 13, and OPG 14.

Now read LRM 181, CPG 14, and OPG 15.

We have skipped over many of the details of how indexed datasets are managed by the system. Details are to be found in passages previously cited. The addition of records to an existing indexed dataset causes it to become disorganized, and efficiency is severely degraded (your WAIT time can go through the roof!). It is a good idea to "reorganize" an indexed dataset occasionally. See CPG 15 and OPG 16. For some complete examples of the use of indexed datasets, see CPG 16 and OPG 17.

## 9.8. Regional datasets.

The third alternative dataset organization is regional organization. A regional dataset is thought of as being divided into regions numbered consecutively starting with zero. A region can hold one or more records. As we shall see, the sequential order defined for regional datasets has certain aspects of physical sequence (as for consecutive datasets) and certain aspects of key sequence (as for indexed datasets). By specifying region numbers the programmer may optimize (or at least have some control over) the placement of records in the dataset. Source keys are used to communicate region numbers and possibly also key values (to correspond to the values in recorded keys).

There are three subvarieties of regional organization, denoted respectively by the ENV options REGIONAL(1), REGIONAL(2), and REGIONAL(3).

Regional(1) datasets are non-keyed datasets. Each region contains exactly

one record, hence a region number is a relative record number. Source keys, when used, are interpreted as region numbers only. Access may be sequential or direct. Sequential access is in region number order, hence the physical sequence characteristics. Direct access operations go directly to the record identified by the region number given. Regional(1) datasets can contain unblocked fixed-format records only.

Regional(2) and regional(3) datasets are both keyed datasets. Records are always accompanied by (non-embedded) recorded keys. Sequential access is again in region number order, unaffected by recorded keys. Direct access operations start at the region number specified in the source key and scan from that point forward (wrapping around to the beginning of the dataset if the end is reached) for the record identified by the value of a recorded key. The source key is used to specify both the region number at which the search is to begin and the recorded key value to be searched for.

The main difference between regional(2) and regional(3) datasets is that regions in regional(2) datasets correspond to records, as in regional(1) datasets, whereas regions in regional(3) datasets correspond to tracks. Thus, regional(3) regions may contain more than one record.

Regional datasets employ dummy records to mark records as having been deleted (or never written in the first place). There is no choice as to whether dummy records can be detected by the program or not, as with indexed datasets. In some cases they are made available, in other cases they are not, as we shall see.

The above material is reviewed at LRM 182, CPG 17, and OPG 18.

## 9.9. Regional(1) datasets.

Regional(1) datasets may be created sequentially or by direct access.

In sequential creation, the file is opened for keyed sequential output. Records are presented using WRITE...FROM...KEYFROM. The source key value is a character representation of the region number. Records must be presented in order of increasing region number (hence the aspects of key sequence). However, some region numbers may be skipped over; the system implicitly writes a dummy record in each region skipped over. Also, when the dataset is closed the remainder of the space in its current extent is filled with dummy records so that all of the space (through that extent) is filled either with real or dummy records.

In direct creation, the file is opened for keyed direct output. At that time the first extent of the dataset is preformatted by filling it with

dummy records (this can take quite a while). Records are presented as above, but the region numbers may be given in any order. A region number may even repeat; the record previously written in the region will be overwritten. At the conclusion of the creation process the first extent of the dataset will contain the records written and those dummy records not overwritten with real records.

After creation, the records of an existing regional(1) dataset can be retrieved sequentially or directly. Sequential access, using either SEQL INPUT and READ...INTO or KEYED SEQL INPUT and READ...INTO...KEYTO, is in region number order. All records are retrieved, whether dummy or not. The value returned to the key variable named in the KEYTO option is the character representation of the region number. A combination permitted for indexed datasets, READ...INTO...KEY, which is used during sequential input operations to skip ahead in the sequence, is not permitted for regional datasets.

Direct access uses KEYED DIRECT INPUT and READ...INTO...KEY. Records may be retrieved in any order, and dummy records are made available.

The basic facilities for altering a regional(1) dataset are as follows.

<u>SEQL UPDATE</u>

| | |
|---|---|
| READ...INTO | Get next record, real or dummy. |
| REWRITE...FROM | Replace it after changing. |

<u>KEYED SEQL UPDATE</u>

Same as above with addition of KEYTO option to the READ statement.

<u>KEYED DIRECT UPDATE</u>

| | |
|---|---|
| READ...INTO...KEY | Get any record, real or dummy. |
| REWRITE...FROM...KEY | Replace <u>any</u> record, real or dummy. |
| WRITE...FROM...KEYFROM | Same effect as REWRITE. |
| DELETE...KEY | Change any record to dummy. |

Note that the DELETE statement is only allowed with direct update files, whereas with indexed datasets it was also allowed with sequential update files.

If you are wondering why dummy records are retrieved, why you can rewrite or delete a non-existent (dummy) record, and why you can write over an existing record -- who knows? The application of language features to native I/O facilities of the system would be smoother if these things weren't permitted for regional(1) datasets.

See LRM 183, CPG 18, and OPG 19.

9.10. Regional(2) and regional(3) datasets.

These are processed using exactly the same FDA's, statements, and statement options as for regional(1) datasets. The differences are as follows.

Dummy records are not retrieved during read operations. REWRITE can only replace existing (non-dummy records), and DELETE can only delete existing records.

The source key value used in a KEY or KEYFROM option has two parts: a region number and a string corresponding to a recorded key. READ, REWRITE, and DELETE operations "search" for the designated record by starting at the track implied by the region number and actually looking for the given recorded key. (The number of tracks spanned in the search, before giving up, is governed by a JCL parameter.) WRITE operations start at the track designated and look for a dummy record to replace in the same manner. Note that duplicate recorded keys can exist in the dataset. Also note that a record retrieved or written may actually belong to a different region than the one at which the search started, yet no feedback is given concerning the actual region.

In a regional(3) dataset, a dummy record created by a DELETE statement, though not made available to a READ operation, is unfortunately not available for re-use by a WRITE statement. Only dummy records left over from the dataset's creation are available for the addition of new records. All dummy records in regional(2) datasets represent space available for new records.

In sequential output operations it is only the region number part of the source key value that is checked for ascending sequence. There are no requirements on the part of the source key value to be used for the recorded key.

In sequential input (or update) operations, records are retrieved in their physical sequence. If the file is keyed and the KEYTO option is being used to receive the key of the record read, the value assigned to the key variable is the recorded key value only. These will not necessarily be in any particular order.

See LRM 184 and LRM 185, CPG 19 and CPG 20, and OPG 20 and OPG 21. The programmer's guide references contain examples. JCL considerations are given at CPG 21 and CPG 22, and OPG 22 and OPG 23.

## 9.11. The EXCLUSIVE attribute and locked records.

You probably know that the difference between the JCL disposition parameters OLD and SHR is that the first is used to prevent the scheduling of another job that needs to use the dataset to which it applies, the idea being that you intend to write into the dataset and it would be meaningless for another job to access the dataset while you are writing into it, whereas the second says that you don't intend to write in it and hence another job that also needs it, but not for writing, can be scheduled concurrently. By use of appropriate PL/I facilities to "synchronize" access, you actually can permit two jobs that update a given dataset to be safely scheduled together; that is, you can use DISP=SHR in both sets of JCL. This is accomplished as follows. The facilities apply only to direct update files; they are to be used in the way described in both programs.

Use the EXCLUSIVE FDA. When a READ statement is executed on an exclusive file, the record involved (not the whole dataset) is "locked" so that another program cannot access it. If another program tries to, it will just wait for the record to be unlocked. The record is unlocked automatically when your READ statement is followed up by a REWRITE or DELETE statement addressing the same record, or when the file is closed. Alternatively, if you decide after reading the record that you don't want to alter it, you can explicitly unlock it by executing an UNLOCK statement, as in

```
UNLOCK FILE (F) KEY (K);
```

Finally, to suppress the automatic locking that occurs on a READ, add the NOLOCK option to the READ statement. See LRM 191 through LRM 193. These features are not in the ANSI standard. We will see them again in Lesson 14.

## 9.12. Conditions applicable to indexed and regional I/O.

There is one remaining qualified I/O condition to be described. The KEY condition occurs whenever an invalid key value is presented in a KEY or KEYFROM option. Some of the common cases of occurrence are as follows.

Requested record having designated key doesn't exist.
Keyed sequential: key is out of sequence.
No space in dataset to add keyed record.

Other cases are described in the relevant entry in LRM 116. Default status is enabled; KEY cannot be disabled. Standard system action is to issue a message and raise ERROR. On normal return from a KEY on unit, execution continues with the statement following the one whose execution raised KEY.

## 9.13. Review

To review all the record I/O transmission statements, read LRM 186 through LRM 190. Ignore all discussion of the EVENT option. For the READ statement ignore the SET option.

## 9.14. Comparison to FORTRAN.

The ability to read or write a given record identified by its relative record number, provided for regional(1) datasets, is roughly comparable to FORTRAN "direct-access" reads and writes. In FORTRAN, however, the records can be processed as unformatted data transfers or under format control; the latter option is not directly reflected in the PL/I capabilities.

FORTRAN has no processing capability comparable to that provided in PL/I for indexed, regional(2), or regional(3) datasets.

## 9.15. Unanswered questions.

In Lesson 10 we will see how the different causes of the KEY condition may be distinguished.

## 9.16. Homework problems.

(#9A) Why do you think the REWRITE statement uses the KEY option instead of the KEYFROM option?

(#9B) State the distinction between the REWRITE statement and the WRITE statement. (We saw that the distinction was blurred in the case of regional(1) datasets.)

(#9C) State the rule relating READ statements and REWRITE statements for sequential update files.

(#9D) Is the following sequence permitted for sequential update files? If so, what does it mean?

```
READ...INTO
READ...IGNORE
REWRITE...FROM.
```

(#9E) Suppose no records have been added to an indexed dataset since its creation. Considering what the index is used for, how much I/O is involved in finding a record having a given recorded key, by direct access? Specifically, is it a fixed amount or does it depend on the size of the dataset? Answer the same question for regional(2) or regional(3) datasets (assume unique recorded keys). What can the system designer (programmer and data base designer) do to minimize the search time for regional(2) or regional(3) datasets? Can you see any realistic applications for these kinds of regional datasets?

(#9F) If you had a dataset consisting of fixed-length unblocked records that in fact contained the character representations of problem data (perhaps several per record), what PL/I language features could you combine in order to select these records in arbitrary order (by relative record number) and yet still decode them under format control?

(#9G) Which condition, ENDFILE or KEY, do you think is raised by a READ...INTO...KEY statement on a keyed sequential file when, in positioning ahead in key sequence to the record with the designated key, the end of file is encountered?

(#9H) Recall problem #8B (stimulus-response data-base). What advantages might indexed organization yield? Discuss how you would use it.

10. (a) Builtin functions and pseudo-variables.
(b) Interlanguage communication.

## 10.1. Review of builtin functions and pseudo-variables.

In the first three lessons we discussed a good many builtin functions, and some pseudo-variables, without saying too much about them.

Builtin functions are functions that can be invoked for their returned value and which are provided by the language; that is, one need not code procedures to compute the desired function. An implementation may supplement those defined by the language. The function is supported either by in-line code or by a library routine. Builtin functions are provided for purposes of convenience to the programmer, or for common computational needs or because the compiler can generate better code, or sometimes because the function involved simply can't be expressed by the programmer using other language features. All of the builtin functions that we have seen so far take arguments; we will soon see others that don't.

Pseudo-variables are similar in that references to them look just like function references. However, they don't denote a value. Instead, they denote a variable or a portion of a variable and in fact their use is as an assignment target. The pseudo-variables generally have counterparts as builtin functions. For example, SUBSTR is both a pseudo-variable and a builtin function. A reference to SUBSTR means a substring of the string which is the first argument. When it is used as a builtin function reference, the first argument may be an arbitrary expression, because the result of the reference to SUBSTR needs only to have a value that can be used in the context of the expression in which the SUBSTR reference is embedded. When it is used as a pseudo-variable reference, the first argument can <u>only</u> be a <u>string variable</u>, because the result of the reference to SUBSTR in this case needs to denote some "storage" to which a value may be assigned.

## 10.2. Names of builtin functions and pseudo-variables.

In all our examples so far we have used builtin functions and pseudo-variables just by using their names in appropriately constructed syntactical function references. It has been tacitly assumed that the names do not appear in declarations of other objects. If they do, then any use of the name within its scope denotes the object declared, and not the builtin function or pseudo-variable. This means that you can use SIN as the name of a variable, for instance, and you can use LOG as the name of a procedure (internal or external) assuming these names were properly declared. However, within the scopes of their declarations, these names are not available to mean the builtin functions.

## 10.3. The BUILTIN attribute.

The name of a builtin function which has been usurped for some other purpose may be restored to its original meaning, inside a nested block, by declaring it with the BUILTIN attribute. No other attributes, except INTERNAL, can be used with it. Example:

```
P:  PROC;
    DCL INDEX FIXED BIN;
    INDEX = 0;
    BEGIN;
      DCL INDEX BUILTIN;
      I = INDEX(S,'/*');
    END;
END;
```

The first reference to INDEX (INDEX = 0;) is a reference to a FIXED BINARY variable. The second, inside the begin block, is a reference to the INDEX builtin function.

BUILTIN carries no implications for data type, etc. There are no values or variables, or constants, of type "builtin." It is incorrect to declare as BUILTIN an identifier which is not the name of a builtin function. See LRM 194.

It is also improper to think of a builtin function as a kind of "entry" value having, maybe, special properties. The current language, however, blurs the distinction in one special case. The names of the mathematical builtin functions (SIN, SQRT, EXP, etc.) may be used as entry constants in the context of an argument being associated with an entry parameter, e.g.,

```
CALL F(SIN);
CALL F(COS);
F:  PROC (Q);
    DCL Q ENTRY (FLOAT BIN (21))
          RETURNS (FLOAT BIN (21));
    Y = Q(X);
END;
```

See LRM 195. This limited facility is not available in the ANSI standard.

## 10.4. Parameterless builtin functions and pseudo-variables.

Why is it unnecessary, except for the reasons demonstrated above, to declare a builtin function name such as SQRT as BUILTIN? The reason is as follows. A reference such as SQRT(X), in the absence of a declaration for the name SQRT, cannot possibly be anything else. It cannot be a reference to an external entry constant because they must be declared (as we saw in Lesson 4). It cannot be a reference to an array because an undeclared identifier cannot require the dimension attribute by default.

As we shall soon see, certain builtin functions and pseudo-variables do not take any arguments. A reference to one of them, when written without an argument list, would look like a reference to a simple variable, and we have seen that they acquire the attributes of an arithmetic variable by default. There is a potential conflict, then, when a name of a parameterless builtin function, such as TIME, is written by itself. In the absence of an explicit declaration for TIME, shall TIME by itself denote the TIME builtin function, or shall it be a FLOAT DECIMAL (6) variable? It must be the latter; a homework problem will help you understand why.

However, if we want TIME to mean the builtin function rather than a variable declared implicitly (with default attributes), then we may do one of two things. We may explicitly declare TIME as builtin, or we may write the builtin function reference as TIME(), i.e., with an argument list, albeit an empty one. The argument list puts us back in the situation of SQRT(X) which, we argued, cannot be anything but a builtin function reference.

An empty argument list may also be written after the name of a parameterless entry to be invoked, as in

```
FUNCTION:  PROC RETURNS (CHAR(1));
   RETURN (SUBSTR(S, I, 1));
END;
T = FUNCTION();
```

In the ANSI language you will be required to write an empty argument list to get FUNCTION invoked, although in the current language you are not. (Review the discussion in Section 6.8.). It is very good documentary practice, in any event, to write an argument list (if only an empty one) with every function reference.

## 10.5. Additional specific builtin functions and pseudo-variables.

In Lesson 1 we examined all of the arithmetic builtin functions and mathematical builtin functions. In Lesson 2 we had most of the string-handling builtin functions. In Lesson 3 we had some of the array-handling builtin functions.

Remaining string-handling builtin-functions:

STRING This effectively concatenates the elements of its aggregate argument, which must be an array or structure containing string elements. The result is a scalar value. It is as if a scalar string variable were string-overlay defined (Lesson 3) on the argument. The STRING builtin function may also be used as a pseudo-variable. There are minor differences in the ANSI standard.

UNSPEC This effectively allows the storage occupied by its argument to be viewed as a bit string. Since the storage required for a variable of a given data type is implementation-defined, so is the length of the bit string. UNSPEC is also a pseudo-variable. Examples:

```
DCL I FIXED BIN (31),
    X FLOAT BIN (21),
    U BIT (32);
U = UNSPEC(I);  The 32 bits occupied by I (in our
                implementation) are moved into U.
U = UNSPEC(X);  Ditto for X, i.e., it is interpreted
                as a bit string of length 32.
UNSPEC(X) = '0101110...0110'B;
                Store the bit pattern in X.
UNSPEC(I) = UNSPEC(X);
                This moves the contents of X into I
                without conversion. The value of the
                floating point variable X can then be
                manipulated as if its internal representa-
                tion were an integer (by manipulating I
                instead).
```

UNSPEC gives you a legal, though inevitably implementation-dependent, way of looking at the storage occupied by any variable through other attributes.

Remaining array-handling builtin functions:

SUM Takes an array argument and returns the sum of its elements.

PROD Same, but returns the product.

ANY Same, but the array argument is an array of bit strings and the operation is logical "or." The i-th bit in the resulting scalar bit string is 1 if and only if the i-th bit of any element of the array is 1.

ALL Same as ANY except the operation is logical "and." ANY and ALL are renamed SOME and EVERY, and slightly changed, in the ANSI language.

POLY Computes, for array A and value X, essentially $\Sigma a_i x^i$. Also a more general form. Not in ANSI.

Details of the above may be found at LRM 18.

Having covered conditions in Lesson 6 and later, we are now ready to look at the condition-handling builtin functions. See LRM 196 and LRM 18. All of these are parameterless builtin functions (some are pseudo-variables as well) that give you certain information about the interrupt in whose on unit (or descendant block thereof) they are referenced.

ONCODE Returns an implementation-defined integer value specifying the cause of the interrupt. See LRM 197. Can be used, for instance, to determine whether a condition occurred naturally or was signaled or to distinguish between many different reasons for the occurrence of a condition.

ONLOC Returns, as a character string value, the name of the procedure in which the interrupt occurred (more precisely, the name of the entry point at which it was entered).

ONFILE For an I/O condition, the name of the file. Though an on unit can be established separately for each file, making this determination unnecessary, should standard system action in the absence of such an on unit take you into an ERROR on unit, you would need ONFILE to determine the file on which the condition occurred.

DATAFIELD The contents of the bad field that caused the NAME condition to occur (Lesson 7). Called ONFIELD in ANSI language.

ONCOUNT See description in LRM 18 and LRM 198; also OPG 25.

ONKEY The value of a bad key causing the KEY condition.

ONSOURCE The contents of the bad character string value whose attempted conversion to something else failed, causing the CONVERSION condition. Can be used as a pseudo-variable in a CONV on unit to replace that bad string for purposes of recovery; the replacement value is used when the conversion is reattempted on normal return from the on unit (see Lessons 2 and 6).

ONCHAR Denotes the single character (one of those in the string represented by ONSOURCE) which caused the conversion to fail. May be used as a pseudo-variable to replace the single character in a recovery attempt.

Stream I/O builtin functions:

COUNT The number of data items transmitted during the last GET or PUT operation on the specified file.

LINENO The current line number of the specified print file.

One of the storage control builtin functions (others will be discussed in Lesson 11):

ALLOCATION The number of generations in the stack for the given controlled variable (see Lesson 5).

Miscellaneous builtin functions:

DATE Parameterless; returns the current date as an implementation-defined character string value (YYMMDD in our system).

TIME Similarly, the current time (HHMMSSTTT; TTT is milliseconds).

Others in ANSI (which have not been previously mentioned):

| | |
|---|---|
| COLLATE | Returns a character string value containing the character set in the implementation's collating sequence. |
| DOT | Dot product of two vectors. |
| PAGENO | Like LINENO; returns the current page number of the specified print file. |
| SUBTRACT | An arithmetic builtin function, like ADD. |
| VALID | Tests whether a given computational value conforms to a given picture specification. Returns a BIT (1) result without raising the CONVERSION condition. |

Good news! The mathematical builtin functions which we remarked in Lesson 1 had been deleted in the ANSI standard are now back in!

## 10.6. Overview of interlanguage communication facilities.

A natural question to ask is whether or not FORTRAN and PL/I routines can be intermixed. Can a routine written in one language invoke one written in the other? Clearly, if this were possible one could receive that much more value from his accumulation of FORTRAN subprograms, for instance. Or one might extend the usefulness of existing FORTRAN programs by having them interface with PL/I procedures to do update operations on datasets, say.

All of this is possible--but only because certain facilities are specifically provided to meet these needs. These facilities deal with the impediments to free communication between languages. Some of these impediments are described below.

The primary problem is that different languages generally have different run-time environments. This is true in IBM systems but not, apparently, in Univac systems. The differences in environment involve, among other things:

(a) the handling of hardware interrupts, such as overflow;
(b) the addressing of arguments and parameters (arguments and parameters may be addressed on different sides of an interlanguage boundary);
(c) the mapping of aggregates, such as arrays;
(d) defined actions on program termination.

The ILC (interlanguage communication) facilities of the Checkout and Optimizing compilers permit useful communication between PL/I, FORTRAN, COBOL, and Assembler routines. In this course we will study only FORTRAN-PL/I communication. The details of PL/I-Assembler communication are well covered in OPG 24 and CPG 23.

When communicating with FORTRAN, the main program may be of either language. There are no special requirements for the contents of the FORTRAN routines; existing ones may be used with PL/I without recompiling. All of the services required are performed by the PL/I system in accord with specifications made in the PL/I routines. See LRM 199.

## 10.7. FORTRAN calling PL/I.

An external PL/I procedure to be called by FORTRAN must use the OPTIONS option of the PROCEDURE statement to announce this fact. Example:

```
PLISUB:  PROC(X,Y) RETURNS (FLOAT)
                OPTIONS (FORTRAN);
```

OPTIONS (FORTRAN) may also be specified on an ENTRY statement of an external procedure. In fact, a procedure may have several different entry points, some to be entered from a FORTRAN routine and others from PL/I. Any given entry point cannot, unfortunately, be invoked equally well by both.

If any of the parameters of a PL/I procedure called by FORTRAN are arrays, their bounds must be declared in the PL/I procedure as constants (unfortunately). Recall from Lesson 5 that the only form of "adjustable extent" permitted in parameter declarations is an "asterisk extent," denoting that the bounds are inherited from the actual argument. Unfortunately, FORTRAN doesn't make that information available. Since PL/I won't allow expressions in declarations of parameters, the following "FORTRAN-style" construction is not allowed:

```
P:  PROC (X,N) OPTIONS (FORTRAN);
    DCL X(N) FLOAT;
```

Because arrays of more than one dimension are mapped differently in PL/I and FORTRAN (row-major order in PL/I and column-major in FORTRAN), one of the services provided by the OPTIONS(FORTRAN) specification is the remapping of an array (of more than one dimension) on entry and on return. That is, on entry to the PL/I procedure storage is acquired dynamically, the FORTRAN array is copied into it (in transposed order), and the copy is then used in the PL/I procedure. On return from the procedure the transposed copy of the array (which may have been the target of some assignments) is copied back into the actual FORTRAN array in the proper order, and the dynamic storage is released. Thus, the fact that arrays are mapped differently in our implementations of these two languages need not be a concern.

However, the remapping of arrays can be "expensive" if they are large (extra storage requirements for the remapped copy) or if it occurs frequently (extra execution time for the copying). If these factors are important, the PL/I programmer has several options at his disposal to refine or control the services provided automatically.

First of all, if an array parameter is not changed by assignment in the PL/I procedure, the transposed copy need not be written back into the original FORTRAN array on return from PL/I. To suppress that, use the NOMAPOUT option of the OPTIONS option. Example:

```
P: PROC (X,Y,Z) OPTIONS (FORTRAN NOMAPOUT(X,Z));
```

Alternatively, an array which is not assumed to have a value on input i.e., whose elements are not fetched in the PL/I procedure before being assigned values by it, does not have to have the remapped copy initialized with the elements of the FORTRAN array on entry. Specify this with the NOMAPIN option, e.g.,

```
P:  PROC (X,Y,Z) OPTIONS (FORTRAN
                          NOMAPOUT(X,Z)
                          NOMAPIN(Y));
```

The programmer may also suppress entirely the creation of a copy (which saves space as well as time) if he is willing to reverse the order of subscript expressions in subscripted array references in the PL/I procedure. To do that, use the NOMAP option (syntax same as for previous options). Actually, the need to reverse the order of the subscript expressions can be avoided by the use of ISUB-defining (Lesson 3).

See LRM 200 - LRM 204.

## 10.8 PL/I calling FORTRAN.

As with all external entry constants, the name of the FORTRAN routine must be declared as EXTERNAL ENTRY in the PL/I procedure (Lesson 4). To indicate that it is a FORTRAN, and not a PL/I, routine, the OPTIONS attribute is also used. Example:

```
DCL COMP ENTRY (FLOAT(*), FIXED BIN (31))
         RETURNS (FLOAT)
         OPTIONS (FORTRAN)
         EXT;
```

The OPTIONS attribute is much like the OPTIONS option. It can include the NOMAP, NOMAPIN, and NOMAPOUT options to control the automatic remapping of multidimensional arrays. The individual arguments to which these options apply are indicated in the way demonstrated below.

```
DCL FORTSUB ENTRY ((*,*) FLOAT, (*,*) FLOAT,
                   (*,*) FLOAT, (*,*) FLOAT)
            OPTIONS (FORTRAN
                     NOMAPIN(ARG1)
                     NOMAPOUT(ARG4)
                     NOMAP(ARG3))
            EXT;
```

With the above declaration, in a call such as

```
CALL FORTSUB(A(*,*), B(*,*), C(*,*), D(*,*));
```

the third argument, C, will be passed as is without remapping, while dummy arguments (copies) will be made for the others just before

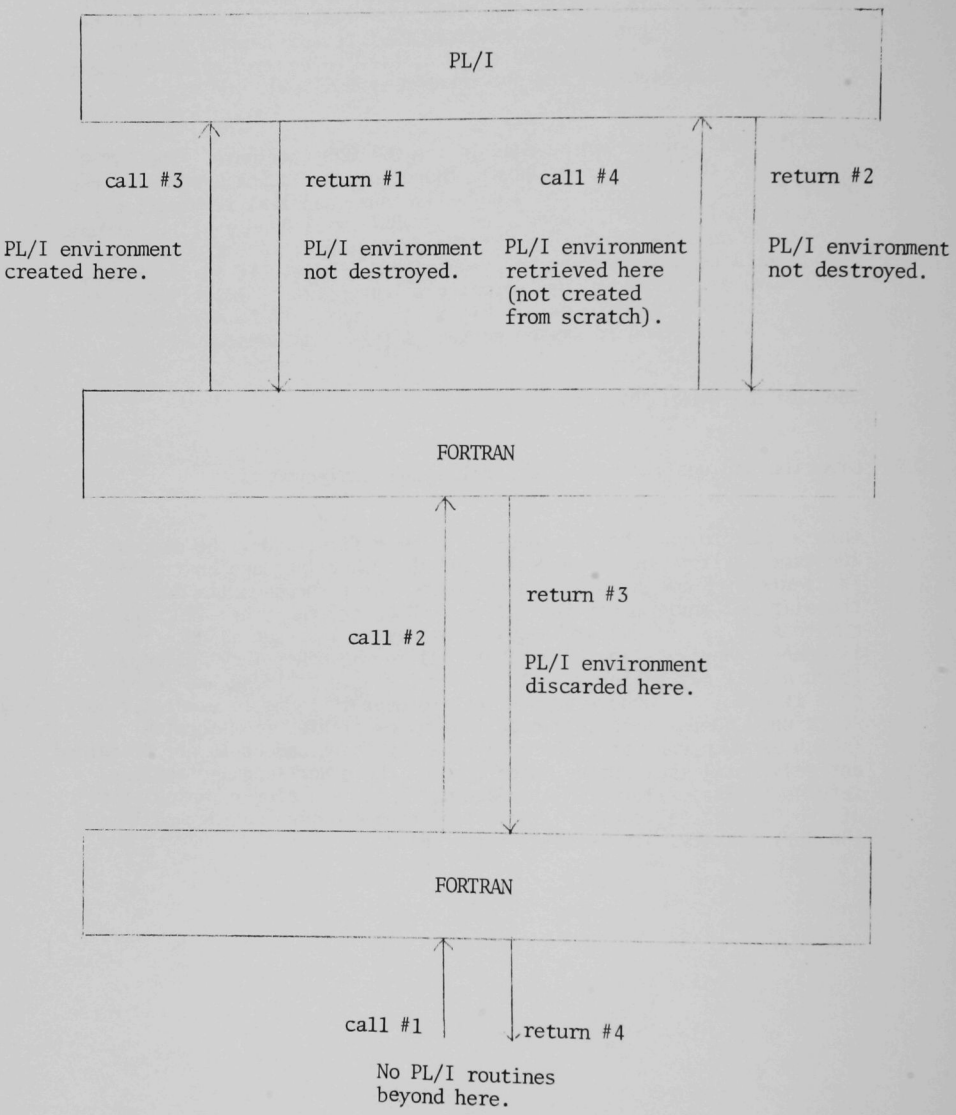
invoking FORTSUB. The copies of B and D will be initialized to the transposes of B and D during this process, but A's will not be initialized (A is presumed to be an "output" argument); and on return from FORTSUB the elements of the transposed copies of A and B will be assigned back to A and B, respectively, before the dynamic storage for all the copies is released (D is assumed to be an "input" argument, i.e., one whose elements are not changed by FORTSUB).

An additional option can be used in the OPTIONS attribute. The INTER option says that PL/I is to handle those interrupts not handled by the other system (ones which would normally cause abnormal termination). For the Model 195, INTER specifies that PL/I will handle all interrupts. By "handle an interrupt" is meant the following: the chain of active blocks will be searched for an established on unit. If one is found, it is invoked; it may return normally to the point of interrupt or it may terminate by a GO TO out of block, as usual. If no established on unit is found, standard system action is taken, as usual.

See LRM 205 - LRM 207.

## 10.9. Creation and destruction of other-language environments.

When a call to an other-language routine is first made, the current run-time environment is set aside and the other-language environment is created. When the other-language routine returns to its caller, the original environment is restored. However, the other-language environment is not discarded quite yet; it is just set aside. This is done in anticipation of another call to the other-language routine (such a call may be inside a loop, for instance). If such a repeat call is made, the other-language environment is found to exist already, so it only needs to be retrieved instead of created from scratch (which is much cheaper). An other-language environment is not discarded entirely until the routine which invoked the other-language routine returns to its caller. This is accomplished by a clever manipulation of the "save area" chain by the interlanguage communication modules of the PL/I library. It is illustrated below.


PL/I
call #3
PL/I environment created here.
return #1
PL/I environment not destroyed.
call #4
PL/I environment retrieved here (not created from scratch).
return #2
PL/I environment not destroyed.
FORTRAN
call #2
return #3
PL/I environment discarded here.
FORTRAN
call #1
return #4
No PL/I routines beyond here.

One consequence of the destruction of a language's environment is that files opened while in that environment are closed when the environment is terminated. Several things can be done to retain an other-language environment for a longer time than the preceding diagram shows. One involves crossing several boundaries. If, in the preceding diagram, an active PL/I procedure exists somewhere in the chain of calls where the comment "No PL/I routines beyond here" is, then the PL/I environment is not created at "call #3" (it is merely retrieved) and it is not discarded at "return #3"; it is not discarded until the original or first caller of a PL/I procedure returns to its caller.

Another technique is demonstrated by the following. Assume the main program is FORTRAN. Let it call a dummy PL/I procedure which merely returns to its caller. This establishes the PL/I environment; it is not discarded until the caller of the PL/I procedure, i.e., the FORTRAN main program, returns to its caller (which in this case is the operating system). Thus, the chain of calls subsequently initiated by the FORTRAN main program may cross language boundaries any number of times, at any depth, without destruction of the PL/I environment.

See LRM 208. Read LRM 209 to review. LRM 210 contains a discussion of communicating via common storage ("named COMMON" in FORTRAN, STATIC EXTERNAL in PL/I).

## 10.10. JCL considerations.

It is recommended that you use PL/I cataloged procedures to link edit, load, and execute mixed PL/I-FORTRAN programs. (These are discussed in Lesson 13.) You will need to make the FORTRAN library available to the linkage editor or loader whenever a PL/I procedure contains OPTIONS (FORTRAN). One way this may be accomplished is by use of the symbolic parameter POSTLIB:

```
POSTLIB = 'SYS1.FORTLIB'
```

Also, you will need to supply a DD statement for FT06F001 in the GO step, even if the FORTRAN routines never write to unit 6, since one of the actions performed during creation of the FORTRAN environment is the opening of FT06F001. See OPG 25 and CPG 24.

## 10.11. Homework problems.

(#10A) Explain what each of the following means:

```
B =  SUBSTR(UNSPEC(X), 4, 2);
B =  UNSPEC(SUBSTR(S, 4, 2));
```

Are the following allowed? If so, what do they mean?

```
SUBSTR(UNSPEC(X), 4, 2) = B;
UNSPEC(SUBSTR(S, 4, 2)) = B;
```

If you are having trouble deciding whether these are allowed, consider which of the following are allowed and which aren't. F, G, and H are user-defined functions.

```
SUBSTR(F(X), 4, 2) = B;
SUBSTR(S, G(A), 2) = B;
SUBSTR(S, 4, H(A)) = B;
UNSPEC(F(C)) = B;
```

(#10B) In the notes we said an undeclared identifier cannot acquire the dimension attribute by default. Let's explore this further.

Under what conditions are identifiers declared contextually? (Review LRM 71.)

If A, B, and SIN are not explicitly declared in a program, are they contextually declared by their appearances in the following statement? If so, as what?

```
A = B(1) + SIN(1);
```

Is there any error here, i.e., will the compiler balk?

Suppose the program contained a DEFAULT statement which specifies the dimension attribute as a default:

```
DFT RANGE (*) (2);
```

When are defaults applied (in particular, before or after acquisition of attributes by context)? (Review LRM 72.) To which of A, B, and SIN would this default apply? Is the program now legal?

Leave the DEFAULT statement in and consider the following. What is the effect of the addition of the declaration

```
DCL (A, B);
```

to the program? Note that A and B are explicitly declared, but with no attributes. Do they acquire any attributes by context? By default? Is the program now legal?

What happens if we also add the following?

```
DCL SIN;
```

Consider the program in any of its intermediate stages as it was developed above. If it was legal at a given stage, would its meaning have changed if B were all of a sudden added to the language as the name of a builtin function? Is there any way a program can have its meaning changed by the addition of a builtin function? Comment on what might happen in FORTRAN if a new intrinsic function were added.

(#10C) Comment on why TIME is not known as a builtin function in

```
A = TIME;      (no declarations)
```

but is in

```
A = TIME();
```

or

```
DCL TIME BUILTIN;
A = TIME;
```

(#10D) During an attempt to add a keyed record to an indexed dataset using a direct update file and a WRITE...FROM...KEYFROM statement, the KEY condition may occur either because there is no space for the new record or because a record containing the specified key already exists. How can these cases be distinguished?

(#10E) Suppose a CONVERSION on unit has been established as follows:

```
ON CONV ONCHAR() = '0';
```

What happens when each of the following character string values undergoes conversion to numeric? What is the final numeric result?

```
12X3
1X4Y
12EF3  (tricky)
```

What happens in the following?

```
CDEFGH  (very tricky)
```

(Hint: how many digits can appear in an exponent field of a floating-point constant, in our implementation?)

(#10F) In Lesson 6 we remarked that

```
IF A = B THEN ...;
```

is illegal if A and B are arrays, because the comparison operator applied to arrays yields an array of BIT (1) results and the IF statement requires a scalar expression. Show how the ALL builtin function can be used to achieve the desired meaning in the IF statement.

# 11. List processing and locate-mode I/O.

## 11.1. Pointers.

In this lesson we will encounter three new types of program-control data. The first of these is "pointer."

A pointer value is an address of some variable. A pointer value must not be thought of as an integer; it cannot be used in the ways integers can be used. For instance, you cannot do arithmetic with it and you cannot write a pointer value out with stream output.

The POINTER attribute is used to declare a pointer variable (usually called simply a pointer), i.e., a variable, the data type of whose possible values is "pointer." The abbreviation of POINTER is PTR. Like other variables, pointers may be internal or external, of any storage class, aligned or unaligned, parameters, defined on other pointers, arrays, structure base elements, initialized, etc.

## 11.2. The ADDR builtin function.

One way that pointer values may be "generated" is by reference to the ADDR builtin function (one of the storage-handling builtin functions). The argument of ADDR may be any variable reference (but it must denote a variable in connected storage). The result of the builtin function reference is a pointer value which is the address of the argument.
Examples:

```
P:  PROC (X) RECURSIVE;
    DCL X FIXED BIN (15);
    DCL Y FLOAT DEC (6) CTL;
    DCL Z CHAR (20) VAR AUTO;
    DCL A (15) CHAR (1) STATIC;
    DCL 1 S STATIC,
          2 T,
          2 U,
            3 V FIXED DEC (5,-2),
            3 W FLOAT BIN (100);
```

| | |
|---|---|
| ADDR(X) | is the address of the actual argument associated with X in the current invocation of P. |
| ADDR(Y) | is the address of the current generation of the controlled variable Y, i.e., the one on top of the stack for Y. |
| ADDR(Z) | is the address of the generation of Z allocated on entry to the current invocation of A. |
| ADDR(A) | is the address of the whole array A. |
| ADDR(A(3)) | is the address of the third element of the array A. |
| ADDR(A(I)) | is the address of the I-th element of the array A. |

| | |
|---|---|
| ADDR(S) | is the address of the structure S. |
| ADDR(S.U) | is the address of the substructure U within S. |
| ADDR(S.U.W) | is the address of the structure base element S.U.W. |
| ADDR(5) | is illegal because the argument is not a variable. |

We will presently see the unique function pointer values serve. For the time being, note that they may be assigned to pointer variables and they may be compared for equality (as with all program-control data, only the comparison operators = and ¬= are allowed).

## 11.3. The BASED storage class and based variables.

In Lesson 5 we saw three of the four storage classes, namely those described by the storage class attributes AUTOMATIC, STATIC, and CONTROLLED. We will now describe the remaining storage class, denoted by the BASED attribute. A variable having this storage class is called a based variable.

The unique significance of based variables lies entirely in the meaning of a reference to one and in how they are allocated.

To begin with, let's look at references to based variables ("based references"). Suppose we have a based variable B:

```
DCL B FIXED BINARY (31) BASED;
```

We may think of B as not having any storage of its own, i.e., not having a unique, assigned location. A reference to B nevertheless denotes a reference to a FIXED BINARY (31) ALIGNED variable residing somewhere. We are responsible for saying where. We do that by providing, with the written reference to B, a pointer-valued expression whose value is taken to be the address of B for that reference. The syntax, in general, is:

*pointer-expression → based-variable*

A simple example is P → B where P is a pointer variable. NOTE: The symbol "→" appearing in these notes is represented in a PL/I program by a minus sign immediately followed by a "greater than" sign, i.e., "->".

Let's examine this closely. We read it as "the B pointed to by P." It is a variable reference like any other: it denotes a location having a value understood in the context of certain attributes (FIXED BINARY (31) ALIGNED in this case). It may be used anywhere a variable reference is permitted, as in the following examples:

```
P → B = P → B + 1;
IF P → B > 0 THEN CALL F(C, P → B);
GET LIST (P → B);
DO P → B = 1 TO 10;
A(P → B) = C(P → B) / 3;
```

In the above examples it has been assumed that P has been assigned a value which is the address of a FIXED BINARY (31) ALIGNED variable. The reference P → B is not legal unless this is so. Examples:

```
DCL (E, F, G,) FIXED BIN (31);
DCL AR (10) FIXED BIN (31);
DCL 1 S,
      2 T FLOAT DEC (6);
      2 U FIXED BIN (31);
      Before P is assigned a value, a reference to P → B is illegal.
      P = ADDR(E);
      P → B now denotes E, i.e., it is a reference to the storage "occu-
           pied by" E.
      P = ADDR(F);
      P → B now refers to the storage occupied by F.
      P = ADDR(AR(I));
      P → B now refers to the storage occupied by AR(I), i.e., by AR(i) if
           I had the value i when the address of AR(I) was taken.
      P = ADDR(S.U);
      P → B now refers to the storage occupied by S.U.
      P = ADDR(S.T);
      A reference to P → B is now illegal; we will examine why later.
```

You can see from these examples that the location referenced in a based reference is determined by the current value of the pointer expression written with the reference. Actually, in all of the examples that expression was merely a scalar pointer variable; we will see more general forms shortly. The thing to note is that the same pointer expression may have different values at different times, and thus a given based reference may denote different locations at different times. Example:

```
DO P = ADDR(E), ADDR(AR(I)), ADDR(F);
     P → B = P → B + 1;
END;
```

This loop causes 1 to be added successively to the three FIXED BINARY (31) ALIGNED variables E, AR(I), and F.

Of course, two different based references involving the same based variable may have different pointer expressions. Suppose P and Q were both pointer variables. Then P → B and Q → B denote FIXED BINARY (31) ALIGNED variables having potentially different locations. They would denote the same thing only if the values of P and Q were equal. Example:

```
P = ADDR(E);
Q = ADDR(F);
P → B = P → B + Q → B;
```

The effect of this is to add F to E.

All of the pointer expressions shown so far have been simple scalar variables. Slightly more complicated instances of variables used in based references are as follows:

```
DCL P PTR STATIC;
DCL Q PTR BASED;
```

```
DCL B ··· BASED;
P → Q → B is read "the B pointed to by the Q pointed to by P."  B is lo-
     cated by a pointer variable, as in the previous examples; however,
     that pointer variable, Q, is itself based and is located by P.
DCL R(10) PTR;
R(I) → B denotes the B pointed to by the I-th element of the pointer
     array R.
DCL 1 PTRS,
     2 FIRST PTR,
     2 LAST PTR;
PTRS.FIRST → B denotes the B pointed to by PTRS.FIRST.
```

Function procedures can return pointer values, i.e., you can write

```
DCL SUB ENTRY (FIXED) RETURNS (PTR) EXT;
SUB: PROC (I) RETURNS (PTR);
  :
  :
  RETURN (P);
  :
  :
  RETURN (ADDR(E));
etc.
```

An example of a pointer expression which is not a pointer variable (possibly subscripted or structure-qualified) is a function reference:

```
SUB(J*K-2) → B
```

Such a function reference may be a builtin function reference which returns a pointer value:

```
ADDR(E) → B.
```

We conclude by saying that a pointer expression is either a pointer variable or a function reference; there are no "operators" that yield a pointer value as result.

This discussion of based variables has served to show the uses of pointer values: they are used to "locate" based variables--that's all!

The process of locating a based variable is called <u>pointer qualification</u>. All of our examples have been examples of <u>explicit pointer qualification</u> (in which the pointer expression used to locate the based variable is written explicitly as part of the based reference, using the pointer qualification symbol → ). In another form, <u>implicit pointer qualification</u>, the qualifying pointer expression is written as part of the BASED attribute and is omitted from the based reference, as in:

```
DCL B FIXED BINARY (31) BASED (P);
P = ADDR(E);
B = B + 1;
P = ADDR(F);
B = B + 1;
```

This causes 1 to be added first to E, then to F. The implicit pointer qualification can be overridden on a particular based reference, as in

```
B = Q → B + 1;
```

There is not much going for implicit pointer qualification. It is just a convenience feature that saves writing in certain cases. An unqualified based reference such as B fails to convey to the reader that the location of B is determined dynamically and is given by the value of an expression appearing elsewhere in the program. We thus recommend that explicit pointer qualification always be used.

We must emphasize that a based variable, B, declared with data type attributes *attr*, can only be used to access storage belonging to a variable having the attributes *attr*. Thus, execution of the statement labeled L in

```
DCL V1 attr1;
DCL V2 attr2 BASED;
DCL P PTR;
P = ADDR(V1);
L: some reference to P → V2;
```

is in error unless *attr1* and *attr2* are the same. (*attr1* and *attr2* need not be explicitly declared, as shown; they may, of course, be attributes acquired contextually or implicitly in the general case.) One slight exception to the requirement for matching of attributes is given later.

As a consequence of the above rule,

```
DCL V1 FLOAT DEC (6);
DCL V2 BIT (32) ALIGNED BASED;
DCL P PTR;
P = ADDR(V1);
DCL B32 BIT (32);
L: B32 = P → V2;
```

is illegal (execution of the statement labeled L is in error, even though a FLOAT DEC (6) variable occupies 32 bits in our implementation). It is just as illegal to look at storage through "different" attributes than the ones implied upon its allocation, using based variables, as it is with defined variables (see Lesson 3). The purpose of this is to guarantee that a legal program has the same meaning in all implementations.

Now what does a based reference such as P → B(I), where B is a based array, mean? This is read "the I-th element of the array B pointed to by P." That is, the value of P must be the address of an array having the same attributes as B (including the dimension attribute). Note that it is the address of the whole array and not the address of the I-th element. Example:

```
DCL B (10) FIXED BIN (31) BASED;
DCL C FIXED BIN (31) BASED;
DCL P PTR;
DCL V (10) FIXED BIN (31);
P = ADDR(V);
```

P → B is a reference to the whole array V.
P → B(I) is a reference to V(I).
P → C is illegal, because P does not point to a scalar FIXED BINARY (31) variable; it points to an array.
P = ADDR(V(J));
P → C is now legal. It is a reference to V(J) - or, more precisely, V(*j*), where *j* was the value of J when the address of V(J) was taken. C has the same data type (and structuring) attributes as an element of V (that is, as what P point to), namely, scalar FIXED BINARY (31).
P → B is illegal because P doesn't point to an array; it points to a scalar.
P → B(I) is illegal for the same reason.

By the same token, P → S.T means "the T component of the structure S pointed to by P." P must have as value the address of a variable having the attributes (including structuring) of S. Example:

```
DCL 1 S BASED,
      2 T FLOAT,
      2 U,
        3 V FIXED BIN (15),
        3 W CHAR (3);
DCL 1 X LIKE S STATIC;
DCL 1 Y LIKE S.U BASED;
DCL P PTR;
P = ADDR(X);
```

P → S is a reference to X.
P → S.U is a reference to X.U.
P → S.U.V is a reference to X.U.V.
P → Y is illegal, because the attributes of Y are not the same as what P points to, i.e., are not the same as those of X.
P = ADDR(X.U);
P → Y is now legal, since Y has the same attributes as what P points to, i.e., as X.U, namely, a structure consisting of a FIXED BIN (15) item followed by a CHAR(3) item, both one level removed from the parental level.

Of what use is any of this? So far we have seen how variables that already exist, i.e., that have had storage allocated to them presumably as the result of a static, automatic, or controlled allocation, may be accessed through a similarly-structured based variable, which does not have any storage of its own (i.e., which serves only as a sort of template that can be moved around). But this doesn't provide much more facility than other techniques for looking at the same storage through different variables (e.g., through parameters or defined variables, neither of which has any storage of its own).

The main use for based storage is a technique called "list processing" which utilizes certain things we haven't described yet (next section). But the facilities we have described so far form the basis of many "system programming" applications when combined with something like "UNSPEC"-ing an absolute

integer value into a pointer variable to gain access to an absolute memory location not "belonging" to a variable allocated during the execution of the PL/I program. That such applications are made implementation-dependent by the use of UNSPEC is not objectionable because, after all, the "system" which is the object of its processing is what defines the "implementation," in a sense. You do not require a program that accesses IBM OS control blocks, for instance, to run on a Univac system.

Any of the "illegal" uses of pointer qualification demonstrated above represent violations of the ANSI standard. As with any such language violation, the meaning of the program in not defined by the language rules and an implementation may do what it wants. Three possible ways an implementation may "react" to a language violation are as follows:

(a) Generate code which assumes no violation has occurred. This approach leads to the most efficient program when, in fact, no violation of language rules occurs. When one does, however, the result is often unpredictable. Sometimes it may be predictable and useful, and many technically illegal (and potentially unexportable) programs are written on this basis. In any event, the result is not documented officially: you either hear about it from someone else, discover it by accident, discover it by looking at generated code, or assume something incorrect about the language itself which turns out to be a property of the implementation and not the language.

(b) Permit the violation and document the consequences. This is often done when those consequences are useful and consistent within an implementation, and when they don't change the meaning of a program which doesn't rely on them. This is called an "implementation extension."

(c) Check for the violation and report an error. This differs from (b) in two respects: extra code is specifically generated (or executed) to detect violations; and when a violation is detected, no way is provided to extract a useful result from it. For instance, an implementation may raise the ERROR condition in such an instance. Recall from Lesson 6 that there is no way to return to the point of interrupt after ERROR is raised.

With respect to illegal uses of pointers, the Optimizing compiler takes approach (a) and the Checkout compiler takes approach (c).

It is time to catch up with references: read LRM 211 - LRM 215 and the entry for ADDR in LRM 18.

## 11.4. Allocating based storage.

A based variable can be used in another way: to allocate some storage dynamically. For example:

```
DCL B (10) FIXED BIN (31) BASED;
DCL P PTR;
ALLOCATE B SET (P);
```

The ALLOCATE statement causes a generation of storage sufficient to hold a variable having the attributes of the based variable to be allocated, and it causes the address of that generation of storage to be assigned to the indicated pointer variable. Note that if the BASED attribute in the declaration of the based variable contains a reference to a pointer variable, the SET option of the ALLOCATE statement may be omitted; the address of the new generation is assigned to that pointer variable. Example:

```
DCL X ... BASED (P);
ALLOCATE X;
```

Here, the address of the storage allocated dynamically is assigned to P.

We saw in Lesson 5 how the ALLOCATE statement is used to allocate a new generation of storage for a controlled variable. No address is returned. The previous generation is "stacked," and subsequent references to the controlled variable refer to its most recent generation. A FREE statement returns the storage belonging to the current generation and "unstacks" the previous one, making it current.

When the ALLOCATE statement is used to allocate storage for a based variable, you are handed the address of the new generation. You use that subsequently to locate the storage for that generation. You may allocate multiple generations. Providing you save their addresses in different pointer variables you will be able to access all of them (compare to controlled variables, where you can only access the most recent generation). Example:

```
DCL B ... BASED;
DCL (P,Q) PTR;
ALLOCATE B SET (P), B SET (Q);
P → B = 1;
Q → B = 2;
P → B = P → B / Q → B;
```

Note that if you are not careful you can easily lose track of storage allocated dynamically through a based variable. For example,

```
ALLOCATE B SET (P);
:
:
ALLOCATE B SET (P);
```

It is assumed that the value of P is not copied into any other pointer variable between these two statements. Then, the second allocation assigns a new value to P. You subsequently have no way of accessing the first allocation of B - or of recovering its storage. It is lost forever!

The FREE statement is used to release storage acquired dynamically for a based variable (we saw it used in Lesson 5 for controlled variables). You have to qualify the generation of the based variable being freed, using either explicit or implicit pointer qualification. Example:

```
FREE P → B;
```

Clearly, you can free generations of based variables in any order; they are not required to be freed in the order of allocation, or the reverse of that, etc.

Once a generation of storage has been allocated for a based variable, that storage bears no intimate relationship to <u>that</u> based variable. Any based variable having the same attributes (including structuring) may be used to access it (in conjunction with the appropriate pointer value). In this regard, the generation of storage is indistinguishable from that belonging to a non-based variable which happens to be accessed through a based variable and a pointer value obtained by taking the ADDR of the non-based variable, as was demonstrated earlier in this lesson. Note, however, that the only kind of storage that can be freed by a FREE statement naming a based variable is storage that was allocated by an ALLOCATE statement for a similar based variable.

In Lesson 5 we said that initialization implied by the INITIAL attribute takes place upon allocation. This is, of course, true also for based variables.

We have now seen two ways that new pointer values are generated: by reference to the ADDR builtin function and by allocation of a based variable.

See LRM 216, LRM 217, and relevant parts of LRM 90.

## 11.5. Adjustable extents for based variables.

Based variables can have adjustable extents, that is, array bounds and string lengths given by the values of expressions appearing in their declarations. However, in the current language you must use another option in conjunction with these, and the items containing adjustable extents can furthermore only exist as members of a based structure. The additional option is called the <u>REFER option.</u> An example follows:

```
DCL N FIXED BINARY (15);
DCL 1 S BASED,
      2 L FIXED BIN (15),
      2 A (N REFER (S.L)) FLOAT;
```

A is a one-dimensional array of FLOAT elements and is a member of the structure S. On allocation of S, the upper bound of A is taken to be the value of N. That determines how much storage is allocated. That value is then automatically assigned to S.L in the newly allocated generation of S. On any subsequent reference to this generation of S, the element S.L is consulted to find the upper bound of S.A (if that is needed for anything). Clearly, you can freely assign values to S.L after allocation of S, but on any reference to S.A (or an element thereof) the value of S.L must be what it was when S was allocated. In particular, S.L must have its original value when S is freed.

For different allocations of S, N may have a different value. Thus, the different generations of S would contain arrays with different upper bounds. Each generation of S would contain the upper bound (in S.L) of its own component A. We call such structures self-defining data.

A based structure declaration may contain any number of adjustable extents and REFER options. There are a few rules that guarantee that a structure can be "mapped" when a reference is made to it. For example, each refer object (the structure base element named in a REFER option) must precede the component whose declaration contains it. See LRM 218 through LRM 220.

The REFER option exists in the same form in the ANSI standard. In addition, any extent may be given by an expression without the REFER option, and such an extent need not belong to a structure member. The expression is evaluated upon allocation and whenever necessary to "map" the variable to which it applies, subsequently; in the latter case it must give the same value as it did on allocation. Example:

```
DCL N FIXED BIN;
DCL A (N) FLOAT BASED;
DCL (P,Q) PTR;
N = some value (value 1);
ALLOCATE A SET (P);
N = some other value (value 2);
ALLOCATE A SET (Q);
```

On any reference to the generation of A located by P, N must have "value 1" and on any reference to Q → A it must have "value 2."

## 11.6. List processing.

The value of allocating multiple generations of a based variable is limited by your ability to store all the pointer values used to locate them. For instance, if you use an array of 100 elements to store pointer values, you can't keep track of more than 100 simultaneous generations, even though you could allocate more.

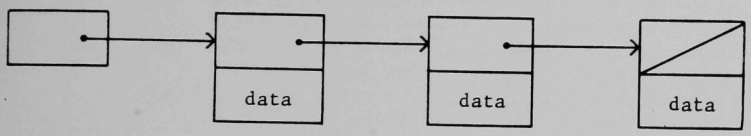
The truly outstanding value of based variables and pointers is that the generations of the based variables themselves can contain the pointer values used to access "related" generations. This is the essence of list processing. It is a way of organizing, allocating, manipulating, and referencing an unbounded amount of data related in some useful way (bounded only by the total amount of memory available). The relationships between data items (or organization imposed on them) characterize certain logical properties of the data and define how you may access them.

Such a collection of data items is called a list structure. To repeat: it is a collection of multiple generations of based variables in which each generation is an aggregate which contains both problem-dependent data and pointer variables used to reach related generations. Initial entry into such a

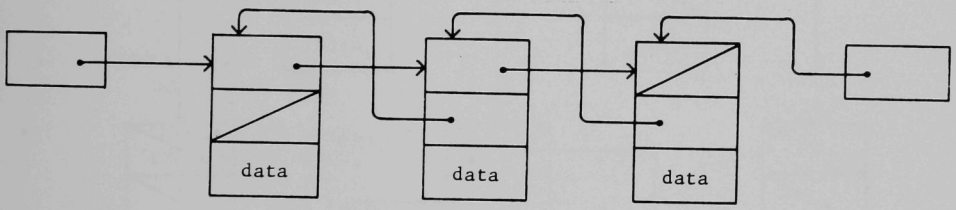
collection is by means of a pointer value (or maybe several) held external to the collection itself.

A list structure may take many forms. Some examples are pictorialized below.

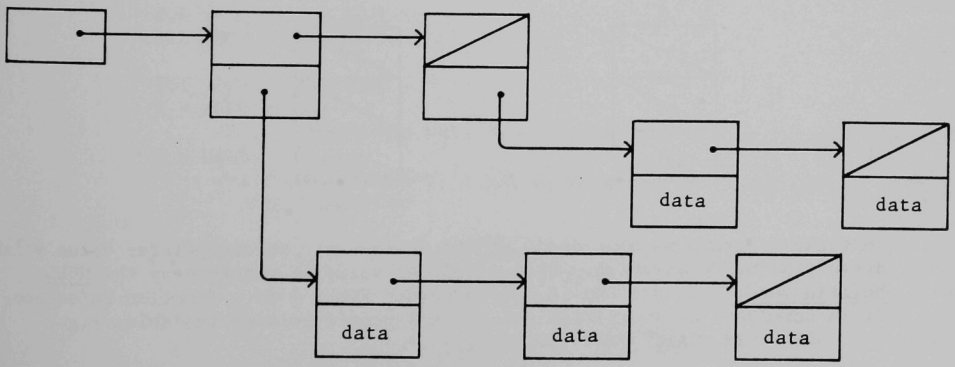
One-way linked list



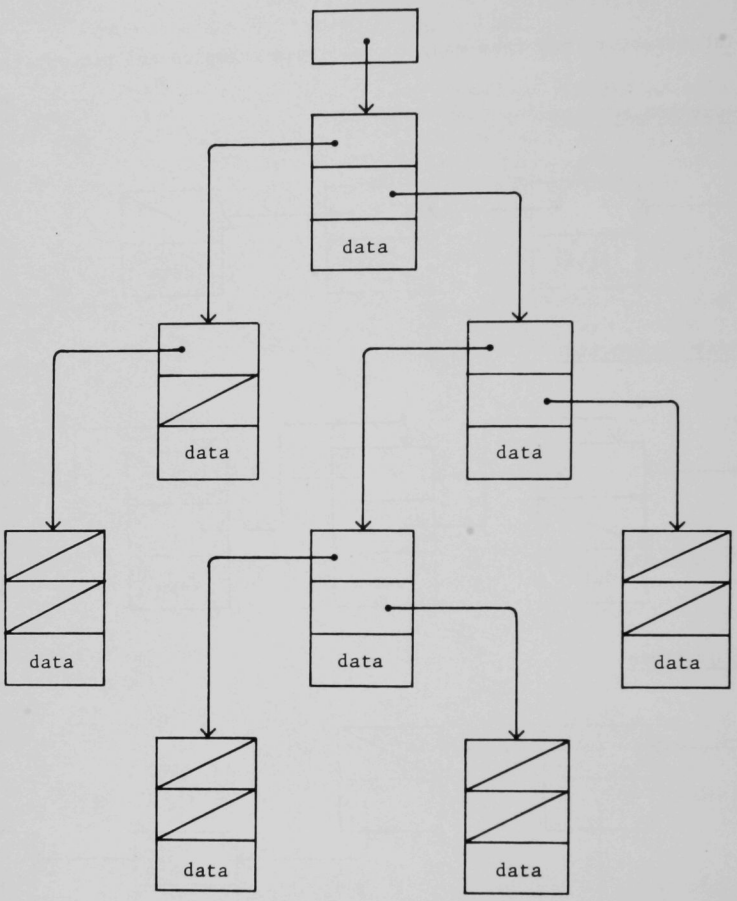

Two-way linked list




List of lists



Binary tree




In these diagrams we have used ▨ to designate a unique pointer value which doesn't point to anything. Such a pointer value is returned by the NULL builtin function. See LRM 18. Even though NULL( ) is a function reference, it is permitted as an initial value for a static pointer variable, e.g.,

```
DCL P PTR STATIC INIT (NULL( ));
```

The following example shows a procedure OBSERVE that maintains a list of the unique character string values that are presented to it along with a count of the number of times each has been observed. The list is initially empty. The pointer variable HEAD gives access to the list. At all times entries in

the list are maintained in "sorted" order, so that we don't generally have to scan the whole list to determine that an entry is not present. Study it carefully and convince yourself that the algorithm works in all these cases:

(a) An entry is being added to an empty list.
(b) An entry is being added before the first entry.
(c) An entry is being added after the last entry.
(d) An entry is being inserted between two existing entries.

The special requirements of these cases are as follows (check that they are met):

(a) HEAD has to be made to point to the new entry. The new entry's NEXT component must be set to the null pointer value.
(b) HEAD has to be made to point to the new entry. The new entry's NEXT component must be made to point to the previously first entry.
(c) The last entry's NEXT component must be made to point to the new entry. The new entry's NEXT component must be set to the null pointer value.
(d) The "previous" entry's NEXT component must be made to point to the new entry. The new entry's NEXT component must be made to point to the entry that was after the "previous" entry, i.e., to the "next" entry.

```
OBSERVE:  PROC (S);
     DCL S CHAR (10);

     DCL HEAD PTR STATIC EXT INIT (NULL( ));
     DCL 1 ENTRY BASED,
           2 NEXT PTR,
           2 T CHAR (10),
           2 COUNT FIXED BIN (15);
     DCL NEXT_FIELD PTR BASED;
          /* NEXT_FIELD IS A TEMPLATE GIVING ACCESS EITHER TO HEAD OR TO SOME
             ENTRY.NEXT */
     DCL (P,Q,R) PTR STATIC;
     DCL (NOT_FOUND, SEARCHING) BIT (1);

     NOT_FOUND, SEARCHING = '1'B;
     P = ADDR(HEAD);
          /* IF P STILL HAS THIS VALUE LATER, P → NEXT_FIELD ACCESSES HEAD */
     Q = HEAD;
          /* IF LIST IS EMPTY, Q HAS VALUE NULL ( ); OTHERWISE, IT POINTS TO
             FIRST ENTRY */

     DO WHILE (NOT_FOUND & SEARCHING & Q ¬= NULL( ));
          IF Q → ENTRY.T < S THEN DO; /*KEEP GOING*/
               P = ADDR(Q → ENTRY.NEXT);
                    /* REFERENCE TO P → NEXT_FIELD
                       LATER REFERENCES AN ENTRY.NEXT */
               Q = Q → ENTRY.NEXT;
          END;
          ELSE IF Q → ENTRY.T = S THEN /*FOUND IT*/
               NOT_FOUND = '0'B;
          ELSE SEARCHING = '0'B; /*WENT BEYOND*/
     END;
```

```
    IF NOT_FOUND THEN DO;
        R = Q;  /*MAY HAVE VALUE NULL( ) */
        ALLOCATE ENTRY SET (Q);
        Q → ENTRY.NEXT = R; /*CHAIN NEW TO RIGHT*/
        Q → ENTRY.T = S;
        Q → ENTRY.COUNT = 0;
        P → NEXT_FIELD = Q; /*CHAIN LEFT TO NEW*/
    END;

    Q → ENTRY.COUNT = Q → ENTRY.COUNT + 1;
END;
```

List structures may be employed in engineering applications to "model" sophisticated physical systems, such as physical or chemical structures. The relationships between data items linked by pointers represent information in an abstract sense. Exploitation of this can lead to newer, more natural ways of processing information. See LRM 221 through LRM 222.

## 11.7. Areas.

One normally has no control over where in storage a based variable is allocated. Generations of based variables could be scattered all over storage. For certain operations you would like to draw a box around a particular list structure and then treat the whole list structure (i.e., the contents of the box) as a single object. It is possible to do essentially that, by restricting certain based allocations to a particular <u>area</u> of storage and treating that area as a whole object.

For this purpose we introduce another program-control data type, "area." An area variable is declared with the <u>AREA attribute</u>, which includes an area size (which has an implementation-defined meaning; in our implementation, it is the number of bytes reserved for the area). Example:

```
    DCL A AREA (5000);
```

This declares an area variable of size 5000 bytes (plus 16 more for control information). The "value" of an area variable is its contents, including the control information.

Area variables may have any storage class, and internal or external scope; they may be parameters, elements of arrays, elements of structures, etc. Because they can be of any storage class, they can even be based.

The area size specification is the third and final type of "extent." (The other two were array bounds and string lengths.) Static area variables can have only constant extents (as is true of extents of any static variables). Area variables of the three dynamic storage classes can have their sizes given by expressions (for based areas in the current language, the REFER option must be used and the area must be a component of a structure). Area parameters may have an "asterisk extent" indicating inheritance of the extent from the

actual area argument (which may be different in different invocations).

The main purpose of an area variable is to mark off an area of storage inside which based allocations may be made. Generations of based variables in areas can also be freed. (We will see how to do these things later.) The system manages the space within an area; space which is freed can be allocated to something else. An area variable is automatically initialized to the "empty" state on allocation.

Areas may be passed as arguments and returned as function values. They may be assigned to other area variables. They may serve as record variables in record transmission statements. Movement of an area value (by assignment, record I/O, etc.) consists of the mass movement of its contents and control information, preserving intact any list structures that happen to exist within it. More on this later. See LRM 223 and LRM 224.

## 11.8. The EMPTY builtin function.

The contents of an area variable may be reset to the initial, "empty" state by assigning the value of the EMPTY builtin function to it. This has the effect of freeing all the based generations inside the area at once. See LRM 225 and the entry for EMPTY in LRM 18.

## 11.9. Area assignment and the AREA condition.

There is a certain point in each area beyond which no generations of based variables exist; beyond that point is free space. Up to that point is "used" space. Note that the used space may contain holes representing freed generations (this space, like the free space at the end, is available for subsequent allocations in the area).

When an area value (i.e., the value of an area variable or a function reference that returns an area value) is assigned to an area variable, only the used portion is copied to the target. The control information which is also moved identifies the portion of the area which is used. If the size of the target area is insufficient to contain the used portion of the area value being assigned, the AREA condition occurs.

Default status for the AREA condition is enabled; it cannot be disabled. In the absence of an established on unit, standard system action is to issue a message and raise the ERROR condition.

The AREA condition is one of the few for which a useful action is defined on normal return from an on unit. The target area reference is re-evaluated and the assignment is re-attempted. In other words, in an AREA on unit you may

free the target area and allocate a larger one, change the value of a subscript used in the target area reference, or change the value of the pointer used to locate a based area target. See LRM 226 and the entry for AREA in LRM 116.

## 11.10. Allocation in an area.

To allocate a based variable inside an area, use the IN option of the ALLOCATE statement.

```
ALLOCATE B IN (A) SET (P);
```

Here, A is an area variable. The IN option is also used in the FREE statement to denote freeing in an area:

```
FREE P → B IN (A);
```

If an attempt at allocation in an area fails (because of insufficient free space) the AREA condition occurs. On normal return from an AREA on unit entered for this reason, the allocation is reattempted after re-evaluating the area named in the IN option (which presumably has been changed in the on unit).

Question: If a list structure is built up in an area, the values of the pointer variables involved will be absolute addresses of locations inside that area; what purpose, then, can area assignment serve? Even though the based variable generations are copied in such an assignment, none of the pointer values is changed.

To overcome this problem we introduce another type of program-control data item.

## 11.11. Offsets.

An "offset" value is an address relative to the start of the storage allocated to a particular area variable. An offset variable is a variable which can hold such a value. An example of a declaration of an offset variable is:

```
DCL OFST OFFSET (A);
```

where A is an area variable.

Offset variables may be used essentially interchangeably with pointer variables. Offset and pointer values may be converted into each other. Both kinds of variables are called, because of their use and their interchangeability, locator variables.

When an offset variable is used to locate a based variable, either in explicit or implicit locator qualification (generalizing now on the earlier term

"pointer qualification"), the offset value is implicitly converted to a pointer value by adding to it the address of the area named in its declaration.

When a based allocation is made in an area, and the SET option names an offset variable, the value assigned to the offset variable is the offset of the allocated generation relative to the area. (Actually, attention may focus on several different areas here: the one named in the IN option and the one named in the declaration of the offset variable. Furthermore, either of these may be omitted and still implied by various things. See LRM 227. However, to keep things simple assume both areas are the same.)

By using the facilities described here, list structures built up within areas can be made totally relocatable, i.e., they won't contain any absolute addresses--only relative ones. Thus, the list structures retain their validity when area values are assigned, and when they are written out and later read back in (even if they are read in to a different location). Thus, whole list structures may be stored and retrieved very efficiently as records in record datasets.

See LRM 228 through LRM 231.

## 11.12. Explicit offset/pointer conversion.

Besides the implicit offset to pointer conversion discussed already, that conversion may be forced explicitly using the <u>POINTER builtin function.</u> Suppose a based variable is allocated in area A and offset variable O is set to its offset in A. Suppose the area A is assigned to B. B now contains a generation of the based variable (call it Q) at the same offset as the one in A. O may be used to locate either the one in A or the one in B. If O was declared as

```
DCL O OFFSET (A);
```

then O → Q locates the one in A because O undergoes implicit conversion to pointer relative to the area (A) with which it was declared. To locate the Q in B we may write

```
POINTER(O, B) → Q
```

or we may assign O to, say, M, declared as

```
DCL M OFFSET (B);
```

and then write

```
M → Q.
```

The <u>OFFSET builtin function</u> converts a pointer value to an offset relative to the given area. The pointer value must be an address within the area.

See the relevant parts of LRM 18.

## 11.13. Locate-mode I/O.

The kind of record I/O demonstrated in Lessons 8 and 9 is called move-mode I/O because data may be transferred, or moved, between buffers and variables in the program. (Buffers are used for blocked records and in other circumstances.) It is possible by using based variables to gain access to data right in the buffers. The technique is called locate-mode I/O because data is located directly in the buffers and not moved between them and program variables. This constitutes a use of based variables entirely distinct from list processing or system programming.

## 11.14. READ statement with the SET option.

The INTO option of a READ statement may be replaced by the SET option in the case of a sequential, keyed or non-keyed, input or update file. The SET option contains a reference to a pointer variable, e.g.,

```
READ FILE (F) SET (P);
```

The next record (or the desired record, in the case of a keyed file--i.e., when the KEY option is used) is read and left in the buffer (in the case of blocked records it was probably already there); its address is returned in the pointer variable. That pointer variable may be used to locate a based variable, the effect of which is to access the record right there in the buffer.

In the current language the file must have the BUFFERED attribute. This attribute is not in the ANSI standard, and locate-mode I/O can be done without it.

Once the next READ statement for the same file is executed, or if the file is closed, the pointer value obtained on the previous read may not be used to locate a based variable. This is because the contents of the buffer may have been changed by the subsequent read, or the buffer may have disappeared because of the file closing.

READ...SET and READ...INTO may be intermixed on the same file (as well as READ...IGNORE).

How do we know what based variable to use to look at a record in a buffer? The READ...SET, although it does generate a pointer value, is unlike the ALLOCATE statement and the ADDR builtin function because no attributes are implied for the storage whose address is being returned. If, in fact, different kinds of records can exist in the dataset, and if different based variables would be appropriate for the different records, then the program must anticipate what kind of record comes next and use the right based variable. It is illegal to use the "wrong" one because you might address storage outside the buffer, or get the wrong attributes for storage inside the buffer. The "right" based variable, of course, is one which has the same attributes and structuring as the variable from which the record was previously written.

One additional freedom is permitted to ease the burden of logically anticipating what kind of record comes next. The attributes of the based variable through which you access the record in the buffer only need to describe a "head" or initial portion of the record. I.e., if a record is really described by a structure such as

```
DCL 1 S1 BASED,
      2 CODE attr,
      2 ...
      :
      : ;
```

then it is legal to refer to that storage through a based variable declared as

```
DCL 1 S2 BASED,
      2 CODE attr;
```

In other words, the earlier rule that the attributes of the based variable must exactly match those of the generation of storage being accessed was too strong; they only need to match as far as they go. This permits the <u>beginning</u> of the record to be accessed through S2.CODE. Depending on what is found there, you may then use the same pointer value with some other appropriate based variable to access the whole record.

In other words, the structure mapping rules are guaranteed by the language to map a structure that matches the beginning of another structure exactly the same as the beginning of that other structure.

See LRM 232 and LRM 233.

Note that a REWRITE statement without the FROM option which follows a READ statement with the SET option is very efficient indeed; this is effectively a no-op. The whole buffer is eventually written back out to the dataset (as the result of executing one or more REWRITE statements for records in the buffer), but only after the whole buffer has been processed (i.e., when a subsequent read designates a record not in the buffer, or when the file is closed).

## 11.15. The LOCATE statement.

Locate-mode output is performed by executing a <u>LOCATE statement</u> instead of a WRITE statement. It applies to sequential output files which have the BUFFERED attribute.

A statement such as

```
LOCATE B FILE (F) SET (P);
```

causes a generation of storage for the based variable B to be allocated in the next available slot in the buffer for file F. The address of that generation is returned in P. P may subsequently be used to address the record in the buffer by locating B.

Notice we said B is allocated in the buffer. That means that adjustable extents are evaluated at that time and any initializations specified by the declaration of B are carried out then.

The generation of the based variable allocated in the buffer remains accessible until the next execution of either a LOCATE statement or a WRITE statement for the same file, or until the file is closed. At that time (but not before) the buffer is eligible for transmission to the dataset.

See LRM 234 through LRM 236.

11.16. Review.

New pointer values are "generated" by:
- (a) Reference to the ADDR builtin function.
- (b) Reference to the NULL builtin function.
- (c) By allocation of a based variable not in an area.
- (d) Locate-mode input (READ...SET).
- (e) Locate-mode output (LOCATE).
- (f) Conversion from an offset value.
- (g) Record input operations (the value may not be valid).

They are propagated by assignment.
They may be used in the following ways:
- (a) To locate a generation of a based variable.
- (b) In equality comparison operations.
- (c) In record output operations.

New offset values are "generated" by:
- (a) Allocation of a based variable in an area.
- (b) Conversion from a pointer value.
- (c) Record input operations (the value is valid)

They are propagated by assignement.
They may be used as follows:
- (a) To locate a generation of a based variable (after conversion to pointer).
- (b) and (c): Same as for pointer.

New area values are "generated" by:
- (a) Reference to the EMPTY builtin function.
- (b) Updating an area variable by allocating or freeing a based variable in it.
- (c) Record input operations.

They are propagated by assignment.
They may be used as follows:
- (a) To localize a based allocation.
- (b) In record output operations.

11.17. Homework problems.

(#11A) What simpler expression has the same value as ADDR(P → B)? As ADDR(X) → B?

(#11B) In the statement labeled L, is the reference to the based variable B a reference to E or to F?

```
DCL P PTR;
DCL (E,F) ...;
DCL B ... BASED (P);
P = ADDR(E);
BEGIN;
  DCL P PTR;
  P = ADDR(F);
  L:  B = B + 1;
END;
```

(#11C) What problems or errors do you see here? Assume appropriate declarations.

(a)
```
DO I = 1 TO 10;
   ALLOCATE X SET (P);
   P → X = A(I);
END;
DO I = 10 TO 1 BY -1;
   B(11 - I) = P → X;
   FREE P → X;
END;
```

(b)
```
DCL S FLOAT STATIC,
    T FLOAT BASED;
P = ADDR(S);
ALLOCATE T SET (Q);
Q → T = 20:
P → T = 35 * Q → T;
FREE Q → T, P → T;
```

(#11D) What does the compiled code have to do on any reference to P → S.U with S declared as follows?

```
DCL 1 S BASED,
      2 N FIXED BIN,
      2 T (K REFER (S.N)) FLOAT,
      2 U CHAR (1);
```

Why is the following not permitted?

```
DCL 1 S BASED,
      2 T (K REFER (S.N)) FLOAT,
      2 N FIXED BIN,
      2 U CHAR (1);
```

(#11E) Contrast based and defined variables.

(#11F) What do the following mean?

```
GO TO P → L;
CALL P → Q;
```

(#11G) Recall the example of the procedure OBSERVE in Section 11.6. A typical realization of this procedure frequently omits the based variable NEXT_FIELD and replaces two statements with others, as follows:

```
     P = ADDR(Q → ENTRY.NEXT);
 by  P = Q;
and  P → NEXT_FIELD = Q
 by  P → ENTRY.NEXT = Q;
```

Under the Optimizing compiler the modified program works and has the desired effect. In fact, it generates the same code as the one in Section 11.6. However, it is technically illegal, and won't get past the Checkout compiler. Why is it illegal? Hint: When control reaches the modified statement

```
     P → ENTRY.NEXT = Q;
```

the first time, i.e., when the first entry is being added to the (currently empty) list, to what does P really point? I.e., what are the attributes of the generation of storage to which P points? Are these the same as those of the based variable located by P? Would the modified program work if NEXT were the second or third component of ENTRY instead of the first? What about the original program?

(#11H) Suppose a call has just been made to OBSERVE. P and Q are left pointing into the list. When the next call is made to OBSERVE, they will be initialized to new values in preparation for a new traversal of the list. However, if the new character string value presented on that call collates higher than the one in the entry to which Q was left pointing, resetting P and Q turns out to be wasteful. Modify OBSERVE to do an initial test of Q → ENTRY.T against S, and avoid resetting P and Q when that is unnecessary. Make sure this works the first time OBSERVE is entered (what will Q be pointing to then?).

P and Q have already been declared STATIC anticipating this change. The original program did not require that.

(#11I) Write a procedure to traverse the list built by repeated calls to OBSERVE. At each entry, print ENTRY.T and ENTRY.COUNT. Free the entry before going on to the next one.

Code the loop using a WHILE-only DO group, i.e., DO WHILE (...); Then try to code the loop using the DO...REPEAT of the ANSI language (see Section 6.5). The form will be something like

```
DO Q = initval REPEAT (nextval) WHILE (cond);
```

What common, potential error is avoided by using this form?

(#11J) Suppose you have a card-image dataset containing a source program. Sequence information exists in columns 73-80, but is has been corrupted. Write a program that updates the dataset by replacing the contents of the sequence field of successive cards by 00000010, 00000020, etc. Use a sequential update file, READ...SET, and REWRITE without FROM. Comment on the amount of physical I/O performed. The based variable used to access a card in the buffer should be a structure. Consecutive sequence numbers can be generated conveniently by using a numeric picture variable (see Lesson 2).

(#11K) The technique demonstrated in Section 11.14 for decoding a record whose address has been supplied by a READ...SET can be avoided if the "record type code" for a record is kept in the previous record. Then, <u>every</u> access to a given record can be made by using the correct based variable. (The program has to have some convention about the <u>first</u> record, however.)

Let us focus on the creation of such a dataset, i.e., one in which each record contains information about the "type" of the next record in sequence. Suppose that a program which writes such a dataset cannot know the type of a record to be produced until it is finished producing the previous one. What feature of locate-mode output (LOCATE) permits the "type" of the next record to be put in the previous record after it is logically completed?

(#11L) Consider the use of a based self-defining structure to represent character string data of fixed, but adjustable, length. Different generations of the based variable will contain character string values of different lengths. What advantage is gained by representing the data this way, instead of using a based varying-length string with a <u>fixed</u> maximum length? Show a suitable declaration of of such a structure. Write a procedure which accepts a pair of pointers to two generations of such a based variable, allocates a third whose character string part contains the concatenation of their character string parts, frees the two generations, and returns a pointer to the new generation.

12. (a) Miscellaneous features.
(b) Preprocessor.

This lesson deals almost exclusively with useful features of our implementation which have not been standardized by ANSI.

## 12.1. DISPLAY statement.

The DISPLAY statement allows communication with the operator in the form

DISPLAY(*expr*);

the value of *expr* is converted (if necessary) to character and written on the operator's console.

In our environment messages to the operator are not encouraged and really serve no useful purpose. However, they are copied to a job's SYSMSG output, which may be useful. It is probably a good idea, for instance, to open file SYSPRINT explicitly very early in the execution of a program just to know that it is definitely "available" for program output and system error messages. Before opening the file, an UNDEFINEDFILE on unit should be established for SYSPRINT. If this on unit should be entered it means there is no way the system will be able to deliver PL/I error messages to the user in the normal way. The on unit can explain that to the user, via SYSMSG, by executing some DISPLAY statements. (The operator probably won't even notice...)

By using the REPLY option on the DISPLAY statement, the program will print a message to the operator, then wait for his reply. The reply, when issued, is assigned to the character string variable named in the REPLY option. You should not use this form here without submitting special instructions with your job; even then, you cannot count on the operator remaining free enough to notice your message and reply to it in a timely fashion. You are charged for the WAIT time accrued while waiting for the reply.

See LRM 237.

## 12.2. FETCH and RELEASE statements.

An external procedure link edited into a program occupies core storage for the duration of the program's execution, even if it is rarely (or, in the extreme case, never) invoked. Better use of core storage can often be made either by employing overlay structures in the load module (see OPG 26 and CPG 25) or by using FETCH and RELEASE statements.

An external procedure named in a FETCH or RELEASE statement is not link edited in with the rest of the program (no external reference is generated). Rather, it is loaded into core on execution of a FETCH statement and deleted on execution of a RELEASE statement. Execution of a CALL statement naming the procedure causes it to be loaded before being invoked if it is not already in core.

"Fetchable" external procedures must be declared with the attributes ENTRY EXTERNAL like any other external procedures. They are known as fetchable procedures by virtue of the appearance of their names in FETCH or RELEASE statements, or both. The entry names appearing in these statements must be entry constants; they cannot be entry variables. The facility is very limited and has many restrictions. See LRM 238 through LRM 241. JCL considerations will be discussed in Lesson 13.

## 12.3. PLIRETC builtin procedure.

PLIRETC is the first of several builtin procedures defined by this implementation. A builtin procedure is like a builtin function except that it is invoked by a CALL statement. Its name is known to the compiler and generally doesn't have to be declared. See Lesson 10.

The PLIRETC builtin procedure allows you to set a step return code which can be tested in JCL to determine whether a succeeding job step should be executed or bypassed. For instance,

```
CALL PLIRETC(8);
```

sets a step return code, or completion code, of 8. The user must restrict himself to codes between 1 and 999. If a job

terminates abnormally (see the discussion in Section 6.9), a return code of 1000 or 2000 will be added to the value set by the programmer. If the environment becomes hopelessly destroyed, a code of 4000 or higher, signifying total disaster, will be returned. See CPG 3 and OPG 3, also CPG 26 and OPG 27. If a job terminates normally and the programmer has not set a return code, 0 is returned.

## 12.4. PLISRTx builtin procedures.

This implementation also provides direct and dynamic access to the system SORT utility via four builtin procedures, PLISRTA through PLISRTD. These are completely described in CPG 27 and OPG 28.

## 12.5. Other facilities.

This implementation has builtin procedures for access to the system Checkpoint/Restart facilities, but these are not implemented in our system.

Other facilities useful primarily in debugging will be described in Lesson 13.

## 12.6. The preprocessor.

IBM's PL/I has always had a preprocessor or compile-time facility that allows the programmer to write macros, arrange for text substitutions in his source program during compilation, compile certain parts of the program conditionally, etc. Perhaps due to some of the inadequacies of the preprocessor, the rest of the world has not considered it to be a part of PL/I. Other vendors have not implemented it, and it is not in the ANSI standard.

The compile-time facility is not invoked unless certain compiler options, discussed in Lesson 13, are elected. We will assume in this lesson that the necessary options have been turned on.

The preprocessor can be used advantageously for simple purposes such as systematic changing of identifiers, parameterization of a program, and introduction of personal abbreviations, or for more advanced purposes, such as the wholesale mechanical generation or derivation of programs from minimal specifications.

See LRM 242.

12.7. The preprocessor scan.

The preprocessor, when invoked, "works on" the source program before the compiler proper sees it. The output of the preprocessor is what gets compiled.

The preprocessor scans the source program for preprocessor statements, which are executed when they are encountered and not transmitted to the output, and "active" identifiers, which are replaced in the output by some replacement text. Any part of the source program scanned in this process which is not a preprocessor statement or an active identifier is carried through intact to the output.

Every preprocessor statement starts with a percent sign (%) and ends with a semicolon. Each preprocessor statement type has, furthermore, a particular syntax. In other words, once the preprocessor encounters a % in its scan, what follows up to the next semicolon must be a syntactically valid preprocessor statement. Outside of preprocessor statements, however, anything goes. The text outside of preprocessor statements is "atomized" into identifiers, constants, comments, parentheses, commas, and "everything else," but that is all; in other words, these atoms need not (at this stage) be related by any higher level syntax. The sole purpose of this atomization is to be able to detect active identifiers and preprocessor statements without confusing them with parts of constants (e.g., the E in 5E-03 will never be taken for an active identifier) or with the contents of character string constants or comments. See LRM 243.

12.8. %DECLARE statement.

Initially, identifiers in the source program are inactive

and thus not subject to replacement. When the preprocessor scan encounters a %DECLARE statement, the named identifiers are activated.

An active identifier, declared in a %DECLARE statement, represents a preprocessor variable. When it appears subsequently in source text outside of preprocessor statements it is replaced by its value. The mechanism for assigning values to preprocessor variables will be shown shortly.

Preprocessor variables make take on integer numeric or character string values only. The %DECLARE statement, in addition to activating an identifier as a preprocessor variable, assigns it some attributes used to describe the kinds of values it may acquire. The two kinds of values are respectively declared by the FIXED and CHARACTER attributes. No other attributes may be included. A FIXED preprocessor variable behaves like a FIXED DECIMAL (5,0) PL/I variable; a CHARACTER proprocessor variable behaves like a CHARACTER VARYING PL/I variable with no maximum length. %DECLARE statements can be used to declare certain other objects, too, as we will see later. See LRM 244.

## 12.9. % assignment statement.

The preprocessor assignment statement is used to assign a value to a preprocessor variable. The form is

% *variable* = *expression*;

The *expression* cannot have the full generality of PL/I expressions. Its operands can be only preprocessor variables, preprocessor function references (see below), decimal integer constants, string constants, and certain builtin function references. The exponentiation operator is not allowed. The operands of arithmetic operators are converted, if necessary, to FIXED DECIMAL (5,0). All arithmetic is performed in this precision; note, therefore, that division behaves more like FORTRAN integer division than regular PL/I fixed-point division.

The *expression*, called a preprocessor expression, is evaluated and its value is assigned to the *variable* whenever the preprocessor scan encounters the % assignment statement. See LRM 245 and LRM 246.

Note that this is a preprocessor statement, hence no replacement activity is triggered by the appearance of an active identifier in it. The identifier is used in the way dictated by the particular preprocessor statement.

## 12.10. Rescanning and replacement.

When an active identifier which is the name of a preprocessor variable is encountered outside of preprocessor statements during the preprocessor scan, it is removed from the source text and its current value replaces it in the output. If the preprocessor variable has the FIXED attribute, its value is converted from FIXED DECIMAL (5,0) to CHAR (8) for this purpose.

Before the replacement value is placed into the output it is, in general, first rescanned for other possible active identifiers. Replacement of them, and rescanning, continues until no further active identifiers remain in the value; it is then placed in the output text. The rescanning of the replacement value of an active identifier can be suppressed, as explained below.

Example:

```
        % DCL A CHAR, B FIXED:
        % A = 'C+B';          The value of A is now the
                               3-character string C+B.
        % B = 9;              The value of B is now 9.
          X = A+E;             This text lies outside of pre-
                               processor statements. The
      after                    identifier A is active. It is
    initial                    removed and replaced by its
replacement                    replacement value, which is then
                               rescanned.
          X = C+B+E;           The replacement value, C+B, con-
                               tains an active identifier, B.
      after                    It is removed and replaced by
  rescanning                   its replacement value, which is
                               converted to CHAR (8) for this
                               purpose.
          X = C+_____9+E;      The final result is as shown.
```

Had B been declared as CHAR instead of FIXED, and had its value been assigned by

```
% B = '9';
```

then the final result would have been

```
X = C+9+E;
```

12.11. % DEACTIVATE statement.

The % DEACTIVATE statement (abbreviated % DEACT) makes a preprocessor variable inactive. When its name is encountered subsequently, no replacement activity occurs. The variable retains its value, because it may be reactivated. See LRM 247.

12.12. % ACTIVATE statement.

When the preprocessor scan encounters a % ACTIVATE statement (abbreviated % ACT), the identifier is again activated for replacement. One of two options, RESCAN and NORESCAN, may be included. The RESCAN option (which is the default if both are omitted) specifies that the replacement value of the active identifier is to be rescanned for possible additional replacement activity before being placed in the output. This is also the behavior described above for identifiers initially activated by the % DECLARE statement. The NORESCAN option says that the replacement value is to be placed in the output text without rescanning for further possible replacements.

Example: The "expansion" of

```
% DCL (A,B) CHAR;
% A = 'C+B';
% B = 'D';
  X = A+E;
% DEACT A;
  X = A+E;
% ACT A NORESCAN;
  X = A+E;
```

yields

```
  X = C+D+E;
  X = A+E;
  X = C+B+E;
```

See LRM 248 through LRM 250.

## 12.13. % IF statement.

The <u>% IF statement</u> has one of the forms

```
% IF preprocessor-expr % THEN true-part;
```

or

```
% IF preprocessor-expr % THEN true-part;
% ELSE false-part;
```

The *true-part* and *false-part* must be single preprocessor statements or preprocessor DO groups (see below).

The *preprocessor-expr*, which is just like an expression on the right-hand side of a % assignment statement, is evaluated and converted to a bit string. The bit string is interpreted as "true" or "false" in the same way as for normal PL/I IF statements (see Lesson 6.) The preprocessor scan resumes at the *true-part*, or the *false-part*, or the text after the *true-part* if the expression is false and there is no % ELSE clause.

Examples:

```
% IF A=B+1 % THEN % A=A-1;
% IF B<C&C=D % THEN
   % IF WORD = 'STOP' % THEN % WORD = '';
   % ELSE % WORD = WORD || NEXT;
```

See LRM 251 and LRM 252.

## 12.14. % DO statement.

<u>Preprocessor DO groups</u> may be of the non-iterative kind, % DO; ...; % END; or the iterative kind. In the latter case only the controlled, or indexed, type of group with one specification is allowed.

The non-iterative preprocessor DO group is particularly useful with % IF statements. The contents of the DO group may be a mixture of preprocessor statements and non-preprocessor text.

Example:

```
% IF TYPE = 'TEST' % THEN % DO;
   PUT FILE (SYSPRINT) DATA (X,Y,Z);
% END;
```

As a result of the above, the PUT statement is generated in the source program if the preprocessor variable TYPE has the value TEST.

```
% DCL (I,J) FIXED;
% DO I = 1 TO 5;
  % J = 2*I + 5;
  A(I) = B(J);
% END;
This generates:
  A(      1) = B(      7);
  A(      2) = B(      9);
  A(      3) = B(     11);
  A(      4) = B(     13);
  A(      5) = B(     15);
```

See LRM 253 and LRM 254.

12.15. % GO TO statement.

The preprocessor GO TO statement causes the preprocessor scan to be resumed from a different point in the source program.

Any preprocessor statement may have a label. The label, and its following colon, are placed between the percent sign and the statement keyword. E.g.,

```
% LAB: A = B;
% LAB1: IF A < B % THEN % GO TO LAB;
```

See LRM 255 and LRM 256.

12.16. % null statement.

The preprocessor null statement, which looks like

```
%;
```

can be used to match nested % ELSE clauses against the proper % IF, as in

```
% IF ... % THEN
   % IF ... % THEN ...;
   % ELSE %;
% ELSE ...;
```

It can also be used to insert a label anywhere to serve as the target of a preprocessor GO TO statement. Example:

```
% DCL I FIXED;
% I = 0;
% L: ;
  :
% IF I < 10 % THEN % GO TO L;
```

See LRM 257 and LRM 258.

## 12.17. Preprocessor procedures.

The preprocessor features we have seen so far allow for simple calculations, simple replacement of identifiers, and conditional or unconditional redirection of the preprocessor scan. Preprocessor procedures permit more complex flow patterns to be set up during the preprocessor scan, and they allow functions of arguments to be computed during compilation.

A preprocessor procedure is like a normal function procedure, but it can be invoked only at compile time. Both the PROCEDURE statement and matching END statement must be marked by leading percent signs. Statements in the body of the procedure are interpreted as preprocessor statements but their percent signs are omitted. Only the statements described above can be used in preprocessor procedures, plus the RETURN statement. Preprocessor procedures may not be nested.

Declarations made inside a preprocessor procedure obey the normal scope rules for internal names, i.e., the items declared are not known outside the procedure. Variables declared in a preprocessor procedure behave as if they had static storage class; that is, they retain their former value across invocations of the procedure. A preprocessor procedure may also reference preprocessor variables declared outside the procedure (their scope is the whole source program, except preprocessor procedures in which they are redeclared).

A preprocessor procedure must return a value (which may be of type FIXED or type CHAR). Therefore:

The PROCEDURE statement must include RETURNS(FIXED) or RETURNS(CHAR).
The procedure can only be invoked by a function reference.
It must execute a RETURN statement containing an expression for the returned value.

The parameters of a preprocessor procedure are declared, inside the procedure, in the normal way. It is interesting to note that the number of arguments supplied in an invocation of a preprocessor procedure need not match the number of its parameters. Excess arguments are ignored; excess parameters are initialized to 0 or the null string depending on their attributes (FIXED or CHAR, respectively).

A preprocessor procedure may be invoked either from a preprocessor statement (in which its name appears in a function reference in a preprocessor expression) or from non-preprocessor text. We will examine these cases separately.

When a preprocessor procedure is invoked from a function reference in a preprocessor expression in a preprocessor statement, the association of arguments and parameters occurs in the normal way, and the returned value is used in the normal way in the preprocessor expression. The arguments in the function reference must all be preprocessor expressions. Dummy arguments are created, as usual, if conversion is required to match the data type of the argument to that of the parameter.

Example:

```
% DCL (A,B,C) FIXED;
% P: PROC (X,Y) RETURNS (FIXED);
     DCL (X,Y) FIXED;
     IF X >= 0 THEN Y = B - 1;
     RETURN (Y*X - A);
% END;
% A = 3;
% B = 8;
% C = 10;
% A = B + P(A+1, C);
% A = B + P(A-40, C);
```

On the first invocation of P, the parameters X and Y have the values 4 and 10. The IF statement references B, declared outside of P; it sets Y (and hence C) to B-1, i.e., 7. The RETURN statement also references a variable, A, declared outside of P. It returns the value 7*4-3, or 25. The % assignment statement that invoked P thus assigns

8 + 25, or 33, to A. A, B, and C now have values 33, 8, and 7. On the second invocation of P, X and Y have values -7 and 7. Y, and thus C, are not altered further by the IF statement. The value returned is 7*(-7)-(-7), or -42. The final values of A, B, and C are thus -34, 8, and 7.

Somewhat different rules apply to the invocation of a preprocessor procedure when its name appears with an argument list as a function reference in non-preprocessor text. The general idea is that the returned value replaces the function reference. However, this replacement activity only occurs if the procedure name is active. Preprocessor procedure names are activated by their appearance, during the preprocessor scan, in a % DECLARE statement with the ENTRY attribute, or by their appearance in a % ACTIVATE statement. The concept of rescanning applies to replacement values of preprocessor procedure references just like it does to replacement values of preprocessor variables. The % DEACTIVATE statement is used to prevent the name of a preprocessor procedure from initiating replacement activity in non-preprocessor text; the procedure is not even invoked when it is inactive.

Perhaps the greatest difference between the two environments in which preprocessor procedures can be invoked lies in the interpretation of the argument list. In non-preprocessor text, the rules for argument lists of preprocessor function references are as follows. The text between consecutive "unprotected" commas (or between one of these commas and the parenthesis at either end of the argument list, or between the parentheses when there are no commas) is considered to be an argument. The literal sequence of characters comprising the argument is scanned for active identifiers (and active procedure references, too!); replacements are performed and rescanned if indicated; and when no further replacement activity can be performed the resulting sequence of characters is considered to be a character string valued argument, and that is what is associated with the parameter. In the case of a FIXED parameter, the character string value of the argument is converted. In any case, a dummy is made.

By "unprotected comma" we mean a comma not inside character string delimiters, comment delimiters, or balanced parentheses. This rule is required in order to recognize another function reference in the argument list. I.e., in

P(Q(A,B))

we have one argument for P, "Q(A,B)", not two, "Q(A" and "B)".

Although this may seem obvious, recall that very little syntax is imposed on non-preprocessor text during the preprocessor scan.

See LRM 259 through LRM 261.

## 12.18. An example.

Suppose we wish to code a table, in a PL/I program, as a static initialized array of structures, for example:

```
DCL 1 TABLE (4) STATIC,
      2 HEIGHT FLOAT INIT (3, 1.5, 1.5, 0.5),
      2 RADIUS FLOAT INIT (.32, .15, 1, .8),
      2 POLISHED BIT(1)
          INIT ('1'B, '1'B, '0'B, '0'B),
      2 STYLE CHAR(1)
          INIT ('A', 'L', 'E', 'A');
```

Each element of the array TABLE is a structure carrying four properties of a cylindrical object. For instance, TABLE(3) is an entry describing a single object having height 1.5, radius 1, style 'E', and which is not polished.

The problem of maintaining such a table quickly becomes tedious. Each time we wish to add a new entry we have to increase the upper bound in the first line and change four "initial" lists. When these have become long enough to be spread over several lines, it then becomes difficult to tell, at a glance, what all the properties of the i-th entry are.

We will define some preprocessor variables and procedures that permit us to produce the table simply by writing

```
TABLE(3,     .32,    '1'B,    'A')
TABLE(1.5,   .15,    '1'B,    'L')
TABLE(1.5,   1,      '0'B,    'E')
TABLE(0.5,   .8,     '0'B,    'A')
END_TABLE
```

It is now obviously easy to make a new entry in the table, and the properties of an entry can be seen at a glance.

The following declarations and definitions suffice. Five "global" preprocessor variables are declared and initialized. Procedures TABLE and END_TABLE are defined, plus another, APPEND, which is invoked from inside TABLE. Note

that END_TABLE must be activated for NORESCAN so that its replacement value, which contains the identifier TABLE, will not be rescanned. the purpose of the TABLE procedure is merely to append the values of its four parameters to four global variables which END_TABLE will use to "emit" the four lists of initial values. TABLE itself generates a null string for replacement value.

```
% DCL (HEIGHT_INIT,
      RADIUS_INIT,
      POLISHED_INIT,
      STYLE_INIT) CHAR;
% DCL #ENTRIES FIXED;
% #ENTRIES = 0;
% HEIGHT_INIT = '';
% RADIUS_INIT = '';
% POLISHED_INIT = '';
% STYLE_INIT = '';
% TABLE: PROC (HEIGHT, RADIUS, POLISHED, STYLE)
            RETURNS (CHAR);
    DCL (HEIGHT,
         RADIUS,
         POLISHED,
         STYLE) CHAR;
    #ENTRIES = #ENTRIES + 1;
    HEIGHT_INIT = APPEND(HEIGHT_INIT, HEIGHT);
    RADIUS_INIT = APPEND(RADIUS_INIT, RADIUS);
    POLISHED_INIT = APPEND(POLISHED_INIT, POLISHED);
    STYLE_INIT = APPEND(STYLE_INIT, STYLE);
    RETURN ('');
% END;
% APPEND: PROC (INIT_LIST, ITEM) RETURNS (CHAR);
    DCL (INIT_LIST, ITEM) CHAR;
    IF #ENTRIES > 1 THEN
       INIT_LIST = INIT_LIST || ',';
    RETURN (INIT_LIST || ITEM);
% END;
% END_TABLE: PROC RETURNS (CHAR);
    RETURN ('DCL 1 TABLE ('
        ||    #ENTRIES
        ||    ') STATIC,'
        ||  '2 HEIGHT FLOAT INIT ('
        ||     HEIGHT_INIT
        ||     '),'
        ||  '2 RADIUS FLOAT INIT ('
        ||     RADIUS_INIT
```

```
            ||      ),'
            ||  '2 POLISHED BIT (1) INIT ('
            ||      POLISHED_INIT
            ||      ),'
            ||  '2 STYLE CHAR (1) INIT ('
            ||      STYLE_INIT
            ||      );'    );
% END;
% ACTIVATE TABLE NORESCAN,
           END_TABLE NORESCAN;
```

12.19. % INCLUDE statement.

It is frequently extremely useful to be able to include text from a library into a source program. For instance, common declarations need not always be written out but may be included from a library. This is particularly valuable when the declarations are those of external variables and need, therefore, to be exactly the same in all external procedures containing them.

For the syntax of the % INCLUDE statement, see LRM 262 and LRM 263. JCL considerations will be taken up in Lesson 13.

The facility provided by the % INCLUDE statement was recently added to the ANSI version of PL/I.

12.20. Builtin functions available in the preprocessor.

The LENGTH, SUBSTR, and INDEX builtin functions may be used inside preprocessor procedures and elsewhere in preprocessor expressions. They may be used in non-preprocessor text only if they are specifically declared BUILTIN in a % DECLARE statement (which also activates them). Example:

```
% DCL SUBSTR BUILTIN;
% DCL S CHAR;
% S = 'STRING';
  X = SUBSTR(S, 3);
% DEACT SUBSTR;
  X = SUBSTR(S, 3);
% S = SUBSTR(S, 3);
  X = S;
```

generates the following:

```
  X = RING;
  X = SUBSTR(STRING, 3);
  X = RING;
```

See LRM 264.

12.21. Homework problems.

(#12A) What do you expect to happen here?

```
% P: PROC (S) RETURNS (FIXED);
     DCL S FIXED;
     RETURN (S + 1);
% END;
% ACT P;
  P(X)
```

(#12B) This widely circulated puzzle has an absurd answer. What do you have to write in place of the "?" so that the program will print out a single quote? In particular, how many single quotes?

```
PROG: PROC OPTIONS (MAIN);
   % DCL S CHAR;
   % S = ? ;
   DCL C CHAR (100) VAR;
   GET STRING (S) LIST (C);
   PUT FILE (SYSPRINT) EDIT (C) (A);
END;
```

Actually, the key to getting the correct answer doesn't have much to do with the preprocessor.

(#12C) What happens here?

```
% DCL I FIXED;
% I = 0;
% L: ;
  A(I) = A(I) + 1;
% IF I > 0 % THEN % I = I + 1;
% GO TO L;
```

(#12D) How can preprocessor variables and procedures "stack" information at compile time? For what kinds of "language extensions," implemented with preprocessor facilities, might this capability be useful?

(#12E) Suppose a preprocessor procedure, P, has one parameter declared with the CHAR attribute. What is the value of that parameter, on entry to P, when P is invoked with the argument list shown

```
P('ABC')
```

(a) in a preprocessor statement?
(b) in non-preprocessor text?

(#12F) Write a "macro" (preprocessor procedure) called STRG, meant to be used in DECLARE statements as follows:

```
DCL C STRG('ABCDEFGH');
      generates
DCL C CHAR (8) INIT ('ABCDEFGH');
         and
DCL D STRG('ISN''T');
      generates
DCL D CHAR (5) INIT ('ISN''T');
```

Note the string length (5) in the second expansion.

(#12G) Write a macro, HEX, that translates HEX('12FC') into '0001001011111100'B, etc.

. (a) Advanced JCL and compiler options.
(b) Program development and debugging.

In this lesson we will explore some of the non-language related features of our implementations of PL/I that enhance the useability of the language. In addition, we will consider some of the extensions to the language, present in our implementations, which aid in the debugging process.

## 13.1. Organization of the Checkout compiler.

The Checkout compiler is designed to meet the requirements of the program testing and debugging part of the program development cycle. It is not intended for the generation and running of production code.

The compiler is organized as a translator and interpreter. The translator phase replaces the conventional compilation phase. Its goal is to produce intermediate output for the interpretation phase. The intermediate output is a coded representation of the source program that permits the interpreter to "execute" the program without repeatedly scanning and parsing the source, applying defaults, etc. The translation phase is generally faster than a traditional compilation because less work is performed; optimized machine code is not produced. The translator concentrates on reporting source program errors in helpful, high-level terms. It also repairs syntax errors very effectively.

The interpreter phase, on the other hand, is much slower than execution of a program from machine code. This is justified because one generally makes very few passes of a program though the interpreter during the program's development; most of the program's useful life will be represented by optimized production runs. The interpreter does far more "consistency" checking than it would be profitable for generated machine code to do. It is capable of detecting errors that would go undectected in an optimized, production version of a program and which could lead to unpredictable program failures ranging from wrong results to an abend (abort). Furthermore, errors are detected and reported as soon as they occur; in an optimized machine code environment the observable effects of such errors are often far removed in time from their causes, making debugging hopelessly difficult. Finally, because the interpreter

has the complete coded form of the source program available to it, it is able to report errors in very high level source program terms. As an example, the message for an out-of-bounds subscript value tells you the name of the array; the number of the dimension involved; the value of the subscript itself; and, if the subscript value was supplied by a simple variable, the name of that variable. You are also told the statement number of the statement containing the error.

The multiple functions of the Checkout compiler are reflected in the variety of ways it can be used. The most straightforward mode of use is employed when a single external procedure, a main procedure which doesn't need any subroutines, is to be translated and interpreted. (Note: builtin functions, whether supported by "library routines" or not, are not considered to be subroutines in this context.) In this mode the translator produces its output directly in core (some of it may spill onto a temporary dataset), and the translation phase is followed immediately, in the same job step, by the interpretation phase. Thus, neither the linkage editor nor the loader is used. See CPG 28. This mode of use is called "compile and go"; like the usual "compile, load and go" mode of other compilers, no "object module" survives after the run.

It is also possible to translate an external procedure and save the output of translation for later execution by the interpreter phase. This mode of use is mandated by the need to link-edit (or load) separately translated external procedures together to resolve external references. It is also required when you need to link-edit in AMDLIB routines or FORTRAN routines.

To support this mode of use, it is possible to request output from the translator. Normally, output from a compiler is in the form of an "object module" to be used as subsequent input to the linkage editor or loader. In the case of the Checkout compiler, translator output consists of two separate parts: a normal object module and the intermediate text. The object module contains a minimum of information and is much smaller than usual. Called a "link-edit stub," it basically contains the information needed by the linkage editor to resolve external references, and it contains a little bit of executable machine code. The intermediate text contains most of the information about the external procedure in coded form. It is used

subsequently only by the interpreter; it is not passed through the linkage editor or loader. See CPG 29.

The output described above is produced by the translator in response to the OBJECT compiler option. If you use the appropriate cataloged procedures, such as PLCCP (described in the next section), this option is supplied automatically and you need not concern yourself with it. (Another option, NORUN, is also supplied to tell the compiler to stop after the translation phase rather than go on into interpretation.) The object module output (link-edit stub) is "captured" in the normal way by a SYSOBJ DD card defining a sequential dataset or a member of a partitioned dataset. The intermediate text output is captured by a SYSITEXT DD card defining a partitioned dataset (not a member thereof). This dataset has no counterpart in other IBM compilers. The intermediate text for a given external procedure is stored as a member whose member name is derived from the external procedure name and supplied automatically by the compiler.

The collection of object modules is next processed either by the linkage editor, to form a load module, or by the loader. After that, execution is initiated in the normal way. If the linkage editor has been used, the load module is invoked in a separate job step. If the loader has been used, the loader initiates execution in the same job step in which it resolves external references. In either case, in the job step in which execution takes place the partitioned dataset containing the intermediate text modules created by the translator must be made available via a DD card for SYSITEXT. When execution begins, the executable machine code in the link-edit stub for the main procedure receives control. What it does is invoke the interpreter phase of the Checkout compiler. All of these things are quite transparent when you use the appropriate cataloged procedures.

See CPG 30.

## 13.2. Cataloged procedures for the Checkout compiler.

Each Programmer's Guide contains a chapter on the IBM-supplied cataloged procedures for the compiler in question. Note that we do <u>not</u> use the IBM-supplied cataloged procedures here.

Rather, we use our own. These are tailored somewhat to our environment. In addition, we have arranged to offer a similarly named family of procedures for each compiler.

The family prefix for the Checkout compiler is PLC. Members of the standard family available in the PLC series are PLCCLG, PLCCEG, PLCCP, PLCCEP, PLCC, PLCEP, PLCEG, and PLCLG. One member, PLCCD, is not available because it is not possible to obtain an "object deck" from the Checkout compiler. The PLC series includes two members not in the standard family: PLCCG and PLCG. All are briefly described below.

Step names used in the cataloged procedures are as follows:

- PLC - Translate only. Compiler options OBJECT and NORUN are supplied automatically to cause the translator phase to produce output, then stop.
- EDT - Link-edit step.
- GO - (a) Execution of link-edited program.
  (b) Substitute the loader for the linkage editor, and go right into execution.
  (c) In procedure PLCCG, translation is immediately followed by execution in the single step named GO.

The steps present in each of the procedures are indicated in the following table.

| | PLC | EDT | GO |
|---|---|---|---|
| PLCCG | | | See note |
| PLCCLG | √ | | √ |
| PLCCEG | √ | √ | √ |
| PLCCP | √ | | |
| PLCCEP | √ | √ | |
| PLCC | √ | | |
| PLCEP | | √ | |
| PLCEG | | √ | √ |
| PLCLG | | | √ |
| PLCG | | | √ |

Note: The single step, GO, in PLCCG combines the classical functions of the PLC and GO steps in one step.

The purpose of each procedure is briefly described here.

PLCCG: Translate and interpret a self-contained program in the form of a single external procedure (a main procedure).

PLCCLG: Translate, load, and interpret. This is used if several external procedures are being translated and linked together by the loader. (Note: how several external procedures can be translated in a single PLC step is described later.) There may be other requirements dictating the use of the loader, even if only one external procedure is being translated. PLCCLG can be used where PLCCG will suffice, but resources will be wasted.

PLCCEG: Translate, link edit, and interpret. The linkage editor provides certain services not provided by the loader, however, it is difficult to imagine how these could be of use when the linkage editor output is not saved.

PLCCP: Translate only. The user must capture object module output via PLC.SYSOBJ and intermediate text output via PLC.SYSITEXT.

PLCCEP: Translate and link edit. The user must capture intermediate text output via PLC.SYSITEXT and load module output via EDT.SYSPVT.

PLCC: Translate only, with object module and intermediate text output passed in temporary datasets to another job step.

PLCEP: Link edit only. The user must supply input to the linkage editor (the result of a previous translation) via EDT.SYSIN and capture its load module output via EDT.SYSPVT.

PLCEG: Link edit result of previous translation, supplied via EDT.SYSIN, and interpret it. Again, it is doubtful that the special services offered by the linkage editor, but not the loader, are useful in this context. The user must supply the intermediate text resulting from the previous translation via GO.SYSITEXT.

PLCLG: Process the result of previous translation, supplied via GO.LDRIN, through the loader and interpret it. The user must supply the intermediate text resulting from the previous translation via GO.SYSITEXT.

PLCG: Interpret a previously translated and link-edited program. The user supplies the load module library via GO.STEPLIB and uses the symbolic parameter PROGRAM to name the member to be executed. In addition, the intermediate text is supplied via GO.SYSITEXT.

Typical uses of PLCCG and PLCCLG, which are the most likely to be needed, were shown in Lesson 0. Other information may be found in OTHER 3, and in other publications and courses of the Computer Center.

## 13.3. Source input conventions.

The traditional ddname for source input to compilers is SYSIN. SYSIN may be used also for data input to your program; recall from Lesson 7 that SYSIN is one of the standard files. The dual functions of SYSIN pose problems for the "compile and go" mode of operation: how can both functions be accommodated in a single job step? The Checkout compiler solves this problem by providing two different ddnames for the two functions. SYSCIN ("compiler input") is for source input to the translator, leaving SYSIN for data input to the program during interpretation.

Actually, other solutions to the problem are available also. If you prefer, you may use the traditional SYSIN for source input (instead of the new SYSCIN). If you happen to have data input also, you follow the source program by a control card containing

*DATA;

starting in column 1 and follow that by the data. Finally, you may supply both source and data, separate by a *DATA statement, in SYSCIN. The three choices are demonstrated below.

```
// EXEC PLCCG
//GO.SYSCIN DD *
    source
/*
//GO.SYSIN DD *
    data
/*
```

```
// EXEC PLCCG
//GO.SYSIN DD *
    source
*DATA;
   data
/*
```

```
// EXEC PLCCG
//GO.SYSCIN DD *
    source
*DATA;
   data
/*
```

See CPG 31 and CPG 32. The latter reference also describes how the program can be executed with several different sets of data all in one job step and without retranslating it.

## 13.4. Translating several external procedures at once.

If several external procedures are to be translated, linked together, then executed, it is not necessary to execute one or more PLCC cataloged procedures followed by a PLCCLG cataloged procedure, with each cataloged procedure translating a single PL/I external procedure. You can make do with a single invocation of PLCCLG. All of the external

procedures are translated in the single PLC step. They are separated in the source input dataset by a control card containing

```
*PROCESS;
```

starting in column 1, as shown in Lesson 0. See CPG 33 and CPG 34.

When you use PLCCG, you may actually translate and interpret several different complete one-procedure programs in a single invocation of the compiler. As CPG 34 demonstrates, you can have separate data for each program by coding

```
// EXEC PLCCG
//GO.SYSCIN DD *
    source 1
*DATA;
    data 1
*PROCESS;
    source 2
*DATA;
    data 2
etc.
/*
```

or you can use the same data for all by coding

```
// EXEC PLCCG
//GO.SYSCIN DD *
    source 1
*PROCESS;
    source 2
etc.
/*
//GO.SYSIN DD *
        common data
/*
```

All sorts of intermediate combinations are possible.

13.5. Checkout compiler options.

See CPG 35 and CTUG 4 for a complete description of compiler options. Note, however, that in some cases our installation defaults differ from the IBM defaults. A list of our local defaults, reprinted from OTHER 3, is attached to these notes.

Certain compiler options are effective during translation, while others apply during interpretation; some apply during both. Two symbolic parameters are provided in our cataloged procedures to pass options to the compiler: OPTIONS is to

be used for translate (PLC) steps and GOOPTS for interpret (GO) steps. (Both are defined in the single-step procedure PLCCG.) A simple example of their use in an EXEC statement follows:

```
// EXEC PLCCLG,OPTIONS='FORMAT',GOOPTS='ERRORS(20)'
```

Cataloged procedures with translate-only (PLC) steps supply OBJECT and NORUN for you; whatever you may specify via OPTIONS supplements these.

Compiler options may also be specified on a *PROCESS statement as described in CPG 33. These modify the options specified in the symbolic parameters, or defaulted, for the following external procedure only.

You may pass an argument to your main procedure. The parameter must be declared as CHAR (100) VAR (see LRM 265). The argument is supplied via the GOPARM symbolic parameters, as in

```
// EXEC PLCCLG,GOPARM='3,UPDATE'
```

If you have no argument to pass in, and indeed have no parameter in the main procedure to receive one, you will nevertheless have to suffer message

```
IEN1207I AN ARGUMENT IS BEING PASSED TO MAIN PROCEDURE
         XXX, BUT THE PROCEDURE HAS NO PARAMETER LIST.
         ARGUMENT IGNORED.
```

This occurs because our cataloged procedures make it look like a null string is being passed in as an argument when you do not use the symbolic parameter GOPARM.

An argument to the main procedure may also be specified on a *DATA statement as described in CPG 32.

## 13.6. Specific Checkout compiler options.

We cannot hope to describe all the available options. However, a few will be mentioned here and more will be covered later. When you have time, read about the complete set of options in the references previously cited.

The Checkout compiler is constantly monitoring for references to unitialized variables. To detect them, it actually initializes variables which <u>you</u> don't initialize by using particular unlikely bit patterns (see LRM 266). In rare cases the patterns used for uninitialized FIXED BINARY or CHARACTER variables may actually represent values your

program can produce and deal with. In these cases you will need to disable the automatic checking by specifying GOOPTS='NODIAGNOSE'. Don't, however, do this as a matter of routine, since the service is one of the most valuable performed by the Checkout compiler.

Our default for the ERRORS option, ERRORS(10), tells the interpreter to report and then recover from the first ten errors that result in the raising of the ERROR condition. Its recovery action is tailored to the specific cause of error; in some cases it is taken after normal return from an established ERROR on unit, while in other cases it is taken in lieu of raising the condition. The repair of errors is surprisingly successful. More often than not, it permits the program to proceed to where other, unrelated, errors are discovered in the same run.

The FORMAT option can be used to obtain a formatted source listing--one which is "properly" indented, having no more than one statement per line, etc., and generally easier to read.

Our default for the SIZE option is SIZE(MAX). This tells the compiler to make use of all the storage available to it; thus, increasing the region request will automatically give the compiler more core storage to work with. It should be noted that the IBM PL/I compilers, unlike the FORTRAN compilers, are designed to work in surprisingly small amounts of storage. There is no lower limit below which the compiler will cease to work (however, interpretation cannot proceed if insufficient storage is available for the allocation of all of your PL/I variables). Generally, if insufficient storage is available to keep everything in core, the compiler will "spill" onto a temporary dataset. The amount of spilling that occurs is a function of space available to the program, amount of PL/I storage allocated (during interpretation), size of the program, complexity of the program (mix of language features used), etc. If spilling becomes excessive the extra I/O can cause your job charges to increase very rapidly. The compiler monitors the activity on the spill file; it will report a "thrashing" condition if one should develop. It should be pointed out that the default region of 150K established in our cataloged procedures is probably too small except for very simple programs; 250K will usually result in a cheaper, faster job.

You do not have to worry about space for buffers for your open datasets when you specify SIZE(MAX). File openings are performed under the control of the interpreter, and it turns out the routines are smart enough to know how much OS space will be required for buffers; the space is made available (by spilling, if necessary) before it is requested. However, certain requests for OS core storage may be made without the Checkout compiler's knowledge. This can happen in the following three cases:

(a) You invoke the SORT utility dynamically.
(b) You invoke an other-language routine which obtains storage by executing a GETMAIN.
(c) You load a fetchable load module by executing a FETCH statement.

In these cases you cannot permit the Checkout compiler to use all the storage available to it; you must reserve some. You can reserve, say, 30K of the region by coding GOOPTS='SIZE(-30K)'.

## 13.7. Cataloged procedures for the Optimizing compiler.

The Optimizing compiler produces an object module as output. It must be link-edited or loaded prior to execution, even if no subroutines are needed (certain housekeeping library routines are always needed). The cataloged procedures available for this compiler comprise a standard family of procedures whose prefix is PLO. Members of the PLO series, and the names of the steps they contain, are indicated below.

| | PLO | EDT | GO |
|---|---|---|---|
| PLOCD | √ | | |
| PLOCLG | √ | | √ |
| PLOCEG | √ | √ | √ |
| PLOCP | √ | | |
| PLOCEP | √ | √ | |
| PLOC | √ | | |
| PLOEP | | √ | |
| PLOEG | | √ | √ |
| PLOLG | | | √ |

The PLOCD cataloged procedure automatically supplies the compiler options DECK and NOLOAD to override the opposite defaults. The user is responsible for supplying inputs and capturing outputs in the ways described for the PLC series of procedures. There is, of course, no need for SYSITEXT.

For consistency with the Checkout compiler, the Optimizer will also accept its source input from SYSCIN or from SYSIN. Multiple external procedures can be compiled in a single PLO step; they are, as before, separated by *PROCESS statements. There is no use for the *DATA statement with this compiler. See OPG 29 through OPG 31.

13.8. Optimizing compiler options.

For the Optimizing compiler, a clear distinction is made between compiler and execution options. Although the compiler itself is not present during execution, certain options may be specified then to select certain services from the run-time support or to "tune" the environment. OPTIONS, GOOPTS, and GOPARM are used exactly as they are in PLC procedures. Note, however, that we do not currently have a PLOG procedure. To execute a previously link-edited production program, you will need to code "bare" JCL, as in the following (which demonstrates how you communicate both execution-time options and an argument to the main procedure).

```
// EXEC PGM=member,PARM='exec-options/main-arg'
//STEPLIB DD DISP=SHR,DSN=pds.containing.member
//SYSPRINT and other DD statements, as needed.
```

Compiler and execution options are described in OPG 32 and OTUG 5. As with the Checker, we have established defaults that differ in some instances from the IBM defaults. Ours are tabulated at the end of these notes.

13.9. Specific Optimizing compiler options.

To obtain maximum optimization you need to specify OPTIONS='OPT(2)'. This will increase the cost of compilation to a degree, but the gains achieved during execution will be worth it if the program is executed often and if other optimization options (REORDER, TOTAL, and CONNECTED) are specified in the program itself.

Two execution options, ISASIZE and REPORT, are worth studying carefully. Dynamic PL/I storage is allocated in an area called the ISA (Initial Storage Area), which is obtained at program initialization time. The allocations performed within the ISA are reasonably efficient (in any event, better than performing a GETMAIN to obtain the storage from OS). If the ISA proves insufficient, addi-

tional storage will be obtained, as needed, by GETMAIN; the program will continue to run (as long as the additional storage is available in the region), but performance will be degraded relative to a run performed with a larger ISA. The reason you can't generally specify ISASIZE(MAX) is that you must leave behind whatever storage will be needed for buffers and dynamically loaded library modules. The spectrum of requirements of different programs cannot optimally be accommodated by a single default. Ours, ISASIZE(8K), differs from the IBM default for subtle reasons. You can specify your own better guess. If you know that the space required for buffers, etc., is relatively constant, while the amount of PL/I storage required depends on the inputs in a particular run (as it well might in a list-processing application), you can reserve a fixed amount of storage for OS and let the ISA track the region request by coding, for instance, GOOPTS='ISASIZE(-20K)'.

In any event, you can ask the system to monitor its own storage management activities and report on them at the end of a run. For this purpose, you use the REPORT option, e.g., GOOPTS='REPORT'. You can specify both together, using abbreviations, as in GOOPTS='R,ISA(-20K)'. The storage management report, which tells you, among other things, an optimal ISASIZE, is produced on the file with ddname PLIDUMP. Thus, when you specify the REPORT option you must add to your GO step

```
//GO.PLIDUMP DD SYSOUT=A
```

A good discussion of these very important options and storage management considerations is in OPG 33.

## 13.10. Source record formats, margins, and sequence fields.

Both the Checker and Optimizer can accept source input in a variety of record formats and record lengths.

Unless you use the MARGINS compiler option to specify otherwise, the default source margins for fixed-format (blocked or unblocked) records are 2 and 72, while for variable-format (blocked or unblocked) they are 10 and 100. At the same time, unless you specify otherwise with the SEQUENCE compiler option, the compilers will assume columns 73 to 80 of fixed-format records, or 1 to 8 of variable-format records, have been reserved for sequence information.

Column 1 (F-format) or 9 (V-format) is assumed to contain a listing-control character (as described under the MARGINS option).

These two defaults automatically match source records created by EDIT in TSO. There, as we will see in Lesson 15, you have a choice of two dataset "types": PLI and PLIF. The former results in V-format blocked records with sequence information in 1 to 8, while the latter creates an F-format dataset with sequence information in 73 to 80. V-format records are generally more economical because trailing blanks are not included to "complete" the record beyond the last character you type. Note that in either case, the first character that you type goes into the listing-control column and is not read as part of the source.

The F-format records are also the standard "card-image" format for card decks.

Each statement is numbered by the compiler so that it may be uniquely referenced in any error messages. You have your choice as to whether the statement numbers are to be assigned by the compiler from the sequence 1,2,... or are to be taken from the sequence field of the record on which the statement begins. The latter choice is the default (determined by the compiler options NUMBER and NOSTMT) when the compilation is performed in TSO, because there you do not get a source listing by default (the option determing that is NOSOURCE). When the compilation is performed in the batch system, the default options are STMT, NONUMBER, and SOURCE. The compiler assigns consecutive statement numbers, which are shown on the source listing. Note that if the source dataset happens to contain sequence information in this case, as it would if it had been created by EDIT in TSO, the sequence information is also listed on the source listing.

In the Checkout compiler, execution-time error messages are always accompanied by statement numbers. Under the Optimizing compiler, you have your choice as to whether statement numbers are to accompany the code offset in run-time error messages. The cost of having them do so is a table, kept in core during execution, and consulted on the occasion of producing any system message. If your program is compiled in the batch system, the defaults there (NOGOSTMT, NOGONUMBER, OFFSET) suppress the inclusion of

this table in the load module but print it out as part of the compilation listing. Error messages at run time will not contain a statement number, but you can look up the offset appearing in the message in the offset table in the listing to find the statement number. If your program is compiled in TSO, you do not get a listing of the offset table by default, so statement numbers (derived from the sequence information) are obtained from the in-core table and used in run-time error messages (the governing options are NOOFFSET, GONUMBER, NOGOSTMT).

## 13.11. Using the preprocessor.

To use the facilities of the preprocessor (Lesson 12) you must specify the MACRO compiler option. If the SOURCE option applies, the source listing produced represents the output of the preprocessor. To obtain, in addition, a listing of the input to the preprocessor, use the INSOURCE compiler option. If you wish to capture the output of the preprocessor on cards, use the MDECK option. Read about these in the references previously cited for compiler options.

If you use the %INCLUDE statement to include source text from a library of source text members, the library (or libraries) will have to be named in DD statements in the compile (PLO) or translation (PLC, or, in the cataloged procedure PLCCG, GO) step. The ddname is either the one you use in the %INCLUDE statement or, if that includes only a member name and no ddname, SYSLIB.

Included text need not have the same record format as the primary source. And if %INCLUDE is the only preprocessor statement used, you need not specify the MACRO option; specify the INCLUDE option instead (it is more efficient).

See OPG 34 and CPG 36.

## 13.12. Mixing PL/I and FORTRAN.

We mentioned in Lesson 10, and will repeat here, that when you link-edit PL/I and FORTRAN mixtures you must make the FORTRAN library available. Use the POSTLIB symbolic

parameter of link-edit (EDT) or loader (GO) steps for that purpose. Also, FT06F001 must be defined in the GO step, even if the FORTRAN program does not write on unit 6. Review Sections 10.10 and 13.6 (for the need to use the SIZE option of the Checker with interlanguage communication) and CPG 24. A JCL sample follows.

```
// EXEC FTHC
//FTH.SYSIN DD *
      FORTRAN source
/*
// EXEC PLOCLG,POSTLIB='SYS1.FORTLIB'
//PLO.SYSCIN DD *
      PL/I source
/*
//GO.FT06F001 DD SYSOUT=A
//    other DD statements, as needed
```

## 13.13. JCL considerations for fetchable procedures.

A "fetchable" procedure, i.e., an external procedure to be loaded dynamically before invocation, must be completely link-edited with any other external procedures it invokes and stored as a member of the load module library named in the STEPLIB DD statement of the execution step.

Normally you do not have to worry about specifying an <u>entry point</u> to the linkage editor or loader; the standard entry point of PL/I load modules, PLISTART, is communicated automatically. However, you <u>must</u> intervene to specify a different entry point for a fetchable load module. The entry point name (and the member name under which it is stored) must both be the same as the external procedure name.

Example: A main procedure called PROG includes the following statements.

```
DCL (SUBR1, SUBR2) ENTRY EXT;
IF TYPE = 1 THEN DO;
   FETCH SUBR1;
   CALL SUBR1;
   RELEASE SUBR1;
END;
ELSE DO;
   FETCH SUBR2;
   CALL SUBR2;
   RELEASE SUBR2;
END;
```

Complete JCL (except for JOB and account cards) for creating a production version of the program and executing it follows:

```
// EXEC PLOCEP
//PLO.SYSCIN DD *
      PROG source
/*
//EDT.SYSPVT DD DISP=(NEW,CATLG),DSN=load.lib(PROG),
// UNIT=unit,SPACE=space

// EXEC PLOCEP
//PLO.SYSCIN DD *
      SUBR1 source
/*
//EDT.SYSPVT DD DISP=OLD,DSN=load.lib(SUBR1)
//EDT.SYSIN DD *
   ENTRY SUBR1
/*

// EXEC PLOCEP
//PLO.SYSCIN DD *
      SUBR2 source
/*
//EDT.SYSPVT DD DISP=OLD,DSN=load.lib(SUBR2)
//EDT.SYSIN DD *
      ENTRY SUBR2
/*

// EXEC PGM=PROG
//STEPLIB DD DISP=SHR,DSN=load.lib
//SYSPRINT and other DD statements, as needed
```

See OPG 35 and CPG 37.

13.14. DD statements for SYSPRINT.

The following DD statement for SYSPRINT is contained in the GO step of all PL/I cataloged procedures:

```
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=1511)
```

This effectively overrides the default linesize of 120 for print files and gives you 132 instead. When you use "bare" JCL for production runs of optimized code, you should write the SYSPRINT statement as above. (We may add a PLOG cataloged procedure, analogous to PLCG, in the future.)

13.15. General plan for program development.

We cannot be too emphatic in our recommendations that the Checkout compiler be used while a program is undergoing development and testing. Although its greatest utility is experienced in the interactive mode (Lesson 15), it is by far still the best debugging tool we have in batch.

As program testing proceeds, external procedures considered debugged may be compiled under the Optimizing compiler and link-edited with those still being tested under the Checkout compiler (we will demonstrate this later).

If bugs are by and by disclosed in production (optimized) code, certain features of the Optimizing compiler (also described later) may be helpful in identifying them. Alternatively, one or more external procedures can be put back through the Checkout compiler.

13.16. Special features for debugging.

The language itself has several debugging features to offer. (These have not been standardized.) Both the Checker and Optimizer implement the CHECK condition, and the SNAP option of ON statements. The Checker (only) implements, in addition, the FLOW, SNAP, and ALL options of the PUT statement, and the CHECK and FLOW statements. And each compiler implements certain compiler options useful in debugging situations. A general reference for the Checker's special features is LRM 267; others will be given later. Also review LRM 124.

13.17. The CHECK condition.

The <u>CHECK condition</u> occurs whenever a variable to which it applies is assigned a value, or a procedure or label to which it applies is reached. The condition is normally disabled. Like the I/O conditions and the CONDITION condition, it is a qualified condition, meaning that you state the individual items to which it applies. Standard system action for the CHECK condition is to write a comment on SYSPRINT showing the procedure or label reached, or the name of the variable and its new value. CHECK can be applied to all known names by leaving out the list of qualifying names. See LRM 268 and the entry for CHECK in LRM 116.

## 13.18. The CHECK statement.

The CHECK statement dynamically enables the CHECK condition for variables, etc., referenced subsequently. Its primary use is in an interactive environment, however it is also useful in batch. The simplest way to get a complete trace of assignments is to execute a CHECK statement as part of your initialization in the main procedure. The NOCHECK statement nullifies the effect of the CHECK statement. See LRM 269 through LRM 271. The Optimizer analyzes these statements for correct syntax, then ignores them.

## 13.19. The SNAP option of the ON statement.

An ON statement may include the SNAP option. The effect of this is to produce a traceback through active blocks, on SYSPRINT, whenever the action specified by the ON statement (whether that be standard system action or execution of an on unit) is taken. This feature, which is in the ANSI standard, is useful in determining the cause of the condition. See LRM 272 and LRM 273.

## 13.20. Checkout compiler extensions of PUT statement.

Under the Checkout compiler (only), program-control variables can be transmitted by LIST- or DATA-directed output. The value transmitted is an implementation-defined high-level interpretation of the value. For example, the value printed for a label variable is the name of the label constant and information from which you can deduce the "environment" part of the label value; for a file variable, it is the name of the file constant which provided its value, a list of file description attributes, and an indication of whether the file is open or closed and a count of the number of records processed; etc. Any value which has not been initialized, or which is invalid or inaccessible, is indicated by a comment.

In addition, other options are permitted on the PUT statement under the Checkout compiler. The SNAP option causes a traceback through active blocks to be printed. The FLOW option causes a table of the last few changes in the flow of control to be printed. The ALL option includes the effects of SNAP and FLOW; in addition, for all active blocks the following is printed:

(a) The block identification.
(b) The enablement/disablement status, in the block, of each PL/I condition.
(c) The values, in the block, of all of the "ON" builtin functions (ONCODE, etc.) .
(d) The names and values of all variables declared in the block.

It may be seen from the above that an extremely useful high-level debugging printout of the status of just about everything can be printed on the occasion of any error by executing, early in your program,

```
ON ERROR BEGIN;
  ON ERROR SYSTEM;
  PUT ALL;
END;
```

See LRM 274 and LRM 275.

13.21. Flow information.

There are two ways that "flow information," i.e., information about any action resulting in the interruption of sequential statement execution, such as a procedure invocation or return, a branch resulting from a GO TO, IF, or DO statement, or the raising of a condition, can be obtained from the Checkout compiler. This information, also, can be useful in determining what is actually happening in a malfunctioning program.

Data on the last several changes in the flow of control are kept in a "flow table." The size of this table is determined by an execution-time option, the FLOW option. Our default is 20 entries. The flow table is dumped onto SYSPRINT by executing a PUT FLOW or PUT ALL statement or, incidentally, whenever SNAP action is taken for a condition.

Alternatively, by executing a FLOW statement you cause the flow data to be written on SYSPRINT as it is generated. The NOFLOW statement turns dynamic flow tracing off. See LRM 276 through LRM 278.

## 13.22. Checkout compiler options for special debugging situations.

Each execution-time message from the Checkout compiler includes a count of the number of statements interpreted up to that point. Suppose the first error occurs after 10000 statements have been interpreted, and suppose any output your program may have produced before that doesn't help you find the cause of error. Furthermore, it is assumed that any FLOW or SNAP output produced with the error message doesn't help. You would like to dynamically enable the CHECK condition by executing a CHECK statement, but you don't know where in the program to execute that. If you execute that too early, you will get too much CHECK output.

What you do is execute the CHECK statement early and block its output until, say, 9900 statements have been executed. The blocking is accomplished by the BLOCK option:

```
GOOPTS='BLOCK(9900)'
```

Another situation that you can get a handle on by using appropriate execution options is an apparent infinite loop. You can break the loop after execution of a given number of statements or after a given number of lines are printed on SYSPRINT by using the STEP or STEPLINES execution options, respectively. When the appropriate limit is reached, the ERROR condition is raised. A small further allotment of statements or lines permits you to print out some debugging output. (The ERROR on unit shown in Section 13.20 is about the best you can do.) This may be usefully combined with a CHECK statement and the BLOCK execution option to produce a trace of assignments that occur in the statements executed just prior to interruption of the loop.

An additional source of information on debugging techniques for batch use of the Checkout compiler is CPG 38.

## 13.23. Mixing Optimizer and Checker compiled procedures.

Once one external procedure from a large program has been debugged to your satisfaction, it may be compiled under the Optimizing compiler and link-edited with procedures compiled by the Checkout compiler. Execution still occurs under the control of the Checkout compiler, but whenever control reaches an Optimizer-compiled procedure the procedure is executed at full machine speed.

There are two precautions you must observe when you mix modules in this way. First of all, if any locator variables (pointers or offsets) are communicated between Checker and Optimizer procedures, then the Checker procedures involved must be compiled in conjunction with the COMPATIBLE compiler option, and execution (under the Checker) must also be conditioned by the COMPATIBLE execution option. Secondly, the first input seen by the linkage editor or loader must have been produced by the Checkout compiler (this is to ensure that the Checker will have control over all storage allocation.

You have a choice of two libraries that may be used to resolve the library external references in the Optimizer-produced code. The normal Optimizer library is SYS1.PLIBASE and is provided automatically in PLO series cataloged procedures. This contains the full code to support library services. An alternative, SYS1.PLICMIX, is selected automatically by PLC series cataloged procedures. This brings in much smaller amounts of code whose function it is to bootstrap into the proper Checkout compiler interpreter routines, which perform the services. The first library results in a larger, faster program compared to the second. You can specify either library in either series of procedures by using the LIBRARY symbolic parameter. See CPG 39 and OPG 36.

## 13.24. Debugging with the Optimizer.

There are a few things you can do to find problems in pure-Optimizer code.

You can use the SUBSCRIPTRANGE, STRINGRANGE, SIZE, and CHECK conditions, and the SNAP option of the ON statement. Enablement of the above conditions causes extra code to be generated, degrading performance and increasing core requirements.

Flow information can also be obtained from the Optimizing compiler, providing the FLOW compiler option is used during execution. The table is dumped whenever SNAP action is taken for a condition.

A similar option, the COUNT option, can be used to print a table of statement execution counts at the end of execution. Available in the Checker, too, this option is defaulted on for the Checker and off for the Optimizer. See OPG 33.

As a final resort, a dump can be requested. For this purpose you call the PLIDUMP builtin procedure provided by our implementation. This gives you quite a bit of high-level (i.e., interpreted and formatted) information first, including the contents of buffers of opened files, followed (if the appropriate option has been specified to PLIDUMP) by a hexadecimal dump of storage. You will need to add

```
//GO.PLIDUMP DD SYSOUT=A
```

to your JCL.

All of these debugging facilities are discussed in OPG 37.

13.25. A library maintenance technique for program development.

Let us present some JCL, then discuss it.

```
// EXEC PLOCEP,EDTIF='(16,LT,PLO)',EDTOPTS=NCAL
//PLO.SYSCIN DD *
*PROCESS NAME('PROC1');
 PROC1: PROC ...
    :
    :
 END;
*PROCESS NAME('PROC2');
 PROC2: PROC ...
    :
    :
 END;
/*
//EDT.SYSPVT DD DISP=OLD,DSN=auto.call.lib
```

We assume a partitioned load-module dataset whose name replaces "auto.call.lib" above has been previously created. This dataset will contain one member for each external procedure in a program under development; the member name is the same as the external procedure name. The members have been processed through the linkage editor, so each is a load module. However, no member is executable as it stands, because it has been link-edited with the NCAL linkage editor option, which leaves external references unresolved. The dataset will serve as an "automatic call library" in a later link-edit or loader step that will bring all the modules together into an executable load module.

You run a job such as the one shown above either to compile some external procedures for the first time or to recompile some after making changes. In the job shown above, two procedures, PROC1 and PROC2, are compiled. Note that the *PROCESS statement in front of the source for each procedure specifies the NAME compiler option. The string given with the NAME option will become the member name under which the external procedure will be stored in the automatic call library. The symbolic parameter EDTIF is assigned the value (16,LT,PLO) so that the link-edit step will be executed regardless of the severity of errors discovered by the compiler in any procedure (without this, a sufficiently severe error in one external procedure will prevent the link-editing of any of them, and they will all have to be recompiled).

The goals of the above JCL are to make it unnecessary to recompile any external procedure that hasn't been changed when you recompile some that have; to maintain fully up-to-date object code at all times; and to ease the burden of tailoring JCL. Assuming that you keep the *PROCESS statement for a procedure with the source itself, as the first card, then you never have to change any JCL. You merely grab whatever decks you wish to compile or recompile and put them in the "fixed" JCL between

```
//PLO.SYSCIN DD *
     and
/*
```

Some people prefer to maintain the source for each external procedure as a member of a partitioned source module dataset. Assuming the source for each procedure starts with a *PROCESS statement, then the extent of your "variable' JCL for the above job would be the minimum necessary, namely:

```
//PLO.SYSCIN DD DISP=SHR,DSN=source.lib(PROC1)
//           DD DISP=SHR,DSN=source.lib(PROC2)
```

Assuming you have named your main procedure DRIVER (for example), you can execute your program subsequently by collecting the pieces and resolving external references with the loader, using the following JCL:

```
// EXEC PLOLG,PRELIB='auto.call.lib',EP=PLISTART
//GO.LDRIN DD DISP=SHR,DSN=auto.call.lib(DRIVER)
// other DD statements, as needed.
```

Instead of resolving external references every time you execute the program, you may do that just once. One way of doing this is by adding the following JCL to the job which recompiles and updates your automatic call library.

It assumes you have an existing executable program library, a partitioned dataset containing one member. This dataset is scratched and reallocated each time the following JCL is run.

```
// EXEC PGM=IEFBR14
//DD1 DD DISP=(MOD,DELETE),DSN=exec.prog.lib,
// UNIT=unit,SPACE=space
// EXEC PLOEP,PRELIB='auto.call.lib'
//EDT.INCLIB DD DISP=SHR,DSN=auto.call.lib
//EDT.SYSPVT DD DISP=(NEW,CATLG),DSN=exec.prog.lib,
// UNIT=unit,SPACE=space
//EDT.SYSIN DD *
    INCLUDE INCLIB(DRIVER)
    ENTRY PLISTART
    NAME DRIVER
/*
```

Note that this JCL does not have any names in it that need to be changed depending on which external procedures have just been compiled. To execute your program, use the following "bare" JCL:

```
// EXEC PGM=DRIVER
//STEPLIB DD DISP=SHR,DSN=exec.prog.lib
//SYSPRINT and other DD statements, as needed
```

The above JCL can be easily adapted for use with the Check-out compiler, or for Checker/Optimizer mixtures. (In the latter case, the requirement that a Checker module be the first presented to the linkage editor can be met by keeping DRIVER, i.e., the main procedure, at the Checker level.)

Finally, this JCL can also be adapted for use with a program containing fetchable procedures. Suppose in addition to the main program, DRIVER, you have two fetchable procedures, FP1 and FP2. The only necessary modification is

```
//EDT.SYSIN DD *
   INCLUDE INCLIB(DRIVER)
   ENTRY PLISTART
   NAME DRIVER
   INCLUDE INCLIB(FP1)
   ENTRY FP1
   NAME FP1
   INCLUDE INCLIB(FP2)
   ENTRY FP2
   NAME FP2
/*
```

Although all three load modules are re-created each time any procedure is compiled, the advantages of this technique are:

(a) The JCL shown above still doesn't depend on which procedures are compiled.

(b) If a particular external procedure happens to be referenced by two or more of the fetchable procedures, it is contained only once in your automatic call library, yet it is automatically brought in to each fetchable load module that needs it.

## CHECKOUT COMPILER AND EXECUTION OPTIONS

[ ] are used to denote text that may be omitted.

| COMPILER OR EXECUTION OPTION | ABBREVIATED NAME | DEFAULT |
|---|---|---|
| AGGREGATE\|NOAGGREGATE | AG\|NAG | AG in batch<br>NAG in TSO |
| ATTRIBUTES\|NOATTRIBUTES | A\|NA | A in batch<br>NA in TSO |
| BLOCK(n) | BL(n) | BL(0) |
| CAPS\|ASIS | - | CAPS |
| CHARSET([48\|60][EBCDIC\|BCD]) | CS([48\|60][EB\|B]) | CS(60 EB) |
| COMPATIBLE\|NOCOMPATIBLE | COM\|NCOM | NCOM |
| COUNT\|NOCOUNT | CT\|NCT | CT in batch<br>NCT in TSO |
| DIAGNOSE\|NODIAGNOSE | DIAG\|NDIAG | DIAG |
| DUMP\|NODUMP | DU\|NDU | NDU |
| ERRORS(n) | - | ERRORS(10) in batch<br>ERRORS(0) in TSO |
| ESD\|NOESD | - | ESD in batch<br>NOESD in TSO |
| FLAG[(I\|W\|E\|S)] | F[(I\|W\|E\|S)] | F(I) in batch<br>F(W) in TSO |
| FLOW(n,m)\|NOFLOW | - | FLOW(20,20) |
| FORMAT\|NOFORMAT | FOR\|NFOR | NFOR |
| HALT\|NOHALT | - | NOHALT |
| INSOURCE\|NOINSOURCE | IS\|NIS | IS in batch<br>NIS in TSO |
| ISASIZE([x],[y],[z]) | - | ISASIZE(8192,<br>8192,20) |
| LINECOUNT(n) | LC(n) | LC(55) |
| LMESSAGE\|SMESSAGE | LMSG\|SMSG | LMSG in batch<br>SMSG in TSO |
| MACRO\|NOMACRO | M\|NM | NM |
| MARGINI('c')\|NOMARGINI | MI('c')\|NMI | NMI |
| MARGINS(m,n[,c]) | MAR(m,n[,c]) | MAR(2,72,1) for<br>F-format<br>MAR(10,100,9) for<br>V,U-format |
| MDECK\|NOMDECK | MD\|NMD | NMD |
| NAME('aaaaaaaa') | N('aaaaaaaa') | - |
| NEST\|NONEST | - | NEST |
| NUMBER\|NONUMBER | NUM\|NNUM | NNUM in batch<br>NUM in TSO |
| OBJECT\|NOOBJECT | OBJ\|NOBJ | NOBJ |
| OPTIONS\|NOOPTIONS | OP\|NOP | OP in batch<br>NOP in TSO |
| RUN\|NORUN[(W\|E\|S)] | - | NORUN(S) in batch<br>NORUN(E) in TSO |
| SEQUENCE(m,n)\|NOSEQUENCE | SEQ(m,n)\|NOSEQ | SEQ(73,80) for<br>F-format<br>SEQ(1,8) for<br>V,U-format |
| SIZE([-]n\|[-]nK\|MAX) | SZ([-]n\|[-]nK\|MAX) | SZ(MAX) |
| SMAN\|NOSMAN | - | NOSMAN in batch<br>SMAN in TSO |
| SOURCE\|NOSOURCE | S\|NS | S in batch |

| | | |
|---|---|---|
| | | NS in TSO |
| STEP(n[,m])\|NOSTEP | ST(n[,m])\|NST | NST |
| STEPLINES(n)\|NOSTEPLINES | STL(n)\|NSTL | NSTL |
| STMT\|NOSTMT | - | STMT in batch |
| | | NOSTMT in TSO |
| STORAGE\|NOSTORAGE | STG\|NSTG | STG in batch |
| | | NSTG in TSO |
| SYNTAX\|NOSYNTAX[(W\|E\|S)] | SYN\|NSYN[(W\|E\|S)] | NSYN(S) in batch |
| | | NSYN(E) in TSO |
| TERMINAL[(options)]\|NOTERMINAL | TERM[(options)]\|NTERM | NTERM in batch |
| | | TERM in TSO |
| VERIFY\|NOVERIFY | V\|NV | V |
| XREF\|NOXREF | X\|NX | X in batch |
| | | NX in TSO |

# OPTIMIZER COMPILER OPTIONS

[ ] are used to denote text that may be omitted.

| COMPILER OPTION | ABBREVIATED NAME | DEFAULT |
|---|---|---|
| AGGREGATE\|NOAGGREGATE | AG\|NAG | AG in batch<br>NAG in TSO |
| ATTRIBUTES\|NOATTRIBUTES | A\|NA | A in batch<br>NA in TSO |
| CHARSET([48\|60][EBCDIC\|BCD]) | CS([48\|60][EB\|B]) | CS(60 EB) |
| COMPILE\|NOCOMPILE[(W\|E\|S)] | C\|NC[(W\|E\|S)] | NC(S) |
| COUNT\|NOCOUNT | CT\|NCT | NCT |
| DECK\|NODECK | D\|ND | ND |
| DUMP\|NODUMP | DU\|NDU | NDU |
| ESD\|NOESD | - | ESD in batch<br>NOESD in TSO |
| FLAG[(I\|W\|E\|S)] | F[(I\|W\|E\|S)] | F(I) in batch<br>F(W) in TSO |
| FLOW[(n,m)]\|NOFLOW | - | NOFLOW |
| GONUMBER\|NOGONUMBER | GN\|NGN | NGN in batch<br>GN in TSO |
| GOSTMT\|NOGOSTMT | GS\|NGS | NGS |
| IMPRECISE\|NOIMPRECISE | IMP\|NIMP | IMP in batch<br>NIMP in TSO |
| INCLUDE\|NOINCLUDE | INC\|NINC | NINC |
| INSOURCE\|NOINSOURCE | IS\|NIS | IS in batch<br>NIS in TSO |
| LINECOUNT(n) | LC(n) | LC(55) |
| LIST[(n,m)]\|NOLIST | - | NOLIST |
| LMESSAGE\|SMESSAGE | LMSG\|SMSG | LMSG in batch<br>SMSG in TSO |
| MACRO\|NOMACRO | M\|NM | NM |
| MAP\|NOMAP | - | NOMAP |
| MARGINI('c')\|NOMARGINI | MI('c')\|NMI | NMI |
| MARGINS(m,n[,c]) | MAR(m,n[,c]) | MAR(2,72,1) for F-format<br>MAR(10,100,9) for V,U-format |
| MDECK\|NOMDECK | MD\|NMD | NMD |
| NAME('aaaaaaaa') | N('aaaaaaaa') | - |
| NEST\|NONEST | - | NEST |
| NUMBER\|NONUMBER | NUM\|NNUM | NNUM in batch<br>NUM in TSO |
| OBJECT\|NOOBJECT | OBJ\|NOBJ | OBJ |
| OFFSET\|NOOFFSET | OF\|NOF | OF in batch<br>NOF in TSO |
| OPTIMIZE(TIME\|0\|2)\|NOOPTIMIZE | OPT(TIME\|0\|2)\|NOPT | NOPT |
| OPTIONS\|NOOPTIONS | OP\|NOP | OP in batch<br>NOP in TSO |
| SEQUENCE(m,n)\|NOSEQUENCE | SEQ(m,n)\|NSEQ | SEQ(73,80) for F-format<br>SEQ(1,8) for V,U-format |
| SIZE(n\|nK\|MAX) | SZ(n\|nK\|MAX) | SZ(MAX) |
| SOURCE\|NOSOURCE | S\|NS | S in batch<br>NS in TSO |

| | | |
|---|---|---|
| STMT\|NOSTMT | - | STMT in batch<br>NOSTMT in TSO |
| STORAGE\|NOSTORAGE | STG\|NSTG | STG in batch<br>NSTG in TSO |
| SYNTAX\|NOSYNTAX[ (W\|E\|S) ] | SYN\|NSYN[ (W\|E\|S) ] | NSYN(S) |
| TERMINAL[ (options) ]\|NOTERMINAL | TERM[ (options) ]\|NTERM | NTERM in batch<br>TERM in TSO |
| XREF\|NOXREF | X\|NX | X in batch<br>NX in TSO |

## OPTIMIZER EXECUTION OPTIONS

[ ] are used to denote text that may be omitted.

| EXECUTION OPTION | ABBREVIATED NAME | DEFAULT |
|---|---|---|
| COUNT\|NOCOUNT | CT\|NCT | Compile-time |
| FLOW[ (n,m) ]\|NOFLOW | - | Compile-time |
| ISASIZE([x],[y],[z]) | ISA([x],[y],[z]) | ISA(8K,8K,20) |
| REPORT\|NOREPORT | R\|NR | NR |
| STAE\|NOSTAE | - | STAE |
| SPIE\|NOSPIE | - | SPIE |

## 14. Multitasking and asynchronous I/O.

Multitasking is perhaps the single most unique feature of PL/I, having no parallel in other popular high-level languages. The feature allows one to express algorithms for parallel computation (concurrent processing) in a natural way. However, the multitasking feature of the language, like the preprocessor, has deficiencies. Until a little over a year ago, ANSI was well along with an improved version of the multitasking feature. Then the committee began to have second thoughts. They eventually decided that new developments in computer architecture and operating system capabilities were coming forth so rapidly that the standardization of multitasking, which is intimately related to operating system capabilities, was actually premature. As a result, multitasking was entirely withdrawn from the proposed standard; and, since asynchronous I/O uses some of the same language elements, that went too. These features will undoubtedly be standardized in the future. For the time being, you can expect IBM to continue to offer their version of multitasking as an extension to the standard. Univac is offering an amalgam of that and the earlier proposal from ANSI.

### 14.1 Concept of flow of control.

We may think of the execution of a "conventional" program, i.e., the kind we have been talking about all along, as being "tracked" by a cursor that points to the instruction or statement being currently executed. Normally this cursor moves forward, or down, in the program. When it encounters an IF statement it may skip ahead. On encountering the END statement of a DO group it may back up to the DO statement. For a GO TO statement, it jumps to some other place. For a procedure call, it also jumps, and if control reaches the procedure's END statement or one of its RETURN statements, the cursor jumps back to the point just beyond the "point of invocation." Finally, whenever a condition is raised and an on unit entered, the cursor jumps in the same way it does for a procedure call, with the expectation of a return jump back to the "point of interrupt" when (and if) the on unit returns normally.

A delay (some WAIT time) that occurs for I/O activity when the "cursor" is at an I/O transmission statement doesn't change the picture in any way. The program may be temporarily suspended, in the sense that its cursor is not progressing, but the point is that it still identifies some statement (and only one) as the current statement.

In describing the peregrinations of the cursor we are describing what is commonly referred to as the "flow of control." The essential feature of a conventional program which distinguishes it from a multitasking program is that its behavior is described by a single flow of control; in procedure and on unit invocations we have to "remember" the point of call or point of interrupt, so the cursor can be restored to that point on the appropriate action by the program, but that housekeeping doesn't alter the fact that there is a single flow of control.

In a multitasking program there may be an arbitrary number of concurrently active "cursors" or "flows of control." That is, several different statements may all be identified as "current." The cursor for each one moves along in the program in the normal way — jumping around, etc.

How, you say, can several statements be in a state of concurrent execution on a single computer? Well, on our system there is in fact only one processing element. It can be servicing only one flow of control at a time. Various things may cause the processing element to temporarily divorce itself from one flow of control and begin (or resume) servicing another. So the several statements are not being executed exactly at the same time. But for all practical purposes, they logically are because there is a degree of unpredictability in the extent to which the one processing element will service one flow of control before something causes it to switch to another. In any event, other computer systems may have multiple processing elements all sharing common storage, so it is entirely conceivable that several statements of a multitasking program may be in states of physically simultaneous, and not just logically concurrent, execution.

Another feature of multitasking programs is that they start off with a single flow of control, looking for all the world like a conventional program. At some point, however, they do something internally to establish an additional flow of control. These additional flows of control may continue for a while, then terminate, leaving others, including the original one, still progressing. A particular flow of control, from the moment of its birth until its death, is called a task. To rephrase what we have already said:

(a) A conventional program has a single task.
(b) A multitasking program starts out in the conventional way, but after a while it creates (starts, or "attaches") new tasks. These may create yet others, etc.
(c) All of the tasks proceed concurrently, eventually dying.

Generally, individual tasks can and do proceed independently of each other. They may each execute different portions of the code in a program, or they may execute the same portion. This poses no problem because the code is not self-modifying (it is "read-only") and each task can be given its own separate work area. Tasks may also share data defined by the program, that is, several statements being executed concurrently in different tasks can access common data. When this, in fact, occurs in a program, that program will generally need to employ some means of synchronizing the accesses of the tasks to the common data. Synchronization is accomplished through the temporary "suspension" of one or more tasks, if necessary.

## 14.2. Overview of PL/I multitasking facilities.

PL/I provides language primitives for:

(a) Creating tasks.
(b) Synchronizing tasks.
(c) Terminating tasks.

## 14.3. When to use, and not use, multitasking.

It is natural to conclude that the ability to code "parallel processes" by using multitasking may gain you the advantage of additional I/O overlap (several tasks can do I/O simultaneously, and another can be using the CPU). Actually, this overlap can be achieved in a conventional, non-multitasking program by using the asynchronous I/O facilities to be described later in this lesson. But, in any event, whether there is any advantage in going out of your way to achieve extra overlap depends on several factors. Two aspects to be considered are possible system-wide gains in throughput that work to everyone's advantage and possible lower job charges (due to decreased WAIT time) that work to yours.

In the case of a job that occupies all, or most of, core storage, the only thing of importance is to minimize the total residence time of the job since it effectively has control of the whole machine during the time it is resident. Clearly, if additional overlap allows it to complete sooner, everyone gains by the increased throughput. The person who runs the job should be, and in our system will be, rewarded through lower costs.

For a job that occupies a small amount of core storage, overlap achieved by it is not so important in terms of overall system throughput, since lots of core storage remains for the scheduling of other jobs which

can provide overlap on an inter-job basis. Because of this, it can be argued that a "small-core" job should be neither rewarded for extra overlap nor penalized for not achieving it. Unfortunately, our system tends to reward even small-core jobs for extra overlap. What is worse, it is just those jobs that are subject most to certain kinds of contention that can inhibit potential overlap, contributing to the variation in recorded WAIT time.

Independently of the above, one must consider the difficulty of designing and debugging a multitasking program. Also, one must recognize that multitasking programs incur additional operating system overhead. (See LRM 279.) And finally, at least for sequential I/O you can achieve the benefits of I/O-CPU overlap without going out of your way merely by employing buffered files (which are, in fact, the default).

What, then, are the logical uses for multitasking? Basically, multitasking is used to express, in a natural way, algorithms exhibiting a high degree of parallelism. An example in system programming is the implementation of a time-sharing system supervisor: the parallel activities are the independent, simultaneous services requested by logged on users. An example in engineering or science fields might be the simultaneous search for a solution by different methods where the convergence of any method is unpredictable; another example might be the simulation of a physical system characterized by competing or cooperating, random or probabilistic, concurrent processes.

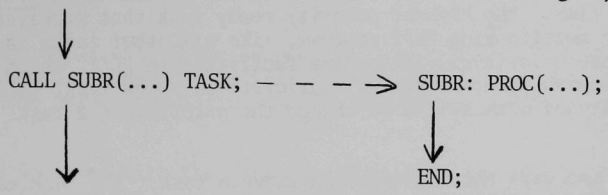
## 14.4. Attaching a task.

In PL/I, an additional independent flow of control is started by invoking a procedure with one of the multitasking options. For example:

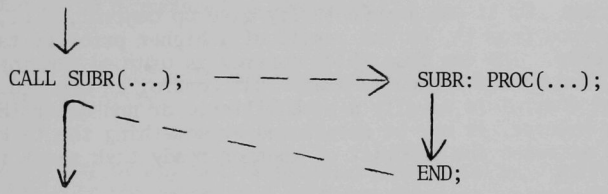
CALL SUBR(*arg-list*) TASK;

In this CALL statement, the TASK option is used to denote that the execution of the procedure SUBR should constitute a new task. The invoking procedure does not transfer control to SUBR in the normal way. Rather, it goes right on with the execution of the next statement; it does not wait for SUBR to return. SUBR is free to execute "in parallel" with the invoking procedure. The task represented by the execution of SUBR survives until the procedure SUBR returns. At that moment, the task (i.e., the flow of control) comes to an end. (As we will see later, there are also several other ways that tasks can end.)

The situation described can be diagrammed in the following way.




Contrast this to the normal invocation of a proecdure.




When SUBR is executed as a task, it is called a subtask of the task represented by the execution of the invoking procedure; that task, in turn, is called the parent task. The main procedure has no parent task. It is called the major task.

Any task can start any number (in theory) of subtasks; these may start other subtasks, etc. A given procedure may have several concurrent invocations (as tasks). This situation should be compared to recursion, in which the several concurrent invocations of a given procedure are all part of the same task: the one flow of control is in the most recent invocation.

See LRM 280 and LRM 281.

## 14.5. Scheduling of tasks; priorities.

When several tasks are active simultaneously they compete for CPU service. Some of them may be waiting for the completion of I/O (or for other things) and are not demanding CPU service at the moment. They are said not to be "ready." Amongst the ready tasks, however, only one can be receiving service from the single CPU. The algorithm which determines which ready task receives CPU service is the scheduling policy of the operating system. Many different scheduling policies are imaginable, such as "least recently served," "round robin," etc.

In the case of OS, however, each task in the system has a priority, and it is always the highest priority ready task that receives control. Tasks of a multitasking PL/I program, like all other tasks in the system, have priorities. There are facilities in PL/I, which will be described shortly, to assign a priority to a new task, to determine the priority of a task, and to change the priority of a task.

There are two ways that control can leave a task. The task may give up control voluntarily by becoming "not ready," as, for instance, by arriving at a point where it must wait for the completion of an I/O operation. Or it may involuntarily give up control, i.e., have control usurped from it, as the result of a higher priority task becoming ready. The one from which control is usurped remains ready, of course, and is not logically aware that control is usurped. The usurping of control is usually a probabilistic or non-deterministic happening; however, it may be occasioned by something the task itself does (such as raise the priority of another ready task above its own).

When a task is created as shown in the previous section, its initial priority is set equal to that of its parent task. The PRIORITY option of the CALL statement, however, can be used to assign either a higher or lower initial priority to the new task. The effect of

CALL SUBR(...) TASK PRIORITY (*expr*);

is to attach SUBR as a task with a priority of n relative to that of the parent task, where n is the value of *expr* (converted, if necessary, to a binary integer).

In PL/I, priorities are always relative (ultimately to that of the major task, which is set initially by the operating system). But this is always sufficient because, as far as your program is concerned, its use of priorities is for the logical purpose of controlling which of its ready tasks is to be selected for CPU service in preference to the others.

The maximum and minimum absolute priorities of tasks are determined by factors outside PL/I. Generally, you can expect ten or so priority levels below that of the major task to be reachable. It would be wise, however, to assume that no higher levels are reachable.

How essential is the use of priorities for logical purposes in PL/I? It turns out that in the current language their use is essential, but only to "simulate" a primitive multitasking service which is not in PL/I but is essential. It is interesting to note that the revised multitasking language earlier proposed by ANSI had this other service ("locking" or "enqueueing") and eliminated the concept of priority.

See LRM 282 and LRM 283.

## 14.6. Task values and variables.

In this section we will describe how the priority of a task may be determined or changed. For this purpose we will need task variables and the PRIORITY builtin function and pseudo-variable.

A variable declared with the TASK attribute is known as a task variable. Task values (i.e., values of task variables), which are a new kind of program control data, have very limited use. They may, of course, be propagated by assignment.

The value of a task variable may be thought of as a binary integer representing a priority.

Task variables may or may not be associated with tasks. Initially, a task variable is not associated with any task; it is said to be an "inactive task variable." An inactive task variable is associated with a task by referencing the variable in the TASK option of the CALL statement which creates the task. Examples:

```
CALL SUBR1 TASK (T1) PRIORITY (-1);
CALL SUBR2 TASK (T2(I)) PRIORITY (1);
CALL SUBR2 TASK (P → T) PRIORITY (N);
CALL SUBR3 TASK (S.T) PRIORITY (0);
```

In these examples, four tasks are created. They are represented by the execution of procedures SUBR1, SUBR2 (for two of the tasks), and SUBR3. With each is associated a task variable (respectively, the element task variable T1, the I-th element of the array T2 of task variables, the based element task variable T located by the value of the pointer variable P, and the task variable T which is a component of the structure S).

Once a task variable is associated with a task, it is called an "active task variable." It remains active until the task with which it is associated terminates. It is illegal to attempt to associate an active task variable with a task by referencing it in a TASK option. In other words, the following is illegal:

```
CALL SUBR1 TASK (T) PRIORITY (-1);
CALL SUBR2 TASK (T) PRIORITY (-2);
```

When a task variable is made active by associateing it with a task as shown above, it is given a value (as a priority value) representing the priority of the newly created task. The priority of the task

may be subsequently examined by use of the PRIORITY builtin function. For example,

```
N = PRIORITY(T);
```

assigns to N the priority of the task associated with the task variable T. In keeping with the spirit of relative priorities, this value is not the absolute priority of that task, but rather the value relative to the priority of the task in which the statement is executed. Hence the reference PRIORITY(T) may return different values depending on which task it is evaluated in.

The priority of a task, after it is initially established, may be changed in either of two ways. One is by assigning a new task value to the associated task variable, as in

```
CALL SUBR1 TASK (T) PRIORITY (-1);
CALL SUBR2 TASK (U) PRIORITY (N);
   .
   .
   .
T = U;
```

The effect of this is to change the priority of the task associated with T to that of the task associated with U.

The second way to change the priority of a task is by using the PRIORITY pseudo-variable. For instance,

```
PRIORITY(T) = -1;
```

This causes the priority of the task associated with T to be set to one less than the priority of the task executing this statement (observe the use of relative priorities again).

A task can increase or decrease its own priority. One way of accomplishing that is shown by the following example:

```
DCL T TASK;
CALL SUBR TASK (T) PRIORITY (-1);
   .
   .
   .
SUBR:  PROC;
          .
          .
          .
       PRIORITY(T) = 1;
          .
          .
          .
END;
```

By the normal scope rules, the variable T declared outside of SUBR is also known inside SUBR. It is associated with the task represented by the execution of SUBR by the CALL statement. The statement

```
PRIORITY(T) = 1;
```

increases the priority of the named task by one relative to the current task. But the named task is the current task in this case.

This method of changing a task's own priority would not be available to it if no task variable was named in the TASK option of the CALL statement that created the task. In particular, it is not available to the major task. Thus, the second method of changing a task's own priority is to use the PRIORITY pseudo-variable with an empty argument list:

```
PRIORITY( ) = 1;
```

It should be remarked that priority values can be held by inactive task variables. Although in this case the priority value is not that of an actual task, it behaves as if it were. One (non-essential) use for this is the following. If the CALL statement that creates a task uses the TASK option containing a task variable, but no PRIORITY option, then the priority of the newly created task is set equal to that held in the (inactive) task variable. Thus,

```
CALL SUBR TASK (T);
```

is equivalent to

```
CALL SUBR TASK (T) PRIORITY (PRIORITY(T));
```

Both of these statements would be in error if T had not previously been assigned a value (as a priority).

See LRM 284 through LRM 287 and the entries for PRIORITY in LRM 18.

## 14.7. Event variables and values.

Suppose a parent task needs to know if one of its subtasks is still active, i.e., in existence. You might think it would suffice to set a BIT (1) variable to '1'B just before creating the subtask, then arrange for the procedure whose execution represents that subtask to set the same variable to '0'B just before it returns. The parent task could then test that variable at any time. The problem with this is that the subtask can terminate in other ways (which we haven't seen yet) besides executing a RETURN or END statement; it would not have a chance to set the bit variable to '0'B in these other cases.

Event variables (those declared with the EVENT attribute) may be employed to keep track of the status of a task. (They may be used for a lot of other things, as we shall see.)

Event variables have two parts to their value: a "completion" part represented as a BIT (1) value, and a "status" part represented as a binary integer.

Event variables may or may not be associated with tasks. Initially, an event variable is not associated with a task (or certain other

things described later) and is said to be an "inactive event variable." An inactive event variable may be associated with a task by referencing the event variable in the EVENT option of the CALL statement that creates the task, as in the following.

```
CALL SUBR TASK EVENT (E);
```

This serves three purposes:

(a) To make the event variable E active. It remains active until the associated task terminates. During this time it may not be associated with another task (or certain other things).

(b) To set the completion part of E to the value '0'B. It remains '0'B until the task terminates, at which time it is set to '1'B automatically.

(c) To set the status part of E to the value 0. Further use of the status part is described later.

The parent task, or any other task for that matter, can subsequently examine the value of the completion part of E to determine whether the subtask is still active. Access to the completion part of an event variable is gained by use of the COMPLETION builtin function. For example

```
IF COMPLETION(E) THEN ...;
```

When an event value is assigned to an event variable, the effect is to assign the completion and status parts simultaneously. No interrupt can occur in this operation, and no task switch can occur until it is complete. But assignment to event variables (particularly to their completion parts) will be saved until later, because it is illegal to do anything to the completion part of an active event variable (except look at it). Assignment to an inactive event variable is not useful, as yet, because any value it may have is not used, and in fact is overwritten, when the event variable is made active.

See LRM 288 through LRM 291 and the entry for the COMPLETION builtin function in LRM 18.

## 14.8. The WAIT statement.

Suppose now that a parent task, which has been executing in parallel with a subtask it has created, has reached a point in its logic where it absolutely must wait until the subtask reaches its end before going on. (Perhaps it needs a value which is set by the subtask just before it finishes.) One thing the parent task could do is "spin" in a tight loop, repeatedly looking at the completion

part of the event variable associated with the subtask. This would be extremely unwise, however, given a suitable alternative. For it would waste CPU cycles. Worse than that, if the subtask does not have a higher priority than the parent task, the loop would be infinite since the subtask would never again get control and could not terminate. (It is assumed the spin loop in the parent task does no I/O or anything else to voluntarily relinquish control of the CPU.)

To accomplish what is needed here, the parent task executes a WAIT statement naming the event variable. If, at that instant, the completion part of the event variable has the value '1'B (indicating the associated subtask has already terminated), the parent task merely proceeds to the statement after the WAIT statement. On the other hand, if it has the value '0'B (indicating the subtask is still executing), then the execution of the parent task is temporarily suspended; i.e., that task is made "not ready." It remains suspended until the subtask terminates (more precisely, until the event variable has been marked "complete" by the termination of the subtask), whereupon it again becomes "ready." This situation is not unlike what happens when a task waits for the completion of an I/O operation. During the period of waiting, other tasks (including, obviously, lower priority ones) may receive CPU service.

The form of the WAIT statement used for the above is:

```
WAIT  (E);
```

In general, a WAIT statement may wait on the completion of any number of events. A list may be specified, e.g.,

```
WAIT  (E1, E2, E3);
```

This WAIT statement will cause the task that executes it to wait for the completion of the three events (i.e., subtask terminations, for now) with which the event variables E1, E2, and E3 have been associated.

In addition, any item in the list of event variables may be an aggregate of event variables. The meaning is the same as writing all the contained element event variables in the list. E.g., for an array E of four event variables,

```
WAIT  (E(*));
```

is the same as

```
WAIT  (E(1), E(2), E(3), E(4));
```

Finally, a task can wait on the completion of any number of the event variables specified in the list; it need not wait on the

completion of all of them. For example,

```
WAIT (E1, E2) (1);
```

causes the current task to be suspended until at least one of E1 and E2 is marked complete. In general, an expression may be given for the wait count.

See LRM 292 and LRM 293.

## 14.9. Termination of tasks.

Termination of a task is said to be normal if the procedure whose execution represents the task reaches a RETURN statement or its END statement. In this case, the status part of the associated event variable, which was set to 0 automatically when the task was created, is left with this value. There are several ways a task can terminate abnormally. Two we will consider here are the following:

(a) It may execute an EXIT statement. See LRM 294.

(b) The block which created it can terminate. This forces termination of the subtask. Example:

```
  :
BEGIN;
  CALL SUBR1 TASK EVENT (E1);
  CALL SUBR2 TASK EVENT (E2);
  WAIT (E1, E2) (1);
END;
  :
```

Two subtasks are started by the begin block which then waits for the completion of either one. As soon as one completes (normally, presumably), the WAIT statement is satisfied, so the parent task proceeds. This causes termination of the begin block. That will force the abnormal termination of the remaining subtask.

When a task terminates abnormally, the status part of its associated event variable is set to 1 if it is still 0. See LRM 295.

A parent task may determine whether a subtask has terminated normally or abnormally by examining the status part of its associated event variable after termination. This is achieved by use of the STATUS builtin function. See the entry for the STATUS builtin function in LRM 18.

Although it is illegal to assign an event value to an active event variable (because, in particular, an attempt may not be made to affect its completion part in this or any other way), it is legal to assign a value to its status part only. This is accomplished by use of the STATUS pseudo-variable. (See entry in LRM 18.) A non-zero value assigned in this way will be left untouched if the task terminates abnormally.

Note that execution of an EXIT statement in the major task is equivalent to execution of a STOP statement in any task. Review LRM 112 and LRM 113.

14.10. Sharing data among tasks.

In general, two tasks can communicate through any variables known to both of them. Which variables are known is determined in the usual way by the block structure and scope of names. The programmer, however, is responsible for synchronizing (by using event variables and the WAIT statement in ways to be shown later) simultaneous references, one of which may change the value of a variable, in two or more tasks. Further details are given at LRM 296. Considerations for the sharing of files between tasks are given at LRM 297.

14.11. Inheritance of on units across tasks.

In Lesson 6 we decribed the search process for an established on unit to handle the occurrence of a condition. The search proceeded from the current block out along the chain of active blocks to the main procedure (if necessary). The same process is used in multi-tasking situations, i.e., if an established on unit is not found in any of the active blocks of a subtask in which the condition occurs, the search continues in the active blocks of the parent task. (Note that those blocks must still be active, for otherwise the subtask would have been terminated abnormally earlier.)

If an on unit is found in one of the blocks of the parent task, it is invoked in the normal way. However, note that its execution is part of the flow of control through the subtask, not through the parent task. The parent task continues doing whatever it was doing when the condition occurred in the subtask. That parent task could even raise the same condition and execute the same on unit in parallel with the subtask! See LRM 298.

The situation described in the last paragraph is only one way in which the same section of code can be in simultaneous execution by different tasks. Other ways result from the following:

(a) The same procedure is attached multiple times as a task.
(b) An internal or external procedure is called in the normal way by two or more tasks concurrently.
(c) A program, even one which does no multitasking, can be made resident in the operating system, so that if several users happen to execute it simultaneously they will share one copy of the code and all static data rather than having their own separate copy. As far as the operating system is concerned, the execution of that common code by each of those users is a separate task. This situation, in which a program has several concurrent executions as separate tasks without itself creating any subtasks, is called multiprogramming.

Whenever part of the code of any external procedure can be executed by two or more tasks simultaneously, that external procedure must specify OPTIONS(REENTRANT) on its procedure statement. This will tell our compiler to use dynamic storage (behaving, in fact, like automatic storage) for any writeable workspace it needs for the procedure, so that each of its "executors" will have a separate copy. (It might otherwise use static storage — which is shared amongst all the executors.) The programmer, too, must observe the same requirement and use automatic storage for any local variables of a reentrant procedure. See LRM 299 and LRM 300.

Before leaving this subject, we will give emphasis to a point not adequately made in the LRM. The establishment status of on units in a parent task is essentially "frozen" when it attaches a subtask, as far as the subtask is concerned. That is, if the search for an established on unit should go back as far as the blocks of the parent task, the effects of any ON or REVERT statements executed there after the creation of the subtask will not be observed by the subtask. They will, however, have their usual effect on the on units that may be entered by the occurrence of a condition in the parent task. For example,

```
ON FOFL X = 1;
CALL SUBR TASK;
ON FOFL X = 2;
L1:  A statement that raises FOFL;
SUBR:  PROC;
   L2:  A statement that raises FOFL;
END;
```

Execution of the statement labeled L1 causes X to be assigned the value 2. Execution of L2 causes it to be assigned the value 1, even if the parent task has already executed its second ON statement.

Also, at this point it should be remarked that standard system action for the FINISH condition, which was said in Lesson 6 to terminate the program, actually only terminates the task in which it is raised. The termination is normal only if the raising of FINISH results from the execution of a procedure END or RETURN statement for the task. In particular, standard system action for ERROR, raised in a task, causes a message to be printed, FINISH to be raised, and that task (only) to be terminated (abnormally). Thus, another legitimate use of multitasking might be merely to isolate the effects of catastrophic errors and prevent termination of the whole program by one.

## 14.12. EVENT option of DISPLAY statement.

Recall, from Lesson 12, that execution of

DISPLAY (*expr*) REPLY (*variable*);

prints a message (the value of *expr*) to the operator, then causes the program to be suspended until his reply is received (and stored in *variable*). The wait is just like that for an I/O operation. By adding the EVENT option to the DISPLAY statement, the program, instead, goes on with the next statement without waiting for a reply. The event variable named must be inactive, i.e., not currently associated with a task or, we can now add, with an operator reply. Its use in a DISPLAY statement makes it active and initializes the completion part to '0'B. It remains in this state until <u>two</u> things have happened:

(a) The operator's reply has been received.
(b) A WAIT statement referencing the event variable has been executed. Note that examination of the completion part by use of the COMPLETION builtin function will reveal a value of '0'B if it is performed before a WAIT statement references the event variable, even if the operator's reply has already been received.

When both conditions are met, the event variable is set inactive and its completion part is set to '1'B. Its status part is not used. See LRM 301.

## 14.13. Asynchronous I/O.

We have already remarked several times in passing that execution of I/O transmission statements may result in an implicit wait for the completion of the I/O operation, during which time the task executing the statement is in a "not ready" state. As with the DISPLAY statement, the EVENT option can be added to most transmission statements to allow the task executing the transmission statement to proceed to the following statement. The WAIT statement is used subsequently

when the task finally arrives at that point where it absolutely needs to be sure the I/O operation has completed before proceeding.

Rules for the use and management of I/O event variables are similar to those for display events. Specifically:

(a) The event variable referenced by the EVENT option of an asynchronous transmission statement must be inactive, i.e., not currently associated with a task, or an outstanding operator reply, or, we can now add, an asynchronous I/O operation.

(b) Execution of the transmission statement causes the event variable to become active, sets its completion part to the value '0'B and its status part to 0.

(c) While the event variable is active, it is illegal to change the value of its completion part (as by assignment to the event variable) or to associate the event variable with a task, operator reply, or another asynchronous I/O operation.

(d) The event variable remains active, and its completion part continues to have the value '0'B, until two conditions are met: the I/O operation has physically ended, and the event variable has been referenced in a WAIT statement. As with display events, but unlike task events, execution of the WAIT statement (eventually) is a prerequisite to setting the completion part of the event variable to '1'B.

References will be given later.

Asynchronous I/O can be used to overlap CPU use with I/O operations, to overlap I/O operations on different files, or even to overlap I/O operations on the same file. In certain cases of the latter use, the NCP ENVIRONMENT option (or JCL DCB parameter) must be employed to specify an upper bound on the number of I/O operations that can be outstanding simultaneously for a given file. See LRM 302 and LRM 303; further information is in the Programmer's Guides.

Note that the performance of asynchronous I/O operations does not involve the creation of new tasks. (Neither does the execution of a DISPLAY...REPLY...EVENT statement.) The facility may be used in a conventional program. Thus, the WAIT statement and event variables have uses both in multitasking and in conventional programs.

## 14.14. Conditions in asynchronous I/O.

We have seen that execution of an I/O transmission statement can raise a variety of conditions (ENDFILE, TRANSMIT, RECORD, etc.). If an asynchronous I/O statement causes one or more of these conditions to occur, the on units are not entered until the WAIT statement is executed. That is, their execution is made synchronous with respect to the flow of control through the task. If an on unit is to be entered, then the following occurs when the WAIT statement is finally executed:

(a) The event variable remains active.
(b) Its status part is set to 1.
(c) Its completion part remains at '0'B.
(d) The on unit is entered.
(e) If the on unit returns normally to the point of interrupt (the WAIT statement), a further on unit may be entered.
(f) When all on units have been executed and have returned normally, the completion part of the event variable is set to '1'B and the event variable is made inactive.
(g) The event variable is similarly marked complete and made inactive if an on unit terminates abnormally, i.e., by a GO TO out of block. Any additional pending on units will not be entered.

The EVENT option can be added to READ, WRITE, REWRITE, or DELETE statements in cases described at LRM 304. The file must have the UNBUFFERED attribute. Additional information is found in LRM 305 through LRM 311 as well as the entries for the applicable I/O conditions in LRM 116.

## 14.15. Review of exclusive files.

In Lesson 9 we briefly introduced the EXCLUSIVE attribute, NOLOCK option of the READ statement, and the UNLOCK statement. These were shown to be of use in synchronizing two independent concurrent conventional programs which update a common data base. They may also be employed to synchronize multiple tasks of a single program which independently update a shared data base. Review the material and references from Lesson 9.

## 14.16. "Physical" events.

We may characterize the task completion events, operator reply events, and asynchronous I/O completion events with which event variables may be associated, as described above, as "physical" events. The event variables are marked complete automatically when the associated physical activity comes to an end. (In the case of display and I/O events, but not task completion events, a WAIT statement naming the event variable must also be executed before the variable can be marked complete.) Generally, the program is largely unable to influence directly the completion of one of these kinds of physical activities; the activity completes in due course. (This is not literally true in the case of task events. We have seen how a task can be terminated abnormally, essentially at will, by having the block which created it terminate. And, of course, even the normal completion of a task is guided by program logic within the task. Generally, however, the task proceeds until normal completion while making unpredictable progress, and therefore it is useful to think of its completion as a physical event.)

## 14.17. "Abstract," or programmed, events.

The PL/I programmer may also define abstract or logical events that do not necessarily correspond to particular physical activity. Rather, they correspond to the program having reached a certain "state," which can have any meaning the programmer desires. The "completion" and "status" of these abstract events can be freely set by the programmer, and tasks can be made to wait for the completion of an abstract event.

So far we have seen no use for "inactive" event variables except that they are available for association with a physical event. As soon as they are associated with it, they become active and essentially must be left alone until the physical event runs its course. They are automatically marked "incomplete" at the start of the physical activity and "complete" at its end. During this time they may be the subject of a WAIT statement.

Inactive event variables, we can say now, can be used to mark the completion of abstract events. The general technique is to set one's completion part to '0'B at some point in time and to '1'B at a later point in time. Once it has been assigned a value, any task can WAIT on the event variable. (There is never any restriction on which event variables may be waited upon except that their

completion parts must have been assigned a definite value before they are referenced in a WAIT statement.) The task is suspended if, at the time the WAIT statement is encountered, the referenced event variable has the completion value '0'B, but it is not suspended if at that moment it has the value '1'B. If it is suspended, it remains suspended until some other task assigns the value '1'B to the event variable's completion part. Note, of course, that in general a WAIT statement may present a list of event variables and a number of them which must be marked complete before the wait is satisfied.

How does one assign a value to the completion part of an event variable? One way is to use the COMPLETION pseudo-variable, as in

```
COMPLETION(E) = '1'B;
```

or

```
COMPLETION(E) = N > 0;
```

(See the entry in LRM 18.) This leaves the status part of the event variable unchanged. Another way is to assign an event value (obtained by referencing another event variable or invoking a function that returns an event value) to the event variable. This propagates the completion and status parts simultaneously, with no possibility of interrupt or a task switch until the whole assignment is complete. Examples will be presented later. See LRM 312.

## 14.18. The DELAY statement.

Another statement that may have a marginal use in multitasking situations is the DELAY statement. The form is

```
DELAY (expr);
```

The current task is suspended for a number of milliseconds given by the value of *expr*. It is exactly as if an I/O operation that required the specified amount of WAIT time were being performed. The DELAY statement may usefully be employed in a loop in a high priority task to let a lower priority task gain control until the expiration of the delay, whereupon the higher priority task will usurp control from it. It may then examine the progress of the lower priority task (by accessing shared variables, for instance) and either go back through the loop and delay again, or do something else. See LRM 313 and LRM 314.

## 14.19. Examples of abstract events.

The first example will be developed in stages.

We assume a program will "produce" and "consume" 100 (say) values. A value is available for consumption as soon as it is produced. A very simple conventional program can be built around a loop such as

```
DO I = 1 TO 100;
  CALL PRODUCE(X);
  CALL CONSUME(X);
END;
```

However, we assume that the acts of producing and consuming a single value are each very laborious, involving a lot of I/O activity that could be overlapped (i.e., logically the consumer's I/O activity is independent of the producer's and can be overlapped with it). So one improvement using task events might be

```
CALL PRODUCE(X);
DO I = 1 TO 99;
  Y = X;
  CALL CONSUME(Y) TASK EVENT (EY);
  CALL PRODUCE(X) TASK EVENT (EX);
  WAIT (EX,EY);
END;
CALL CONSUME(X);
```

Here, the 2nd value is being produced while the 1st is being consumed, and so on.

The only real criticism of this multitasking solution is that subtasks are created 198 times (only two are active simultaneously, of course). There is a considerable overhead involved in creating a task which we would like to avoid.

Our solution will be to have the main program (the major task) create two subtasks, just once, and wait for both of them to complete. One will be responsible for producing values, the other for consuming them. We will arrange for the consumer to wait until the producer has produced a value in some workspace belonging to the producer. When the producer has produced such a value, it will inform the consumer that it may proceed. We will make it the responsibility of the consumer, when it receives the signal to proceed, to move the value to its own workspace. The producer will wait for this action to be completed. The consumer will signal the producer when it has completed the move. At that point, the producer will be free to produce another value while the consumer is busy consuming the last one. If the consumer happens to finish first, it will wait for another value to be made available (as signaled by the producer). If the producer happens to finish first, <u>it</u> will wait for the consumer to catch up and signal that it has moved the new value to its workspace.

Study the following solution carefully!

```
PROG:  PROC OPTIONS (MAIN);
   DCL (PRODUCE, CONSUME) ENTRY EXT;
   DCL (X, Y) ...;
   DCL (PRODUCED, MOVED, E1, E2) EVENT;
   COMPLETION(PRODUCED), COMPLETION(MOVED) = '0'B;
   CALL PRODUCER TASK EVENT (E1);
   CALL CONSUMER TASK EVENT (E2);
   WAIT (E1, E2);

PRODUCER:  PROC;
   DCL I FIXED BIN;
   DO I = 1 TO 100;
      CALL PRODUCE(X);
      COMPLETION(PRODUCED) = '1'B;
      WAIT (MOVED);
      COMPLETION(MOVED) = '0'B;
   END;
END;

CONSUMER:  PROC;
   DCL I FIXED BIN;
   DO I = 1 TO 100;
      WAIT (PRODUCED);
      COMPLETION(PRODUCED) = '0'B;
      Y = X;
      COMPLETION(MOVED) = '1'B;
      CALL CONSUME(Y);
   END;
END;

END; /* PROG */
```

The kind of control flow achieved in this program is known as a classical "coroutine" structure. It is characterized by an orderly "handshaking" of two parallel processes. Notice the symmetry: the operations performed by one task on PRODUCED and MOVED are the same as those performed by the other task on MOVED and PRODUCED, respectively. Also notice that all the tasks can execute at the same priority.

In the second example, we will assume that we have two tasks doing "mostly independent" things. Every once in a while, however, each task needs to update some data shared between the two tasks. In other words, there is a region in each task, called a "critical region," in which all operations on the shared data are performed, having the following properties: If neither task is in its critical region, the first one to arrive at its critical region is permitted

unconditionally to enter it. If, however, a task arrives at its critical region while the other is already in its, the one just arriving must wait until the other leaves its critical region. Thus, we are guaranteed that both tasks will not be in their critical regions simultaneously.

It might appear that this problem is solved by having both tasks execute code like the following:

```
WAIT (NOT_IN_CRITICAL_REGION);
COMPLETION(NOT_IN_CRITICAL_REGION) = '0'B;
    .
    .
    .
    critical region
    .
    .
    .
COMPLETION(NOT_IN_CRITICAL_REGION) = '1'B;
```

Here, NOT_IN_CRITICAL_REGION is initially complete. Let us suppose task 1 arrives at this code well in advance of task 2, so that it is, say, in the middle of its critical region when task 2 arrives at its WAIT statement. The event variable will be found to be marked incomplete. So task 2 will indeed wait until task 1 leaves its critical region and sets the event variable complete. But the danger here is that the two tasks may arrive at their WAIT statements nearly simultaneously. They will both find the event variable marked complete and both will proceed. (For example, assume task 2 has a higher priority than task 1 but is not, at the moment, "ready," i.e., assume it is waiting for I/O. Task 1 is proceeding. It passes its WAIT statement but before it has a chance to set the event variable incomplete, task 2 finishes its I/O, becomes ready, and usurps control from task 1 by virtue of its higher priority. It executes its WAIT statement and also proceeds. Being at a higher priority, task 2 continues and sets the event variable incomplete, then enters its critical region. Let's assume it does some I/O in there, so it relinquishes control to task 1. Task 1, already past its WAIT statement, sets the already incomplete event variable incomplete and proceeds into its critical region. Both tasks are now in their critical regions, and that is what we wanted to avoid.)

You might say "The probability that the adverse timing hypothesized above permits both tasks to enter their critical regions is incredibly and acceptably small; don't worry about it." But is the risk worth taking? The failure of the program's logic, in that one chance out of a million, could cause the destruction of a crucial data base! It is the essence of multitasking that programs be provably correct regardless of the sequence of any actions that can be performed in parallel.

Our solution will be shown in pieces. We will need a special task, represented by the procedure GRANTOR, which will execute at a higher priority than any other task in the program. This is an absolute necessity. GRANTOR will spend most of its time waiting for a request to perform a service, so it will not consume much of the CPU resource and won't generally interfere with the lower priority tasks. The idea, however, is that when one of the two tasks asks GRANTOR to perform a service, GRANTOR must get control immediately and in particular not let the other task proceed. It must also not relinquish control until it is finished performing the service requested. This, too, is essential and is achieved by coding GRANTOR so that it does no I/O or anything else that can cause it to wait until it has finished its duty.

The main program is shown first. It starts by initializing some event variables, lowering its own priority, then creating GRANTOR at a higher priority. GRANTOR will immediately take control. It will wait for a request. The main program will resume at that point and will then initiate the two tasks containing the critical regions. (These must never execute at a priority as high as GRANTOR's.) When both of these tasks have finished, the main program tells GRANTOR to terminate normally (that is one of the services it can be asked to perform) then waits for it to do so.

```
PROG:  PROC OPTIONS (MAIN);
   DCL (GRANTOR, TASK1, TASK2) ENTRY EXT;
   DCL (G, T1, T2) EVENT;
   DCL (TAKE, GIVE, QUIT, LOCK(2)) EVENT EXT;
   COMPLETION(TAKE), COMPLETION(GIVE), COMPLETION(QUIT) = '0'B;
   COMPLETION(LOCK(*)) = '1'B;
   PRIORITY() = -1
   CALL GRANTOR TASK PRIORITY (+1) EVENT (G);
   CALL TASK1 TASK EVENT (T1);
   CALL TASK2 TASK EVENT (T2);
   WAIT (T1, T2);
   COMPLETION(QUIT) = '1'B; /* TELL GRANTOR TO END */
   WAIT (G); /* WAIT FOR IT TO DO SO */
END;
```

The code in the vicinity of TASK1's critical region will look like

```
     TAKE = EV1;
     WAIT (LOCK(1));
          :
          :
          critical region
          :
          :
     GIVE = EV1;
```

Here, EV1 is a local event variable declared and initialized in TASK1 as follows:

```
DCL EV1 EVENT;
COMPLETION(EV1) = '1'B;
STATUS(EV1) = 1;
```

LOCK is an array of event variables declared in TASK1, TASK2, and GRANTOR as in PROG:

```
DCL LOCK (2) EVENT EXT;
```

The code in the vicinity of TASK2's critical region will be similar, namely:

```
TAKE = EV2;
WAIT (LOCK(2));
      .
      .
      .
      critical region
      .
      .
      .
GIVE = EV2;
```

In TASK2, EV2 is declared and initialized as follows:

```
DCL EV2 EVENT;
COMPLETION(EV2) = '1'B;
STATUS(EV2) = 2;
```

Obviously, TASK1, TASK2, and GRANTOR all declare TAKE and GIVE as external event variables. (Remember, they were initialized by the main program.)

As soon as either task assigns to TAKE, GRANTOR will proceed (because it is waiting for the completion of any one of several event variables, including TAKE, and it has a higher priority). Note that the task performing the assignment also succeeds in communicating the value 1 or 2 (used to identify the requesting task) to GRANTOR via the status part of TAKE. The assignment statements TAKE = EV1 and TAKE = EV2 are not interruptible. When GRANTOR gets control, it will set the element of the LOCK array corresponding to the requesting task either to complete or to incomplete, depending on whether the other task is not, or is, already in its critical region. It will do some other housekeeping, then go dormant again waiting for another request. The task which made the request will either wait or not, depending on the value assigned to its element of LOCK. Even if the other task has become ready before the requesting task executes its WAIT statement (i.e., while GRANTOR has control), when it makes its request to GRANTOR for permission to enter its critical region, GRANTOR will observe that it has already granted that permission to the first task and will set the second task's element of LOCK to incomplete. Finally, when either task leaves its critical region, it "gives back" the permission it was granted by making another request to GRANTOR. Note that if the other task is waiting for permission to enter, GRANTOR must now set that task's element of LOCK complete.

The code for GRANTOR is as follows:

```
GRANTOR:  PROC;
   DCL (TAKE, GIVE, QUIT) EVENT EXT;
   DCL LOCK (2) EVENT EXT;
   DCL (WANTOR, GIVER, OTHER) FIXED BIN;
   DCL OWNED BIT (1) INIT ('0'B);
   DO WHILE ('1'B); /* LOOP TERMINATED BY RETURN */
      WAIT (TAKE, GIVE, QUIT) (1); /* AWAIT A REQUEST */
      IF COMPLETION(TAKE) THEN DO; /* WANTS TO ENTER */
         WANTOR = STATUS(TAKE); /* WHO WANTS ? */
         COMPLETION(LOCK(WANTOR)) = ¬ OWNED;
            /* GRANT PERMISSION IFF RIGHT TO ENTER
               NOT ALREADY OWNED BY OTHER TASK */
         OWNED = '1'B;
         COMPLETION(TAKE) = '0'B; /* RESET */
      END;
      ELSE IF COMPLETION(GIVE) THEN DO; /* READY TO LEAVE */
         GIVER = STATUS(GIVE); /* WHO WANTS TO LEAVE ? */
         OTHER = 3 - GIVER; /* INDEX OF OTHER TASK */
         IF ¬ COMPLETION(LOCK(OTHER)) THEN /* IT WANTS IN */
            COMPLETION(LOCK(OTHER)) = '1'B;
            /* LET IT IN, BUT LEAVE OWNED ON */
         ELSE OWNED = '0'B;
         COMPLETION(GIVE) = '0'B;
      END;
      ELSE /* REQUEST TO QUIT */ RETURN;
   END;
END;
```

14.20. JCL considerations.

Whenever you use a cataloged procedure that link edits or loads a multitasking program, you <u>must</u> use the TASKLIB symbolic parameter in the way shown below

```
TASKLIB = 'SYS1.PLITASK'
```

See OPG 38.

In addition, the ISASIZE execution option may be used to specify, via its second and third operands, the size of the ISA acquired for each task other than the major task, and the maximum number of tasks (including the major task) that can be active simultaneously. The first operand of ISASIZE is used, as shown in Section 13.9, to specify the size of the major task's ISA. (Reread that section and the reference given there, OPG 33.) Note that ISASIZE is an execution option of the Checkout compiler as well as the Optimizer. No mention was made of ISASIZE in connection with the Checker, in Lesson 13, because in that compiler its first operand is ignored and the value specified for the SIZE operand is used instead. See CPG 40.

In our system, the default for ISASIZE in a multitasking program is ISASIZE(8K,8K,20). Typical use of TASKLIB and ISASIZE (together) is demonstrated in the following:

```
// EXEC PLOCLG,TASKLIB='SYS1.PLITASK',GOOPTS='ISA(30K,10K,4)'
```

14.21. Homework problems.

(#14A) Describe the differences between multiple concurrent invocations of a given procedure as separate tasks and multiple concurrent invocations of a given procedure by recursion.

(#14B) List all the PL/I actions you can think of that will cause the current task to relinquish control to another ready task in your program.

(#14C) How can you create a subtask at a priority higher than that of the major task without reaching a priority level higher than that assigned initially to the major task by the operating system?

(#14D) Execution of the GO TO statement in the following example is illegal. Can you explain why?

```
CALL SUBR TASK;
L:  ...
  :
  :
SUBR:  PROC;
  :
  :
  GO TO L;
  :
  :
END;
```

Under what conditions is execution of the GO TO statement in the following example legal? Illegal?

```
ON FOFL GO TO L;
CALL SUBR TASK;
L:  ...
```

(#14E) Give several reasons why a task must be (abnormally) terminated when the block containing the CALL statement that created it terminates.

(#14F) Recall that the event variable associated with an asynchronous I/O operation is marked complete only as part of the execution of a WAIT statement referencing it, even if the I/O operation is physically complete earlier. Thus, it would appear there is no way to "test" whether an I/O operation is complete or not without being forced to wait if it isn't. There is a tricky (though legal) way to test its completion periodically, however, without being forced to wait. Can you find it? Weak hint: You will need a second event variable.

(#14G) (Very difficult) Generalize the "critical region" problem in the following way:

(a) Permit any number of tasks to have critical regions, rather than just two.

(b) Permit any task to have any number of critical regions, each identified in some convenient way.

(c) Make sure that only one task at a time is permitted to enter a critical region of type "x". A given task may have several different critical regions of type "x", as well as critical regions of other types. While a task is in a critical region of type "x", another task may be in a critical region of a different type.

Hints: Since no bound is set on the number of tasks or critical regions, you will need to use list processing techniques (based variables, pointers, etc.). Be sure that based storage is freed in the same task in which it is allocated. You will need a task with the "high priority, non-interruptible" properties of GRANTOR. You will need to communicate more information to it with each request than you can conveniently represent in the status part of an event variable, so instead create the service task each time you need a service from it and communicate via arguments; it will end normally when it has provided the service.

# 15. The Checkout compiler in TSO.

In Lesson 13, the use of the Checkout compiler in the batch system was outlined. In addition, special features of PL/I useful in debugging, particularly in a batch environment, were described. While the value of the Checkout compiler in batch cannot be belittled, neither can its unique capabilities in a conversational environment be overstressed. The Checkout compiler is "at home" in TSO, and in this lesson we hope to convey a sense of excitement about its truly outstanding potential in this environment for contributing to productivity in the development and debugging process.

The notes for this lesson cover a brief orientation lecture which is meant to precede a taped demonstration of the Checker in TSO.

## 15.1. Creating a PL/I source dataset.

To create a source dataset containing a PL/I program, in TSO, enter the EDIT command with either of the "dataset" types" PLI or PLIF, as in

```
EDIT PROG PLI NEW
```

You will be prompted with line numbers. As you type each line, remember that the first character you type goes into the column reserved (by our default compiler options) for a listing control character; it is not part of the source program. (Except when you want a blank line, overprinting, or a page eject in the listing, type a blank as the first character.)

PLIF dataset type produces a dataset having FB-format records with an LRECL of 80. The EDIT line numbers are placed in columns 73-80. The first character you type goes into column 1; the next 71 or less go into columns 2 through 72. These are the default conventions assumed by the compiler for source margins and sequence information for source datasets consisting of fixed-length records.

PLI dataset type produces a dataset having VB-format records with an LRECL of 104. The first four bytes of a record are used by the system to indicate the length of the remainder of the record. The EDIT line numbers are placed in columns 1-8 of the data portion of the record, i.e., immediately following that length prefix. The first character you type goes into column 9. The next 91 or less go into columns 10 through 100. Short records are produced if you do not type

all 91 possible characters. These conventions match those assumed by the compiler, by default, for variable-length records.

Note that the use of either dataset type results in the appending of the "dataset qualifier" PLI to the dataset name given in the EDIT command.

PLI dataset type is generally more efficient than PLIF dataset type, in that short lines won't waste space. However, you cannot conveniently dump such a dataset onto cards. You would have to use the COPY command of TSO first to make a copy of the dataset in card image format. You will need several operands of the COPY command to arrange for this change of format and movement of the line numbers. Note that lines containing in excess of 71 source characters would be truncated during the copy.

A guide to the use of the editor may be found in CTUG 5 and OTUG 6. The two terminal users guides should be consulted, by those new to TSO, for chapters on other basic aspects of using TSO.

## 15.2. Invoking the Checker.

The Checkout compiler is invoked with the PLIC command in TSO. Before we get into that, we must mention the need for you to issue the IPLIC command first. This is used once per session, before the Checker is entered. It allocates the file SYSPLIC to the system dataset SYS1.PLICLNK required by the Checker. In addition, it allocates files SYSIN and SYSPRINT to the terminal. Use of the IPLIC command will not be necessary if you use the PLICKLGN logon procedure (it performs the above three allocations). If you have not used IPLIC or PLICKLGN, the response to your PLIC command will be a rather fast READY not accompanied by any further information.

The Checkout compiler absolutely cannot run in our 70K regions. For very small programs it might squeeze by in 140K. If the program is of moderate size, the amount of "spilling" performed when only 140K is available will be painfully slow and expensive for you, and probably detrimental to the performance of TSO for everyone. The use of the 200K region will result in more efficient processing of typical programs.

The PLIC command has the general form

PLIC *dsn keyword-operands*

For example,

PLIC PROG

or

PLIC PROG LMSG HALT MACRO

The *dsn* is the source dataset name. (PLI is appended as a dataset qualifier automatically.) You are actually invoking what is known as a "prompter" for the Checkout compiler. Its main function is to allocate files and datasets required for your compilation and then invoke the compiler itself. Some of the *keyword-operands* are defined by, and used by, the prompter only. The majority of them, however, translate into compiler options and are assembled by the prompter as a string of compiler options to be passed to the compiler. If you specify operands erroneously, you will be prompted by the prompter for corrections.

How can you find out about operands of the PLIC command? Two ways:

(a) Use the TSO HELP command. This is available for the purpose of finding out about the operands of any TSO command.

(b) See CTUG 6.

Typical or ordinary use of the PLIC command serves the same purpose as the PLCCG cataloged procedure (Lesson 13), i.e., the compiler proceeds from translation into interpretation without creating an object module. This is suitable for the execution of a self-contained main program not requiring link editing with other external procedures. Object modules and intermediate text modules may be created for later combining by the linkage editor or loader and execution under the interpreter phase of the Checker (references and a few brief notes will be given later).

We will here mention a few essential operands of PLIC.

One of the most essential is the PRINT operand. It controls the allocation of the file used for SYSPRINT. Note that when you use the PLIC command, the allocation of SYSPRINT to the terminal previously established by IPLIC or PLICKLGN is not actually used (it would be used for isolated execution under the interpreter phase). The default for the PRINT operand is PRINT(*), which says to allocate the file

to be used for SYSPRINT to the terminal. Generally, you can rely on this default. Recall that SYSPRINT is used by the translator phase for listings and similar outputs you select via options, and it is used (generally) by the program itself (and by the system) as a standard output file during execution. So, allocation to the terminal is quite reasonable. Note that you will not be flooded by listings, etc., since the compiler options for them have been set "off" in the defaults that apply when the compiler is used conversationally (see OTHER 3 again).

Another very useful choice is PRINT(*dsname*). The file used for SYSPRINT will be allocated to a dataset having dataset name *dsname*.LIST. If it doesn't exist, it is created for you with record format VBA. You can submit a job to list it later. What makes this especially valuable is that any subsequent interactions that you have with the Checker (its prompts and your replies) will be recorded on that dataset. Thus, you will have a "hardcopy" record of your session, which is nice if you are at a tube.

But, you ask, wouldn't you miss seeing SYSPRINT ouput produced by your program? Yes, but you can arrange to have a copy of what goes to the dataset *dsname*.LIST during interpretation sent to the terminal at the same time (shown later). Also, note that all compiler diagnostics (both during translation and interpretation) are automatically copied to the terminal, without a specific request from you, if you have used PRINT(*dsname*).

If you use an operand for a compiler option, such as SOURCE, the listing is produced on the file used for SYSPRINT, as governed by the PRINT operand. You can also embed that option in the TERMINAL operand (which is, itself, a compiler option), as in TERMINAL(SOURCE). The listing requested is produced on the terminal independent of the allocation of the file used for SYSPRINT.

All diagnostic messages have a long form and a short form. Which you get is governed by the compiler option LMESSAGE/SMESSAGE.

In TSO the default (using the abbreviation) is SMSG. Note two things:

(a) The long messages are generally much more informative, and you would do well to request them while you are still a beginner. Invoke PLIC as follows:

PLIC *dsn* LMSG

(b) If you have started a session with the default SMSG, you can change to LMSG during the session (demonstrated later). Or, there is a way you can ask for the text of the long form of a specific diagnostic you have just been given (they are always accompanied by their message numbers).

## 15.3. General behavior of the Checker in TSO.

As a consequence of our default options, the Checker proceeds as follows.

First it proceeds through translation. Syntax checking occurs first. If sufficiently severe syntax errors are found, control is turned over to you at the terminal. You can use various facilities of the Checker to correct the syntax errors that are reported, then go on. Next "global" checking of the program for consistency is performed. Again, if sufficiently severe errors are found, control is sent to the terminal and you are given a chance to correct them.

Following that, interpretation begins. As the program proceeds, various things can happen which again cause control to be sent to the terminal. You can interact with the program in several ways, modify it, etc., and go on.

Whenever control is sent to the terminal, you are prompted for a request. A prompt always ends in a "?" but that may be preceded by other characters which denote the state of the Checkout compiler. A variety of responses from you are permitted, depending on the state.

## 15.4. When control is passed to the terminal.

In general, when control has been passed to the terminal you may issue a subcommand. These are considered to be subcommands of the PLIC command just like CHANGE, LIST,

SAVE, etc., are subcommands of EDIT. There are a very large number of subcommands of PLIC.

In addition, when control is passed to the terminal during interpretation (identified by the prompt "?" without any preceding characters), you may enter PL/I statements from the terminal ("immediate-mode PL/I"). These are immediately translated and interpreted. Almost any PL/I statement, no matter how complex, is allowed. You may enter a DO group, a begin block, etc.

One of the subcommands is HELP. It serves various purposes, depending on the operands written with the subcommand. HELP is valid in response to any prompt. When used without any operands, i.e., as just HELP or H, the reply will be an explanation of the current state followed by a list of subcommands valid in that state. Whenever you don't know what is expected of you, type H.

Two other uses of the HELP command are as follows:

(a) To ask for an explanation of a particular subcommand. For instance, H LIST (or just H L) requests information on the LIST subcommand. H H requests information on the HELP subcommand.

(b) To ask for the long form of a particular compiler diagnostic whose short form has just been given to you. Example: H 1093 (here, we assume the short-form message was prefixed by the message number IEN1093I).

Another useful subcommand, valid most of the time, is OPTIONS. It can be used to list or change compiler options. For instance, OP LMSG sets the LMESSAGE option for subsequent diagnostics.

The MONITOR subcommand is used to initiate the copying at the terminal of all output directed to a stream file allocated to a dataset. For instance, if SYSPRINT has been effectively allocated to a dataset (by use of the PRINT operand of PLIC as shown above), you can get a copy of SYSPRINT output at the terminal by issuing the subcommand MONITOR SYSPRINT. NOMONITOR terminates monitoring.

How can you force control to be passed to the terminal before execution starts, so that you can issue a MONITOR

subcommand? One way is to use the HALT compiler option, specified as an operand on the PLIC command. It causes control to be passed to the terminal when the main procedure (any external procedure, actually) is entered for the first time. When that happens, you can type

MON (abbreviated; SYSPRINT implied)
GO (causes execution to resume).

Several other subcommands will be described later. A wealth of information is found in CTUG 7 and CTUG 8.

15.5. What sends control to the terminal?

We have already mentioned that the translator sends control to the terminal if it finds severe enough errors (the required severity is determined by the setting of certain compiler options). It also sends control to the terminal if you interrupt it (by depressing the BREAK key, for instance). In all cases, the prompt is "T?" to indicate that the translator has sent control to the terminal.

During execution there are many ways control can be sent to the terminal. Some are the result of unique extensions to the language implemented only by the Checkout compiler. Others are the result of slight redefinitions of the language as implemented by the Checkout compiler. A few of these are as follows.

(a) Execution of a HALT statement sends control to the terminal. See LRM 315.
(b) Standard system action for the FINISH condition has been redefined to send control to the terminal. You are thus given a chance to re-execute the program, possibly after modifying it, before terminating your session.
(c) Standard system action for the ERROR condition has been redefined to send control to the terminal. By use of appropriate subcommands you can determine the cause of the error and correct it, then resume execution from an appropriate point in the program.
(d) An additional condition, the ATTENTION condition, is available in the interactive environment. The ATTN condition "occurs" during execution when the BREAK key is depressed. Standard system action is to send control to the terminal. You can, of course, establish an ATTN on unit and thereby

use the BREAK key to affect the logic of your program (but not in a program compiled by the Optimizer).

See LRM 316.

In all of the above cases, a message is printed at the terminal explaining why control was passed to it. You are then prompted with "?".

In the case of either prompt, "T?" or "?", you may issue various subcommands. One of these, the GO subcommand, is used to resume processing just after the point from which control was passed to the terminal. (GO may be abbreviated by a null line.) In response to a "?" prompt you may, in addition, enter immediate-mode PL/I statements. The GO TO statement is an immediate-mode statement useful in this context to resume execution at a designated statement. The language has been extended to allow a line number after the keyword GO TO, so that you may resume execution at an unlabeled statement. In connection with this, GO TO 0 is taken by convention to mean "start execution again from the beginning."

Further information on the passing of control to the terminal is in CTUG 9.

## 15.6. Interactive debugging.

Rather than write a lot about this subject, we will demonstrate it. The main point, however, is that one does not need to switch back and forth between different TSO processors (EDIT, a compiler, LOADGO). One can do all one's debugging and program amending within the environment of the Checkout compiler, generally without even retranslating the program as it is amended. This is possible because

(a) An internal copy of the original source dataset is available for a variety of purposes at all times. When control is at the terminal, subcommands can be used to list it, modify it, and save it in an external dataset.

(b) PL/I statements may be executed in immediate mode to try to understand the nature of an execution error that has caused control to be

passed to the terminal. Through the use of subcommands, "breakpoints" may be established in the program and execution resumed. You can arrange to execute statements attached at the breakpoints or to have control return to the terminal when one is reached.

(c) Statements, or groups of statements, may be added, changed, or deleted without requiring retranslation of the whole program or loss of the execution environment.

(d) Rather general text editing subcommands are provided within the Checker to cope with more extensive or arbitrary source program changes. Their use mandates a retranslation, but that is accomplished by another subcommand.

Extensive information on the facilities for, and techniques of, interactive debugging may be found in LRM 317 and CTUG 10 through CTUG 12 (CTUG 11 contains numerous examples).

## 15.7. Topics for further study.

Consult the two TSO User's Guides (CTUG and OTUG) for information on the following topics, not covered in these notes.

(a) Use of the Checker in TSO to translate several external procedures, followed by their linking and execution (using LOADGO, or LINK and CALL). Interactive execution (program amending, etc.) is still possible, but if one retranslates an external procedure he will need to leave the Checker environment to use LOADGO or LINK again.

(b) Mixing Checker and Optimizer modules in TSO.

(c) Compiling under the Optimizer in TSO.

(d) Operands of LINK and LOADGO (PLIBASE and PLICMIX) that imply the PL/I libraries.

Also review Section 7.25, "Stream I/O to a Terminal."

Fetchable load modules (see Section 12.2) may be used in TSO providing execution of the program is initiated by the CALL command and the fetchable load modules are members of the partitioned dataset named in the CALL command.

Key to versions of manuals referenced.

| | |
|---|---|
| LRM | OS PL/I Checkout and Optimizing Compilers: Language Reference Manual GC33-0009-3 |
| OPG | OS PL/I Optimizing Compiler: Programmer's Guide SC33-0006-3 |
| OTUG | OS PL/I Optimizing Compiler: TSO User's Guide SC33-0029-2 |
| CPG | OS PL/I Checkout Compiler: Programmer's Guide SC33-0007-2 |
| CTUG | OS PL/I Checkout Compiler: TSO User's Guide SC33-0033-2 plus TNL SN33-6132 |

## REFERENCES FOR LESSONS 1-5

| | | |
|---|---|---|
| LRM 1. | p. 10, | "Identifiers" |
| LRM 2. | p. 15, | "The characteristics...these features." |
| LRM 3. | p. 49, | "When a...is compiled." |
| LRM 4. | p. 432-433, | "DECLARE" |
| LRM 5. | p. 49, | "DECLARE AND DEFAULT STATEMENTS" |
| LRM 6. | p. 73-75, | Up to "Examples of Declarations" |
| LRM 7. | p. 289, | Figure 19.4 |
| LRM 8. | p. 392, | Figure I.1, note 1. |
| LRM 9. | p. 396, | "BINARY and DECIMAL" |
| LRM 10. | p. 398, | "COMPLEX and REAL" |
| LRM 11. | p. 410, | "FIXED and FLOAT" |
| LRM 12. | p. 421, | "Precision Attribute" |
| LRM 13. | p. 15-19, | "ARITHMETIC DATA" stopping at "Numeric Character Data" |
| LRM 14. | p. 427, | General rule 2. |
| LRM 15. | p. 339, | Explanation and Figures F.4a and F.4d. |
| LRM 16. | p. 324-339, | Section F |
| LRM 17. | p. 345, | "Mathematical Built-In Functions" and "ACCURACY OF THE MATHEMATICAL FUNCTIONS" |
| | p. 347-352, | Figures G.1 and G.2 |
| LRM 18. | p. 353-367, | (Descriptions of each built-in function) |
| LRM 19. | p. 344, | "Arithmetic Built-In Functions" |
| LRM 20. | p. 472-473, | Figure K.1 |
| LRM 21. | p. 21, | "Character-String Data" stopping before the discussion of VARYING. |
| LRM 22. | p. 22, | "Bit-String Data" stopping before the discussion of VARYING. |
| LRM 23. | p. 197, | "Editing by Assignment" stopping before the discussion of VARYING. |

| | | |
|---|---|---|
| LRM 24. | p. 39-40 | "COMPARISON OPERATIONS" |
| LRM 25. | p. 341, | Explanation and Figures F.5a and F.5b. |
| LRM 26. | p. 40, | "CONCATENATION OPERATIONS" |
| LRM 27. | p. 38-39, | "BIT-STRING OPERATIONS" stopping at "Boolean Built-In Function" |
| LRM 28. | p. 38, | "Operations using Built-In Functions" |
| LRM 29. | p. 37, | "USE OF BUILT-IN FUNCTIONS" |
| LRM 30. | p. 40-42, | "COMBINATIONS OF OPERATIONS" |
| LRM 31. | p. 324, | Figure F.1. |
| LRM 32. | p. 21-22, | "Character-string variables may also be declared ...current length, in bytes." |
| LRM 33. | p. 23, | "A bit-string variable may be given...current length of the string, in bits." |
| LRM 34. | p. 197, | "A string value...varying length string variable." |
| LRM 35 | (none) | |
| LRM 36. | p. 396-397, | "BIT, CHARACTER, and VARYING" |
| LRM 37. | p. 344, | "String-handling Built-In Functions" |
| LRM 38. | p. 420-421, | "PICTURE" |
| LRM 39. | p. 22, | "Character-string variables...only a digit." |
| LRM 40. | p. 202-203, | "Character-String Picture Specifications" |
| LRM 41. | p. 305, | "Data assigned to a variable declared with a character-string picture specification is raised." |
| LRM 42. | p. 305-306, | "Picture Characters for Character-string Data" |
| LRM 43. | p. 19-21, | "Numeric Character Data" |
| LRM 44. | p. 199-202, | "PICTURE SPECIFICATION" (all except "Character-String Picture Specifications") |
| LRM 45. | p. 305, | Two paragraphs, beginning "Arithmetic data..." |
| LRM 46. | p. 306-314, | "Picture Characters for Numeric Character Data" |

| | | |
|---|---|---|
| LRM 47. | p. 25-27, | "ARRAYS" stopping at "Cross-Sections of Arrays" |
| LRM 48. | p. 403, | "Dimension Attribute" |
| LRM 49. | p. 432-433, | "Factoring of Attributes" |
| LRM 50. | p. 345-346, | "AGGREGATE ARGUMENTS" |
| LRM 51. | p. 27, | "Cross-Sections of Arrays" |
| LRM 52. | p. 43-45, | "Array Expressions" stopping at "Array-and-Structure Operations" |
| LRM 53. | p. 27-29, | "STRUCTURES" |
| LRM 54. | p. 31, | "LIKE Attribute" |
| LRM 55. | p. 416-417, | "LIKE" |
| LRM 56. | p. 469-483, | "Structure Mapping" |
| LRM 57. | p. 31-32, | "ALIGNED and UNALIGNED Attributes" |
| LRM 58. | p. 391,394, | "ALIGNED and UNALIGNED" |
| LRM 59. | p. 45-46, | "Structure Expressions" stopping at "Structure Assignment BY NAME" |
| LRM 60. | p. 29-30, | "ARRAYS OF STRUCTURES" |
| LRM 61. | p. 45, | "Array-and-Structure Operations" |
| LRM 62. | p. 46, | "Structure Assignment BY NAME" |
| LRM 63. | p. 427-429, | "Assignment Statement" |
| LRM 64. | p. 399, | General rule 2. |
| LRM 65. | p. 400, | Figure I.3. |
| LRM 66. | p. 30, | "DEFINED Attribute" |
| LRM 67. | p. 399-403, | "DEFINED" |
| LRM 68. | p. 52, | "PROCEDURE STATEMENT" |
| LRM 69. | p. 61, | "PROCEDURE BLOCKS" |
| LRM 70. | p. 73-74, | "It is not...two uses of the name C." |

LRM 71. p. 74-75, "Contextual Declaration"

LRM 72. p. 75, "Implicit Declaration"

LRM 73. p. 73-74, "The appearance of...the same block)."

LRM 74. p. 75, "Since a...in error."

LRM 75. p. 76-77, "Internal and External Attributes" stopping at "Note."

LRM 76. p. 409, "EXTERNAL and INTERNAL"

LRM 77. p. 78, "Scope of Member Names of External Structures"

LRM 78. p. 13, "A block...PROCEDURE statement."

LRM 79. p. 61-63, "Blocks"

LRM 80. p. 79-81, "Default Attributes"

LRM 81. p. 81-83, "DEFAULT Statement"

LRM 82. p. 433-435, "DEFAULT"

LRM 83. p. 78-79, "Multiple Declarations and Ambiguous References"

LRM 84. p. 85, "The purpose...class of storage used."

LRM 85. p. 32-33, "INITIAL Attribute"

LRM 86. p. 412-414, "INITIAL"

LRM 87. p. 85, "Static Storage"

LRM 88. p. 85-86, "Automatic Storage" stopping before "EFFECT OF RECURSION ON AUTOMATIC VARIABLES"

LRM 89. p. 86-89, "Controlled Storage"

LRM 90. p. 394-396, "AUTOMATIC, STATIC, CONTROLLED and BASED"

LRM 91. p. 399-400, General rules 3 and 7a.

LRM 92. p. 404-405, "ENTRY" stopping at General rule 6.

LRM 93. p. 112-113, "ENTRY attribute" stopping at "Entry Expressions as Arguments"

LRM 94. p. 398, "CONNECTED"

LRM 95. p. 457, "RETURN"

LRM 96. p. 440-441, "ENTRY"

LRM 97. p. 410-412, "GENERIC"

LRM 98. p. 103-118, "Subroutines and Functions"

LRM 99. p. 450-453, "PROCEDURE"

- - - - - - -

CPG 1. p. 27, "AGGREGATE Option"

CPG 2. p. 40, "AGGREGATE LENGTH TABLE"

- - - - - - -

OPG 1. p. 22, "AGGREGATE Option"

OPG 2. p. 37, "AGGREGATE LENGTH TABLE"

- - - - - - -

CTUG 1. p. 116, "AGGREGATE | NOAGGREGATE"

OTUG 1. p. 59, "AGGREGATE | NOAGGREGATE"

- - - - - - -

OTHER 1. G. Weinberg, PL/I Programming: A Manual of Style, Section 1.5.1. McGraw-Hill, (1970)

OTHER 2. K. Dritz, The Precision of Fixed-Point Arithmetic Operations in PL/I. Proceedings, SHARE XLV (1975).

## REFERENCES FOR LESSONS 6-10

| | | |
|---|---|---|
| LRM 100. | p. 446-447, | "IF" |
| LRM 101. | p. 448, | "Null Statement" |
| LRM 102. | p. 437-438, | General rules 1 and 6. |
| LRM 103. | p. 438, | General rule 2. |
| LRM 104. | p. 437-440, | "DO" |
| LRM 105. | p. 415-416, | "LABEL" |
| LRM 106. | p. 413, | Rule 16. |
| LRM 107. | p. 69-70, | "Reactivation of an Active Procedure (Recursion)" |
| LRM 108. | p. 445-446, | "GO TO" |
| LRM 109. | p. 113-115, | "Entry Expressions as Arguments" |
| LRM 110. | p. 406, | General rules 8 and 9. |
| LRM 111. | p. 404-406, | "ENTRY" |
| LRM 112. | p. 67, | "PROGRAM TERMINATION" |
| LRM 113. | p. 459, | "STOP" |
| LRM 114. | p. 207, | "When a...Checkout Compiler." |
| LRM 115. | p. 378, | "Classification of Conditions" |
| LRM 116. | p. 379-390, | Descriptions of individual conditions |
| LRM 117 | p. 12-13, | "A condition prefix...Program Checkout." |
| LRM 118. | p. 207, | "The programmer...when they occur." |
| LRM 119. | p. 207-208, | "Condition Prefixes" |
| LRM 120. | p. 208, | "Scope of the Condition Prefix" |
| LRM 121. | p. 457, | "REVERT" |
| LRM 122. | p. 459, | "SIGNAL" |
| LRM 123. | p. 398, | "CONDITION" |

| | | |
|---|---|---|
| LRM 124. | p. 207-215, | "Exceptional Condition Handling and Program Checkout" |
| LRM 125. | p. 448-449, | "ON" |
| LRM 126. | p. 250-252, | "ORDER AND REORDER OPTION" |
| LRM 127. | p. 429-430, | "BEGIN" |
| LRM 128. | p. 452, | General rule 5. |
| LRM 129. | p. 249-278, | "Efficient Programming" |
| LRM 130. | p. 119, | "PL/I includes...execution of a program." |
| LRM 131. | p. 432, | "CLOSE" |
| LRM 132. | p. 120-123, | "Files" |
| LRM 133. | p. 391-424, | "Attributes" |
| LRM 134. | p. 124-129, | "Opening and Closing Files" |
| LRM 135. | p. 449-450, | "OPEN" |
| LRM 136. | p. 122, | "STREAM and RECORD Attributes" |
| LRM 137. | p. 122, | "INPUT, OUTPUT, and UPDATE Attributes" |
| LRM 138. | p. 123, | "PRINT Attribute" |
| LRM 139. | p. 123, | "ENVIRONMENT Attribute" |
| LRM 140. | p. 146-152, | "ENVIRONMENT Attribute" |
| LRM 141. | p. 393, | Figure I.2. |
| LRM 142. | p. 134-136, | "Data Specifications" |
| LRM 143. | p. 131, | "LIST-DIRECTED TRANSMISSION" |
| LRM 144. | p. 136-138, | "List-directed Data specification" |
| LRM 145. | p. 131-132, | "DATA-DIRECTED TRANSMISSION" |
| LRM 146. | p. 138-141, | "Data-directed Data Specification" |
| LRM 147. | p. 132, | "EDIT-DIRECTED TRANSMISSION" |
| LRM 148. | p. 141-142, | "Edit-directed Data Specification" stopping at "General rule" |

| | | |
|---|---|---|
| LRM 149. | p. 142-144, | "Data Format Items" |
| LRM 150. | p. 315-322, | "Edit-directed Format Items" |
| LRM 151. | p. 144, | "Control Format Items" |
| LRM 152. | p. 144-145, | "Remote Format Items" |
| LRM 153. | p. 198-199, | "STRING Option in GET and PUT Statements" |
| LRM 154. | p. 132-133, | "Data Transmission Statements" |
| LRM 155. | p. 133-134, | "Options of Transmission Statements" |
| LRM 156. | p. 444-445, | "GET" |
| LRM 157. | p. 453-454, | "PUT" |
| LRM 158. | p. 128-129, | "STANDARD FILES" |
| LRM 159. | p. 145-146, | "Print Files" |
| LRM 160. | p. 153, | "In record-oriented...deblocked automatically." |
| LRM 161. | p. 122, | "SEQUENTIAL, DIRECT and TRANSIENT Attributes" |
| LRM 162. | p. 123, | "KEYED Attribute" |
| LRM 163. | p. 160-176, | "Environment Attribute" |
| LRM 164. | p. 166, | "CONSECUTIVE, INDEXED, and REGIONAL Data Sets" |
| LRM 165. | p. 155, | "INTO Option" |
| LRM 166. | p. 155, | "FROM Option" |
| LRM 167. | p. 122, | "BUFFERED and UNBUFFERED Attributes" |
| LRM 168. | p. 122, | "BACKWARDS Attribute" |
| LRM 169. | p. 155-156, | "IGNORE Option" |
| LRM 170. | p. 176, | "Consecutive Organization" stopping before "SEQUENTIAL UPDATE" |
| LRM 171. | p. 176, | "SEQUENTIAL UPDATE" |
| LRM 172. | p. 170, | "IN-LINE CODE OPTIMIZATION (TOTAL)" |
| LRM 173. | p. 166-167, | "Optimization of Input/Output Operations" |

| | | |
|---|---|---|
| LRM 174. | p. 176, | "Indexed Organization" stopping at "KEYS" |
| LRM 175. | p. 156, | "KEYFROM and KEYTO Options" |
| LRM 176. | p. 156, | "KEY Option" |
| LRM 177. | p. 177-181, | "KEYS" |
| LRM 178. | p. 182, | "CREATING A DATA SET" |
| LRM 179. | p. 170-171, | "KEY CLASSIFICATION (GENKEY)" |
| LRM 180. | p. 182, | "DUMMY RECORDS" |
| LRM 181. | p. 182-183, | "SEQUENTIAL ACCESS" and "DIRECT ACCESS" |
| LRM 182. | p. 183-184, | "Regional Organization" stopping at "REGIONAL(1) ORGANIZATION" |
| LRM 183. | p. 184-187, | "REGIONAL(1) ORGANIZATION" |
| LRM 184. | p. 187-189, | "REGIONAL(2) ORGANIZATION" |
| LRM 185. | p. 189-191, | "REGIONAL(3) ORGANIZATION" |
| LRM 186. | p. 436-437, | "DELETE" |
| LRM 187. | p. 454-456, | "READ" |
| LRM 188. | p. 457-459, | "REWRITE" |
| LRM 189. | p. 460-461, | "WRITE" |
| LRM 190. | p. 459, | "UNLOCK" |
| LRM 191. | p. 123, | "EXCLUSIVE Attribute" |
| LRM 192. | p. 154, | "UNLOCK Statement" |
| LRM 193. | p. 157, | "NOLOCK Option" |
| LRM 194. | p. 397-398, | "BUILTIN" |
| LRM 195. | p. 117, | "If the parameter...can be passed." |
| LRM 196. | p. 344, | "Condition-handling Built-in Functions" |
| LRM 197. | p. 369-377, | "Condition Codes (ON Codes)" |
| LRM 198. | p. 377, | "Multiple Interrupts" |

| | | |
|---|---|---|
| LRM 199. | p. 279, | "Interlanguage Communication Facilities" through "or FORTRAN routines." |
| LRM 200. | p. 282-283, | "Passing Arguments to a PL/I Procedure" and "Invocation" |
| LRM 201. | p. 451, | Syntax rule 4. |
| LRM 202. | p. 452, | General rule 8. |
| LRM 203. | p. 440, | Syntax rule 5. |
| LRM 204. | p. 441, | General rule 7. |
| LRM 205. | p. 280-281, | "Passing Arguments to a COBOL or FORTRAN Routine" and "Invocation" |
| LRM 206. | p. 284-285, | "Interrupt Handling" |
| LRM 207. | p. 418-419, | "OPTIONS" |
| LRM 208. | p. 284, | "Establishing the PL/I Environment" |
| LRM 209. | p. 287, | "FORTRAN INTERFACE" |
| LRM 210. | p. 283-284, | "Using Common Storage" |

- - - - - - - -

| | | |
|---|---|---|
| CPG 3. | p. 160, | Figure 12.1. |
| CPG 4. | p. 65-81. | "Data Sets and Files" |
| CPG 5. | p. 83-90, | "Defining Data Sets for Stream Files" |
| CPG 6. | p. 88-90, | "Tab Control Table" |
| CPG 7. | p. 90, | "STANDARD FILES" |
| CPG 8. | p. 91, | "CREATING A CONSECUTIVE DATA SET" |
| CPG 9. | p. 91-95, | "ACCESSING A CONSECUTIVE DATA SET" |
| CPG 10. | p. 95-96, | "EXAMPLE OF CONSECUTIVE DATA SETS and "PUNCHING CARDS AND PRINTING" |
| CPG 11. | p. 96-99, | "INDEXED Data Sets" stopping at "CREATING AN INDEXED DATA SET" |

| | | |
|---|---|---|
| CPG 12. | p. 99-103, | "CREATING AN INDEXED DATA SET" stopping at "Dummy Records" |
| CPG 13. | p. 103, | "Dummy Records" |
| CPG 14. | p. 103-104, | "ACCESSING AN INDEXED DATA SET" |
| CPG 15. | p. 104, | "REORGANIZING AN INDEXED DATA SET" |
| CPG 16. | p. 104-105, | "EXAMPLES OF INDEXED DATA SETS" |
| CPG 17. | p. 105-106, | "REGIONAL Data Sets" stopping at "CREATING A REGIONAL DATA SET" |
| CPG 18. | p. 109-110, | "REGIONAL(1) Data Sets" |
| CPG 19. | p. 110, | "Regional(2) Data Sets" |
| CPG 20. | p. 110-118, | "Regional(3) Data Sets" |
| CPG 21. | p. 107-109, | "CREATING A REGIONAL DATA SET" |
| CPG 22. | p. 109, | "ACCESSING A REGIONAL DATA SET" |
| CPG 23. | p. 163-173, | "Linking PL/I and Assembler Language Modules" |
| CPG 24. | p. 48, | "MIXING OBJECT MODULES" through "bytes available" |

- - - - - - - - -

| | | |
|---|---|---|
| OPG 3. | p. 163, | Figure 12-1. |
| OPG 4. | p. 28, | "OPTIMIZE Option" |
| OPG 5. | p. 73-89, | "Data Sets and Files" |
| OPG 6. | p. 91-99, | "Defining Data Sets for Stream Files" |
| OPG 7. | p. 97-98, | "Tab Control Table" |
| OPG 8. | p. 98-99, | "STANDARD FILES" |
| OPG 9. | p. 101-102, | "CREATING A CONSECUTIVE DATA SET" |
| OPG 10. | p. 102-103, | "ACCESSING A CONSECUTIVE DATA SET" |
| OPG 11. | p. 103-105, | "EXAMPLE OF CONSECUTIVE DATA SETS" and "PUNCHING CARDS AND PRINTING" |

| | | |
|---|---|---|
| OPG 12. | p. 107-108, | "Indexed Data Sets" stopping at "CREATING AN INDEXED DATA SET" |
| OPG 13. | p. 108-114, | "CREATING AN INDEXED DATA SET" stopping at "Dummy Records" |
| OPG 14. | p. 114, | "Dummy Records" |
| OPG 15. | p. 114-115, | "ACCESSING AN INDEXED DATA SET" |
| OPG 16. | p. 115, | "REORGANIZING AN INDEXED DATA SET" |
| OPG 17. | p. 115-116, | "EXAMPLES OF INDEXED DATA SETS" |
| OPG 18. | p. 116-118, | "Regional Data Sets" stopping at "CREATING A REGIONAL DATA SET" |
| OPG 19. | p. 120-121, | "REGIONAL(1) Data Sets" |
| OPG 20. | p. 121-122, | "Regional(2) Data Sets" |
| OPG 21. | p. 122-124, | "Regional(3) Data Sets" |
| OPG 22. | p. 118-120, | "CREATING A REGIONAL DATA SET" |
| OPG 23. | p. 120, | "ACCESSING A REGIONAL DATA SET" |
| OPG 24. | p. 165-175, | "Linking PL/I and Assembler-Language Modules" |
| OPG 25. | p. 211-212, | "IBM System/360 Models 91 and 195" |

- - - - - - - - - - -

| | | |
|---|---|---|
| CTUG 2. | p. 85-87, | "Conversational Input" |
| CTUG 3. | p. 87-89, | "Conversational Output" |

- - - - - - - - - - -

| | | |
|---|---|---|
| OTUG 2. | p. 67, | "OPTIMIZE(TIME\|0\|2)\|NOOPTIMIZE" |
| OTUG 3. | p. 39-41, | "Conversational Input" |
| OTUG 4. | p. 41-42, | "Conversational Output" |

## REFERENCES FOR LESSONS 11-15

| | | |
|---|---|---|
| LRM 211. | p. 89, | "A based variable...area variables." |
| LRM 212. | p. 89, | "BASED VARIABLES" |
| LRM 213. | p. 89-90, | "LOCATOR QUALIFICATION" |
| LRM 214. | p. 90, | "POINTER VARIABLES" stopping at "Setting Pointer Variables" |
| LRM 215. | p. 91, | "ADDR BUILT-IN FUNCTION" |
| LRM 216. | p. 426, | General rules 7, 8, 12. |
| LRM 217. | p. 444, | General rules 3-5. |
| LRM 218. | p. 93-94, | "SELF-DEFINING DATA (REFER OPTION)" |
| LRM 219. | p. 397, | General rule 6. |
| LRM 220. | p. 403, | General rule 5. |
| LRM 221. | p. 94-95, | "LIST PROCESSING" |
| LRM 222. | p. 95-96, | "MULTIPLE GENERATIONS OF BASED VARIABLES", "NULL BUILT-IN FUNCTION", and "TYPES OF LIST |
| LRM 223. | p. 96-97, | "AREAS" stopping at "Offset Variables" |
| LRM 224. | p. 394, | "AREA" |
| LRM 225. | p. 99, | "EMPTY Built-In Function" |
| LRM 226. | p. 100, | "AREA ON-Condition" |
| LRM 227. | p. 98-99, | "ALLOCATE Statement with the IN Option" |
| LRM 228. | p. 97, | "Offset Variables" |
| LRM 229. | p. 98, | "Offset Expressions" |
| LRM 230. | p. 99-100, | "AREA ASSIGNMENT" |
| LRM 231. | p. 100, | "INPUT/OUTPUT OF AREAS" |
| LRM 232. | p. 156, | "SET Option" |
| LRM 233. | p. 456, | General rule 9. |

| | | |
|---|---|---|
| LRM 234. | p. 154, | "LOCATE Statement" |
| LRM 235. | p. 157-160, | "Processing Modes" |
| LRM 236. | p. 447, | "LOCATE" |
| LRM 237. | p. 437, | "DISPLAY" except General rule 4. |
| LRM 238. | p. 53, | "FETCH AND RELEASE STATEMENTS" |
| LRM 239. | p. 67-68, | "Dynamic Loading of an External Procedure" |
| LRM 240. | p. 442, | "FETCH" |
| LRM 241. | p. 457-458, | "RELEASE" |
| LRM 242. | p. 229, | Up to "Preprocessor Input and Output" |
| LRM 243. | p. 229-230, | "Preprocessor Input and Output" stopping at "Rescanning and Replacement" |
| LRM 244. | p. 463, | "%DECLARE" stopping at General rule 4. |
| LRM 245. | p. 232-233, | "Preprocessor Expressions" |
| LRM 246. | p. 462, | "%assignment Statement" |
| LRM 247. | p. 462-463, | "%DEACTIVATE" |
| LRM 248. | p. 461-462, | "%ACTIVATE" |
| LRM 249. | p. 230-231, | "Rescanning and Replacement" |
| LRM 250. | p. 232, | "Preprocessor Variables" |
| LRM 251. | p. 237, | "The % IF statement...IF statement." |
| LRM 252. | p. 464, | "%IF" |
| LRM 253. | p. 235, | "Preprocessor Do-group" |
| LRM 254. | p. 463-464, | "%DO" |
| LRM 255. | p. 237, | "The %GO TO statement...avoiding text." |
| LRM 256. | p. 464, | "%GO TO" |
| LRM 257. | p. 237, | "The preprocessor null statement...ELSE clause." |
| LRM 258. | p. 465, | "%null Statement" |

| | | |
|---|---|---|
| LRM 259. | p. 233-235, | "Preprocessor Procedures" stopping at "SUBSTR...Functions" |
| LRM 260. | p. 465-466, | "%PROCEDURE" |
| LRM 261. | p. 466, | "Preprocessor RETURN" |
| LRM 262. | p. 235-236, | "Inclusion of External Text" |
| LRM 263. | p. 464-465, | "%INCLUDE" |
| LRM 264. | p. 235, | "SUBSTR, LENGTH, and INDEX Built-In Functions" |
| LRM 265. | p. 118, | "Passing an Argument to the Main Procedure" |
| LRM 266. | p. 23, | "UNINITIALIZED VARIABLES" |
| LRM 267. | p. 217-227, | "Execution-time Facilities of the Checkout Compiler" |
| LRM 268. | p. 211-212, | "CHECK Condition" |
| LRM 269. | p. 218-220, | "CHECK and NOCHECK Statements" |
| LRM 270. | p. 431-432, | "CHECK" |
| LRM 271. | p. 447-448, | "NOCHECK" |
| LRM 272. | p. 208-209, | "ON Statement" |
| LRM 273. | p. 449, | General rule 6. |
| LRM 274. | p. 222-226, | "Current Status List" |
| LRM 275. | p. 453, | Syntax rules 3 through 5. |
| LRM 276. | p. 220-222, | "FLOW Statement" and "NOFLOW Statement" |
| LRM 277. | p. 442, | "FLOW" |
| LRM 278. | p. 448, | "NOFLOW" |
| LRM 279. | p. 240, | "Multitasking may allow...system overheads." |
| LRM 280. | p. 239, | First five paragraphs of "Multitasking" |
| LRM 281. | p. 241, | "Creation of Tasks" stopping at "CALL STATEMENT" |
| LRM 282. | p. 242, | "PRIORITY Option" up to "If the option does not appear..." |

| | | |
|---|---|---|
| LRM 283. | p. 242, | "PRIORITY OF TASKS" |
| LRM 284. | p. 24-25, | "TASK DATA" |
| LRM 285. | p. 242-243, | "PRIORITY Option" and "PRIORITY BUILT-IN FUNCTION AND PSEUDOVARIABLE" |
| LRM 286. | p. 430, | General rules 1, 2, 4, and 5. |
| LRM 287. | p. 423-424, | "TASK" |
| LRM 288. | p. 24, | "EVENT DATA" |
| LRM 289. | p. 241, | "EVENT Option" except last paragraph |
| LRM 290. | p. 430, | General rule 3. |
| LRM 291. | p. 407-408, | "EVENT" except General rule 10. |
| LRM 292. | p. 244, | Two paragraphs of "WAIT STATEMENT" |
| LRM 293. | p. 459-460, | General rules 1 through 4. |
| LRM 294. | p. 441-442, | "EXIT" |
| LRM 295. | p. 245-246, | "Termination of Tasks" |
| LRM 296. | p. 243-244, | "SHARING DATA BETWEEN TASKS" |
| LRM 297. | p. 244, | "SHARING FILES BETWEEN TASKS" |
| LRM 298. | p. 240, | "In general, the rules...in this chapter." |
| LRM 299. | p. 240-241, | "Tasking and Reentrability" |
| LRM 300. | p. 451, | Syntax rule 4. |
| LRM 301. | p. 437, | General rule 4. |
| LRM 302. | p. 171, | "NUMBER OF CHANNEL PROGRAMS (NCP)" |
| LRM 303. | p. 157, | "Note that...waited for." |
| LRM 304. | p. 156-157, | "EVENT Option" |
| LRM 305. | p. 244, | "An event variable...abnormal return." |
| LRM 306. | p. 408, | General rule 10. |
| LRM 307. | p. 436, | General rule 5. |

| | | |
|---|---|---|
| LRM 308. | p. 455,456, | General rule 7. |
| LRM 309. | p. 458-459, | General rule 5. |
| LRM 310. | p. 461, | General rule 3. |
| LRM 311. | p. 460, | General rules 5 through 8. |
| LRM 312. | p. 244-245, | "TESTING AND SETTING EVENT VARIABLES" |
| LRM 313. | p. 245, | "DELAY STATEMENT" |
| LRM 314. | p. 436, | "DELAY" |
| LRM 315. | p. 446, | "HALT" |
| LRM 316. | p. 217-218, | "Execution-time Facilities of the Checkout Compiler" up to "Tracing Facilities" |
| LRM 317. | p. 227, | "Program Amending" |

- - - - - - - - - - -

| | | |
|---|---|---|
| CPG 25. | p. 53, | "OVERLAY" |
| CPG 26. | p. 160-161, | "Return Codes" |
| CPG 27. | p. 175-186, | "PL/I Sort" |
| CPG 28. | p. 47, | "When Link-editing is Required." |
| CPG 29. | p. 47, | Two paragraphs of "Link-edit Stubs and Object Modules" |
| CPG 30. | p. 22, | "Link-edit Stub (SYSLIN)" and "Intermediate Text and Dictionary (SYSITEXT)" |
| CPG 31. | p. 19, | First paragraph of "Primary Input (SYSCIN or SYSIN)" |
| CPG 32. | p. 18, | "DATA STATEMENT" |
| CPG 33. | p. 17-18, | "PROCESS STATEMENT" |
| CPG 34. | p. 41-43, | "Batched Compilation" |
| CPG 35. | p. 23-35, | "Optional Facilities" |

| | | |
|---|---|---|
| CPG 36. | p. 43-45, | "Compile-time Processing (preprocessing)" |
| CPG 37. | p. 51-52, | "LINK EDITING FETCHABLE LOAD MODULES" |
| CPG 38. | p. 151-161, | "Program Checkout" |
| CPG 39. | p. 48-50, 221, | "Combining PL/I Modules from the Optimizing and Checkout Compilers" |
| CPG 40. | p. 30, | "ISASIZE Option" |

- - - - - - - - - - - -

| | | |
|---|---|---|
| OPG 26. | p. 59-62, | "OVERLAY STRUCTURES" |
| OPG 27. | p. 163-164, | "Return Codes" |
| OPG 28. | p. 177-187, | "PL/I Sort" |
| OPG 29. | p. 19, | "Input (SYSIN, or SYSCIN)" |
| OPG 30. | p. 22, | "Specifying Compiler Options in the PROCESS Statement" |
| OPG 31. | p. 41-43, | "Batched Compilation" |
| OPG 32. | p. 20-34, | "Optional Facilities" |
| OPG 33. | p. 31-34, | "EXECUTION-TIME OPTIONS" |
| OPG 34. | P. 43-45, | "Compile-time Processing (preprocessing)" |
| OPG 35. | p. 62-63, | "LINK EDITING FETCHABLE LOAD MODULES" |
| OPG 36. | p. 63, | "Combining PL/I Modules from the Optimizing and Checkout Compilers" |
| OPG 37. | p. 157-164, | "Program Checkout" |
| OPG 38. | p. 150-151, | "Multitasking Using Cataloged Procedures" |

- - - - - - - - - - - -

| | | |
|---|---|---|
| CTUG 4. | p. 111-131, | "Compiler Options" |
| CTUG 5. | p. 11-23, | "Creating and Updating PL/I Programs" |

| | | |
|---|---|---|
| CTUG 6. | p. 107-110, | "PLIC Command" |
| CTUG 7. | p. 133-191, | "Subcommands of PLIC Command" |
| CTUG 8. | p. 40-41, | Figure 1.6. |
| CTUG 9. | p. 42-44, | "Control Passing to Terminal" |
| CTUG 10. | p. 25-44, | "Debugging a Program" |
| CTUG 11. | p. 45-83, | "Debugging Techniques" |
| CTUG 12. | p. 133-191, | "Subcommands of PLIC Command" |

- - - - - - - - - - -

| | | |
|---|---|---|
| OTUG 5. | p. 55-70, | "Compiler Options" |
| OTUG 6. | p. 11-25, | "Creating and Updating PL/I Programs" |

- - - - - - - - - - -

OTHER 3. S. M. Prastein, editor. Argonne National Laboratory Computer User's Guide, chapters 9 and 12.

INDEX

*DATA control statement .... 13.3, 13.4, 13.5, 13.7
*PROCESS control statement . 13.4, 13.5, 13.7, 13.25

% symbol ..... 12.7
%ACTIVATE statement ..... 12.12, 12.17
%Assignment statement ..... 12.9
%DEACTIVATE statement ..... 12.11, 12.17
%DECLARE statement ... 12.8, 12.17
%DO statement ..... 12.14
%ELSE clause .. 12.13, 12.16
%END statement ..... 12.14, 12.17
%GO TO statement .... 12.15, 12.16
%IF
  clause ..... 12.13
  statement ... 12.13, 12.16
%INCLUDE statement .. 12.19, 13.11
%Null statement ..... 12.16
%PROCEDURE statement ..... 12.17
%THEN clause ..... 12.13

A format item ..... 7.19
Abnormal termination
  of a program ... 6.9, 12.3
  of a task ... 14.9, 14.11, 14.16
ABS builtin function .. 1.17
Abstract events ..... 14.17, 14.19
ACOS builtin function ..... 1.20
Active
  event variables .... 14.7, 14.14, 14.17
  identifiers ..... 12.7
  preprocessor function references ..... 12.17
  task variables ..... 14.6
Actual arguments ..... 5.9, 5.11
ADD builtin function .. 1.17
ADDR builtin function ..... 11.2
Address ..... 11.1
Adjustable extents .... 5.4, 5.6, 5.7, 5.10, 5.11, 10.7, 11.5, 11.7, 11.15
AFTER builtin function (ANSI) ..... 2.18
AGGREGATE compiler option .. 3.10
Aggregate parameters .. 5.13
Aggregates ..... 3.1, 14.8
ALIGNED attribute ..... 3.11
Alignment
  attributes ..... 3.11
  requirements .. 3.10, 3.11
ALL
  builtin function .... 10.5
  option ..... 13.16, 13.20, 13.21
ALLOCATE statement .... 5.7, 11.4, 11.10
ALLOCATION builtin function 10.5
AMDLIB ..... 13.1
ANSI Standard ... 0.2, 1.18, 1.20, 2.18, 4.9, 5.7, 5.9, 5.10, 5.12, 5.13, 5.14, 6.5, 6.7, 6.18, 6.19, 7.10, 7.15, 7.21, 7.22, 8.9, 9.11, 10.3, 10.4, 10.5, 11.3, 11.5, 11.14, 12.0, 12.6, 12.19, 13.16, 13.19, 14.0, 14.5
ANY builtin function .. 10.5
Area
  assignment .... 11.7, 11.9
  size ..... 11.7, 11.9
AREA
  attribute ..... 11.7
  condition ..... 11.9
Areas ..... 11.7
  empty ..... 11.7, 11.8
Argument lists
  empty ..... 10.4
Arguments ..... 5.9, 5.11, 12.17
  passing to main procedure 13.5
Arithmetic operations ..... 1.14
  conversion rules .... 1.15
  precision rules ..... 1.16
Arrays ..... 3.2
Arrays as parameters ..... 5.11, 5.13
Arrays of structures .. 3.14
ASIN builtin function ..... 1.20
Assignment statement ..... 1.12, 2.6, 2.9, 2.16
Assignments
  area ..... 11.7, 11.9

INDEX


arithmetic ..... 1.12
array ..... 3.4
bit string ..... 2.9, 2.16
BY NAME ..... 3.15
character string .... 2.6, 2.16
entry ..... 6.8
event ..... 14.7, 14.17
file ..... 7.3
fixed-length bit string .. 2.9
fixed-length character string ..... 2.6
label ..... 6.7
structure ..... 3.12, 3.15
varying-length bit string 2.16
varying-length character string ..... 2.16
Asterisk extents ..... 5.11, 10.7, 11.7
Asynchronous I/O ..... 14.0, 14.3, 14.13, 14.14
ATAN builtin function ..... 1.20
ATAND builtin function ..... 1.20
ATANH builtin function ..... 1.20
Attaching a task ..... 14.4, 14.5, 14.6, 14.7, 14.11
ATTENTION condition ... 15.5
Attributes
alignment ..... 3.11
arithmetic ..... 1.5, 1.6, 1.19
default ..... 1.4
definition of ..... 1.2
dimension ..... 3.3
file description .... 7.3, 7.8, 7.10
of a returned value ..... 5.14
of constants ..... 1.11
of parameters ..... 5.11, 5.12
role of ..... 1.2
scope ..... 4.4
storage class ..... 5.12, 11.3
string ..... 2.3
structure ..... 3.8, 3.9
AUTOMATIC attribute .... 5.6
Automatic call library ..... 13.25
Automatic variables ... 5.6, 5.15, 14.11
B format item ..... 7.19
BACKWARDS attribute .... 8.9
Balanced parentheses ..... 12.17
Base attributes ..... 1.5
Base elements (of structures) ..... 3.7
Based
references ..... 11.3
variables ..... 11.3
BASED attribute ..... 11.3
Based variables ..... 11.3
accessing records in buffers ..... 11.13
in list processing .. 11.6
in system programming .... 11.3
Batched compilation ... 13.4
BEFORE builtin function (ANSI) ..... 2.18
Begin blocks ..... 4.8, 6.2
BEGIN statement .. 4.8, 6.19
Belonging to ..... 4.3
BINARY
attribute ..... 1.5
builtin function .... 1.17
Binary tree ..... 11.6
BIT
attribute ..... 2.3
builtin function .... 2.18
Bit string
values ..... 2.2
variables ..... 2.3, 2.15
Bit strings
fixed-length ..... 2.9
in %IF clause ..... 12.13
in IF clause ..... 6.1
in WHILE clause ..... 6.4
varying-length ..... 2.16
Block
entry ..... 5.6
termination .... 5.6, 6.7, 14.9, 14.16
BLOCK execution option ..... 13.22
Block structure ..... 14.10
Blocks ..... 4.8, 6.7
BOOL builtin function ..... 2.18
Bounds ..... 3.3, 5.4, 5.11
BREAK key ..... 15.5
BUFFERED attribute .... 8.9, 11.14, 11.15
Buffers ..... 11.14, 11.15
BUILTIN attribute .... 10.3, 12.20
when required ..... 10.4
Builtin functions .... 5.12,

INDEX

10.1, 12.20, 13.1
arithmetic ..... 1.17
array arguments ..... 3.5
array-handling ..... 5.11, 10.5
condition-handling .. 10.5
mathematical .. 1.20, 10.5
miscellaneous ..... 10.5
multitasking ..... 14.6, 14.7, 14.9
storage control .... 10.5, 11.2, 11.6, 11.8, 11.12
stream I/O ..... 10.5
string-handling ..... 2.18
Builtin procedures ... 5.12, 12.3
BY clause ..... 6.5
BY NAME option ..... 3.15

C format item ..... 7.19
CALL command of TSO ... 15.7
CALL statement ... 4.1, 5.9, 12.3, 14.4, 14.5, 14.6, 14.7
Cataloged procedures ... 0.7
Checkout compiler ... 13.2
Optimizing compiler ..... 13.7
PLC series ... 13.2, 13.4, 13.23
PLO series ... 13.7, 13.23
CEIL builtin function ..... 1.17
CHAR builtin function ..... 2.18
CHARACTER attribute ... 2.3, 12.8
Character string
values ..... 2.1
variables ..... 2.3, 2.15
Character strings
fixed-length ..... 2.6
varying-length ..... 2.16
CHECK
condition .. 13.16, 13.17, 13.18, 13.24
statement .. 13.16, 13.18, 13.22
Checkout compiler
cataloged procedures ..... 13.2
compiler options ... 13.5, 13.6, 15.2
develcpment and testing .. 13.15
in TSO ..... 15.2
special debugging features 13.16
storage management .. 13.6
Checkpoint/Restart facilities ..... 12.5
CLOSE statement ..... 7.6
Closing a file ... 7.6, 7.8, 11.14, 11.15
COLLATE builtin function (ANSI) ..... 10.5
COLUMN format item .... 7.20
Comparison operations ..... 2.14, 6.1, 6.4
COMPATIBLE compiler and execution option ..... 13.23
Compile-time facility ..... 12.6
Compiler options ..... 13.5, 13.6, 13.8, 13.9, 15.2
Compiling
in batch ..... 13.10
in TSO ..... 13.10, 15.2, 15.7
several external procedures at once ... 13.4, 15.7
COMPLETION
builtin function ... 14.7, 14.12
pseudo-variable .... 14.17
Completion codes ..... 6.9, 12.3
Completion part of event variables ..... 14.7, 14.8, 14.12, 14.14, 14.17
COMPLEX
attribute ..... 1.5
builtin function .... 1.17
pseudo-variable ..... 1.18
Concatenation ..... 2.12
Concurrent execution ..... 14.1, 14.11
CONDITION
attribute ..... 6.17
condition ..... 6.17
Condition codes .. 6.9, 12.3
Condition prefixes .... 6.12
Conditions
computational ..... 6.11, 6.19
conversational processing 15.5
disablement of ..... 6.12
enablement of ..... 6.12
establishment of ... 6.13, 6.15, 14.11

INDEX


exceptional ..... 6.10
I/O ..... 7.5, 7.24, 8.17, 14.14
occurrence of ..... 6.11, 6.16, 14.11
program checkout ... 6.11, 13.17
programmer named .... 6.17
raising of ..... 6.13
status of ..... 6.12
storage control ..... 11.9
system action ..... 6.11, 15.5
CONJG builtin function ..... 1.17
CONNECTED attribute .. 5.13, 13.9
Connected references .. 3.6, 5.13
Consecutive datasets .. 8.12
altering ..... 8.15
creating ..... 8.13
retrieving ..... 8.14
CONSECUTIVE suboption ..... 8.11, 8.12
Constants
arithmetic ..... 1.11
bit string ..... 2.8
character string ..... 2.5
entry ..... 4.6, 12.2
file ..... 7.2
label ..... 6.7
named ..... 4.3
Containing procedure ... 4.3
Contextual declarations .... 1.4, 4.3
Control flow ..... 14.1
Control format items ..... 7.18, 7.20
Control variables ..... 6.5
Controlled
arguments ..... 5.9
parameters ..... 5.9
variables ..... 5.7, 5.9, 11.4
CONTROLLED attribute ... 5.7
Controlled DO groups ... 6.5
Conversational debugging ... 15.6
CONVERSION condition ..... 6.11, 7.19, 7.24, 7.26
Conversions ..... 1.12, 1.13
arithmetic to string ..... 2.11
between pointer and offset 11.12
during I/O ... 7.15, 7.16, 7.19
in argument/parameter matching ..... 5.12
in arithmetic assignments 1.13
in arithmetic operations . 1.15, 2.11
in string assignments .... 2.10
in string operations ..... 2.11
string to arithmetic ..... 2.11
COPY
builtin function (ANSI) .. 2.18
command of TSO ..... 15.1
option ..... 7.22
Coroutines ..... 14.19
COS builtin function .. 1.20
COSD builtin function ..... 1.20
COSH builtin function ..... 1.20
COUNT builtin function ..... 10.5
COUNT compiler option ..... 13.24
Creating a task ..... 14.4, 14.5, 14.6, 14.7, 14.11
Critical regions ..... 14.19
Cross sections of arrays ... 3.6

Data
format items .. 7.18, 7.19
list items ..... 7.13
lists ..... 7.13
problem ..... 2.11
program control ..... 4.7, 7.2, 11.1, 11.7, 11.10, 13.20, 14.7
DATA option ..... 7.16, 7.22
Data-directed transmission . 7.16, 13.20
DATAFIELD builtin function . 10.5
Dataset organization .. 8.11
Datasets ..... 7.1
DATE builtin function ..... 10.5
Ddname ..... 7.4
Deactivation of active identifiers .... 12.11
Debugging ..... 13.1, 15.6
features ... 13.16, 13.22, 13.24
DECAT builtin function

INDEX

(ANSI) ..... 2.18
DECIMAL
attribute ..... 1.5
builtin function .... 1.17
DECK compiler option .. 13.7
Declarations
inside preprocessor procedures ..... 12.17
of entry constants .. 4.7, 5.11, 5.12, 5.14
of entry variables ... 6.8
of label constants ... 6.7
role of ..... 1.3
scope of .. 4.3, 4.7, 4.8, 6.7
types of ..... 1.4
DECLARE statement ..... 1.3, 1.4, 4.3
Default
attributes ..... 1.4, 1.8, 3.11, 4.5, 4.9
declarations ... 5.8, 5.14
DEFAULT statement ..... 1.4, 4.9, 5.8
Default status of conditions 6.12
DEFINED attribute .... 3.16, 5.10
Defining ..... 3.16
ISUB ..... 3.18, 10.7
simple ..... 3.17
string overlay ..... 3.19
types of ..... 3.20
DELAY statement ..... 14.18
DELETE statement ..... 8.6, 9.7, 9.9, 9.10, 9.11, 14.14
Determining state of processor in TSO ..... 15.4
DIM builtin function .. 5.11
Dimension attribute .... 3.3
Direct access .... 8.4, 9.1, 9.2, 9.5, 9.6, 9.8, 9.9
DIRECT attribute ..... 8.4
Disablement of a condition . 6.12
DISPLAY statement .... 12.1, 14.12
DIVIDE builtin function .... 1.17
DO
groups ..... 6.2, 6.3, 6.5
specifications ..... 6.5
statements ..... 6.2, 6.4, 6.5
DO groups .... 6.2, 6.3, 6.5
controlled ..... 6.5
indexed ..... 6.5
iterative ..... 6.3, 6.5
non-iterative ..... 6.2
preprocessor ..... 12.14
WHILE-only ..... 6.4
DOT builtin function (ANSI) 10.5
Dummy
arguments .... 5.12, 12.17
records ... 9.7, 9.8, 9.9, 9.10
Dumps ..... 13.24
Dynamic loading ..... 12.2, 13.13
Dynamic storage ..... 11.7, 14.11
allocation ..... 5.1, 5.6, 5.7, 11.4, 11.10
deallocation (freeing) ... 5.6, 5.7, 11.4, 11.10

E format item ..... 7.19
EDIT
command of TSO .... 13.10, 15.1
option ..... 7.17, 7.22
Edit-directed transmission . 7.17
Editing features of Checkout compiler ..... 15.6
EDTIF symbolic parameter ... 13.25
ELSE clause ..... 6.1
Empty areas ..... 11.7, 11.8
EMPTY builtin function ..... 11.8
Enablement of a condition .. 6.12
END statement .... 4.1, 4.8, 6.2, 6.4, 6.5, 6.9, 14.9, 14.11
ENDFILE condition .... 7.24, 8.17, 14.14
ENDPAGE condition ..... 7.24
Entry
constants ..... 4.6, 4.7, 12.2
declarations .. 4.7, 5.11, 5.12, 5.14, 6.8
labels ..... 4.1, 4.6
values ..... 6.8
variables ..... 6.8
ENTRY
attribute ..... 4.7, 5.12, 12.17
statement ..... 5.16
Entry points ... 5.16, 13.13

INDEX

Environment .... 10.6, 13.20
other-language ..... 10.9
part of entry value .. 6.8
part of label value .. 6.7
ENVIRONMENT attribute .....
7.10, 8.10, 8.11, 8.12, 9.3
Equivalencing of data .....
3.16
ERF builtin function .. 1.20
ERFC builtin function .....
1.20
Error
detection ..... 13.1
messages .... 13.10, 15.2, 15.4
ERROR condition ..... 6.11, 6.16, 7.5, 7.8, 7.24, 11.3, 13.20, 14.11, 15.5
ERRORS compiler option .....
13.6
Establishment of conditions
6.13, 6.15, 14.11
Event
values ..... 14.7, 14.17
variables .... 14.7, 14.8, 14.10, 14.12, 14.14, 14.17, 14.19
EVENT
attribute ..... 14.7
option ..... 14.7, 14.12, 14.13
Events
abstract .... 14.17, 14.19
display ..... 14.12, 14.16
I/O ..... 14.13, 14.16
operator reply .... 14.12, 14.16
physical .... 14.16, 14.17
programmed .. 14.17, 14.19
task ..... 14.7
task completion .... 14.7, 14.8, 14.9, 14.16
Exceptional conditions .....
6.10, 10.8
EXCLUSIVE attribute .. 9.11, 14.15
Exclusive files ..... 9.11, 14.15
EXIT statement ..... 14.9
EXP builtin function .. 1.20
Explicit declarations .....
1.4, 4.3, 4.7, 6.7
Explicit opening .. 7.4, 7.8
Explicit pointer
qualification ... 11.3
Expressions ..... 2.14
arrays as operands in ....
3.5
element ..... 6.1
preprocessor ..... 12.9
structures as operands in
3.13
Extents .... 5.4, 5.10, 11.5
External
entry constants ..... 4.7
names ..... 4.4
procedures ..... 4.1, 4.3, 5.11, 12.2, 13.4, 13.13, 13.25
variables ..... 4.5, 5.8
EXTERNAL attribute .... 4.4, 4.7

F format item ..... 7.19
FETCH statement ..... 12.2
Fetchable procedures .....
12.2, 13.13, 13.25, 15.7
Field width ..... 7.19
File
constants ..... 7.2
values ..... 7.2
variables ..... 7.2
FILE
attribute ..... 7.2
option ..... 7.4, 7.22
File description attributes
7.3, 7.8, 7.10
Files ..... 7.1
FINISH condition ..... 6.11, 6.16, 14.11, 15.5
FIXED
attribute ..... 1.5, 12.8
builtin function .... 1.17
Fixed-length bit strings ...
2.9
Fixed-length character
strings ..... 2.6
FIXEDOVERFLOW condition ....
6.11
FLOAT
attribute ..... 1.5
builtin function .... 1.17
FLOOR builtin function .....
1.17
FLOW
compiler option .... 13.24
execution option ... 13.21
option ..... 13.16, 13.20, 13.21
statement ... 13.16, 13.21
Flow of control ..... 14.1, 14.11, 14.14
establishing .. 14.1, 14.4
terminating ..... 14.1

INDEX

Flow table .... 13.21, 13.24
Flow tracing .. 13.21, 13.24
Formal parameters ..... 5.9, 5.11
Format
  items ... 7.17, 7.18, 7.26
  list ..... 7.17, 7.18
FORMAT
  compiler option ..... 13.6
  statement ..... 7.21
FORTRAN
  communication with PL/I .. 10.6, 10.7, 10.8, 10.9, 10.10, 13.12
  comparison to PL/I .. 0.8, 1.5, 1.7, 1.11, 1.14, 1.16, 1.17, 1.20, 3.2, 3.3, 3.16, 4.1, 4.2, 4.5, 5.1, 5.9, 5.11, 5.12, 5.14, 5.16, 6.5, 7.13, 7.26, 8.18, 9.14, 10.6, 10.9
  library ..... 13.12
  suboption ..... 10.7
FREE statement .. 5.7, 11.4, 11.10
FROM option ..... 8.7, 11.14
FT06F001 DD statement ..... 13.12
Function references ... 4.1, 5.9, 5.14, 12.17

Generations of storage ..... 5.7, 5.15, 11.4, 11.15
GENERIC attribute ..... 5.17
Generic procedures .... 5.17
Generic selection ..... 5.17
GENKEY suboption ..... 9.5
GET statement ... 7.12, 7.25
Global checking ..... 15.3
GO subcommand of PLIC ..... 15.4, 15.5
GO TO
  out of block .. 6.7, 6.13, 14.14
  statement ..... 6.6, 15.5
GONUMBER compiler option ... 13.10
GOOPTS symbolic parameter .. 13.5, 13.8
GOPARM symbolic parameter .. 13.5, 13.8

HALT
  compiler option ..... 15.4
  statement ..... 15.5
Hardcopy record of PLIC use 15.2
HBOUND builtin function .... 5.11
HELP
  command of TSO ..... 15.2
  subcommand of PLIC .. 15.4
HIGH builtin function ..... 2.18

I/O transmission statements 7.7
  record ..... 8.6, 14.13
  stream ..... 7.12
Identifiers ..... 1.1
  active ..... 12.7, 12.10
IF
  clause ..... 6.1
  statement ..... 6.1, 6.2
IGNORE option .... 8.9, 9.2, 11.14
IMAG
  builtin function .... 1.17
  pseudo-variable ..... 1.18
Immediate-mode PL/I statements ..... 15.4, 15.5
Implementation extensions .. 11.3
Implementation-defined features ..... 0.2
Implicit declarations ..... 1.4, 4.3, 4.9
Implicit opening .. 7.4, 7.8
Implicit pointer qualification ... 11.3
IN option ..... 11.10
In-line I/O ..... 8.16
Inactive
  blocks ..... 6.7
  event variables .... 14.7, 14.17
  preprocessor procedures .. 12.17
  preprocessor variables ... 12.11
  task variables ..... 14.6
INCLUDE compiler option .... 13.11
Inclusion of text from a library ..... 12.19, 13.11
Index ..... 9.1
INDEX builtin function ..... 2.18, 12.20
Indexed datasets ..... 9.1
  altering ..... 9.6, 9.7

INDEX


creating ..... 9.4
retrieving ..... 9.5
Indexed DO group ..... 6.5
INDEXED suboption .... 8.11, 9.1
Infinite loops ..... 13.22, 14.8
Inheritance of on units .... 6.13, 14.11
INITIAL attribute ..... 5.3, 11.4
Initial Storage Area ..... 13.9, 14.20
Initial values ..... 5.3
Initialization ... 5.1, 5.3, 5.6, 5.7, 11.4, 11.15
INPUT attribute .. 7.10, 8.8
Input stream
data-directed ..... 7.16
edit-directed ..... 7.17, 7.19, 7.20
list-directed ..... 7.15
INSOURCE compiler option ... 13.11
INTER suboption ..... 10.8
Interactive debugging ..... 15.6
Interlanguage communication 13.12
common storage ..... 10.9
overview ..... 10.6
Intermediate text .... 13.1, 15.2
Internal
entry constants ..... 4.7
names ..... 4.4
procedures ..... 4.2
INTERNAL attribute .... 4.4, 4.7
Interpretation ..... 15.3
Interpreter phase ..... 13.1
Interrupting the terminal .. 15.5
INTO option ..... 8.7, 11.14
Invocation ..... 5.9, 5.15, 5.18, 6.7, 6.8, 6.13, 14.4, 14.11
IPLIC command of TSO .. 15.2
ISA ..... 13.9
ISASIZE execution option ... 13.9, 14.20
ISUB defining ... 3.18, 10.7
Iteration factor ..... 5.3, 7.18
Iterative DO group .... 6.3, 6.5

JCL
consecutive datasets ..... 8.13, 8.15
DCB parameters ..... 8.11, 14.13
ddname ..... 7.4
disposition parameter .... 9.11
errors and UNDEFINEDFILE condition ..... 7.5
for executing production programs ..... 13.8, 13.14
for fetchable procedures . 13.13
for interlanguage communication .. 10.10
for multitasking ... 14.20
for program development .. 13.25
for source text libraries 13.11
FT06F001 DD statement .... 13.12
indexed datasets ..... 9.3
NCP operand ..... 14.13
parameter for dummy records ..... 9.7
parameter for extended search ..... 9.10
PLC series cataloged procedures ..... 13.2, 13.4
PLIDUMP DD statement ..... 13.9, 13.24
PLO series cataloged procedures ..... 13.7
record format ..... 8.7
record length ..... 7.11
region request ..... 13.6
regional datasets ... 9.10
step condition code ..... 6.9, 12.3
SYSCIN DD statement ..... 13.3, 13.7, 13.25
SYSIN DD statement ..... 13.3, 13.7
SYSITEXT DD statement .... 13.1, 13.7
SYSLIB DD statement ..... 13.11
SYSOBJ DD statement ..... 13.1
SYSPRINT DD statement .... 13.14
testing programs during the course ..... 0.7

KEY
  condition ..... 9.12
  option .... 9.2, 9.6, 9.7, 11.14
Key sequence ..... 8.4, 9.1, 9.4
Keyed
  datasets .. 8.2, 8.5, 9.1, 9.8
  files ..... 11.14
  records ..... 8.2
KEYED attribute ..... 8.5
KEYFROM option ..... 9.2
Keys ..... 8.2
KEYTO option ..... 9.2
Known in ..... 4.3

Label
  constants ..... 6.7
  values ..... 6.8
  variables ..... 6.7
LABEL attribute ..... 6.7
Labels
  entry ..... 4.1, 4.6
  on a preprocessor statement ..... 12.15, 12.16
  statement ..... 6.6, 7.21
Language violations ... 11.3
LBOUND builtin function .... 5.11
LENGTH builtin function .... 2.18, 5.11, 12.20
Level numbers ..... 3.8
Library
  AMDLIB ..... 13.1
  automatic call ..... 13.25
  FORTRAN ..... 10.10, 13.12
  maintenance technique .... 13.25
  of executable programs ... 13.25
  PL/I libraries in TSO .... 15.7
  source text ..... 12.19, 13.11
  SYS1.FORTLIB ..... 13.12
  SYS1.PLIBASE ..... 13.23
  SYS1.PLICMIX ..... 13.23
  SYS1.PLITASK ..... 14.20
LIBRARY symbolic parameter . 13.23
LIKE attribute ..... 3.9
LINE
  format item ..... 7.20
  option ..... 7.22
Line numbers ... 13.10, 15.1
LINENO builtin function .... 10.5
LINESIZE option ..... 7.11
LINK command of TSO ... 15.7
Link editing ..... 12.2
Link-edit stub .. 13.1, 15.2
Linkage editor ..... 13.1, 13.12, 13.13, 13.23, 13.25
Linked list ..... 11.6
LIST option ..... 7.15, 7.22
List processing ..... 11.6
List structures ..... 11.6
  relocatable ..... 11.11
List-directed transmission . 7.15, 13.20
Listing control ..... 13.10, 15.1
LMESSAGE compiler option ... 15.2
Loader ..... 13.1, 13.12, 13.13, 13.23, 13.25
LOADGO command of TSO ..... 15.7
LOCATE statement ..... 11.15
Locate-mode
  I/O ..... 11.13
  input ..... 11.14
  output ..... 11.15
Locator
  qualification ..... 11.11
  variables ... 11.11, 13.23
Locators ..... 11.11
Locked records ..... 9.11
LOG builtin function .. 1.20
Logical operations ... 2.13, 6.1, 6.4
LOG10 builtin function ..... 1.20
LOG2 builtin function ..... 1.20
Loop termination ..... 6.5
Loops ..... 6.5
LOW builtin function .. 2.18

MACRO compiler option ..... 13.11
Macros ..... 12.6
Main procedure ... 4.1, 6.9, 13.5
MAIN suboption ..... 4.1
Major structures ..... 3.7
Major tasks .... 14.4, 14.6, 14.9
Mapping
  of arrays ..... 3.3, 10.7
  of self-defining data ....

11.5
of structures ..... 3.10, 11.14
Margins
source ..... 13.10
MARGINS compiler option .... 13.10
Matching
of arguments and parameters ..... 5.11, 12.17
of attributes in based references ..... 11.3, 11.14
of attributes in defining 3.20
of data lists and format lists ..... 7.18
of sizes of record and record variable .. 8.7
Mathematical builtin functions ..... 1.20
MAX builtin function .. 1.17
Maximum
length ..... 12.8
precision ..... 1.9
Message to the operator .... 12.1, 14.12
MIN builtin function .. 1.17
Minor structures ..... 3.7
Mixing Checker and Optimizer code ... 13.15, 13.23, 13.25, 15.7
MOD builtin function .. 1.17
Mode attributes ..... 1.5
Model 195 ..... 10.8
Modes of stream transmission 7.14
MONITOR subcommand of PLIC . 15.4
Move-mode I/O ..... 11.13
Multiple
entry points ..... 5.16
generations ... 5.7, 5.15, 11.4, 11.6
invocations ... 5.15, 6.7, 6.8, 14.4
Multiple assignment ... 1.12
MULTIPLY builtin function .. 1.17
Multiprogramming ..... 14.11
Multitasking
definition of ..... 14.0
when to use ..... 14.3

NAME
compiler option .... 13.25
condition ..... 7.24
Named constants .. 4.3, 4.4, 4.6, 6.7, 7.2
Names
scope of ..... 4.4
NCAL linkage editor option . 13.25
NCP suboption ..... 14.13
Nesting ..... 4.2, 4.8, 6.2
NOCHECK statement .... 13.18
NODIAGNOSE compiler option . 13.6
NOFLOW statement ..... 13.21
NOGONUMBER compiler option . 13.10
NOGOSTMT compiler option ... 13.10
NOLOAD compiler option ..... 13.7
NOLOCK option .. 9.11, 14.15
NOMAP suboption ..... 10.7, 10.8
NOMAPIN suboption .... 10.7, 10.8
NOMAPOUT suboption ... 10.7, 10.8
Non-iterative DO group ..... 6.2
Non-preprocessor text ..... 12.7, 12.8, 12.10, 12.14, 12.17
NONUMBER compiler option ... 13.10
NOOFFSET compiler option ... 13.10
NORESCAN option ..... 12.12
Normal return (from an on unit) .... 6.13, 6.16, 14.14
Normal termination
of a program ... 6.9, 12.3
of a task ... 14.9, 14.11, 14.16
NORUN compiler option ..... 13.1
NOSOURCE compiler option ... 13.10
NOSTMT compiler option ..... 13.10
Not ready ..... 14.5, 14.8, 14.13, 14.18
NULL builtin function ..... 11.6
Null statement .... 6.1, 6.2
Null string
constant ..... 2.17
value ..... 2.17
NUMBER compiler option .....

13.10
Number of digits attribute . 1.5

OBJECT compiler option ..... 13.1
Object module ..... 15.2
Occurrence of a condition .. 6.11, 6.16, 14.11
OFFSET
  attribute ..... 11.11
  builtin function ... 11.12
  compiler option .... 13.10
Offsets ..... 11.11
ON statement .. 6.13, 13.16, 13.19, 13.24, 14.11
On units ..... 6.13, 14.11, 14.14
ONCHAR
  builtin function .... 10.5
  pseudo-variable ..... 10.5
ONCODE builtin function .... 10.5
ONCOUNT builtin function ... 10.5
ONFILE builtin function .... 10.5
ONKEY builtin function ..... 10.5
ONLOC builtin function ..... 10.5
ONSOURCE
  builtin function .... 10.5
  pseudo-variable ..... 10.5
OPEN statement ... 7.4, 7.8, 7.11
Opening a file .... 7.4, 7.8
Operational expressions .... 2.14
Operations
  arithmetic ..... 1.14
  comparison .... 2.14, 6.1, 6.4
  logical ... 2.13, 6.1, 6.4
  precedence of ..... 2.14
  priority of ..... 2.14
  string ..... 2.12
Operator
  communicating with ..... 12.1, 14.12
  reply to message ... 12.1, 14.12
OPT compiler option ... 13.9
Optimization ..... 13.9
  effect on conditions ..... 6.19
OPTIMIZE compiler option ... 6.19
Optimizing compiler
  cataloged procedures ..... 13.7
  compiler options ... 13.8, 13.9
  debugging ..... 13.24
  optimized production code 13.15
  storage management .. 13.9
Options
  compiler ..... 13.5, 13.6, 13.8, 13.9, 15.2
OPTIONS
  attribute ..... 10.8
  option .. 4.1, 10.7, 14.11
  subcommand of PLIC .. 15.4
  symbolic parameter ..... 13.5, 13.8
ORDER option ..... 6.19
Other-language environment . 10.9
OUTPUT attribute ..... 7.10, 8.8
Output stream
  data-directed ..... 7.16
  edit-directed ..... 7.17, 7.19, 7.20
  list-directed ..... 7.15
OVERFLOW condition .... 6.11
Overlap of CPU and I/O ..... 14.3, 14.13
Overlays ..... 12.2

P format item ..... 7.19
Page
  footings ..... 7.24
  headings ..... 7.24
PAGE
  format item ..... 7.20
  option ..... 7.22
PAGENO builtin function (ANSI) ..... 10.5
PAGESIZE option ..... 7.11, 7.24
Parallel
  computation ..... 14.0
  execution .... 14.4, 14.19
Parameter attribute descriptions ... 5.11, 5.12
Parameter lists .. 4.1, 4.3, 5.16
Parameters ..... 4.3, 5.9, 5.11, 12.17
  declarations of ..... 4.3, 5.9

INDEX

Parent tasks .... 14.4, 14.5
Partial declarations ... 4.9
Passing an argument to the main procedure .. 13.5
Passing control to the terminal ..... 15.3, 15.4, 15.5
Physical events ..... 14.16, 14.17
Physical sequence ..... 8.4, 9.9, 9.10
Picture
  characters ..... 2.21
  specification ..... 2.21
PICTURE attribute ..... 2.20
Pictured data ..... 2.20
  character ..... 2.21
  numeric ..... 2.22
PLC series cataloged procedures ..... 13.2, 13.4, 13.23
PLI dataset type .... 13.10, 15.1
PLIC command of TSO ... 15.2
PLICKLGN logon procedue .... 15.2
PLIDUMP
  builtin procedure .. 13.24
  DD statement ..... 13.9, 13.24
PLIF dataset type ... 13.10, 15.1
PLIRETC builtin procedure .. 12.3
PLISRTA builtin procedure .. 12.4
PLISRTB builtin procedure .. 12.4
PLISRTC builtin procedure .. 12.4
PLISRTD builtin procedure .. 12.4
PLISTART ..... 13.13
PLO series cataloged procedures ..... 13.7, 13.23
Point of interrupt ... 6.13, 14.14
Pointed to by ..... 11.3
Pointer
  qualification ..... 11.3
  values ..... 11.1
POINTER
  attribute ..... 11.1
  builtin function ... 11.12
Pointer qualification
  explicit ..... 11.3
  implicit ..... 11.3
POLY builtin function ..... 10.5
POSITION attribute ... 3.19, 3.20
POSTLIB symbolic parameter . 13.12
Precedence of operations ... 2.14
Precision
  attributes ..... 1.5
  rules ..... 1.16
PRECISION builtin function . 1.17
Precisions
  default ..... 1.8
  maximum ..... 1.9
Preprocessor ... 12.6, 13.11
  DO groups ... 12.13, 12.14
  expressions ..... 12.9, 12.13, 12.20
  procedures ..... 12.17
  scan ..... 12.7
  statements .. 12.7, 12.13, 12.17
  variables ..... 12.8
PRINT
  attribute ..... 7.10
  operand ..... 15.2
Print files .... 7.10, 7.15, 7.20, 7.22, 7.23, 7.26
Priority
  assigning ..... 14.5
  changing ..... 14.5, 14.6
  definition of ..... 14.5
  determining ... 14.5, 14.6
  relative ..... 14.5, 14.6
  used in scheduling tasks . 14.5
PRIORITY
  builtin function .... 14.6
  option ..... 14.5
  pseudo-variable ..... 14.6
Priority of operations ..... 2.14
Problem data ..... 2.11
Procedure
  blocks ..... 4.8
  names ..... 4.6
PROCEDURE statement ... 4.1, 4.8, 5.14, 5.15, 5.16, 6.19
Procedures
  external ..... 4.1, 4.3, 5.11, 12.2, 13.4, 13.13, 13.25
  fetchable ... 12.2, 13.13, 13.25, 15.7
  generic ..... 5.17

internal ..... 4.2
main ..... 4.1, 6.9, 13.5
preprocessor ..... 12.17
recursive ..... 5.6, 5.15, 6.7, 6.8
PROD builtin function ..... 10.5
Program
amending ..... 15.6
development ..... 13.1, 13.15, 13.25
termination of ..... 6.9
Program control data ..... 2.11, 4.7, 7.2, 11.1, 11.7, 11.10, 13.20, 14.7
Programmed events ... 14.17, 14.19
Programmer-named conditions 6.17
Prompter ..... 15.2
Prompting for input ... 7.25
Prompts ..... 15.3, 15.5
Pseudo-variables ..... 10.1
arithmetic ..... 1.18
array arguments ..... 3.5
condition-handling .. 10.5
multitasking ..... 14.9, 14.17
string-handling ..... 2.19
PUT statement .. 7.12, 7.25, 13.16, 13.20, 13.21

Qualified
conditions .... 6.17, 7.5, 7.24, 8.17, 9.12, 13.17
names ..... 3.7

R format item ..... 7.21
Raising of conditions ..... 6.13
Reactivation of inactive identifiers .... 12.12
READ statement ... 8.6, 8.7, 8.9, 8.14, 8.15, 9.2, 9.5, 9.6, 9.9, 9.10, 9.11, 11.14, 14.14, 14.15
Ready ..... 14.5, 14.8
REAL
attribute ..... 1.5
builtin function .... 1.17
pseudo-variable ..... 1.18
Record
datasets ..... 7.9, 8.1
files .... 7.9, 8.1, 11.14
variables ..... 8.1, 8.7
RECORD
attribute ..... 7.9, 8.1
condition .... 8.17, 14.14
Recorded keys .... 8.2, 9.1, 9.2, 9.10
Records ..... 8.2, 11.14
Recursion ... 6.7, 6.8, 14.4
RECURSIVE option ..... 5.15
Recursive procedures .. 5.6, 5.15, 6.7, 6.8
Reentrant code ..... 14.11
REENTRANT suboption .. 14.11
Refer object ..... 11.5
REFER option .... 11.5, 11.7
Region number .... 9.8, 9.9, 9.10
Region request .. 13.6, 15.2
Regional datasets
altering ..... 9.9
creating ..... 9.9
retrieving ..... 9.9
REGIONAL suboption .... 8.11
Regional(1) datasets ... 9.9
REGIONAL(1) suboption .. 9.8
Regional(2) datasets .. 9.10
REGIONAL(2) suboption .. 9.8
Regional(3) datasets .. 9.10
REGIONAL(3) suboption .. 9.8
Regions ..... 9.8
Relative priorities .. 14.5, 14.6, 14.18
RELEASE statement ..... 12.2
Relocatable list structures 11.11
Remote format items .. 7.18, 7.21
REORDER option .. 6.19, 13.9
REPEAT builtin function .... 2.18
Repetition factors ..... 2.5
Repetitive specifications .. 7.13
Replacement
of active identifiers .... 12.10
of active preprocessor function references .. 12.17
REPLY option ... 12.1, 14.12
REPORT execution option .... 13.9
RESCAN option ..... 12.12
Rescanning ..... 12.10
Return codes ..... 6.9, 12.3
RETURN statement ..... 5.14, 5.16, 6.9, 12.17, 14.9, 14.11
Returned values ..... 4.1,

5.14, 12.17
RETURNS
attribute ..... 5.14
option .. 4.1, 5.14, 5.16, 12.17
REVERSE builtin function (ANSI) ..... 2.18
REVERT statement ..... 6.15, 14.11
REWRITE statement ..... 8.6, 8.7, 8.15, 9.2, 9.6, 9.9, 9.10, 9.11, 11.14, 14.14
Rewriting in place .... 8.15
ROUND builtin function ..... 1.17
Run-time environment .. 10.6

Scale attributes ..... 1.5
Scale factor attribute ..... 1.5
Scheduling policy ..... 14.5
Scope ..... 4.3
of a declaration .... 4.3, 4.7, 4.8, 6.7
of a name .... 4.4, 12.17, 14.10
Scope attributes .. 4.4, 5.8
Self-defining data .... 11.5
Sending control to the terminal ..... 15.3, 15.4, 15.5
SEQUENCE compiler option ... 13.10
Sequential access ..... 8.4, 8.12, 9.1, 9.2, 9.4, 9.5, 9.6, 9.8, 9.9, 9.10, 11.14, 11.15
SEQUENTIAL attribute ... 8.4
SET option .... 11.4, 11.14, 11.15
Shared
data ..... 14.10
database ..... 9.11, 14.15
files ..... 14.10, 14.15
variables ..... 14.10
Sharing data among tasks ... 14.10
SIGN builtin function ..... 1.17
SIGNAL statement ..... 6.16
Signaling an attention ..... 15.5
Simple defining ..... 3.17
SIN builtin function .. 1.20
SIND builtin function ..... 1.20
SINH builtin function ..... 1.20
SIZE
compiler option .... 13.6, 13.12, 14.20
condition .... 6.11, 7.24, 13.24
SKIP
format item ..... 7.20
option ..... 7.22
SMESSAGE compiler option ... 15.2
SNAP option .. 13.16, 13.19, 13.20, 13.21, 13.24
SORT utility ..... 12.4
Source
input conventions .. 13.3, 13.7, 13.10, 15.1
listings ..... 13.11, 15.2
margins ..... 13.10
record formats ..... 13.10
sequence field ..... 13.10
SOURCE compiler option ..... 13.10, 13.11, 15.2
Source keys ..... 8.2, 9.2, 9.9, 9.10
Source text
inclusion of ..... 12.19, 13.11
libraries ... 12.19, 13.11
Spill file ..... 13.6, 15.2
SQRT builtin function ..... 1.20
Standard files ..... 7.23
Standard system action ..... 6.13, 6.14
State of processor implied by prompting text .... 15.3
Statement labels ..... 6.6, 7.21
Statement numbers .... 13.10
STATIC attribute ..... 5.5
Static variables ..... 5.5, 5.15
STATUS
builtin function .... 14.9
pseudo-variable ..... 14.9
Status of conditions .. 6.12
Status part of event variables ..... 14.7, 14.9, 14.14, 14.17
STEP execution option ..... 13.22
STEPLINES execution option . 13.22
STMT compiler option ..... 13.10

INDEX

STOP statement ... 6.9, 14.9
Storage allocation .... 5.1, 5.6, 5.7, 11.4
  in a buffer ..... 11.15
  in an area .. 11.7, 11.10, 11.11
Storage class ..... 12.17
  attributes ..... 5.2, 5.8, 5.9, 5.10, 11.3
Storage management
  Checkout compiler ... 13.6
  Optimizing compiler ..... 13.9
  report ..... 13.9
Stream
  datasets ..... 7.9
  files ..... 7.9
STREAM attribute ..... 7.9
STRING
  builtin function ... 2.18, 10.5
  option ..... 7.22
  pseudo-variable .... 2.19, 10.5
String operations ..... 2.12
String overlay defining .... 3.19
STRINGRANGE condition ..... 6.11, 13.24
STRINGSIZE condition .. 6.11
Structure
  mapping ..... 3.10
  qualification ..... 3.7
Structures ..... 3.7
  as record variables .. 8.7
  of arrays ..... 3.14
Structuring attributes ..... 3.8
Subcommands of PLIC ... 15.4
SUBSCRIPTRANGE condition ... 6.11, 13.24
Subscripts ..... 3.2
SUBSTR
  builtin function ... 2.18, 12.20
  pseudo-variable ..... 2.19
Subtasks ..... 14.4, 14.9
SUBTRACT builtin function (ANSI) ..... 10.5
SUM builtin function .. 10.5
Synchronization of tasks ... 14.1, 14.10, 14.15, 14.19
Syntax checking ..... 15.3
SYSCIN DD statement .. 13.3, 13.7, 13.25
SYSIN DD statement ... 7.23, 13.3, 13.7
SYSITEXT DD statement ..... 13.1, 13.7
SYSLIB DD statement .. 13.11
SYSOBJ DD statement ... 13.1
SYSPRINT DD statement ..... 7.23, 13.14, 13.21, 13.22, 15.4
  allocation in TSO ... 15.2
System action ... 6.13, 6.14
SYSTEM option ..... 6.14
System programming .... 11.3

TAN builtin function .. 1.20
TAND builtin function ..... 1.20
TANH builtin function ..... 1.20
Task
  creation ..... 14.1, 14.4, 14.5, 14.6, 14.7, 14.11
  definition of ..... 14.1
  determining normality of termination ..... 14.9
  determining whether still active ..... 14.7
  major ... 14.4, 14.6, 14.9
  parent ..... 14.4, 14.5
  synchronization .... 14.1, 14.10, 14.15, 14.19
  termination .. 14.1, 14.9, 14.11, 14.16
  values ..... 14.6
  variables ..... 14.6
TASK
  attribute ..... 14.6
  option ..... 14.4, 14.6
TASKLIB symbolic parameter . 14.20
Temporaries ..... 5.12
Terminal
  receiving control .. 15.3, 15.4, 15.5
TERMINAL compiler option ... 15.2
Terminal I/O ..... 7.25
Termination
  of a block ..... 5.6, 6.7, 14.9, 14.16
  of a loop ..... 6.5
  of a program ..... 6.9
  of a task ... 14.9, 14.11, 14.16
THEN clause ..... 6.1
TIME builtin function ..... 10.5
TITLE option ..... 7.4
TO clause ..... 6.5

TOTAL
  option ..... 8.16
  suboption ..... 13.9
Traceback ..... 13.19
Tracing
  assignments ..... 13.17, 13.18, 13.22
  flow of control ... 13.21, 13.24
TRANSLATE builtin function . 2.18
Translator phase ..... 13.1
TRANSMIT condition ... 7.24, 8.17, 14.14
Trees ..... 11.6
TRUNC builtin function ..... 1.17
TSO ..... 7.25
  CALL command ..... 15.7
  compiling under TSO ..... 13.10, 15.2, 15.7
  COPY command ..... 15.1
  creating source datasets . 13.10, 15.1
  EDIT command ..... 13.10, 15.1
  HELP command ..... 15.2
  IPLIC command ..... 15.2
  LINK command ..... 15.7
  LOADGO command ..... 15.7
  PLIC command ..... 15.2
  PLICKLGN logon procedure . 15.2

UNALIGNED attribute ... 3.11
UNBUFFERED attribute .. 8.9, 14.14
Unconnected references ..... 3.6, 5.13
UNDEFINEDFILE condition .... 7.5, 7.8, 7.24, 8.17
UNDERFLOW condition ... 6.11
Uninitialized variables .... 5.3, 13.6, 13.20
UNLOCK statement ..... 9.11, 14.15
UNSPEC
  builtin function ... 2.18, 10.5, 11.3
  pseudo-variable .... 2.19, 10.5
UPDATE attribute ..... 8.8

VARIABLE attribute ..... 6.8
Variables ..... 1.1
  aggregate ..... 3.1
  automatic ..... 5.6, 14.11
  base (in defining) .. 3.17
  based ..... 11.3
  controlled ..... 5.7, 11.4
  defined ..... 3.17, 5.10
  element ..... 3.1, 6.5
  entry ..... 6.8
  event ..... 14.7, 14.8, 14.10, 14.12, 14.14, 14.17
  external ..... 4.5, 5.8
  file ..... 7.2
  label ..... 6.7
  locator ..... 11.11, 13.23
  preprocessor ..... 12.8
  program control .... 13.20
  representation of arithmetic data .. 1.7
  representation of string data ..... 2.17
  static ..... 5.5
  task ..... 14.6
  uninitialized ..... 5.3, 13.6, 13.20
VARYING attribute .... 2.15, 2.17
Varying-length bit strings . 2.15, 2.17
Varying-length character strings .. 2.15, 2.17, 12.8
VERIFY builtin function .... 2.18

WAIT statement ..... 14.8, 14.10, 14.12, 14.13, 14.14, 14.16, 14.17, 14.19
Waiting on an event ... 14.8
WHILE clause ..... 6.4, 6.5
WHILE-only DO group .... 6.4
WRITE statement .. 8.6, 8.7, 8.13, 9.2, 9.4, 9.6, 9.9, 9.10, 11.15, 14.14

X format item ..... 7.20

ZERODIVIDE condition .. 6.11

ARGONNE NATIONAL LAB WEST
3 4444 0084702 2