An Informal PL/I Roundtable, Collection One by R.W. Mitchell, Editor Carlos Christensen

Mathew Myszewski Caral Sampson

Massachusetts
COMPUTER ASSOCIATES, Inc.

Massachusetts
COMPUTER ASSOCIATES, Inc. / Lakeside Office Park • Wakefield, Massachusetts 01880 • 245-9540

An Informal PL/I Roundtable, Collection One

by

R.W. Mitchell, Editor Carlos Christensen Mathew Myszewski Caral Sampson

Technical Report CA-6704-0511 April 5, 1967

Abstract

Collection One of the PL/I Roundtable contains informal remarks on several PL/I issues. The conceptual basis of PL/I and its development history are noted for background information. The significance of PL/I for both quasi-programmers and professional programmers is discussed and several technical problems relating to implementation, the host operating system and efficiency are raised. Also included is a concise comparison of PL/I and COBOL.

<u>Preface</u>

The PL/I Roundtable is a collection of informal remarks on PL/I. At this time there has been no attempt to correlate the contributions or to mold them into a single, continuous presentation. Rather, the intent of this document is to provide a mechanism for collecting reactions, suspicions and discoveries relative to PL/I as a result of work by COMPASS staff members with PL/I since its inception. This document is labelled "Collection One" because several of us wish to comment further and these comments will probably be collected together as "Collection Two". PL/I being what it is, there is a likelihood that this flow of reactions and discoveries will continue, so we have left open the possibilities of further collections of the PL/I Roundtable.

R.W.M.

<u>Acknowledgements</u>

The comments documented in this Roundtable were motivated by COMPASS staff members participating in a PL/I Briefing given at MITRE Corporation, seminars on PL/I given at the National Security Agency, and the development of the NICOL Programming Language (a variant of PL/I) for Bell Telephone Laboratories.

Contents

Abstract	i
Preface	ii
Acknowledgements	ii
The Conceptual Basis of PL/ICarlos Christensen	1
The History of PL/IR.W. Mitchell	6
A Comparison Between PL/I and COBOL Caral Sampson	10
The Operating System as a Host for PL/I Mathew Myszewski	15
Some Major Efficiency Questions in PL/I Mathew Myszewski	18
Implications of PL/I for the Professional Programmer Caral Sampson	21
The Quasi-Programmer and PL/I Caral Sampson	2 6
Freedom of Expression is not License for Anarchy Mathew Myszewski	31
PL/I Problems for the Implementor	34

The Conceptual Basis of PL/I

Carlos Christensen

PL/I brings together, in a single unified programming language, facilities for programming in three areas of application:

- -- scientific computing,
- -- business data processing, and
- -- systems programming.

These areas of computing application have previously been served by separate programming languages, and the task of designing PL/I consisted of studying existing languages, rationalizing the conflicts which arose, and joining their various facilities in a single language. Although PL/I does contain some unique and novel features, it principally represents an outstanding effort toward the synthesis of existing and well-known programming features.

We will first consider the sources on which PL/I drew in each of the three applications areas mentioned above. In the area of <u>scientific</u> <u>computing</u>, PL/I had well-developed and well-known precedents in FORTRAN, ALGOL, and others. At the time PL/I development began, FORTRAN IV and ALGOL 60 were complete and well-specified languages, and many of their

unsuspected advantages and shortcomings had come to light. Although the designers of PL/I did extend and generalize some features of FORTRAN and ALGOL, a "purely scientific" program in PL/I program is quite intelligible to a programmer who is familiar with either FORTRAN or ALGOL.

In the area of <u>business</u> <u>data processing</u>, PL/I also had a well established precedent in COBOL. COBOL was the first high-level programming language to receive a careful definition intended for use by the entire computing community, and this definition was available to the designers of PL/I. COBOL's contribution was principally in the areas of data description and input-output. Since FORTRAN and ALGOL had relatively primitive facilities in this area, it was possible to accept COBOL's contribution without serious conflict. On the other hand, COBOL's "English-like" language for arithmetic expressions ('ADD A TO B GIVING C') was in direct conflict with scientific programming language (C = A+B'), and it was discarded.

In the area of <u>systems programming</u>, PL/I did not have a generally accepted precedent. Although systems programming has a long history and although many appropriate languages have been produced, no useful widely accepted standard has appeared. Instead, this area is crowded with competing languages, none of which can claim to be representative of the whole area. Accordingly, the designers of PL/I included a minimum, primitive facility for the list-processing, bit-picking and interrupt-handling required by systems programmers. The list processing capability is in many ways similar to the arbitrary linking of arbitrary nodes in L6 and CORAL.

Thus PL/I is not related in the same way to each of the three applications mentioned above. For scientific computing and business data processing, it represents a synthesis of some very advanced and powerful programming features and purports to represent a final disposition and consensus in these areas. For systems-programming, it represents a basic facility from which the PL/I programmer can build, but does not offer him truly 'high-level' user-oriented facilities.

We have considered the way in which the PL/I facilities were drawn from sources in each of the applications areas. The remaining problem was the joining of these facilities into a single integrated programming language. Here the really basic issues of programming language design arose: the character set for the language, the structuring of the program, the incorporation of operating system features into the language, the flow of control in the program, and so on. Initially, the design committee attempted to adopt FORTRAN as its basic language model; but as work progressed, the basic model moved more and more toward ALGOL until only a residue of FORTRAN concepts remained.

PL/I resembles FORTRAN in certain immediately apparent but relatively unimportant aspects, as follows:

- -- PL/I uses, in addition to special characters and digits, only upper case letters, while ALGOL made use of lower case, upper case, and bold face letters;
- -- PL/I uses the keywords of FORTRAN rather than those of ALGOL. For example, a loop in PL/I and FORTRAN begins with 'DO', while it begins with 'FOR' in ALGOL; and

-- PL/I similarly carries over some of the uses of special characters from FORTRAN. For example, exponentiation is '**', while it is '†' in ALGOL.

The programmer should not be deceived by these superficial similarities. PL/I started out to be FORTRAN VI, but it has become something very different indeed.

On the other hand, PL/I resembles ALGOL 60 in some very basic and important ways, as follows:

- -- PL/I is a format-free language, as is ALGOL; that is, it does not depend on card columns for its interpretation as does FORTRAN;
- -- PL/I is heavily dependent on the concept of declaration of identifiers, which was first introduced and proved in ALGOL;
- -- PL/I uses "block structure" to gather a sequence of statements together into a single unit and to control the scope of declaration; and
- -- PL/I uses a method of subroutine definition which is an improved and extended version of the ALGOL procedure definition.

In many ways, then, PL/I is an improved and extended version of ALGOL. ALGOL has been described in a widely available paper*, and that paper is recommended as collateral reading for the study of PL/I.

^{*} Naur, Peter (Ed.) Revised report on the algorithmic language ALGOL 60. Comm. ACM 6 (January 1963), 1-17.

The designers of PL/I did not, however, accept ALGOL uncritically as a linguistic framework for PL/I. They recognized and, for the most part, corrected the known defects of ALGOL.

For example,

- -- Integer labels were eliminated because they are easily confused with problem data;
- -- The loop parameters were defined statically instead of dynamically to avoid confusing anomalies;
- -- "Call-by-name" was restricted to eliminate cases which cannot be implemented efficiently;
- -- The assumption that all procedures and functions could be used recursively was dropped as an unnecessary bookkeeping expense;
- -- Side effects within the evaluation of an expression were defined in a simpler way; and
- -- Declaration of formal parameters was made mandatory.

Thus the designers of PL/I drew on the experience which had been gained from the "ALGOL experiment" to produce a linguistic base for PL/I which is superior to ALGOL.

The History of PL/I

R.W. Mitchell

The development of PL/I is an outgrowth of a longstanding desire in the SHARE FORTRAN project for a more complete, less restrictive language. The original language specifications in spring of 1964 were easily recognizable as an extension of FORTRAN. The language has undergone many revisions since then and now more closely resembles a conglomerate of FORTRAN, ALGOL, and COBOL.

The three major areas of the PL/I history are language specification, industry activity, and compiler development. The following notes the major events in each area.

-- <u>Industry Activity</u>

October 1963 -	SHARE/IBM 3 \times 3 Committee on advanced language development formed, all members of systems and scientific programming background.
May 1964 -	3 \times 3 Committee expanded by including a GUIDE representative and some of the more vocal responders to Report 1.
August 1964 -	SHARE NPL Project formed with many SHARE participants, one CO-OP observer and one X3.4.3.1/ Honeywell observer.
March 1965 -	SHARE closes door to all outside observers.
March 1965 -	X3.4 votes 'NO' on any NPL activity other than observing progress, this was after an offer of complete information (but no control) from IBM and prolonged debate on language development vs. de facto standardization vs. official standardization.
August 1965 -	BEMA/X3.4 sponsored tutorial on PL/I.

January 1966 - X3.4 votes to encourage some industry wide activity on PL/I.

February 1966 - BEMA sponsored open meeting of formation of a industry activity on PL/I, alternatives were discussed and activity under X3.4 recommended.

March 1966 - X3, X3.4 approve PL/I committee.

April 1966 - X3.4.2C formed to discuss and evaluate PL/I and its specification relative to standardization.

June 1966 - X3.4.2C forms subcommittees

X3.4.2C1 Language Development

X3.4.2C2 Formal Definition

X3.4.2C3 Subset Specifications

-- Language Specifications

March 1964 - SHARE Report 1; Expanded FORTRAN, System/360 limits on data.

June 1964 - SHARE Report 2; data structures, report generator, removal of System/360 restrictions.

December 1964 - NPL Technical Report; complete revision, more complete, delete logical data type, BEGIN-END blocks added.

March 1965 - PL/I - SRL-0; complete revision, first machine producible specification.

May 1965 -PL/I -SRL-1; Cleanup of SRL-0, delete sort and report generation. January 1966 -PL/I - SRL-2; pointer data, completely new I/O, asynchronous operations, removal of dynamic fetching and deleting of programs. June 1966 -PL/I Concrete Syntax (TN 3001) and PL/I Abstract Syntax (TN 3002); first reports of new semi-formal definition (circa SRL-2). PL/I SRL-3; first implications of F compiler. July 1966 -October 1966 -PL/I Translator (TN 3003); more of semi-formal definition (circa SRL-3). November 1966 -PL/I Interpreter (TN 3004); more of semi-formal definition (circa SRL-3). December 1966 -PL/I - SRL-4; moving character strings below arithmetic data in the data hierarchy, deletion of binary pictures. January 1967 -Formal Definition of PL/I (TR 25.071 - Ver I). January 1967 -Revised TN 3001.

-- <u>Compiler Development</u>

Spring 1965 - PL/I Subset by Knuth using TMG (IBM 7090)

? 1965 -SLANG produced compiler, not satisfactory for production use (S/360) NICOL 1 (IBM 7090) Fall 1965 -Fall 1965 -Stanford Subset PL/I (FORTRAN and ALGOL features) (IBM 7090) PL/I (F) (S/360) Summer 1966 -Summer 1966 -EPL (BTL) for MULTICS development (GE 635) PL/I (D) (S/360) Summer 1967 -Digitek PL/I (GE 635/645, SIGMA 7). RCA Subset (Spectra 70) Univac Subset (?) ? -

A Comparison Between PL/I and COBOL

Caral Sampson

The following is a comparison between PL/I and COBOL. The material is taken from "A Guide to PL/I for Commercial Programmers", C20-1651-0, published by International Business Machines Corporation. The comparison is divided into the following categories: language notation, program structure, data description, data manipulation, and input/output.

Language Notation

In general, both languages employ similar notations: programmer-defined words use alphabetic characters and decimal digits; expressions consist of sequences of names, constants, and operators; keywords identify language elements; punctuation characters separate elements; statements have an English-like appearance. However, the notation rules of both languages do differ; the following list contains some of the more significant differences.

- 1. In both PL/I and COBOL, keywords have preassigned meanings. In COBOL, keywords are reserved for their intended purpose and cannot be used for other purposes. In PL/I, however, keywords are not reserved for special purposes and may appear wherever a programmer-defined word is permitted; for example, the keyword READ may be used in a PL/I program as the name of a file. In PL/I, different meanings for the same word are determined from context.
- 2. COBOL requires a programming form, PL/I does not. In PL/I, punctuation characters determine the significance and grouping of language elements. This permits PL/I programs to be treated as long strings of characters and to be transmitted to a computer by means of almost any input medium.
- 3. In COBOL, blank characters must surround arithmetic operators; this is not required in PL/I. In PL/I, blank characters are only required between successive words that are not separated by special characters such as parentheses, operators, and punctuation characters.

- 4. COBOL restricts comments to the Procedure Division and requires that they be written in a NOTE statement. PL/I allows comments to appear throughout the entire program and permits them to be used wherever blank characters may appear (except in a character string constant or in a picture specification).
- 5. The COBOL character set consists of 51 characters; PL/I uses 60 characters.

The following points show some of the less significant differences:

- 1. PL/I terminates all statements with a semicolon; COBOL terminates statements with either a period, a comma, or a semicolon.
- 2. Programmer-defined words in COBOL must not be longer that 30 characters; in PL/I the limit is 31 characters.
- 3. PL/I uses the break character (an underscore) within programmer-defined words to improve readability; COBOL uses the hyphen.

Program Structure

Although both programs are constructed in problem-oriented terms and employ the statement as the basic program element for processing data and for altering the sequence of program execution, it is in the area of program construction that PL/I and COBOL differ the most. The following list points out some of the more significant differences in the program structure of the two languages.

- 1. In general, a COBOL program is equivalent to one external procedure in PL/I. The Data Division and the Environment Division of COBOL correspond for the most part to the DECLARE statement in PL/I. The COBOL Procedure Division is equivalent to the executable statements in a PL/I procedure. The function served by the COBOL Identification Division is provided in PL/I by comments.
- 2. The ENTER statement in COBOL is equivalent to the CALL statement in PL/I when the CALL statement is used to activate separately compiled external procedures. However, the concept of nested procedure blocks (internal procedure blocks) in PL/I has no counterpart in COBOL. Consequently, it is not possible within the same COBOL source program to define internal subprograms to which arguments may be assigned.

- 3. COBOL does not provide the equivalent PL/I facilities for automatic and controlled allocation of storage.
- 4. The AT END, INVALID KEY, and SIZE ERROR options in COBOL are provided in PL/I by the ON statement. However, PL/I provides a fuller range of interruption conditions than does COBOL.
- 5. The effect of the ALTER statement in COBOL is achieved in PL/I by assigning a label constant to the label name in a GO TO statement.
- 6. The PERFORM statement in COBOL and the DO statement in PL/I may be used for similar purposes. The PERFORM statement, however, is used for out-of-line loop control whereas the DO statement is used for in-line loop control.

Data Description

In general, both languages use similar methods for describing the characteristics of data items stored within a computer: programmer-defined words are used to name data items; keywords specify the characteristics of named data items; data items may be collected into aggregates; constants may be specified for each data type; data names may be assigned initial values; and a picture specification may serve as an alternative method for describing data. There are differences, however, between the data description features of both languages; the following points contain some of the more significant differences.

- 1. COBOL describes data in the Data Division; PL/I uses the DECLARE statement. The COBOL Data Division contains separate sections for different kinds of data; PL/I does not require a separation of the various data types in a DECLARE statement.
- 2. COBOL requires all programmer-supplied words to be defined in the Data Division. PL/I allows programmer-supplied words to be used in a program without being defined in a DECLARE statement; the meaning of such words is determined from context and a complete set of default rules is used to determine unspecified data characteristics.
- 3. In COBOL, the description of data on external storage media is specified in the Data Division. In PL/I, the input/output statements specify the description of externally stored data.

- 4. Bit strings and label data are PL/I data types not contained in COBOL.
 - 5. COBOL uses figurative constants; PL/I does not.
- 6. PL/I imposes no limit on the number of dimensions in an array or on the number of levels in a structure. COBOL limits a table (the COBOL equivalent of a PL/I array) to a maximum of three dimensions and a group (the COBOL equivalent to a PL/I structure) to a maximum of 49 levels.
- 7. The order of name qualifiers in COBOL is the reverse of the order used in PL/I. COBOL uses the keywords IN and OF as qualification operators; PL/I uses the period.

Data Manipulation

Both languages use expressions to specify data calculations and the picture specification to edit data. However, PL/I uses a single statement for all types of data manipulation, whereas COBOL uses several different statements. The following list contains some of the more significant differences in the data manipulation features of both languages.

- 1. The effects of the COBOL statements MOVE, COMPUTE, ADD, SUBTRACT, MULTIPLY, and DIVIDE are achieved in PL/I with the assignment statement. However, when the MOVE statement in COBOL is used with groups (the equivalent of PL/I structures), data is moved without regard to the level structure of the groups, and data conversion, if specified, is ignored. When the assignment statement in PL/I is applied to structures, the assignment is performed elementary item by elementary item and all data conversion is done as specified.
- 2. The BY NAME option in PL/I is similar to the CORRESPONDING option in COBOL.
- 3. PL/I permits expressions to use data items that contain edit characters. COBOL does not provide this feature.
- 4. The bit string operators in PL/I are similar to the logical operators in COBOL. However, COBOL has no data type that corresponds to the bit string data type of PL/I.

5. The concatenation operator of PL/I is not available in COBOL.

Input/Output

Both languages employ similar methods for transmitting data between internal and external storage areas to the extent that input/output statements process files and identification records (label records) in files may be processed both on input and on output. The languages differ, however, in input/output capabilities; the following points cover some of the more significant differences.

1. In COBOL, data transmission implies files that are composed of logical records. One or more data items form a logical record; data is transmitted one logical record at a time.

PL/I provides two types of data transmission. Record-oriented transmission, like COBOL, deals with logical records. Stream-oriented transmission handles individual data items; a file is thought of as one continuous stream of data items rather than as a collection of logical records.

- 2. PL/I provides control format specifications that regulate printing and spacing operations.
- 3. In PL/I, identification records (label records) are read or written as a result of the IDENT option in an OPEN or CLOSE statement. In COBOL, label records are specified by a file description entry in the Data Division, and special label procedures are specified in a USE statement.
- 4. COBOL permits a file name to be used as a name qualifier; PL/I does not.
- 5. In PL/I, the characteristics of a file are specified in a DECLARE statement or an OPEN statement. In COBOL, file characteristics are specified in a file description entry in the Data Division.

The Operating System as a Host for PL/I Mathew Myszewski

The PL/I language consists of an impressive array of facilities. While it is conceivable that a single PL/I processor could be built to run on a given barefoot machine, a more likely possibility is that a given PL/I processor will delegate some of its functions to the operating system which forms its environment. In general, the division of labor will vary according to the power of the operating system and other resources available. However, there is no doubt that powerful facilities for I/O, resource allocation, linkage, and control must be provided, if not in the PL/I processor environment proper, then in the processor itself. The next few sections will outline these facilities and how they shape both the environment and processor.

The PL/I facilities for input and output require an assortment of features, some which tax all but the most comprehensive of operating systems. Files, of course, are symbolically referenced, but surprisingly, are defined statically. Thus, one could not read the name of a file at execution time and then access it. However, one is allowed to change other file attributes dynamically.

Random-access is required by the PL/I UPDATE file function attribute, the KEYED file attribute, and the DIRECT file access attribute, as well as by the DELETE and REWRITE statements. The BACKWARDS attribute requires access in the reverse order. A primitive form of non-selective file protection is provided by the UNLOCK statement and the EXCLUSIVE attribute; however, this in itself requires some degree of sophistication either in the environment or in the interface to the environment.

Resource allocation is implied directly by the CONTROLLED and AUTOMATIC attributes. Many other features indirectly require either dynamic or static allocation for data and more especially for bookkeeping information. A list of such features, which is by no means complete, would include the RECURSIVE attribute, the VARYING string attribute, the BUFFERED file attribute, DATA-directed input/output, multiple ENTRY points, the TASK options, and dope vectors which keep track of array bounds. Some of these topics will be treated in more depth under the topic "Some Major Efficiency Questions in PL/I".

Linkage of the usual sort, i.e. procedure-to-procedure and procedure-to-data is required in PL/I. This linkage, for the most part, may be performed prior to execution, as the number, names, and size of external procedures and external data are all known at that time, and hence these objects may be located statically.

Control of flow in PL/I has some rather sophisticated aspects. Parallel paths may be accomplished through use of the TASK option, TASK attribute, the EVENT option, and the PRIORITY option. However, only strict hierarchical control is allowed, i.e. orphan tasks are not permitted. Interrupt actions may be controlled through the use of the ON and REVERT statements, condition prefixes, and the SIGNAL statement. Again, control is tightly linked to the static program structure and the automatic stacking of a given condition may be done at most once per block.

The requirement for environmental support is but one aspect of the question of hosts of PL/I. Another equally important question is whether PL/I precludes the use of available operating system capabilities. We have already mentioned above operating system facilities such as orphan tasks which may exist in operating systems but not in PL/I. However, we are less

concerned that PL/I does not contain all possible variations of all conceivable features. Rather, we are concerned with the "features" which in effect prevent efficient matching of resources to programs.

In particular, either the LABEL variable or the POINTER variable, especially when combined with the CELL concept, virtually prevent dynamic reallocation of data or procedures unless some paging mechanism is used. This is due to the fact that any movement of a piece of data implies the update of all pointers referencing it. However, the CELL attribute allows arbitrary overlay of pointers with other data. So even were it known which pointers referenced a data element or structure, one could not update these without knowing whether or not it were a CELL currently containing a particular pointer value. This, at least to my knowledge, would entail a prohibitive amount of bookkeeping. A similar argument applies to reallocation of procedures and the LABEL variable.

Other questions of efficiency will be covered under the topic "Some Major Efficiency Questions in PL/I". Questions of user protection will be covered under the topic "Freedom of Expression is not License for Anarchy".

The set of transactions between a PL/I processor and an operating system is a large set indeed. How can one hope for an efficient interface if PL/I and the operating system have been developed independently? Surely obtaining a well-matched system would be fortuitous under such conditions. The same argument must be applied to the design of language without regard to its efficient compilation. In short, one should not paste together components designed independently of one another and expect a good programming system to result, no matter how good the individual designs might be.

Some Major Efficiency Questions in PL/I

Mathew Myszewski

PL/I "... is a language designed for efficiency, a language that enables the programmer to use virtually all the power of his computer."*

The two main questions we are concerned with are the following: To what extent do greater generality and more features make code optimization necessary? Impossible? To what extent do greater generality and more features require more storage space?

Before answering the first question, it would be well to explain what is meant by it. If a language has mostly special cases, a compiler may fairly easily produce good code for the one or two possible forms (e.g. the FORTRAN DO statement). As one adds generality, it is not a straightforward matter to determine the special (but frequently used) cases and produce efficient code for them. For example, the PL/I DO statement allows multiple index specifications. Were one to compile all cases identically, the simple case would result in at least two extra transfer instructions per loop. In the case of pointers, side effects may be produced which make it impossible to reorder computations for greatest efficiency of execution. The result of such generality then, is not necessarily more efficient compilers, but quite possibly compilers which either do not optimize, cannot optimize, or do so at some considerable increase in compiler size, complexity, and running time. Bear in mind that we are only considering optimization which we have come to expect in other more simple languages.

Before leaving the subject of loops completely, it should be noted that PL/I allows the programmer to modify the loop index within the loop. In order to avoid excessive register loading for the common case, one must keep track of any possible changes to the index from functions with sideeffects, etc.

^{*} IBM Systems Reference Library, IBM System/360 Operating System, PL/I Language Specifications, File No. S360-29, Form C28-6571-4, p.9.

Although the cost of calling a function or subroutine should be small in relation to the size of the subroutine, several factors combine to make prologue and epilogue coding large. A brief list would include: reentrant save-restore code, initial value loading, passing of parameters whose extents are to be computed, AUTOMATIC data, ON-conditions (all the standard ON-conditions are probably pushed whether or not they are used in the procedure), multiple entry points, and multiple returns. To give an example of how any one of these considerations can increase running time, we will look at multiple entry points. Since each entry point may have different parameters in different order, or have different function attributes, it is necessary to sort the parameters into some canonical order and record the data type to be returned. At the return statement it is determined dynamically which data type is to be returned and conversion is performed to this type. These determinations cannot be made at compile time and thus result in code.

The second question, that of storage, is as the first question, largely dependent on the particular implementation; in most cases, however, added storage will be needed to implement the features described here. For example, code produced without optimization will take more space, as discussed above. Again, we will list those features most apt to use extra space. Any use of the asterisk notation or expressions for bounds or length will necessitate storage for a dope vector for each instance of storage for the data. Prologues and epilogues can be as space-consuming as they are slow. Multiple entry points require at least n x m conversion routines where n is the number of different function data attributes and m is the number of different data types returned in a given procedure. Any DATA-directed GET statement without a data list will create a table of all identifiers known at that point, plus all their attributes and extents. Lastly, a calling procedure

must contain a dope vector for at least each argument passed to a routine which uses the asterisk notation or expressions as bounds of length, and possible for all arguments.

In short, PL/I will not come cheaply. Either compilers will have a great deal of special case machinery to distinguish the general case from the more common case or they will compile inefficient, space-consuming code.

Implications of PL/I for the Professional Programmer

Caral Sampson

Usually programmers are categorized by the application area in which they work. Thus one can speak of scientific, commercial, and systems programmers. Another method of classifying programmers is by their attitude towards the computer. Thus one can speak of non-professional and professional programmers. An arbitrary definition of this latter method of classification follows.

The non-professional programmer considers the computer only as a means to an end. He is not interested in how the computer works or what its limitations are, except as these characteristics affect the perceived utility of the computer. His interest is solving his problem, and he views the computer much the same as a slide rule, desk calculator, tabulating equipment, or any other tool at his disposal. Moreover, the tremendous speed of the computer, coupled with its size, tends to impress the non-professional programmer to such a degree that he considers the computer infallible, forgetting it is a machine which can do only what it can be told, and is told, to do.

On the other hand, the professional programmer is interested in the computer itself. He knows how it works and how to use it efficiently and effectively. He knows its limitations and sees to it that his problem does not exceed the limitations of the computer. The professional programmer knows that the computer is not infallible; it cannot be programmed to do many things which to the non-professional programmer seem possible, even if difficult.

The non-professional programmers are usually subject-matter (scientific or commercial) oriented. The professional programmers are usually either students in a computer science curriculum, software developers, or software maintainers. The major distinction between non-professional and professional programmers is their attitude towards the computer rather than their application area. This paper considers the implications of PL/I for the professional programmer.

One implication of PL/I for the professional programmer is that he has fewer languages, perhaps only one, to learn in detail. A design objective of PL/I is to produce a language that encompasses scientific, commercial, real-time, and systems applications, thereby providing the programmer with a language capable of handling a broad range of applications. Features and concepts in PL/I were taken from many existing languages. The data structure is from COBOL and JOVIAL. The procedure concept is from ALGOL. The pointer facilities are from the list processing languages. Although with PL/I, as it is being implemented, the programmer may have only one major language to master, this language is so complex that the programmer still will have to be knowledgeable in many programming techniques and concepts.

Few programmers are intimately familiar with more than one type of high order language, and fewer with more than two. Types of high order languages include algebraic languages (such as FORTRAN and ALGOL), business languages (such as COBOL), list-processing languages (such as LISP and IPLV), formula manipulation languages (such as FORMAC), and simulation languages (such as SIMSCRIPT). A professional programmer who maintains the systems at a user installation is usually not proficient in more than two languages. He is generally well acquainted with the installation's operating system, assembly language, and prime algebraic language, usually FORTRAN. If the installation uses other languages, such as COBOL, the programmer knows enough about that language to build the proper system tape. It is unlikely that the programmer also knows a list processing language, a formula manipulation language, or even another algebraic language if the major algebraic language is FORTRAN. This type of professional programmer will find PL/I a rich language. He will discover a wealth of applications for features such as string data and data structures. The interrupt facilities, error correction functions, debugging features, and multi-tasking facilities will help him in writing the installation's accounting and other standard routines. Because PL/I is capable of handling a large range of applications, the professional programmer will find his knowledge of types of high order languages extended and it is likely he will find that PL/I suffices for most of his applications.

Another design goal of PL/I is to provide a language that enables the programmer to use virtually all the power of his computer. To help accomplish this PL/I provides facilities for techniques such as shared data processing, asychronous program execution, and real-time processing. However, these techniques are only useful when the operating environment has the proper facilities for data management, task management, and job management. Therefore the programmer will have to know in detail not only PL/I, but the operating environment as well.

To use the multi-tasking facility effectively the programmer must be aware of the relationship between tasks in his PL/I program and load modules in the operating system. The purpose of tasking is to allow more than one set of instructions to be operating asynchronously. If the hardware cannot do this, why use the feature in a PL/I program and pay for the overhead (in time and code) required for multi-tasking? PL/I thus puts a greater demand on the programmer's awareness of the operating environment.

With PL/I, the professional programmer will have to be fully aware of the characteristics of his particular computer. PL/I data attributes are an important reflection of the host computer. In PL/I for the IBM 360, the FIXED DECIMAL data type reflects the packed data type and related character by character arithmetic operations of the computer. FIXED BINARY reflects the half word/full word data representation and related binary arithmetic operations. FLOAT data type (both DECIMAL and BINARY) reflects the machine floating point data and related arithmetic operations. To use efficiently the many data types (over 14 combinations of attributes) that can be specified in PL/I, the programmer must be aware of the machine representation and the type of arithmetic operations used with each.

Because PL/I reflects the operating environment and allows the programmer to interact with the environment, he will be able to write many programs in PL/I that previously were written in machine code. PL/I provides interrupts for conversion errors on data and provides a series of functions and pseudo-variables to enable the programmer to determine the offending character, change the character, and then continue processing

using the new character. Previously on data errors the programmer resorted to assembly code. The interrupt conditions and the debugging facilities of PL/I will enable the programmer to do most of his debugging at the symbolic level.

PL/I covers the largest areas of application of any language yet defined. The implementation problems are many. Most of these problems have been solved in some manner in other compilers prior to being addressed in PL/I. But these problems in combination have never been encountered or solved in one compiler. The IBM 360 PL/I F level compiler is only the first of a long series of implementations. Before work can begin on better implementations the full language must be implemented. Thus the present compiler (and those in the near future) will generate poor quality code for many PL/I statements. The professional programmer must learn which statements are costly to use in terms of storage space and execution time, and develop guidelines for using the compiler efficiently. An example of inefficient versus efficient use of PL/I is a card to tape program described in the paper "The Quasi-Programmer and PL/I".

For the professional programmer PL/I also implies trips to conferences, published papers, job security, and job opportunities. PL/I is a new area of specialization. It is at the stage FORTRAN was in 1957, with everyone asking "How do I use it?", "Is it good?", "How do I use it efficiently?", "What does it cost me to use it?" The professional programmer has an opportunity to become an expert in PL/I and provide answers to some of the many questions being asked about PL/I. There is a lack of general PL/I literature as well as a lack of teaching aids and self study documents. The computer conferences and trade publications will be needing more and more papers on PL/I. The PL/I expert will find himself in great demand, probably more so than the FORTRAN expert of 1957 if only because PL/I is more complex than FORTRAN, the investment in it is greater, and the cost of using it is greater.

In summary, the implications of PL/I for the professional programmer are:

1. He will need to know fewer languages than previously, perhaps only one.

- 2. He will gain a broader knowledge of language and programming concepts.
- 3. He will need a detailed understanding of his particular operating environment, both operating systems and hardware.
- 4. He can program more things in a higher level language than previously.
- 5. He has a golden opportunity to improve his professional reputation.

The Quasi-Programmer and PL/I

Caral Sampson

The paper "Implications of PL/I for the Professional Programmer" classifies programmers as non-professional or professional. The nonprofessional programmer is neither a beginner, a novice, nor a quasiprogrammer. He is well trained in a discipline, recognizes that he is not a computer expert, and freely seeks the services of the professional programmer. There is also a class of programmer one may call a quasiprogrammer. This type thinks he is a professional programmer or computer expert, and is therefore dangerous in any computing facility. Some examples of quasi-programmers are: a programmer working in commercial data processing who has never heard of floating point arithmetic; the programmer who isn't interested in efficient compiler or computer usage; the programmer who refuses to switch from FORTRAN II and the FORTRAN monitor system to FORTRAN IV and IBSYS because IBSYS is too complicated to use; the FORTRAN programmer writing large programs who doesn't use subprograms. All of these are quasi-programmers. Time alone does not make a quasi-programmer into a professional programmer. His attitude towards his job and the computer are the determining factors. The quasiprogrammer's limited knowledge of computers, computer languages, compilers, and operating systems coupled with the multitude of PL/I statements, the multitude of data attributes, and PL/I's interaction with its operating environment, makes learning PL/I (even a subset) a difficult task. The quasi-programmer will have an even more difficult time using PL/I for meaningful work. During a recent series of lectures on PL/I a member of the audience commented that each page of PL/I documentation should be stamped "PROGRAMMER BEWARE". And such is the case, because PL/I is a permissive language allowing many statements and mixtures of data.

The size and complexity of PL/I make learning the language a formidable task. The learning problem is compounded by the lack of PL/I literature. The present literature consists of one book and several documents published by IBM. IBM provides a few introductory student texts, program logic manuals, and an often revised reference manual. The introductory

texts provide a broad brush treatment of PL/I for specific application areas and lack the detail necessary for writing a program. The logic manuals, containing flow charts and descriptions of the PL/I compiler, are written for implementors. The reference manual, sometimes called the SRL manual, is the only available detailed description of PL/I. This manual describes PL/I using a syntax notation unfamiliar to most programmers, and this notation stops many from using the manual. Furthermore, the terminology used to describe PL/I is new or inconsistent with current usage of the same terms. However, most terminology is precisely defined, if one knows how to find the definition. As one student said about the manual, "It was written by a Philadelphia lawyer and must be read as such." If the quasi-programmer gets past the problem of reading the reference manual, he still has problems because the manual does not specify when to use a given feature and there is no literature available on this subject.

The major problems one encounters when learning and using PL/I are a result of the size and complexity of PL/I. For example, PL/I provides:

- a. over eight data types not including over fourteen combinations of arithmetic data attributes;
- b. seventeen operators;
- c. thirty-three statements, most with at least one option; and,
- d. sixteen major classifications of attributes.

The professional programmer has the background necessary to understand the relationships between the different facilities of PL/I. The quasiprogrammer must be taught the relationships between the different data types, operators, statements, and attributes. He cannot be taught this in forty hours of class lectures. Yet one week is the duration of most PL/I training classes. Many pseudo-programmers will attend this type of class and, upon finishing the class, will have the false security that they have been told enough about PL/I to use it effectively. They haven't! The course, as outlined in the IBM PL/I coding Education Guide (R20-1018) only skims the surface of PL/I and presents none of the anomalies of the language.

The following is an example of a typical type of error that arises because of the many combinations of arithmetic data types available with PL/I. The programmer may describe data as having a scale of FIXED or FLOAT and a base of DECIMAL or BINARY. There are thus four possible combinations of scale and base: FIXED DECIMAL, FIXED BINARY, FLOAT DECIMAL, and FLOAT BINARY. In the 360 implementation, a data item described as FLOAT DECIMAL and FLOAT BINARY have the same machine representation; they are represented in hexadecimal floating point format. FIXED DECIMAL and FIXED BINARY data items have different machine representations; they are represented as packed decimal and binary data, respectively. Furthermore, the FIXED scale represents decimal or binary numbers that may have a fractional portion, i.e., a scale factor or precision. This poses the problem of associating precision with a FIXED constant. For purposes of expression evaluation, an apparent precision is defined for real fixed-point constants to be (w,d) where w is the total number of digits in the constant and \underline{d} is the number of digits specified to the right of the decimal point.

Most programmers in the commercial application area are familiar with fixed decimal data. However, most programmers do not understand rational arithimetic. Because of PL/I rules governing evaluation of expressions, the expression (10 + (3/2)) yields a result of 1.5000000000000.* The same result occurs if the constants are replaced by three data items having precisions (2,0), (1,0), and (1,0). Quasi-programmers cannot understand why the result is not 11.5, but the main problem is that they probably may never realize that truncatation can occur from the left.

The quasi-programmer who has been using FORTRAN also has problems. In FORTRAN he uses integer and floating point data and tends to equate the FIXED DECIMAL or FIXED BINARY of PL/I with FORTRAN's integer data. He will have the same problem as the commercial programmer with the expression (10 + (3/2)). Another problem may be that algorithms no longer work when programmed in PL/I. For instance, the algorithm "compute (I-((I/2)*2)) and then check for zero" used for distinguishing between odd and even integers will not work in PL/I because there is no mod (\$ 1) integer arithmetic.

^{*} See footnote page 32 (RWM)

The many operators of PL/I coupled with the many data types and the ability of the compiler to interpret most expressions creates another set of problems for the quasi-programmer. In PL/I there are seventeen different operators divided into four types: arithmetic operators, comparison operators, bit-string operators, and string operators. There are accordingly two types of data which can be used with the operators: arithmetic data and string data. The operands of any operator need not be the same type as the operator nor the same type as each other. Thus it is correct to concatenate (a string operation) a float data item with a character string data item. It is also correct to use the arithmetic addition operator with two character string data items. Expressions with mixed data items, especially with arithmetic and string data, are usually written by mistake; however the compiler provides an interpretation for them. If the programmer is lucky, there may be some indication at execute time that conversion between arithmetic and string data is taking place. The quasi-programmer in his use of PL/I will probably write many statements that are incorrect, are time and space consuming, yet never know about it.

Another area where the quasi-programmer may have trouble using PL/I is the interrupt operations. Although the interrupt operations provide the programmer with a flexible debugging aid, their concept is usually outside the knowledge of the quasi-programmer. The interrupt operation causes the suspension of normal program activities in order to perform a special action; after the special action, program activities may or may not resume at the point where they were suspended. Some of the interrupts are generated by hardware detected errors, some are the result of error detection by system routines, and some interrupts are generated, under programmer control, by the execution of the PL/I SIGNAL statement. Those conditions which are not named and controlled by the programmer have standard actions associated with them which the programmer may override. There are two program checkout conditions which are enabled by the programmer. The first condition, SUBSCRIPTRANGE, occurs when a subscript is evaluated and found to lie outside its specified bounds. The second condition, CHECK (identifier-list) provides a snap/trace of the specified

identifiers. If the quasi-programmer masters the interrupt concept, he will have a powerful debugging aid. However, the debugging facilities of PL/I are a two-edged sword. Although they make debugging much easier than machine level debugging, they will consume vast amounts of time and computer storage if used in a production program. Thus the quasi-programmer will have to learn to use them discriminately and effectively.

Because of the size and complexity of PL/I, there are many ways to program a given problem. One way may be better than another way. The quasi-programmer probably will never realize that he is using a given feature in a manner it was never intended. Also, because he never cares about generated code, he will never realize that the way he defines his problem data drastically affects the running time of his program. An example of data definition seriously affecting execution time follows.

A simple card to tape routine was programmed two ways. In one program the data was declared as 80 one-character items (DCL CARD(80) CHAR(1)), and in the other as one 80 character item (DCL CARD CHAR(80)). Other than the declare statement, both programs were identical. The first program (80 one-character items) took approximately 150 minutes; the second program (one 80 character item) took approximately 7 minutes. A factor of 25!

Unfortunately, there are many quasi-programmers. Hopefully PL/I will force them to understand more about computers, computer languages, compilers, and operating systems. If not, there will be many installations ordering more computers or bigger computers.

Freedom of Expression is not License for Anarchy*

Mathew Myszewski

Although PL/I has many forms for expressing the same problem solution, and it would be hoped that the most natural means of expression were also the most efficient, error free, and free from anomalies; usually this is not the case. It is unfortunate that a language which has aspired to so much is so filled with traps for the unsuspecting. This in spite of the claim that "PL/I is organized so that any programmer, no matter how extensive his experience, can use it easily at his own level."**

There are three kinds of traps that come to mind. First, and least detrimental, is to use a natural form of expression and find that it is inefficient. This has largely been discussed in the topic "Some Major Efficiency Questions in PL/I." Next, the user of PL/I may use a natural form of expression and find that he gets incorrect results quite possibly without any warning. Lastly, simple errors in expression may cause completely anomalous behavior similar to the store in a wild location that one frequently encounters in assembly language programming. Each of these traps is present in PL/I.

Since efficiency questions have been treated elsewhere, we will only pause to note one example of this type of pitfall. Suppose a PL/I programmer has used the debugging forms of the ON statement and condition prefixes. When he is done debugging he removes all the ON statements to eliminate the debugging output. However, inadvertently or through innocence

^{*} Juster Norton, "The Dot and the Line", Random House, New York, 1963.

^{**} IBM Systems Reference Library, IBM System/360 Operations System, PL/I Language Specifications, File No. S360-29, Form C28-6571-4, p.9.

he leaves in one or more CHECK type condition prefixes. These will not cause incorrect results or spurious printout but <u>will</u> use inordinate amounts of machine time. This is the perfidy of inefficiencies; correct results are produced but unnecessary machine time is used, often without the knowledge of the unsuspecting user. And if he does suspect, it is often extremely difficult to track down the cause of such inefficiencies.

A few examples of incorrect results obtained without warning will suffice to give an idea of what is in store for the wary or unwary PL/I user. Other cases will be left as an exercise for the production PL/I programmer. Consider the statement:

$$A = 10 + 3/2$$
:

Who would suspect that the PL/I manual defines the result of this computation to be undefined? The reason is that the definition of fixed point arithmetic is such that the subtraction will cause an overflow which gives an undefined result*. (Note that the literals could have easily been variables of the same data type.) Other cases of this form of trap may be found in operator precedence and other rules for expression evaluation.

Lastly, we will consider some examples of anomalous behavior. With the exception of assembly language programming, pointers were never so accessible to run-of-the-mill programmer before. Since the pointer may be over-laid with, for example, floating point data using the CELL, the possibility exists for execution of random fetches and stores in data, program, and other areas of storage. Fortunately, the programmer in PL/I has the opportunity to do SUBSCRIPTRANGE testing and catch a principal cause of these anomalies in FORTRAN.

^{*} The difference in the statement here and the one on page 28 are a result of the PL/I definition having been changed between the times when the respective authors studied the issue. (RWM)

These few items can and will be extended considerably as more experience is gained with PL/I. I suspect that new categories of traps will arise, such as conflict of user identifiers with PL/I keywords and built-in function names, but those given above will continue to ensnare even the well informed.

PL/I Problems for the Implementor

R.W. Mitchell

There are two general problems facing the implementor of PL/I. The first is "What is PL/I?" and the second is a collection of technical "How should such issues be done?"

Determining "What PL/I is" is a difficult issue because of the size of PL/I and because of the form of the definition of PL/I (which will be considered in another section). One of the most evident characteristics of any of the definitions of PL/I is its size. The total number of pages is always in the hundreds. The intricate interdependies of the language are very difficult to identify and understand under these conditions.

This has resulted in most of the implementations of PL/I having been designed with less than complete understanding of the language and later being revised as a number of issues became apparent.

The unusual technical problems facing the implementor center around the following

- -- attribute structure,
- -- procedure linkage,
- -- ON-conditions,
- -- asynchronous processing, and
- -- optimization.

The attribute structure is large and the rules for completing the set of attributes for a partially declared identifier are fairly complex in some cases. Because of this large attribute structure the processing of all data references is also quite involved.

The basic problem in procedure linkage is the <u>potential variability</u> of entry points (and, hence, argument list), attributes of arguments, and return points. This results in very inefficient code for the simple cases or very complex logic for the implementation.

The ON-conditions and asynchronous processing present the problems of flow of control paths which do not follow a convenient hierarchical pattern.

The question of optimization is dependent on recognizing the special cases which permit deletion of unnecessary actions. Once again the size of the language (hence, number of cases) and the variability of the specifics of the attributes create the problems.

The design goals of PL/I include modularity partially to ease the implementor's task of providing compilers of reasonable size and efficiency. This has led some to suggest very modularized compilers as a possibility. It is my opinion that we will be a couple years just learning how to implement PL/I with all of its features and it won't be until after that time that the potential for modularized PL/I compilers can be properly investigated.