The Early History and Characteristics of PL/I

George Radin
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, N.Y. 10598

Introduction

Source material for a written history of PL/I has been preserved and is available in dozens of cartons, each packed with memos, evaluations, language control logs, etc. A remembered history of PL/I is retrievable by listening to as many people, each of whom was deeply involved in one aspect of its progress. This paper is an attempt to gather together and evaluate what I and some associates could read and recall in a few months. There is enough material left for several dissertations.

The exercise is important, I think, not only because of the importance of PL/I, but because of the breadth of its subject matter. Since PL/I took as its scope of applicability virtually all of programming, the dialogues about its various parts encompass a minor history of computer science in the middle sixties. There are debates among numerical analysts about arithmetic, among language experts about syntax, name scope, block structure, etc., among systems programmers about multi-tasking, exception handling, I/O, and more.

I must acknowledge my debt particularly to Jim Cox for help in the search through the archives, but also to Marc Auslander, David Beech, Pat Goldberg, Marty Hopkins, Brian Marks, Doug McIlroy, John McKeehan, Erich Neuhold, John Nicholls, Bruce Rosenblatt, for help and criticism of early drafts, and to Warner King, Elliot Nohr, and Nat Rochester for making the material available. Finally, because Mike Marcotty and Bob Rosin (the PL/I language coordinators for this conference) were unhappy with the early drafts and provided specific suggestions, the following paper is much improved.

Background

In the early 1960's computer applications, programmers and equipment were much more clearly divided among scientific, commercial, and special-purpose lines than they are today. There were computers designed primarily for scientific applications, such as the IBM 7090 and the IBM 1620, scientific operating systems for these computers, a scientific programming language (FORTRAN), and a user's group for installations whose primary applications were scientific (SHARE). Similarly there were computers oriented toward commercial applications, such as the IBM 7080 and the IBM 1401, system facilities for these, a commercial language (COBOL), and a users' group for installations involved primarily in commercial applications (GUIDE). Finally, there was a bagful of special purpose computers such as the IBM 7750 and Harvest and special languages such as JOVIAL.

Moreover, there were accepted characteristics which distinguished these categories. Scientific users needed floating point arithmetic, arrays, subroutines, fast computation, fast compilation and batch job scheduling. Commercial users needed decimal arithmetic, string-handling instructions, fast and asynchronous I/O, compilation which produced efficient object code, and excellent sort programs. Special purpose users needed capabilities like variable length bit string arithmetic, macro libraries (such as COMPOOL in JOVIAL), pattern matching operators, list handlers, and very fast response for real time.

The trouble was that this elegant separation was no longer a realistic characterization of installations and projections were that matters were going to get worse. Applications and users refused to stay properly compartmentalized.

The stereotype of a scientific programmer who is willing to wait two hours for a blinking computer finally to display the answer on its console lights was giving way to the professional who wanted a camera-ready report in a beautifully readable format. The files that scientists were processing were as voluminous as those in many commercial installations. They too needed sorts and I/O managers.

Commercial users were doing linear regressions and factor analyses on market data. They needed floating point and arrays.

Installation managers objected to buying two sets of hardware, maintaining two operating systems and educating two sets of application programmers who could not talk to each other, or share programs.

Becoming increasingly aware of this situation, IBM reached the conclusion that it could best serve its customers' (and hence its own) interests by developing a single product line and a single family of operating systems which satisfied the requirements of all three user types. Thus S/360 and OS/360 development was begun.

It was for this same reason that IBM and SHARE formed the Advanced Language Development Committee of the SHARE FORTRAN project in October of 1963. This group was charged with specifying a programming language which would meet the needs of all three of these user types as well as those of systems programmers.

I can find neither documentation nor personal recollection to illuminate more clearly the specific motivations behind the formation of this committee or the selection of its members. A meeting was held in October, 1963, at the Motel-on-the-Mountain in Suffern, N. Y., attended by IBM and SHARE management. At this meeting the SHARE committee members were already named and present. They were:

Hans Berg, from Lockheed, Burbank, who had direct experience in training programmers and in the problems of running a computing installation,

Jim Cox, of Union Carbide, who had experience not only in FORTRAN and Assembly language programming, but also in using IBM's existing business language, Commercial Translator,

and Bruce Rosenblatt, of Standard Oil of California (the chairman of the committee) who, in addition to experience in programming and installation management, best understood SHARE's expectations for the committee.

The IBM committee members were:

C. W. Medlock, a FORTRAN compiler expert who had developed loop optimization algorithms,

Bernice Weitzenhoffer, who had wide experience in both FORTRAN and COBOL programming,

and myself (the IBM project leader) whose background lay primarily in scientific programming and programming language theory (pursued in university and consultant environments).

Tom Martin of Westinghouse Corp. (manager of the Systems Division of SHARE) and Paul Rogoway of Hughes Dynamics, Inc. (chairman of the SHARE FORTRAN project within the Systems Division) were not official members but attended almost all meetings and participated in the decision-making process. Similarly, Larry Brown and Ray Larner, IBM FORTRAN compiler developers, were virtually full time committee participants.

The committee met generally every other week for three or four days (long weekends, in fact), mostly in New York and Los Angeles. The non-IBM members retained the demanding responsibilities of their full-time jobs during this period. Language definition was, for them, mostly a moonlighting activity. IBM management, being more directly a beneficiary of the activity, allowed their employees compensatory time off, but this was often an official gesture rather than a practical occurrence. The meetings were held in hotel suites, in New York, primarily, but also in California. At times the most vocal opponents of the activity were not FORTRAN advocates but neglected wives (and husband).

Procedures at the meetings were largely informal. Since the non-IBM committee members had signed a confidential disclosure agreement with IBM, we attempted, at the first several meetings, to learn about IBM's New Product Line (which became S/360) and, particularly, its operating system (later called OS/360) and language processors. Minutes were not officially taken. When an area was to be defined, one or more small subgroups were formed to develop specific proposals. These were then presented to the committee as a whole, preferably preceded by a written description mailed Special Delivery before the meeting.

The major constraint which confronted the committee throughout the development process was a series of very early, but seemingly rigid, deadlines. In order to be included in the first release of IBM's New Product Line we were first informed that the language definition would have to be complete (and frozen) by December, 1963. In December we were given until the first week in January, 1964 and, finally allowed to slip into late February. Thus, not only was the total time for language definition very short, but even this period was punctuated by "deadlines" which gave a sense of crisis to the activity. (An interesting example of the subjectivity of history can be found in my vivid recollections of these deadlines compared to Bruce Rosenblatt's remark in a recent letter to me (19): *"I was not conscious, as a non-IBM'er of any missed deadlines; though I felt earlier commitments were requested, I thought we (the committee) had successfully avoided these when the subject came up."*)

In spite of this pressure, PL/I did not make the first release of OS/360. It has been suggested that this late introduction was primarily what accounted for COBOL's early success over PL/I. While this can never be proven, it is clear that much of the long and demanding work in PL/I language development would have been made much easier had the designing committee been given six months or a year at the beginning to lay down a carefully defined language base.

Since the background of the group was so strongly FORTRAN-oriented, and the committee reported to the SHARE FORTRAN project, starting the language development with FORTRAN as a base seemed an obvious choice. (In fact, for the first several meetings, the language was referred to as FORTRAN VI.) Bruce Rosenblatt recalls that FORTRAN compatibility was part of the original charge to the committee. As part of its New Product Line software development IBM had supported proposals for such a new language. But even in these proposals, strict upward compatibility with FORTRAN was soon abandoned.

In a detailed FORTRAN VI language definition written by Larry Brown and Ray Larner (20) the introduction stated:

> *"FORTRAN VI is not intended to be compatible with any known FORTRAN IV. It includes the functional capabilities of FORTRAN IV as well as those capabilities normally associated with "commercial" and "algorithmic" languages. In order to embrace these capabilities in a usable and practical language, it has been found virtually impossible, and certainly undesirable, to retain FORTRAN IV as a compatible subset."*

Nat Rochester, in "A FORTRAN VI Proposal," (21) wrote:

> *"Compatibility with FORTRAN IV would preclude having a simple elegant streamlined language because FORTRAN IV itself is heavily burdened with curious restrictions and complexities that have accumulated during the long history of additions that maintained an approximate compatibility with earlier versions."*

Finally, a report of a meeting held in Los Angeles on February 11, 1963, (22), where IBM language experts and programming management discussed the future of FORTRAN, began:

> *"The purpose of the meeting was to decide whether there should be a FORTRAN V or whether we should move directly to FORTRAN VI. F5 was defined as a compatible extension of F4 to provide field and character handling capability, making it suitable for 'information processing' as distinct from purely scientific or business applications. F6 was defined as a new language, incompatible with existing FORTRAN's, with a simplified syntax designed in part for ease of compiling, suitable for scientific 'information processing' and business applications, and open-ended with regard to real-time, control and other applications."*

After several weeks of investigation the SHARE-IBM committee unanimously agreed to abandon the direction of compatible extensions to FORTRAN IV and start the language definition from scratch. In a recent letter to me, Doug McIlroy credits (or blames) me with being responsible for this decision. I certainly was responsible for persuading the technical and management community in IBM to accept it. In too many meetings within IBM I was faced with strong objections from FORTRAN language experts to the introduction of yet another language. In addition to the reasons given by Brown and Larner, the arguments I recall which argued for incompatibility are:

- The syntax of FORTRAN does not lend itself to free-form representation on display terminals. In particular, statement labels were recognized by their column position on a card (or card-image file).

- Declarations in FORTRAN are organized by type rather than by identifier. For instance, all 10 by 10 arrays can be declared together and all floating point variables can be declared together. This generally results in the attributes of a single variable being scattered across several statements. The committee wanted to be able to isolate the description of a variable, perhaps even keeping it in some compile-time library.

- Storing arrays by column (as in FORTRAN) rather than by row is awkward even for scientific users but is quite unnatural to commercial application programmers, who think in terms of tables rather than matrices. It is mostly possible to keep this mapping unknown to the programmer, but there are uses of overlays and I/O which expose it.

- In fact, the FORTRAN programmer's capability to take advantage of data mapping by unconstrained overlaying in COMMON was not a practice the committee wanted to perpetuate.

In addition to these specific arguments, two more basic points were made in these debates. First, it was demonstrated that after ALGOL block structure was added (a recognized requirement which, however, did not get defined in time to be included in the first language report) together with all the extensions required for commercial and systems applications, the result would be a language so different from FORTRAN that the benefits to IBM of staying compatible (e.g. compiler development cost) hardly seemed worthwhile. Second, the application programmers who would benefit from FORTRAN compatibility did not include the large and important group of commercial programmers. They would have to learn a new language in any case, and therefore it was unfair to them to compromise this language in order to benefit scientific programmers.

In retrospect, I believe the decision was correct. Its major drawback was that, by taking FORTRAN as a base, we could have gone, in an orderly way, from a well-defined language to an enhanced well-defined language. We could have spent our time designing the new features instead of redesigning existing features. By starting over, we were not required to live with many technical compromises, but the task was made much more difficult.

My recollections of the language debates are that at every stage of its evolving definition, the language was constrained by compiler and object code performance objectives when running under OS/360 (in its changing forms). Thus, for instance, the definitions of I/O features and multi-task execution could not conflict sharply with those functions as provided by the operating system. Facilities for separate compilation were matched to those supported by the link editor. Thus, while the definers always tried to focus on required functions rather than specifically on OS/360 facilities, so that the language would remain "system-independent", it was constrained to be a system independent language which would execute efficiently on a S/360.

Bruce Rosenblatt, in comments on an early draft of this paper (19), reveals the different perspective which the non-IBM committee members had on this issue: "*Your comments on our attempt to achieve compatibility with OS surprised me. I had no idea what OS was and felt frustrated in trying to find out at the time. Maybe in the later development process this held true, but certainly not in our original deliberations.*"

Once the decision was made to become incompatible with FORTRAN, a name other than FORTRAN VI was needed. A language designed for such a wide application set needed a general name. APL was already taken. MPPL (for Multi-Purpose Programming Language) was considered. NPL (for New Programming Language) seemed a good choice, since it would tie the language, within IBM, to the New Product Line. This name remained until, in 1965, its prior use by the National Physical Laboratory in England required a change to PL/I. Again this new name linked the language, within IBM, with S/360 and OS/360. (PL/360 would have made this link too strong since we were determined to be system-independent.)

The first document describing the language, (SHARE, 1964, March) (1) was presented to SHARE at its March meeting in San Francisco. (Appendix A contains several examples developed for the SHARE meeting showing programs written in this "new" programming language compared to the "old" FORTRAN IV.) The reaction was anything but indifferent. It was widely praised for its breadth, its innovations and its attention to programmers of varying sophistication. It was criticized for its formlessness, complexity and redundancy. One SHARE member later compared it to a 100-blade Swiss knife; another wondered why the kitchen sink had been omitted from the language. At the end of the meeting, however, the language was endorsed by SHARE, and IBM decided to develop several processors for S/360. Shortly after the SHARE meeting a more complete version of this document, entitled, "Specifications for the New Programming Language, April, 1964" was distributed (2).

Since the language was to appeal to commercial programmers, the GUIDE users' group appointed Bob Sheppard of Proctor and Gamble as a member of the committee. At this same time SHARE added Doug McIlroy of Bell Telephone Laboratories to the committee because of his strong background in numerical analysis, and systems programming.

In June 1964 the committee produced a second version, "Report II of the SHARE Advanced Language Development Committee" (3). Some of the major differences between this document and the April specification were:

- The explicit verb SET for assignment statements was removed. The language reverted to the FORTRAN (A=B) syntax, i.e. "A=B" rather than "SET A=B".

- Binary numeric data types were introduced. The April document defined only decimal.

- The April document defined a data type called ADDRESS, which could take as values statement labels, procedure names or files. The June document separated these into three separate data types.

- The DO group was introduced. The April document had specified the FORTRAN-like syntax in which the label of the last statement in the group is explicitly named in the DO statement.

- The semi-colon terminator was introduced.

- The storage classes TEMP and HELD, corresponding to ALGOL's distinctions, were introduced.

- The "RETURN TO label variable" form was added.

The level of acceptance of this language by SHARE and by IBM can be seen in a statement attached to this June document, signed jointly by Thomas W. Martin, Manager SHARE Systems Division, and Carl H. Reynolds, Manager of Programming Systems, Data Systems Division, IBM Corporation:

"The attached report represents the efforts of the joint IBM/SHARE Committee to identify the functions and features needed in an extended language and to propose a particular embodiment thereof. Because of its belief that the committee's results are a significant step forward, it is the intention of the IBM Corporation to use this report as the basis of specification for the New Programming Language."

At about this same time IBM decided that the major part of NPL compiler development would be done in its laboratory in Hursley, England. Although product planning and language control remained officially in the U.S. for at least the initial period, NPL (and then PL/I) was primarily a Hursley responsibility. (Responsibility for the language and its compilers has recently moved to the IBM Laboratory in,Santa Teresa, California.)

In November 1964 the Hursley laboratory produced a document much more detailed than the reports of the Advanced Language Development Committee (4). It ran to 200 pages and represented the definition of the language from which the implementors would develop their compilers. Then in December, 1964, this document was published for external distribution as "NPL Technical Report, IBM World Trade Laboratories (Great Britain) Ltd" (5). Some significant language changes between this document and the June report were:

- A 60 character set replaced the original 48 character set. Special characters (such as | for "OR") were added primarily for NPL operations and delimiters.

- "BEGIN...END" blocks were introduced, supported by the members of the IBM laboratory in Vienna, Austria. They argued, in a paper called "Block Concept for NPL," (9) for the elimination of ALLOCATE and the inclusion of Blocks. Paul Rogoway (who had now joined IBM) and I objected because we wanted the function of dynamic storage allocation at other than block entry. We prevailed and both functions were included.

- %INCLUDE was removed, but many new compile-time facilities were added. (%INCLUDE was brought back into PL/I in the language supported by the first available compiler, called the F-compiler.)

- STATIC, AUTOMATIC, and CONTROLLED storage classes replaced TEMP and HELD.

- "IF...THEN...ELSE" was introduced. The June specification did not have the THEN delimiter.

When compared to the relatively small changes undergone by FORTRAN or ALGOL 60 in their first years, the language described in this document is remarkably unlike the PL/I described in later documents such as GC 33-0009-4 (6) which defines the language supported by the present IBM Checkout and Optimizer compilers for PL/I. Further language evolution has resulted in a 1976 PL/I American National Standard (7), and an ECMA Standard (8). An ISO Standard is near approval at this writing.

The period between April and November 1964 was marked by a staggering amount of language definition, debate, presentation and early compiler implementation. It is impossible in this short paper to describe this activity properly. Let me briefly mention three major parts of the discussion.

Literally as soon as the March report was typed, Kurt Bandat, Viktor Kudielka, Peter Lucas and Kurt Walk of the IBM Vienna Laboratory began an intensive critique of the language. As early as April 3 they produced a 52 page "Review of the Report of the SHARE Advanced Language Development Committee" (10). This review disclosed syntactic errors, ambiguities, problems generally, and made recommendations, many of which were accepted.

Also as soon as the March document was available, the Programming Language Evaluation department was formed in IBM, at the Time/Life building in New York City, under Paul Comba. This was a group of nine highly experienced programmers with extensive background in high-level languages. They selected 22 representative programs from various application areas which had already been written in existing languages (6 FORTRAN, 5 COBOL, 5 ALGOL, 6 JOVIAL). They reprogrammed all 22 in NPL and compared this language to the originals with respect to criteria such as ease of programming, size of programs, readability. On July 20, 1964, they produced a 185 page report describing their conclusions: "A Comparative Evaluation of the New Programming Language" (11). As an aid to their own learning the group developed a formal syntax of NPL, which was also published in this report. The following is a direct quote from the overall conclusions of this report:

> ***"Overall Conclusions***
>
> *The goals set by the developers of NPL have been stated in terms of improvements and extensions over the current FORTRAN. Because of this, and because FORTRAN is the most widely used programming language, our overall evaluation gives the greatest weight to the comparison of NPL with FORTRAN. The following remarks refer to Section 1.1 of the April 15 Specifications.*
>
> ***1. Extensions to include additional applications.*** *NPL has undoubtedly succeeded in this stated goal. The programs exhibited in Part 2 of this report are a partial proof of this. The range of these extensions is indeed extremely wide. (Because of these extensions, it is likely that NPL will not need any provision for the inclusion of assembly language subroutines.) Whether the character string operations of NPL will prove adequate for writing programming systems remains doubtful. We have not evaluated this aspect.*
>
> ***2. Clarity, and removal of arbitrary restrictions.*** *NPL has succeeded outstandingly when compared with FORTRAN and with the other languages except ALGOL (Iverson's programming language, while not considered here, is much more general than NPL; however there are serious problems involved in implementing Iverson's language in its totality). NPL is a little more prolix than FORTRAN. In a few cases it requires constructs that are less convenient than FORTRAN and look unnatural to FORTRAN users, but these cases are relatively minor. Some of them may be removed by redefining NPL on a 60 character set.*

As to the claim that NPL is easier to teach and easier to use, we believe it will prove to be valid PROVIDED:

(a) subsets of NPL are defined;
(b) an authoritative formal definition of the syntax of NPL is given;
(c) teaching aids to explain the more complicated parts of NPL (e.g. the valid combinations of attributes and their default interpretations) are developed.

3. Bringing the language in line with current practice of machine use. *The adequacy of NPL with respect to I/O control, trap supervision, monitored operations and multiprogramming is not evaluated in this report, since the I/O facilities were inadequately defined when this study was undertaken and there are, in general, no facilities in the other languages corresponding to trap supervision, monitored operations and multiprogramming. It is clear nevertheless that the inclusion of some of the facilities in NPL is a most important advance.*

•
•
•

The comparison of NPL with COBOL *for commercial programming shows that NPL has most of the facilities of COBOL and some very useful additional ones. NPL is usually more flexible, simpler, and less verbose. Of the COBOL facilities that are lacking in NPL, many are minor or specialized; only a few are substantial. NPL should not be presented as a language that can fully replace COBOL, but it is likely that a large number of commercial shops will find it superior and eventually will want to convert to it.*

The comparison of NPL with ALGOL *shows that NPL has gone a long way toward incorporating many ALGOL concepts, and has a much richer repertoire of extended arithmetic operations than ALGOL does. The structure of NPL statements and programs is not as simple as that of ALGOL; in part this could be remedied by redefining NPL on 60 characters, in part it is a consequence of the additional facilities offered by NPL. We believe that a large number of ALGOL users whose interests extend beyond the formulation of algorithms for numerical analysis will find that the additional facilities of NPL more than compensate for its comparative lack of elegance. As a publication language, NPL cannot compete with ALGOL unless a publication version of NPL is developed.*

The comparison of NPL and JOVIAL *shows that NPL has all the essential capabilities of JOVIAL with the exception of status variables and arrays of string variables, and less convenient switching capabilities. NPL should prove strongly competitive with regard to JOVIAL.*

In conclusion *NPL is a very strong and powerful language. It can be further strengthened by:*

- a definition of NPL on a 60 character set,
- the definition of subsets and the development of teaching aids,
- a formal definition of the syntax,
- a publication version of NPL."

As soon as the language and compiler responsibility was moved to Hursley, England, intensive work began in that laboratory to solidify the language and build compilers. Ray Larner, Larry Brown, and Tom Peters moved to England to participate in this effort. Hursley management (John Fairclough, the lab director, George Bosworth, manager of programming, John Nash, manager of the F-compiler, and I. M. (Nobby) Clarke, manager of the Library) were determined to stabilize the language at a level at which an efficient compiler could be built on schedule. In fact, from 1965 until today, the success of PL/I is due almost entirely to the Hursley programmers who made it well-defined and usable, who built the compilers for it, and who participated in the standards activities (John Nicholls, David Beech, Brian Marks, Roger Rowe, Tony Burbridge among many others).

Questions were being raised about whether it was possible to build an efficient compiler for such a large and complex language. In October 1964 a committee was formed to investigate the compiler activity, analyze the language, and generally predict the characteristics of NPL compilers. This committee was chaired by Mike Roberts and had participants from the original NPL committee (Jim Cox, who had since joined IBM, and myself), managers of compiler groups (Jack Laffan, John Nash, Bob Sibley) and a member of the original FORTRAN compiler group (Dick Goldberg).

In November 1964 (23) the committee reported. Its conclusions are summarized in the following quote from its report:

> *"The mission of the committee was to determine whether NPL as defined by Hursley V (the November document) would necessarily result in slow compilers or inefficient object code. A brief summary statement is that there is no single feature in the language (as distinct from currently planned implementations) that will cause major and unavoidable inefficiencies in object code. The sum total of all the features has a strong effect on compile speed. Object code inefficiencies, where found, can be mitigated by the suitable restriction of, or interpretation of, features in the language or by considerable effort at compile time. There will be a continuing need for such restriction and interpretation especially until a compiler is implemented.*
>
> *The standard for comparison chosen was ASA FORTRAN IV. Every subsequent reference to FORTRAN means ASA FORTRAN IV. The general consensus is that a compiler for NPL will require about four times as much coding as a FORTRAN compiler. It is also generally agreed that those NPL programs which are essentially FORTRAN programs written in NPL will take twice as long to compile as would the same programs written in FORTRAN and compiled with a FORTRAN compiler programmed to the same design criteria. (The agreed figures for the NPL-COBOL comparison are twice as much for coding and a factor of 1.5 for compile time.) It is the expectation of the committee that the compile time per statement for programs using the full facilities of NPL will be at most 20 percent greater than the compile time per statement for those NPL programs which are essentially FORTRAN programs written in NPL."*

A good summary of the reaction of the ALGOL community to NPL is found in a report from Paul Rogoway to Nat Rochester, his manager, describing a meeting in Hursley on July 24, 1964, where NPL was described to "European University Computing Leaders." I quote:

> *"The NPL presentation was not particularly well organized but most of the significant features of the language were presented. In general the audience was*

overwhelmed by the language--some members by its power, most by its apparent largeness. The experience reemphasized the crying need for a tutorial presentation of a subset of the language geared to the computing interests and needs of the scientific or FORTRAN - ALGOL audience. There was nearly unanimous feeling that the language would appear to be a fine one, but more definite judgment was reserved, pending further study of the June 15 document (copies of which were distributed to the attendees Tuesday afternoon).

Most of the attendees are ALGOL users and generally agreed that the language is a superset of ALGOL. Serious concern was raised over any instance (such as the WHILE clause) where the same language has a different meaning in NPL from ALGOL. Also, many persons suggested that the point of departure for the new language development should have been ALGOL rather than FORTRAN.

The most significant observations came from three individuals representing interestingly enough three distinct interest groups. A FORTRAN user involved in application programming asked for certain additional function in the language such as an ON clause for overflow of storage; the director of a university computing center expressed concern about compilation speed and subsetting and/or teachability of the language; and an ALGOL father requested an escape character and a single form of the FOR statement.

There was general agreement that the NPL Task Force, and the ALGOL Committee are attacking the same problem from two opposite directions. SHARE from a pragmatic direction, the ALGOL persons from a theoretical one. It was further agreed that cooperation between the two groups is desirable and that much could be gained by attempting to describe equivalent features in an identical manner."

In the fall of 1964 a Language Control Board was established in the New York Programming Center. In 1966, this Board was moved to Hursley. As its attempts to clarify and improve the language proceeded, it became clear that none of the 1964 language documents were adequate as a base from which to work. Ray Larner and John Nicholls (joined by David Beech, Roger Rowe and, later, David Allen) took the first steps toward developing a formal semantic definition of PL/I (called Universal Language Definition II). This was followed by a formal definition developed in the Vienna Laboratory by a group managed by Kurt Walk ("Abstract syntax and interpretation of PL/I (ULD Version III)") (14). Later, this was superseded again by the ECMA/ANSI definition referred to earlier. Thus, while it began as a language informally specified with many ambiguities and omissions, PL/I has evolved into one of the most precisely defined languages in use.

Since 1964, many languages both within and outside IBM have been developed as variants on PL/I. The most important of these was EPL, a language developed in early 1965 by Bob Morris and Doug McIlroy of Bell Laboratories for use by the MULTICS project. MULTICS (16), a joint Bell Labs, G. E., MIT system project, was looking for a suitable systems programming language at the time NPL was being defined. The only other serious contender was MAD, a language based on ALGOL 58 developed at the University of Michigan, which lacked recursion, and adequate data structures, and did not interface well with the interrupt mechanism of MULTICS.

NPL had all the necessary features and was already designed. Thus EPL was developed as a subset of NPL. (It lacked complex data types, aggregate expressions, and all I/O.) In about a year the EPL compiler was operational. Almost all of MULTICS was written in this language, accounting for much of the productivity and maintainability of the system.

Today, among IBM program products, there are more PL/I compiler licenses sold to IBM S/370 customers than FORTRAN licenses. PL/I is still second to COBOL among IBM customers for commercial applications. PL/I, and a variant for systems programming, has been used successfully to program several large operating systems and compilers (notably MULTICS, OS/VS Release 2 Version 2, the PL/I Checkout compiler). Several pedagogical subsets have been defined and used in introductory programming books (15). Thus to a large degree the goals of the original designers have been realized. It is being widely used for scientific, commercial, and systems programming.

Design Criteria

It will be helpful, in evaluating PL/I, to review the explicit design criteria of its originators, as well as to see what they did not aim for. There are six such objectives, developed by the Advanced Language Committee members and reported in the January, 1965 Communications of the ACM in a paper by Paul Rogoway and myself (13). They are quoted below:

"Freedom of expression: *If a particular combination of symbols has reasonably sensible meaning, that meaning is made official. Invalidity is a last resort. This will help to insure realistic compatibility between different NPL compilers.*

Full access to machine and operating system facilities: *The NPL programmer will rarely if ever need to resort to assembly language coding. No facility was discarded because it belonged more properly to assembly or control card languages.*

Modularity: *NPL has been designed so that many of its features can be ignored by the programmer without fear of penalty in either compile time or object time efficiency. Thus, manuals can be constructed of subsets of the language for different applications and different levels of complexity. These do not even have to mention the unused facilities. To accomplish this end, every attribute of a variable, every option, every specification was given a default interpretation, and this was chosen to be the one most likely to be required by the programmer who doesn't know that the alternatives exist.*

Relative machine independence: *Although NPL allows the programmer to take full advantage of the powerful facilities of System/360 and Operating System/360, it is essentially a machine-independent language. The specific parameters of the language such as size of numbers, kinds of input/output devices, etc. are independent of the characteristics of any particular machine.*

Cater to the novice: *Although the general specification is there for power, beauty, growth, etc., the frequently used special case is specifiable redundantly in an explicit way. As an extreme example, it is possible to invoke in a consistent way any of the sixteen Boolean functions on two logical variables by calling for* $BFb_1b_2b_3b_4(arg_1,arg_2)$ *b_i (i=1,...4) are independently either zero or one. The operators specifying union and inter-*

section are, however, still explicitly allowed as: arg_1 | arg_2,arg_1 & arg_2. *This approach allows the compiler to maximize efficiency for the commonly-used case, and, again, permits the novice to learn only the notation which is most natural to him.*

NPL is a programming, not an algorithmic language: *Programming languages are most often written on coding sheets, punched at key punches or terminals, and listed on printers. While the specification of a publication language is considered essential, the first and most important goal of syntax design has been to make the listings as readable, and to make the writing and punching as error-free, as possible. A free-field format has been chosen to help the terminal programmer."*

This is an unusual set of objectives when viewed after fifteen years, not so much for what it includes, which is generally reasonable, as for what it omits. Nowhere do we find many of the goals in favor today:

e.g. a solid theoretical basis,

extensibility from a simple base language,

interactive facilities,

producing well-structured programs,

ability to prove correctness.

(Bruce Rosenblatt recalls that extensibility was discussed by the committee, and that the decision to avoid reserved key words was a result of this objective.)

It was still of overriding importance, as it was with FORTRAN, that object code be efficient. Convenience was the key word rather than simplicity. This was a language defined more in the spirit of English than of Mathematics. (There were irregular verbs and idioms.) Success was measured primarily by the ease of specifying large classes of programs in the language. If the right primitives were available, the language succeeded; if tedious procedures had to be written to provide the required functions, the language failed.

Rationale and Evaluation of Contents of NPL/PL/I

This section discusses some of the more interesting characteristics of the December 1964 language and evaluates them in the light of our experience.

-Program Structure

The lexical rules in NPL (which remained largely unchanged as PL/I developed) have proven to be easy-to-learn and quite natural (especially as compared to COBOL and FORTRAN). They removed all mention of column numbers, allowed blanks and comments freely wherever separators were allowed, and thus were naturally adaptable to different media.

The use of semi-colons as statement terminators instead of as separators (as in ALGOL) is an interesting small example of the fundamental differences between the two languages. The ALGOL decision provides a more regular basis for extending the language. The PL/I decision is more readable when listed on a printer with only the standard upper case font. Some studies (17) have suggested that this PL/I usage is easier to learn and less prone to errors but, like all human factors studies, this is quite subjective.

The NPL designers decided also to develop a syntax which did not require that keywords be reserved. This decision was made in order to allow language extensions without invalidating old programs.

An interesting example of the subjectivity of language syntax is the debate over the assignment statement. Since all other statements in NPL began with a verb, it would have simplified the syntax (and thus the parse phase of compilation) to have specified assignment as:

SET variable = expression;

which was, in fact, the syntax in the March report.

Going to a distinct assignment operator as in ALGOL:

variable:= expression;

would also have been reasonable. The committee, being uncertain about the correct decision, informally polled the general SHARE body. The results were completely inconclusive. Each choice had its strong adherents and its equally persuasive detractors. As a result the committee decided to revert to the FORTRAN syntax since it was the most widely understood. The impact of this decision on the compiler was very small. The major problem has been the human confusion caused by using the same symbol for the boolean equality operator and the assignment operator, as in:

A=B=C; .

Facilities for defining aggregates of statements had been provided by subroutines in FORTRAN, and by blocks in ALGOL. The NPL design approach here again reveals its concern with performance over simplicity. The functions provided by this level of structure were separated and a different syntactic form defined for each so that their separate usage could easily be recognized for customized implementation.

To delimit a group of statements simply for control (as in IF THEN clauses or DO WHILE iterations) the syntax DO;...END; was defined. This required no more run-time overhead than a branch.

To delimit a group of statements in order to define scope of variables, and to allow storage sharing among blocks of disjoint scope, the ALGOL-like BEGIN;...END; block was used. This defined a dynamic as well as static state and so a prologue was required in order to establish this state on entry to the block.

Finally to delimit a group of statements which can act either as a subroutine (i.e. the object of a CALL) or as a function invoked within an expression, the syntax PROCEDURE;...END; was used.

-Control Flow

The NPL designers attempted to be very ambitious in the definition of dynamic structure and flow, an attempt that still accounts for much of PL/I's strength and usability, but that introduced some complexity and optimization problems. For separate reasons the following facilities were provided, some of which, even individually, were quite unusual in a high level language:

- creation of asynchronous executing tasks,
- ON conditions specifying blocks to be invoked when specific conditions occurred,
- recursive procedures with static, as well as automatic, storage,
- variables whose values are labels (for computed GO TO as well as abnormal returns from procedures).

It was not so much the problems associated with each of these facilities that caused trouble as their interactions. In this area it would seem that the language had not achieved its goal of modularity. However, several PL/I procedure attributes and compiler options were added to allow the programmer explicitly to declare that some particular language feature would not be used (such as recursion or tasking). The fact that such assertions could cause the compiler to produce efficient code without requiring large compiler modifications argues that some sets of these features were indeed defined to be reasonably independent.

An example of this interaction was the definition of label variables. Consider what had to be contained in the value of a label variable. This value had to include the level of recursion and the name of the task as well as the name of the block in the procedure in which the label occurred. Restrictions had to be defined to preclude label variables from containing values which no longer were active.

Thus, in NPL, label variables could not be static or controlled. On assignment, the persistence of the variable on the left was constrained to be no greater than that on the right.

e.g.

```
P: PROCEDURE (A);
        DECLARE A LABEL,
                B LABEL;
L: B=L;
   A=B;
END;
```

would result in the value L persisting in A after return from P.

In

GO TO label variable;

the variable's value was constrained to be in the same block activation as the GO TO statement so that GO TO could not be used for RETURN. But this could not in general be assured at compile time and so code had to be generated to check for it at run time. (This is an example of a facility carefully defined so that a sophisticated optimizing compiler could find the cases where efficient object code could be safely generated. Often, however, in real compilers developed by programmers under schedule, compile speed and space constraints, this sophisticated analysis is not achieved.)

Similarly, because

RETURN TO label variable;

was allowed freely in NPL, such a return could collapse any number of blocks and procedures and commence at any label in the returned-to-procedure. Thus every procedure invocation which passed a label variable to another procedure had to expect this abnormal control flow and suitably materialize its state (that is, ensure that all variables modified in registers were copied back into storage, and all subsequent references got fresh copies from storage). (Eventually, as PL/I developed, the constraints on GO TO were dropped as was the RETURN TO form, since it had become redundant.)

The NPL document said nothing to preclude passing a label variable across tasks. But it did not define what happened if its value were used in a RETURN TO statement. In general, tasking was one of the more inadequately designed features of NPL. This facility was greatly improved as PL/I evolved but still has quite limited use. A tasking proposal considerably different from that in PL/I was proposed for the PL/I ANSI standard. This too was finally rejected primarily because of difficulties in achieving an adequate definition.

In retrospect it is hard to understand how asynchronous processing capability could have been defined in NPL without locking, queueing, event posting, waiting or any method by which to program synchronization. No explicit definition of atomicity or synchronization was given which would restrict a compilers' ability to move code freely. Thus in general it was difficult to allow tasks to cooperate at all.

Programming languages before NPL (and most languages still) had few general facilities to define actions which were to be taken when conditions such as reading an end of file, overflow, bad data, new line, occurred. Often the conditions could not even be sensed in the language, and assembly language patches were required. Even where the condition could be sensed and explicit "IF THEN ELSE" clauses inserted wherever it could occur, the resulting program was quite awkward.

The NPL designers were determined that entire real applications should be programmable in the language and these programs be properly reliable and safe. Thus they introduced 23 types of conditions and the executable statement:

ON condition action;

This facility provided a powerful control structure for coping with these infrequent occurrences. It resulted, however, in a complex total control flow when coupled with the other aspects of dynamic flow in NPL. For instance, inheritance and stacking of actions were defined across procedure and block invocations. When a condition occurred, the block specifying the action was to be executed (almost) in the state which pertained when the ON statement was executed. This required the ability to suspend one state temporarily and reinstate another. Thus some information necessary to handle the exception might be inaccessible in the ON unit since it was accessible only in the context of the exception itself.

ON conditions, as a general mechanism to take action when unusual or infrequent situations occur, are always more difficult to define and sometimes more difficult to use than would be desired. It is hard, however, in retrospect, to point to the subsequent invention of a simpler or more usable facility. The conclusion that, therefore, the facility should not be in a programming language at all simply was inconsistent with the NPL objectives.

-Storage Management

STATIC storage in NPL was directly modelled after FORTRAN storage management. STATIC INTERNAL corresponds to FORTRAN local storage and STATIC EXTERNAL to FORTRAN Named COMMON. Since NPL supported recursion, AUTOMATIC storage had to be introduced. Allowing recursion at the procedure level does not require dynamic stacking at block entry. But AUTOMATIC variables local to blocks were defined so that binding of extents could be defer-

red as late as block entry time. This did require dynamic stack management and has caused considerable run-time complexity. It was included primarily because the NPL designers in 1963/4 were quite concerned with storage efficiency, because late binding offered a function that was considered to be useful in some future interactive environment, and because the designers again reasoned that an intelligent compiler could recognize the cases where extents could be bound earlier.

CONTROLLED storage, like ON conditions, was introduced to provide, at the NPL language level, functions hitherto available only to assembly language programs invoking operating system services. The NPL definition, however, provided more extensive function than that offered by most operating systems.

Generations of CONTROLLED variables were allowed so that programmers could implement dynamic storage allocation with arbitrary numbers of separate allocations. Extents could change between generations; only the number of dimensions and the data type (and structuring) had to be constant. (This last facility has resulted in controlled storage being significantly less efficient than STATIC or AUTOMATIC. But, since there is no facility in PL/I to create new names at run-time or to grow an existing allocation, its removal would result in a real functional restriction in the language.)

While both CONTROLLED storage and multi-tasking are each useful functions, their interaction illustrates the problems which combinations of such functions can introduce into a language. Consider the definition of CONTROLLED storage in a multi-task program as it appears in the November, 1964 NPL document (4).

> *"An attached task has almost the same access to the attaching task's data as it would have if it were executed synchronously; that is, when it is attached, all allocations of CONTROLLED data known to the attaching task are passed to the attached task. However, subsequent allocations in the attached task are known only within the attached task and its subsequent descendants; subsequent allocations in the attaching task are known only within the attaching task and its subsequent descendants. Thus the stack of allocations for a CONTROLLED variable splits into separate stacks, with a common part preceding task initiation and separate parts after task initiation. A task may only free storage which it allocated. All storage allocated within a task is destroyed when that task is completed."*

This definition is also illustrative of one of the most serious and pervasive problems with the NPL document (4) a problem which resulted primarily from the scope of the definition and the pressures of its schedule. It is a short paragraph attempting to define a very complex, powerful facility and thus leaving unanswered many detailed points. For example, can the creating task free the shared generation? If so, it can now access the previous generation; can the created tasks also now access this generation? (It should be noted that even this document represented a major improvement in level of definition over the much shorter, more ambiguous document developed by the Advanced Language Development Committee (3).)

Another innovation in the NPL storage model was the concept of auxiliary storage. In all programming languages, and in most systems without virtual memory, data which cannot fit within available main memory must be explicitly written to auxiliary storage using I/O facilities. This means that files must be defined, opened and written. The mapping of data aggregates to records must be defined by the programmer.

In NPL the statements SAVE and RESTORE were defined to permit moving the current values of a variable to auxiliary storage, and subsequently retrieving them, without requiring that the user cope with the complexities of I/O. The function was provided in great generality and required extensive run-time support. Stacks of generations of SAVE's were allowed, as was retrieval by key. (These stacks were similar to those of controlled variables. One would expect a similar approach to tasking, but in fact tasking was not mentioned in the section on SAVE and RESTORE.) The complexity of implementation, and a general feeling that virtual memory would eventually provide a better solution to the same problem led to the eventual removal of SAVE and RESTORE from NPL.

It is surprising, in retrospect, to recall that one of the most powerful aspects of PL/I, widely-used for systems programming, was not in the original NPL definition at all. Namely, all notions of pointers, areas, offsets and based variables are completely missing. Moreover, none of the criticisms about the various versions of the language noted this omission. (Similarly, the notion of a reference is missing from ALGOL 60.)

This part of the language was introduced when one PL/I compiler group attempted to program their compiler in PL/I, and when General Motors stated their need for pointers in their graphics applications. An early description of these facilities in PL/I was a paper by Bud Lawson, produced in October 1965, entitled "A Guide to PL/I List Processing" (12). After discussing the problem in general, Bud introduces the notion of a based variable (but without that explicit attribute) and a pointer:

> *"This facility to operate upon pointers and list data elements is provided by extending the CONTROLLED storage class to include pointer variable. For example, consider the following declaration.*
>
> *DECLARE ALPHA FLOAT CONTROLLED (P);*
>
> *This declares ALPHA to be a floating point variable, and that the pointer P will be used to identify the location of ALPHA when ALPHA is referenced. P is assumed contextually to have the attribute POINTER due to its position in the CONTROLLED attribute. This new form of controlled variable will be referred to as a based variable. It does not imply stacking of allocations of data. Instead, it provides a descriptive device for describing the structure of data that exists in any storage class (STATIC, AUTOMATIC or CONTROLLED)."*

It has been argued that the UNSPEC built-in function (which returns the underlying bit representation of its argument) and the ADDR built-in function (which returns the address of the first byte of its argument) coupled with free pointers containing real memory addresses, exposed too much of the underlying storage structure. Certainly much sophisticated optimization has been made more difficult because of these features. On the other hand they made feasible the programming of many procedures which would otherwise have been written in Assembly language.

-Data Types

Nowhere is the "swiss knife" aspect of NPL more apparent than in its unparalleled variety of data types. Even though the language still lacked Entry variables, Areas, Offsets, Pointers and Events the interactions of its many data types accounted for much of its apparent complexity.

Except for Label and File variables, data were of two types: arithmetic or string. Strings were character or bit and could be of fixed or varying length. Facilities for substring selection, concatenation, searching, etc. were provided.

The question of whether a language should treat strings as one-dimensional arrays is still debated today. The Bell Labs "C" language (18), for instance, supports only a single character scalar data type which must be declared in an array to represent a string. An insight into the level of discussion in 1964 can be gained from this quote from a memo to Fred Brooks, from me, dated July 24, 1964 (25). Fred, manager of OS/360 at the time, had asked the question at an earlier meeting.

> *"The type string is, as you have remarked, unnecessary. We could have introduced instead a type character and a type bit (or logical), each being of constant length 1. We could then use the mechanism of a vector of characters or bits to accomplish the string facility. This has two interesting consequences:*
>
> *a. People (and there really are some) who have not had any previous exposure to the concepts of vectors, arrays, dimensions, etc., are going to run very quickly back to COBOL unless we publish a manual for them which cleverly disguises these notions in more familiar terms. Even some people who do know about vectors will find it hard not to think of JOE as a scalar. But more important if JOE is really (J,O,E), SAM is really (S,A,M) and DANIEL is really (D,A,N,I,E,L), note that (JOE,SAM,DANIEL) is not an array. I think confusion, rather than clarity, will result.*
>
> *b. Suppose we adopt the proposed approach. A character string is then just a vector. The concatenation of two character strings is just the concatenation of two vectors. We don't have this facility yet, but it's obviously necessary, so we'll introduce it. Analogously, we will have to introduce, for vectors, an infix notation to reference a substring, an index function to search the vector for a given pattern, a variable length facility since fixed length strings are intolerable. Now, having provided all this for arrays of dimensionality one, we will certainly feel compelled to generalize the facilities to higher dimensionality. Of course, here the notations became a bit more complicated since A concat B is ambiguous for arrays unless one states the dimension to be concatenated. You see that what you are asking turns out not to be the treatment of strings as arrays, but quite the contrary. I'm not knocking the proposal, by the way. We suggested it several times at our SHARE committee meetings and were greeted by great reluctance among the implementors (and we really couldn't blame them)."*

Another debatable question was whether bit strings should be treated like character strings or like variable-length binary integers. NPL chose the former direction, which results in left justification and, hence, unnatural conversions between bit and binary. This can result in undesirable effects when the bit strings are meant to represent numbers.

In the April, 1964 paper from the Vienna Lab which reviewed NPL (10), a proposal was made to reverse this decision, that is, to treat bit strings as numbers:

> *"Operations on bit strings are (in this proposed change, ed.) defined as bit-by-bit operations (with the exception of CONCAT), beginning at the right side of the strings. If the strings are of different lengths, the shorter string is extended on the left side with zeros to the length of the longer string, for all operations. The bits are numbered from right to left.*
>
> Remark
>
> *This rule has the advantage that the numeric value assigned to a string...is preserved by the extension.*
>
> •
> •
> •
>
> Remark
>
> *The inconsistency of the rules for bit-and-character strings seems to be unavoidable, since it is due to a fundamental difficulty (writing numbers beginning with the most significant digit)."*

Arithmetic data could be "numeric fields" or in "coded form". (A numeric field was defined as a character string whose PICTURE attribute enabled a unique numeric value to be determined. Thus arithmetic data could be stored with explicit decimal points, dollar signs, leading blanks, etc.) Arithmetic data could be decimal or binary, fixed or floating point, real or complex. The "precision" could be specified by declaring the total number of digits and the number of digits to the right of the decimal (or binary) point.

Arrays were allowed to have arbitrary dimensionality and arbitrary lower and upper bounds for each dimension. Cross sections of arrays were specifiable by inserting asterisks in those dimensions which were to vary (e.g. A(I,*) referred to a vector, which was the i'th row of A). Structures of arbitrary depth were defined with name qualification to ensure uniqueness of references.

The decimal and numeric field facilities and structures were largely taken from COBOL, the fixed binary, floating point and arrays from FORTRAN. What is remarkable is that, with very few exceptions, the attributes were orthogonal. (For instance, any radix was allowed with any mode and any encoding or precision; arrays of structures and structures with array elements were permitted.) The language and compiler complexity that this freedom ultimately caused was not fully anticipated by the NPL designers. This is illustrated by the fact that while, already in the NPL document of 1964 (5), great detail is given to the specification of the PICTURE attribute, conversion is covered entirely in a little over one page. The later PL/I document required 17 double column pages, with many tables, to define all the permissible conversions.

Many combinations are seldom used (e.g. fixed decimal components of complex numbers). Others have, in fact, not been widely used even though one can create examples in which they are appropriate (e.g. fixed binary data which are not integers). The developers of NPL, and PL/I later, always

intended that numeric data attributes describe the ranges of values and the precise meaning of operations on data, and not imply an internal representation. (Honeywell PL/I, for instance, represents FIXED DECIMAL as binary numbers with power of ten scaling; thus DECIMAL and BINARY integers are represented identically.) What has happened, in spite of this intent, is that those data types which correspond closely to S/360 machine-supported operands are used almost entirely (e.g. fixed binary integers of precisions 15 and 31). Using data types which are much different from these and using conversions with loss of information (e.g. float to fixed) often results in unexpected program behavior.

It is always difficult to define arithmetic and assignment for programming languages or for machines. Numerical analysts disagree among themselves; there are conflicting goals which cannot all be satisfied. For instance, a subject of continuing debate in NPL, and later in PL/I, was whether the result of an operation should be uniquely defined by the attributes of its operands (which simplifies language definition and compilation), or whether the result should be influenced by its eventual use (which often leads to more useful and expected answers). Taking the former approach (as in FORTRAN), 3/2+.5 might yield 1+.5=1.5, while 3/2+1/2 might yield 1+0=1. Taking the latter approach might always yield 2, but is difficult to define in general and would make finding common subexpressions for optimization difficult until the text had been translated to a lower level. PL/I rules yield the expected 2 in these cases but surprise the programmer who tries 3/2+9 when OVERFLOW is disabled.

The basic principle which PL/I follows is to preserve precision when the hardware allows it, and to use overflow otherwise. This principle, and the specific PL/I precision rules, were debated at great length during the standard definition process. Every alternative proposal was found to have its own set of problems and so, in the end, the PL/I rules were preserved.

But the debate continues, centered largely around fixed point divide. I quote here from a recent letter to me from Doug McIlroy recalling the NPL debate (24): *"One case in point that I remember especially is the rules for arithmetic: We all had hoped to make definitions that would yield generally agreeable answers for most plausible expressions and work well with constants, which should have indicated value without forcing type. One old shibboleth stuck with us all the way through: Commercial users were expected to balk unless division of fixed point numbers yielded fixed point results. Rules that made sense, gave all the bits you were entitled to, and catered for fixed point divide, eluded us, despite any number of tries at different bottom up or top down ways to arrive at coercions. To this day I believe that had we cut the Gordian knot of fixed point divide, there would have followed a more plausible and less demanding rule. (If you declare your integers to be full single precision numbers, the minute you add two of them you get double precision arithmetic.)"*

Many programming language definers take the easy way out. They simply do not define the results of computation and assignment in any detail. This results in languages which are proudly "machine independent" but programs which are not. Machine architects contain the problem generally by not allowing mixed attribute facilities at all. Where they do, as in the Move Character Long instruction in S/370, the specification is as complex as it is in PL/I.

One of the more controversial PL/I decisions concerned the order of computation and assignment for array expressions. In APL monadic and dyadic operations on arrays are performed right to left, one operator at a time. Thus

```
A←A+A[1]
```

means the same as

```
TEMP←A[1]
A←A+TEMP.
```

In PL/I, however

```
A = A + A(1);
```

meant the same as

```
DO I=1 TO N;
    A(I) = A(I) + A(1);
END;
```

Thus the new value of A(1) is added to A(2)...A(N).

There are good arguments for either decision. The PL/I approach can produce more efficient code and does not require temporaries. There are cases also where storage hierarchies are accessed much more efficiently. The APL approach seems more natural to programmers. (The ANSI standard PL/I has, in fact, changed to the APL approach.) What is most interesting historically, however, is that the problem does not seem to have been recognized in the NPL document. Not until the PL/I (F) document was there an example inserted to call attention to the difference.

-I/O Facilities

ALGOL chose to leave I/O as a facility for each system to provide as it wished. Thus, while algorithms specified in ALGOL will run on different systems, programs may not. FORTRAN invented the concept of a format list and a data list, thus allowing entire programs to run provided the files which they accessed were compatible and the control statements properly modified. COBOL introduced more extensive facilities for the processing of records and preparing of reports than any other language before or since.

The NPL designers were determined that both FORTRAN and COBOL programmers should not find their product lacking in the functions they were accustomed to, and that the external data management facilities which they were told would be available to OS/360 assembly language programmers would be invokable in NPL. Thus more than 10 percent of the NPL definition dealt with I/O. It had more extensive format-list capability than FORTRAN and, while not as rich as COBOL functions generally, it reflected what little was then known of the OS/360-to-be features such as the TITLE option (for DD names), the SORT statement (designed to invoke the SORT utility) and the TASK option on READ and WRITE (for asynchronous I/O). It is interesting to observe that ANSI made very little change to PL/I in the name of operating system independence (although asynchronous I/O has been removed). Hence either the NPL designers succeeded in their goal of generalizing these OS/360 facilities or the OS system facilities have been widely adopted in the industry.

-Compile-Time Facilities

NPL compile-time facilities were developed with conventional macro-assemblers in mind. The functions, with the exception of macro procedures, were all standard. The syntax was constrained to allow one-pass processing. Surprisingly, although equivalence to JOVIAL motivated much of NPL's

fixed binary data type definition, %INCLUDE (PL/I's equivalent of COMPOOL) was removed from the language in the December 1964 document.

-Implicit and Default Attributes

Perhaps because of the FORTRAN background of its designers, NPL (and PL/I) did not require that all attributes be explicitly declared. If identifiers were not declared at all, implicit or contextual data types were assigned consistent with their use in the program (e.g. GO TO L; implied that L was a label variable). If the rules for implicit declaration established the variable as numeric, NPL adapted the FORTRAN convention of assigning fixed binary type to identifiers whose names began with the letters I through N, float otherwise. (This rule has been removed in the PL/I standard.) NPL, however, responding to an outstanding request for extension to FORTRAN, added an IMPLICIT statement, by which any set of attributes could be implied for variables whose names began with specific letters.

But the real complexity (and confusion) entered the language with default, rather than implied attributes. NPL did not require that all attributes be explicitly declared for all variables. The missing attributes were given default values depending on the ones which were explicitly presented. (E.g. explicitly declaring a variable to be BINARY gave it a default scale of FLOAT; explicitly declaring FLOAT gave it a default radix of DECIMAL; the default lower bound of an array dimension was 1). Given all the different kinds of attributes in PL/I, defining natural and consistent defaults for any sensible subset became a sizable task.

Today the computer science community generally does not approve of all this defaulting, nor of implicit conversions. If the program is not explicitly correct, we generally argue that it should not compile. But how much, one may ask, is this change in objectives due to the fast, cheap turn-around offered by contemporary systems? Would we be as willing to be forced to resubmit if each submission meant a day's delay?

Implications for Current and Future Languages

NPL was a language defined too quickly. Its virtues resulted primarily from the combined intuitions (and programming experience) of its definers. (They convened "knowing" what they wanted in the language.) Its faults were largely ambiguity and incompleteness.

In retrospect it appears that these definers correctly assessed the unique strengths of the languages from which they drew inspiration. They took from FORTRAN the notions of separately compiled subroutines sharing common data and the DO loop. From COBOL they introduced data structures, I/O, and report generating facilities. From ALGOL they took block structure and recursion. A major contribution of NPL was not only its original facilities but the combining of these borrowed characteristics into a consistent single language.

If one reads the 1964 NPL documents and compares them with PL/I today it becomes clear that the Hursley language definers did much more than simply clarify and extend. They took an impressive first effort (NPL) and, spending many years in the process, fashioned a well-defined, usable programming language.

In the first years (1964-66), these clarifications were motivated largely by the need to know precisely what to compile, to make compilation tractable, and to make object code reasonably efficient. They were eventually motivated also by users' comments, particularly from SHARE and GUIDE, but, in the years between the NPL document and the release and wide use of the PL/I F compiler in 1966, little real programming experience existed to demonstrate language change requirements.

Like almost all of its contemporary languages, PL/I suffered from being defined at a single level. If it was felt that a facility was useful and could be compiled reasonably it was added to the language. In a subsequent wave of language definition ALGOL 68, SIMULA, PASCAL, and PL/I compile-time facilities turned (in very different ways) to extension facilities as a means of adding data types and operators.

The strongest characteristic of PL/I is its breadth and detail. It is really feasible to write entire application (and even system) programs in the language without resorting to assembly language. It may be that programming in the eighties will be divided between applications programmers, who will want to communicate in many different, very high-level, special purpose languages, and highly skilled systems programmers, who will provide the processors, libraries and services for the former. Even more important than its wide use in scientific and commercial application programming is the role PL/I (and its derivatives) may play as a language for these systems programmers.

Bibliography

1) Report of the SHARE Advanced Language Development Committee, March 1, 1964.

2) Specifications for the New Programming Language, April, 1964.

3) Report II of the SHARE Advanced Language Development Committee, June 25, 1964.

4) The New Programming Language, IBM UK Laboratories, November 30, 1964.

5) NPL Technical Report, IBM World Trade Laboratories (Great Britain) Ltd., December, 1964.

6) OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, GC33-0009-4, IBM Corporation, October, 1976.

7) American National Standards Institute," American National Standard Programming Language PL/I", New York, N. Y. ANSI X3.53, 1976.

8) European Computer Manufacturers Association, "Standard for PL/I," Standard ECMA-50), Geneva, Switzerland, 1976.

9) Bandat, K., Bekic, H., Lucas, P., "Block Concept for NPL," IBM Laboratory, Vienna, July 1964.

10) Bandat, K., Kudielka, V., Lucas, P., Walk, K., "Review of the 'Report of the SHARE Advanced Language Development Committee'," IBM Laboratory, Vienna, April 1964.

11) Comba, P., "A Comparative Evaluation of the New Programming Language", IBM, Time-Life Bldg., New York, July 1964.

12) Lawson, H. W., Jr., "A guide to the PL/I list processing," SDD-PL/I, Time/Life Bldg., N.Y.C., October 1965. (See also Lawson, H. W., Jr., "PL/I List Processing", CACM, Volume 10, No. 6, pp. 358-367, June 1967).

13) Radin, G., Rogoway, H. P., "NPL: Highlights of a New Programming Language," Communications of the ACM, Volume 8, Number 1, January 1965, pp. 9-17.

14) Walk, K., Alber, K., Fleck, M., Goldmann, H., Lauer, P., Moser, E., Oliva, P., Stigleitner, H., Zeisel, G., "Abstract syntax and interpretation of PL/I (ULD Version III)," Technical Report TR 25:098, Vienna: IBM, April 1969.

15) Conway, R. W., Gries, D., "A primer on structured programming using PL/I, PL/C, and PL/CT", Winthrop Publishers, Cambridge, Mass., 1976.

16) Corbato, F. J., Vyssotsky, V. A., "Introduction and overview of the Multics system", Proc. AFIPS 1965 FJCC, Vol. 27, Spartan Books, New York, pp. 185-196.

17) Gannon, John D., Horning, James, "The impact of language design on the production of reliable software", presented at the International Conference on Reliable Software, 1975, and published in Technical Report CSRG-45, Dec. 1974, University of Toronto, Toronto, Ontario.

18) Kernighan, B. W., Ritchie, D. M., "The C Programming Language," Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

19) Rosenblatt, Bruce A., Letter to George Radin, Dec. 1977.

20) Larner, Raymond A., Brown, Lawrence C., "FORTRAN VI Language Notebook," IBM memo, Feb. 1964.

21) Rochester, N., "A FORTRAN VI Proposal," IBM memo, 1963.

22) IBM report, Feb. 1963.

23) IBM memo, Nov. 1964.

24) McIlroy, M.D., Letter to George Radin, Dec. 1977.

25) Radin, George, IBM memo to Dr. F. P. Brooks, July 24, 1964.

Appendix A

(NEW = Language Described in Share Report
of April, 1964)
OLD = FORTRAN IV)

NEW

```
            PROCEDURE PROBLEM1
            GET (A, B, C, D) ' E(14, 5) '
RDACD.   GET (X) 'E(14, 5) '
            ON EOF RETURN
            IF ( X EQ D ) ( SET FOFX = 0 GO TO
               PTALINE )
            IF ( X LT D ) SET FOFX = X*(A*X + B)
               + C
                  ELSE SET FOFX = X*(-A*X + B)
                                 - C
PTALINE. PUT (X, FOFX) ' E(14,5) '
            GO TO RDACD
            END
```

OLD

```
C     PROBLEM 1
         READ (5,1) A, B, C, D
      6  READ (5,1) X
         IF ( X - D) 2,3,4
      2  FOFX = X*(A*X + B) + C
         GO TO 5
      3  FOFX = 0.0
         GO TO 5
      4  FOFX = X*(-A*X + B) - C
      5  WRITE (6,1) X, FOFX
         GO TO 6
      1  FORMAT ( 4F14,5 )
         END
```

NEW

```
            PROCEDURE PROBLEM2
            DECLARE PEOPLE (20), FIXED,
               =20*0 / SALARY (20), =20*0
                     / AGE, FIXED,
                        PICTURE(ZZZ) / INCOME, FLOAT
RDACD.   GET(AGE, INCOME) 'F(3), E(10,2)'
            IF ( AGE NE-1 ) ( SET I = AGE /5 + 1
                  SET PEOPLE (I) = PEOPLE (I) + 1
                  SET SALARY (I) = SALARY (I)
                     + INCOME
                  GO TO RDACD )
            DO LOOP I = (1,20)
            IF ( PEOPLE(I) NE 0 ) SET SALARY (I)
               = SALARY (I)/PEOPLE(I)
            SET AGE = 5*I - 5
LOOP.    PUT(AGE, SALARY(I), PEOPLE(I))
               'F(3), E(10,2), F(7)'
            RETURN
            END
```

OLD

```
C     PROBLEM 2
         INTEGER AGE
         REAL INCOME
         DIMENSION PEOPLE(20), SALARY(20)
         DATA(PEOPLE(I), SALARY(I),I=1,20)/40*0.0/
      1  READ (5,3) AGE, INCOME
         IF ( AGE + 1 ) 8,4,2
      2  I = AGE/5 + 1
         PEOPLE(I) = PEOPLE (I) + 1.0
         SALARY(I) = SALARY(I) + INCOME)
         GO TO 1
      4  DO 7 I = 1,20
         IF ( PEOPLE(I) ) 8,6,5
      5  SALARY(I) = SALARY(I)/PEOPLE(I)
      6  AGE = 5*I - 5
      7  WRITE (6,3) AGE, SALARY(I), PEOPLE(I)
      8  STOP
      3  FORMAT ( 13, F10.2, F7.0 )
         END
```

```
            NEW

            PROCEDURE PROBLEM3
            DECLARE FIXED PRIME / FACTOR
            DO LOOP2 PRIME = (3,1000,2)
            DO LOOP1 FACTOR = (2,SQRT(PRIME))
LOOP1.IF    ( MOD(FACTOR,PRIME) EQ 0 ) GO TO LOOP2
            PUT(PRIME) 'F(4)'
LOOP2.      CONTINUE
            RETURN
            END



        OLD

C       PROBLEM 3
            INTEGER PRIME, FACTOR
            PRIME = 3
        3   A = PRIME
            A = SQRT(A)
            J = A
            DO 1 FACTOR = 2,J
            IF ( MOD(FACTOR,PRIME) ) 1,2,1
        1   CONTINUE
            WRITE (6,5) PRIME
        5   FORMAT ( 120)
        2   PRIME = PRIME + 2
            IF ( PRIME - 1000 ) 3,7,7
        7   STOP
            END
```

```
            NEW

            PROCEDURE PROBLEM4
            DECLARE FLOAT I/L
RDACD.      GET(R, F, L)'E(14,5)'
            ON EOF RETURN
            SET R = R**2
            SET F = 6.2832*F
            SET FACTOR = F*L
            GET(CO, CF) 'E(14,5)'
            PUT(R, F, L) 'E(14,5)'
            DO LOOP E = (1.0,3.0,0.5)
            PUT (E)'E(14,5)'
            DO LOOP C = (CO,CF,1.0E-8)
            SET I = E / SQRT( R +
                (FACTOR-1/(F*C))**2 )
LOOP.       PUT(C, I) 'E(14,5)'
            GO TO RDACD
            END



        OLD

C       PROBLEM 4
            REAL I, L
        7   READ (5,1) R, F, L
            R = R**2
            F = 6.2832*F
            FACTOR = F*L
            READ (5,2) CO, CF
            WRITE (6,1) R, F, L
            E = 1.0
        9   WRITE (6,1) E
            C=CO
        5   I = E / SQRT( R + (FACTOR-1/(F*C))**2 )
            WRITE (6,2) C, I
            C = C + 1.0E-8
            IF ( C - CF ) 5,5,6
        6   E = E + 0.5
            IF ( E - 3.0 ) 9,9,7
        1   FORMAT (3F14.5 )
        2   FORMAT (2E14.5 )
            END
```