PL/I

Mr. C.A.R. HOARE

1. Design Aims.

The PL/I language has been developed by ~~a particular computer manufacturer~~, IBM, as a possible successor to the languages which exist at the moment, ~~namely~~ mainly Fortran and Cobol, ~~and Algol to a certain extent.~~ and ALGOL 60. The objective of the language is very clearly stated. It is to provide a high level programming language which will cover every aspect of the ~~possible~~ use of a computer: it is even hoped that it could replace the use of assembly language for ~~all~~ most purposes, including the writing of software; but it is possible that this hope will not be entirely fulfilled.

In order to achieve this objective, the language ~~is~~ has to be a very large one, including a great number of features and facilities, each of which is designed to look after a particular area of data processing. It is so large a language that it seems quite likely there will be no one person who will ever know the whole of it thoroughly. Each programmer will therefore know as much of the language as is necessary for his purposes, and will be able to neglect those aspects of the language which he does not need. The commercial programmer will find in the language all that he wants for the expression of data processing problems in commercial applications. The scientific programmer will find normal arithmetic facilities equivalent to what he is used to from Fortran and Algol; and if, in fact, his knowledge is confined to these particular aspects that are of interest to him, he need never know of the advanced facilities for list processing or real time data processing which are also included.

This language characteristic is known as "modularity"; it is most clearly exemplified in the language by the default concept. This permits a programmer to use only a very small subset of the vocabulary of the complete language; the language implementation will supply the characteristics of the programme or its data which the user does not specify, and of which he may, in fact, not be ignorant. He can therefore declare his variables simply as either fixed point or floating point, in which case they will be assumed by default to be either integers or real numbers; or, if he knows about it and is interested in it, he can control exactly the precision of his fixed and floating point numbers, and can even specify that they are to be held internally in binary, or in decimal form.

The other very characteristic feature of PL/I, which is one of its design aims, is the freedom of expression with which it allows a programmer to write his programme. In particular, if the programmer wishes to be laconic, he can leave out all the declarations, and just write the statements; the compiler will guess from what he has written, what he meant to write. It will supply the missing information in the form of complete and full specifications of the features which the compiler believes the programmer meant to put in.

This freedom of expression which the programmer has is completely defined in the language. It should be possible by looking in the manual to see what the exact effect of leaving out a particular kind of declaration is going to be.

However, in practice this would be quite difficult as the manual is drafted at propent. There is no authoritative definition of these rules in some simple notation like the Backhus normal form of the Algol Report, and it is doubtful whether any programmer could be expected to know off-hand what default attributes the compiler is going to assign to his variables and to his programme. The attitude towards the more obscure points seems to be that the programmer should try it out, and see whether the compiler can make sense of what he has written, and whether it does so in the way he intended.

These are the four major design aims of the language: multi-purpose application, modularity for ease of instruction and use, default attributes to assist in achieving modularity, and a very wide freedom of expression for the programmer. They are clearly declared as aims of the language, and are described very well in the document which defines the language [1].

2. Data Types.

As an example of the comprehensiveness and the thoroughness with which this language has been designed, it is instructive to consider in greater detail the variety of data types which are provided. An item of data is, in ALGOL for instance, an integer variable or a real variable or a Boolean variable. The power of a language may to some extent be measured by the range of types of quantity which it is capable of manipulating.

PL/I introduces a distinction between two kinds of number, a binary number and a decimal number; and the programmer can specify in which form he wants his numbers to be stored. Both options are provided in the language, without regard to the basic arithmetic capabilities of the computer on which it is implemented; although obviously on exclusively binary computers, the use of decimal data types would result in a very inefficient program; and similarly, the use of binary types on a decimal computer. Another basic distinction between numbers is whether they are fixed point or floating point. A floating point number has a mantissa and an exponent, and a fixed point number has an integral part and a fraction part. Note that a fixed point number does not have to be an integer. Finally, there is the distinction between real and complex numbers, which can also be specified in the language. So we have three criteria on which numbers can be classified:

b binary / decimal
f fixed point / floating point
r real / complex

By Cartesian multiplication, this gives eight basic categories into which numbers can fall, varying from decimal float real, to binary fixed complex; this last is probably the least widely used choice in the range.

For a floating point number one can specify (as an integer constant) the number of significant digits, binary or decimal, which are required in the mantissa. For example, one can specify a 16 digit decimal floating point number, or a 37 digit binary floating point number, as a quantity to be manipulated in the programme; thus:

```
DECLARE X DECIMAL FLOAT(16),
        Y BINARY  FLOAT(37);
```

For a fixed point number, one can specify the total number of digits required, either binary or decimal, depending on the mode, thus:

```
DECLARE I FIXED(6);
```

If the number has a fraction part, the programmer can specify how many of those digits he wishes to appear after the decimal (or binary) point. Thus to specify a fixed point number with six digits before and two digits after the decimal point, he would write:

```
DECLARE J DECIMAL FIXED ( 8 , 2 )
```

The techniques outlined above enable a very wide variety of number ranges to be specified. However, there is also a completely different and even more powerful mechanism (the picture specification) for specifying what a number is to be. That is by the picture specification, which achieves yet further refinements of choice. For example, using the previous method, one cannot specify whether a number is to be signed or not. If there are eight digits, then the numbers can vary from minus 99,999,999 to plus the same amount; and one cannot specify that, in fact, only the positive half of the range is going to be used.

Picture specification makes even that possible:

PICTURE '99999999'

This specifies that the number consists of exactly eight decimal digits, with no room for the sign at all. If there is to be a fractional part as well, two methods are provided:

(1) PICTURE '999999V99'

in which V indicates an "implied" decimal point, which does not actually appear in the storage of the computer, but which is assumed to be there when arithmetic is performed on this quantity.

(2) PICTURE '999999.99'

in which the point is actually stored in the computer's memory. This means that whenever arithmetic is performed on this quantity, the point may have to be taken out by software before the operation can be accomplished. Similar techniques are available for binary numbers. To specify a three-bit binary number which is unsigned (that is all its digits are to be interpreted as binary co-efficients, and there is no sign digit), one can write:

PICTURE '111'

If the number is to be potentially signed, and the sign is to be interpreted in 2's complement notation, one would write:

PICTURE '222'

If the number is to be a three digit binary number, but potentially signed and stored in the sign plus modulus convention, one would write:

PICTURE '333'

The above account demonstrates that the language does give an immense amount of control, not only of the sort of numeric data that is to be manipulated, but also of exactly how it is to be stored and represented. In addition to numeric data, there are also character strings and bit strings included in the language. Any variable can be specified as holding a character string of a certain length, say twelve:

```
CHARACTER (12)
```

or a bit string of a certain length, say three:

```
BIT (3)
```

A character string is considered as just a sequence of characters in Extended Binary Coded Decimal Interchange Code (EBCDIC). A bit string is a series of bits, for example 101, which is interpreted not as an arithmetic quantity but as a series of truth values. Now, of course, a character string can also contain a set of characters which represent a decimal number, or, indeed, a binary number, so that the character string provides yet another way of holding numbers in the computer, which exists in parallel to all the other methods outlined above.

A very striking feature of PL/I is that, in spite of this immense variety of data types, there is virtually no restriction whatsoever on the mixture of quantities of any of these types in arithmetic expressions. One can add a character string to a binary variable stored in 2's complement form, and all the conversions will be inserted automatically in the implementation, to produce a possibly meaningful result.

3. Language Features.

In a brief survey, it is very difficult to cover all the features of the language; but in order to give a rough idea of the magnitude of the language one can consider all the major features of Algol, Fortran and Cobol to be mixed together, giving a basis on which has been built the more novel facilities of PL/I. Many of these facilities are inessential, - they are notational facilities which enable one to express more conveniently things that can already be accomplished in Algol, Fortran or Cobol, but to do so in more convenient ways.

As an example of an inessential facility which is very attractive, consider a procedure INVERT which operates on a two-dimensional array and performs a matrix inversion. Suppose also we have a three-dimensional array A, and we wish to invert the matrix which is formed by one of its 2-dimensional cross-sections. This may be achieved by writing, for example, the statement:

```
CALL INVERT ( A [* , 17 , *] )
```

This fixes the value 17 as the value of the second subscript and treats the first and third subscript positions as being controlled inside the inversion procedure. Similarly, one could write:

```
CALL INVERT ( A [J , * , *] );
```

or

```
CALL INVERT ( A [* , * , K] );
```

This is an example of a non-essential feature. It enables one to express in the language more conveniently something which can be already expressed in languages such as Fortran or Algol, but in a more cumbersome way.

The extra facilities which are really substantial contributions to the power of the language and of the running programme can be categorised as:

(1) the list-processing feature, which is more powerful than Lisp, and contains Lisp as a special case.

(2) Powerful file processing capabilities.

(3) A tasking feature which enables the programme to control simultaneous execution of several branches of the same programme.

and (4) A macro-feature which enables the programmer to use a part of the power of the language itself to process ~~language texts before they are translated~~ the text of the language at compile time before passing it through the rest of the translator.

This ~~third~~ fourth feature is, at the date of the lecture, undergoing a major redesign, and is not further described here.

3.1 Tasking.

The tasking facility is based on allowing the programme to branch at any statement and to carry out a given statement in parallel with the rest of the programme. For example, the statement

```
CALL INVERT ( A [J , * , *] ) TASK (B);
```

specifies that the process of inverting the matrix will be carried out in parallel with the rest of the calculations. In other words, having set up the call of this procedure, and having planted the parameters, the main program is free to carry on, while the subroutine executes its operations in parallel with it.

If the computer has a facility for executing several streams of instructions in parallel, the parallelism will be genuine. Even on conventional single-stream computers a simulated parallelism can be set up, which is useful when the procedure being called contains input-output instructions which may be held up by the peripheral hardware of the computer. In effect, the procedure which is specified to operate in parallel with the main program will proceed in the normal fashion until it gets held up on a peripheral transfer. The processor can then continue with the rest of ~~this~~ the main part of the programme, and only go back to complete the procedure invoked in parallel when an interrupt comes through to indicate that the peripheral operation is ~~over and the waiting time is finished.~~ terminated.

There are various additional facilities provided for synchronising processes which have been called to operate in parallel, to allow them to converge in a particular place, to make sure they keep in step when this is required, and to protect areas of the programme which would yield disastrous results if they were obeyed twice simultaneously. That is one of the more ~~delightful~~ fundamental problems of parallel processing, ~~where you have a statement like this:~~ of which the simplest example is:

```
N = N + 1;
```

If two processes attempt to obey this statement simultaneously, one of them will get hold of the N, increment it, and put it back in N. Meanwhile the other one has got hold of N and also added one to it, and put it back again; but because these events have overlapped, the total effect of adding one to N twice is only to add it once.

A statement or sequence of statements with this property is known as a critical portion, and the programmer has got to ensure that he does not try to obey such ~~statements~~ portions twice simultaneously in parallel processes.

3.2 List Processing.

The list processing facility is, in fact, very similar to that described under the title of Plex Processing, in these lectures [2]. It is based on the structure definitions of the commercial aspect of the language; it enables structures to be dynamically generated, and provides for the use of pointer variables to point to the structure, thus enabling any particular generation of a structure to be identified.

This section gives a brief introduction to the basic concepts.

All declarations in PL/I are introduced by the word DECLARE:

```
DECLARE PERSON,
```

The identifier declared here is "PERSON". Since this is to be a major structure name, a level number is put in front of it:

```
DECLARE 1 PERSON,
```

To specify that this major structure describes the form of a record, and to indicate that the programmer wishes to keep control of the allocation and deallocation of storage to records of this structure, one would write:

```
DECLARE 1 PERSON CONTROLLED,
```

One could have specified that the allocation of storage to a record of this structure was to be AUTOMATIC, in which case the storage would be allocated on entry to the block in which it was declared. Or one could have specified it to be STATIC, in which case storage would have been allocated outside the procedure, and it would have acted like an own variable in the Algol sense. Finally, for list processing one requires a fourth type of control, the one which permits an arbitrary number of records of identical structure to exist simultaneously; and which enables the programmer to select at any time a particular one of them by means of a pointer. In this case one would write:

```
DECLARE 1 PERSON CONTROLLED (W),
```

The identifier W is here declared as a variable capable of containing a pointer which is going to point at any given time to exactly one of the records of this structure which exist in the store. Now, we must define the collection of variables which constitute records of the structure PERSON, and which indicate the salient features of each individual person represented by such a record. The first requirement is perhaps an integer component to indicate the date of a person's birth:

```
    2 DATE_OF_BIRTH FIXED,
```

Then one wishes to indicate whether the person is a man or a woman:

```
    2 MALE BIT (1),
```

One also wishes to record for each person-record, who that person's father was; this is achieved by declaring a component to hold a pointer which is going to point to a person's father, thus:

```
    2 FATHER POINTER,
```

The complete structure declaration is now:

```
DECLARE 1 PERSON CONTROLLED(W),
          2 DATE_OF_BIRTH FIXED,
          2 MALE BIT(1),
          2 FATHER POINTER;
```

The effect of this structure declaration is to cause a computer to be prepared to allocate small groups of consecutive storage locations to records with this particular structure of components. In order to bring a ~~structure~~ record into existence, an ALLOCATION statement is used, which normally takes the form:

```
ALLOCATE PERSON;
```

Having created the storage required, the programmer wishes to gain access to the address of its first word so that he can use it later on in the programme. This is achieved by an extended form of the ALLOCATION statement:

```
ALLOCATE PERSON SET (W);
```

This assigns to the variable W the address of the first word of the area of storage allocated to the new record.

Now, to indicate that this new person which has just been created was born in 1937, the appropriate value must be assigned to the relevant component of the record to which W currently points:

```
W -> DATE_OF_BIRTH = 1937;
```

W's date of birth becomes 1937. The = means the same as the Algol := sign, and the arrow symbol (->) is a new symbol which means: take W, look at the record it is pointing to, and then look at the particular component of that record, and put the value 1937 into that component.

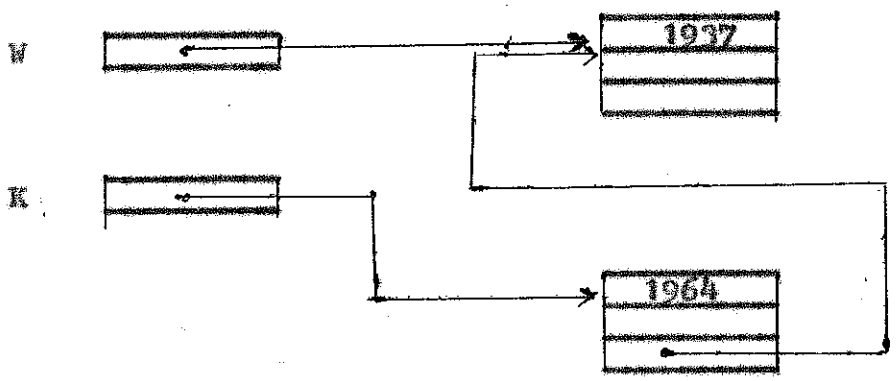
If the programmer wishes to allocate a new person born in, say, 1964, he will write a new allocation statement:

```
ALLOCATE PERSON SET (K);
K -> DATE_OF_BIRTH = 1964;
```

Finally, to indicate that the father of this new person K is, in fact, the old person W, the following statement may be written:

```
K -> FATHER = W;
```

It takes the address out of W and copies it into the FATHER component of the person which is currently pointed to by K. The situation can be pictured thus:

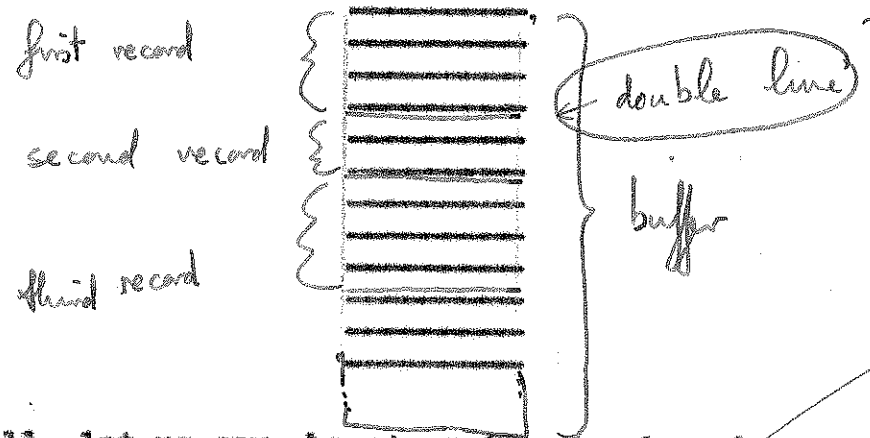


This is a very brief introduction to the list processing feature of PL/I. There are many more facilities provided, for instance, for destroying records as well as creating them, for finding the address of records which have been previously allocated as part of the local work space of the block, and treating them as if they were also dynamically allocated records.

3.3 Files.

One of the interesting applications of the concept of a pointer and a record is that ~~they have been made~~ it forms the basis of the input-output mechanism for records held in files on magnetic tape and discs, the so-called record-oriented input-output. This form of input/output is distinguished from the stream-oriented, which takes in and puts out streams of characters in a form which is legible to a human being; the marks on a magnetic tape or a disc or a drum are not normally accessible ~~to human beings~~ for inspection, and the information is therefore normally transmitted as images of what is held in core store. On a binary machine they will, in fact, always be in binary. There is no point in converting them between binary and decimal if they are merely to be held on magnetic tape or disc storage.

A basic characteristic of transfers between magnetic tape and main store, or between a disc and main store, is that the records being transmitted are significantly smaller than will fit into the natural or economic unit of transfer for the device. In the case of discs there is a fixed sector length, often several hundreds of words long, into which could be fitted a fairly large number of records. Even on magnetic tape, there are very strong economic reasons for blocking records together when they are output onto magnetic tape, rather than outputting them singly and leaving an inter-record gap between every few words. ~~So, in general,~~ Thus, for efficiency or for hardware reasons, input and output on these magnetic devices is usually blocked and buffered. A buffer in the main store consists of a set of storage locations which will, in fact, hold a reasonable number of different records and the actual record boundaries



put the picture here

will, let us say, be at various ~~random places or even~~ regular or possibly irregular ~~regular~~ places throughput the buffer. The buffer forms the physical unit of information transferred between the store and the magnetic medium, although, to the programmer, the record is the logical unit of transfer.

A commonly used method for input of records is provided by a READ statement of the form:

READ FILE (OLD_MASTER) INTO J ;

This bodily copies the "next" record from wherever it happens to be in the buffer into the storage allocated for the structure J. This method has several disadvantages:

(1) For every structure which is to be read there has got to be two areas of storage, one in the buffer and one outside the buffer, which is wasteful of storage.

(2) Every piece of information coming into the computer has got to be copied twice; once into the store and once between the store and the store; and this is wasteful of time.

(3) Even more ~~difficult~~ serious is the fact that one sometimes wishes to mix records of varying structures and size on the same file, and it is impossible to know in advance whether the next record being input

is of appropriate size and structure to fit into J.

~~This technique for reading only works if you happen to know that the record on the file which is actually on the magnetic tape is of exactly the right structure and size to fit into J; and if the records belong to many different structures it is quite difficult to guarantee this in advance.~~ of reading it.

For these reasons, a facility is provided for accessing and processing the records actually within the buffers into which they have been input. To do this the programmer may ask for the address of the record within the buffer, by means of the statement:

READ FILE (OLD_MASTER) SET (P);

where P is a pointer variable to which the address of the record is assigned. Now the programmer can refer to this information by writing expressions of the form:

P -> RATE_OF_INTEREST

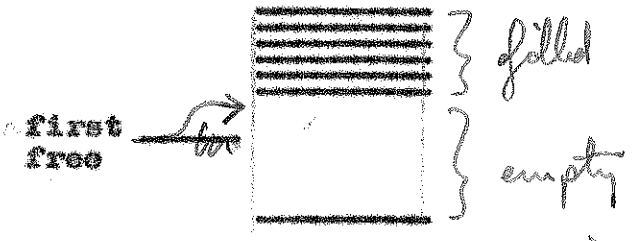
which will refer to the rate of interest component within this record. The programmer, ~~now has the opportunity~~ may use in processing the record, ~~of using~~ the names of components belonging to any structure which he believes the record actually belongs to. He may, for instance, have two different structures, a normal and a special, and he may put a marker as the first component both of normal and of special structures, which indicates whether this particular structure is normal or special. Now, he can use the construction

IF P -> MARKER = 0 THEN ...; ELSE ...;

to determine whether this record belongs to the normal or special structure, so that he can refer to it in the appropriate manner in each case.

N.P.

A similar facility is provided on output; the programmer is given the opportunity of creating new records actually within the buffer from which they are going to be output. An output buffer is at any stage usually only half full:



~~There is therefore a first free location put in the buffer and~~ it is more economical of time and storage space if the programmer actually creates the record to be output at the first free position. He does this by a statement:

LOCATE PERSON FILE (OUT) SET (P);

which sets P equal to the current address of the first free location in the buffer, and moves the first free pointer to just beyond the new record of structure PERSON. If there is insufficinet room in the buffer, the current content of the buffer is output, and the new record is allocated at the beginning of the now empty buffer.

4. Conclusion

This lecture has given an overall survey of a small part of the essential features of the language. It is a language which is still growing. ~~Every six months or so~~ a new manual is produced, which includes, as well as changes to existing facilities, extensions to new areas of application in which it has been found that existing facilities are not adequate. The lecturer must therefore avail himself of the modularity of the language to concentrate on those features of the language which he believes he understands, and cannot claim to know the whole of it.

5.

Edited Questions and Answers.

Q. Does your statement, "P points to Rate of Interest" mean that a field name may appear only in one kind of record, because you have not said what kind of record?

A. The PL/I has a similar qualifying structure to COBOL. If the components of two different structures have the same name, then every occurrence of the name must be qualified by the name of the structure. In fact, in all examples quoted above, it is assumed that no two fields of any two different structures have the same name; or, if they have, then it does not matter, as in the case of marker which occupies the same position in both structures. For example, if the name RATE_OF_INTEREST had been declared in two different structures, it would have had to be qualified by the name of the major structure in which it had been declared, for example:

P -> BANKLOAN. RATE_OF_INTEREST

or P -> REPAYMENT. RATE_OF_INTEREST.

This corresponds to the COBOL feature "RATE-OF-INTEREST IN BANKLOAN" or "RATE-OF-INTEREST OF REPAYMENT".

Q. How do you refer to a record which is not actually in the store at the moment but is somewhere back in the file? For instance, if K points to a record which is no longer in the high speed store?

A. It is not permitted for a pointer to point to a record which is not in main store. Only one record of a file may be in store at a time, and if it is not the one which the programmer wants, he must arrange to input the one he does want. Note also that when a record is either output or input all its pointer components become undefined.

Q. Presumably there are quite a large number of reserved words of one sort or another?

A. There are no reserved words at all in PL/I. Any word can be used by the programmer, and can be either explicitly or implicitly declared to be identical to a basic word of the language. If you happen to place a word of the language like IF in the wrong context, that is, in a place in which it looks to the compiler as though it were expecting an invented word, it will assume that you intended to invent that word, and it will be taken as a fixed point binary variable local to the procedure which you are currently compiling.

Q. You also said that a programmer using a laconic means of representation would be able to establish whether or not a programme had done what he though it had. How can he be sure of that in all circumstances? It seems it might work for a certain set of data but not for another. How would you then know the interpretation which had been given on your behalf was valid for all cases in which you are likely to be dealing?

A. I think that this is a very big problem in the language, and it (illustrates) ~~is one~~ of the dangers of setting out to design a language in this way. I believe that the solution is sought in the excellence of the diagnostic information that comes out in the result of an attempt to do a compilation of the programme.

Q. So it will say what it has assumed?

A. It will, ~~in many cases~~ one hopes, say what it has assumed; but this depends on the characteristics of the implementation, not of the language itself.

Q. There has been a lot of talk about writing supervisors, operating systems, and compilers in PL/I. Do you think this is going to be a successful approach in the interests of machine efficiency?

A. I would reserve judgement on that. I think it is a bit early to say. I would say this though: that as a documentation medium it does provide a standard way of talking about programmes and languages which is more powerful and has a wider range of applications than most existing terminologies. For example, a technical term "tasking" could very well be part of a common jargon which is familiar to programmers from a very wide range of disciplines. The situation at the moment is that scientific programmers cannot properly even talk to commercial programmers, and neither of them can talk to real time programmers, simply because the terms in which they think about their problems is totally different. They have no common jargon. I think that PL/I, even if not universally accepted as a common language to write in, is at least a good common language to talk about.

Q. If it aims to replace the assembly language, then supposing you are starting with a new machine, does this imply you can write the PL/I compiler in PL/I itself.

A. I think that this can be done, probably more easily than in the case of Algol and Fortran. However, the targets for the language are so ambitious that a very great deal must depend on seeing how the implementation in practice works.

One of the features which I think is likely to be critical in the writing of software in PL/I as a practical method of producing machine code software is the expansion ratio of the compiled programme to source code, which I think may prove to be unacceptable for anything but interim use.

Q. Does PL/I in fact allow machine instructions to be inserted in it?

A. I assume that the implementation permits mixture of independently compiled programs in machine code and PL/I.

Q. Is it not another significant consideration that if you say, "Well for my new machine I want to write my supervisor in PL/I" you have first of all to write a compiler for PL/I in the machine language.

A. Not at all, because a bootstrapping technique can be used in future, depending on the compilers which have already been produced by IBM.

Q. PL/I, as you have said, is obviously a very large language, perhaps beyond the ability of a single programmer to fully comprehend. This may make it rather difficult to use successfully, and may also have its effect on the task of implementation.

A. I think there is an element of truth in that. I think that Algol 60 has been a lot more selective in its design. It tends *mainly* (only) to deal with the fairly easy problems and ignore the difficult ones; it makes recommendations only where it is reasonably certain that these are the right recommendations to make. PL/I on the other hand, has been developed over a very short period of time with very ambitious aims and the designers have had to

put a lot of things in it which they have had subsequently to rethink quite considerably. They are much more interested in getting the language really comprehensive than in making sure that in every detail it is tidy and elegant.

Q. Would you agree that whereas PL/I can take Fortran more or less as a subset, there are things that you can do in Algol such as calling by name and the full implications of it which you cannot do in PL/I?

A. Yes, you cannot call by name in PL/I. I think that is almost the only widely known language feature which has not been put into PL/I.

Q. Does PL/I also have mutual recursive procedures?

A. PL/I has a RECURSIVE attribute which must be applied to procedures which are going to be called recursively.

Q. Are there facilities for writing procedures with an indefinite number of arguments?

A. No.

Q. I think the answer to that is that there is a way in which you can do this with a generic attribute.

~~A. Yes.~~

Q. For instance, you can have something like a square root function which can have single precision or double precision arguments, or say, an exponential routine which can have a complex parameter as an argument.

A. Or even two of them. You could, for example, specify an arctan which has either one or two parameters, or even three, if you want. It would be very cumbersome, though, to attempt to specify, for instance,

a READ procedure which has a fully variable number up to ~~then~~ say a hundred. Therefore, in effect, it is not possible to specify a procedure with an indefinite number of parameters.

Q. If you have got parallel processes and processes which are going to be used in real time, perhaps by several different programmers, can you automatically choose a language subset from PL/I which will ensure the code you generate can be used by two programmers simultaneously?

A. You would not have to choose a subset for that. There is such a facility in the full language.

Q. To what extent is garbage collection automatic in this language?

A. There is no automatic garbage collection. It can not be implemented within the current design. For any given pointer variable, you don't know what kind of thing it is pointing to. You don't know whether it is two words long or seven words long. Therefore you could even be pointing to the whole of the rest of the store. This makes automatic garbage collection impossible. But there is a technique which you can use to write your own garbage collector. If, when you ~~do~~ attempt an ~~ALLOCATE~~ and there is not enough room for it, you get an Interrupt which can be directed into a routine which you have written yourself to find records which have become "inaccessible".

References.

[1] I.B.M. Publication
I.B.M. Operating System/360
PL/I Language Specifications
Form C28-6571-2 (now superseded by
C28-6571-3)

[2] M. Woodger. (earlier lecture in the series).