C
p1, 4, 8, 9, 10, 12, 14, 21

Very chatty, but effective - it touches all the important points.

Transcript of a talk:

PL/I AS A TOOL FOR SYSTEM PROGRAMMING

by F. J. Corbató

Presented at a PL/I Seminar for the Air Force Systems Command at Hanscom Field, Bedford, Massachusetts, March 5, 1968 for inclusion in the summary report of the symposium.

Revised Draft

3/27/68

PL/I As A Tool For System Programming

F. J. Corbató

First I have to give some background, because I don't think you can discuss or evaluate a language like PL/I unless you know the background of the speaker. To some extent, PL/I is like getting too close to an elephant. All you can see is the pores and what you see depends on which side of the elephant you're on. For present purposes, I have the advantage of not being a language expert. Instead, my vantage point is that of a system designer-implementer concerned with the overall system performance and the degree that the system meets the goals that it was designed for. This gives me a little more detachment from the issue of whether the language is just right or not. For that reason some of my remarks will not be completely unequivocal but rather will be shaded. The basis of the PL/I experience that I wish to talk about is mostly on the Multics project, which is a cooperative project being done with the Bell Laboratories, the General Electric Company and Project MAC of M.I.T. using the GE 645 computer which is derived from the GE 635. However, I am not giving an official Multics view but rather only my own opinion as a member of the design team. In fact, it's a preliminary view because things are too confused at this point to really be certain that we have analyzed what is happening. (A bit like asking for comment on a battle while it is still in progress; it's too early to know all the answers.) Further, one has to be cautious in forming final judgments on a language even though it is already a de facto standard, since there still is a need for a great deal of diversity in the computing field so that different techniques can be evaluated.

To understand the context in which our systems programming was done, I first have to give you a brief review of what the Multics project's goals are. A set of papers are in the 1965 Fall Joint Computer Conference Proceedings if you wish more detail. Briefly, we are trying to create a computer service utility. In particular, we want continuous operation of pools of identical units. We want to combine in a single complex the goals of interactive time sharing and noninteractive batch processing. We want to combine the goals of remote and local use in one system. The system programming problem is to develop a framework which multiplexes all this equipment at once and yet also allows controlled interaction and sharing between users working in concert on various problems in real time. In short, it's a fairly ambitious project, not because of any single idea but because we're trying to tie together all these ideas at once. It was our judgment that we required new hardware to meet these goals squarely and that, of course, meant that we had to write almost all of our software for ourselves, including, it turned out, even the assembler. We were able to borrow a little, but not as much as we had hoped. Thus, the project began basically at a research and development level, where flexibility is needed. In our view we wanted a small team of people, because the hardest thing to do when you're groping in an unknown area is to coordinate people. We felt strongly that we had to have maximum flexibility in our implementation.

To give you a little bit of the scale of the project, I will discuss briefly the implementation. The project began in earnest in the fall of 1964 and should develop a usable pilot system about the end of this year. This means it will take approximately four years to create a useful system. That's a long time, and I think one has to appreciate the investment of effort that goes into such a venture. If you spend four years developing something, you probably try to exploit it for a period of time greater than that; thus, there is clearly an underlying goal here of wanting to see the project evolve as conditions change. The system itself is described quite tersely at the level suitable for a senior system programmer in about 4,000 single-space typewritten pages in the Multics System-Programmers' Manual. The system in final form seems to project out to about eight hundred to one thousand modules of maybe four pages of source code each on the average, or in other words, between three to four thousand pages of source code. (It's interesting that the amount of description approximates the amount of code, but they're of course written at different levels.) The amount of system program that's in machine code is less than 10% at the source code level; it would be even less except that the compiler did not come along early enough, so some things had to be written in machine code right away. The system projects out to between one and one and a half million 36-bit words which loosely is the supervisor program. In operation, most of it, of course, pages out and is not resident in core, but it is expected to be there, and it is exclusive of all the languages and facilities of that sort, such as COBOL, FORTRAN, and even PL/I itself. The manpower to create the system has ranged from approximately zero to 50 people over a four-year period with roughly an increasing number as time went on. When I say zero to 50 people, I mean effective persons who are involved and working full time. That isn't much for the size of the job that I've described, and there is clearly the need to have maximum leverage at the fingertips of each person.

The next question I want to address myself to is why one uses a compiler at all to do system programming. (I'll take up next the question of why PL/I in particular, but first: why a compiler?) First, there is the ability to describe programs briefly and lucidly. One can clearly obfuscate one's ideas with a compiler language but it's harder. To some extent one is talking about what one wants to do rather than how one wants to do it. The trouble with machine code, of course, is that when you look at a random section of machine code you don't know what properties of the instructions the programmer really wanted to exploit. On the 7094, for example, the fact that the P-bit got cleared by an instruction may or may not be germane to what the program is trying to accomplish. With a compiler language, especially the later ones, one tends to describe what one wants to accomplish in terms of a goal and let the compiler work out the specific detail. This contributes to lucidity, of course. It also gives one the chance for change and redesign, because on a system as large as the one I have just described, the only sensible attitude is to assume that the system is never finished. Although the system obviously goes through phases, one is continually improving and evolving it. We have had this experience on CTSS, our previous time-sharing system and we know it is true. What happens is that users keep having expanding needs and goals as they exploit the facilities and they continually come up with wanted improvements. Certainly, the other extreme -- of assuming that a computer software system is like hardware and can be designed once and for all on a one-shot basis and then left to the hands of some maintainers -- I think has been shown to be a failure.

Another issue, too, in a system of the ambition that we are talking about, the software is at least three-quarters of the design work and yet it usually doesn't get started until the hardware is already firm. Thus, there is a desire to speed up the implementation effort and using a compiler allows each programmer to do more per day. It's our experience that it doesn't matter too much if one is dealing with assembly language or compiler language; the number of debugged lines of source code per day is similar. Another point, too, is that the supervisor is the host of the user services so that the computer time spent in the supervisor is between 10% in some well worked out systems to maybe an extreme of 50% of the total time. Thus, the possibility that the compiler isn't generating the most efficient code isn't a disaster. In other words, one is dealing with code that isn't being exercised <u>all</u> the time. It has to be there, it has to be right, but there is room for some clumsiness. Further, if the system is well designed, the production job will run efficiently and the supervisor will remain out of the picture.

Finally, there is the issue of technical management of programming projects: the problem of trying to maintain a system in the face of personnel turnover and in the face of varying standards of documentation. Personnel turnover is expected on a four-year project. (We didn't think it was a four-year project to begin with; we estimated two.) One has to assume in most organizations somewhere between ten and twenty per cent turnover per year even if everybody is relatively happy. People get married, husbands are transferred, and for a variety of personal reasons, people must leave, carrying with them key knowhow. Training a new person involves a minimum period of six to nine months, even starting with good people, especially if you're faced with a system which has 4,000 pages of description in it. You don't casually sit down and read that, even in a weekend. In fact, it's fair to say that the system is large enough that no single person can remain abreast of all parts at once. Thus, there is a reasonable case for a compiler in developing large systems.

In developing CTSS we used the MAD compiler slightly and it was quite effective. The only problem was that we were cramped for core memory space for the supervisor. The compiler generated object code was somewhat bulkier than hand code, and this was unfortunately a burden we couldn't carry too well, but where we used it, it was very effective.

So the question was: What compiler to use when developing Multics? We chose PL/I. The reasons go somewhat like this. One of the key reasons that we picked the language was the fact that the object code is modular, that is one can compile each subsection of the final program separately, clean up the syntax, and test it on an individual basis. This seems obvious, perhaps because it's in several languages, like JOVIAL, FORTRAN, or MAD, but it isn't in some of the ALGOL implementations and it blocked us from considering the ALGOL implementation we had available. The second reason for picking PL/I was the richness of the constructs, especially the data structures and data types which we considered to be very powerful and important features. We had an unknown task on our hands with fairly strong requirements. We viewed the richness as a mixed blessing, however, because we certainly were a little wary of the possible consequences. But it certainly seemed the right direction to start and maybe to err on and to cut back. As I'll get to later, it was a little too rich. But I'll come back to that. A fifth reason for choosing PL/I was that it was approximately machine independent. Our object in doing the system has not been to compete with normal manufacturing. Instead, our object has been to explore the frontier and see how to put together effectively a system that reaches and satisfies the goals that were set out. We are trying to find out the key design ideas and communicate these to others regardless of what system they are familiar with. Hence, a language that gets above the specific details of the hardware is certainly desirable, and PL/I does a very effective job of that. In other words, it forces one to design, not to bit-twiddle. And, this has turned out to be one of its strong points.

Another reason that we considered PL/I was that we thought the language would have wide support. To date it has had the support of one major manufacturer. And, the final key reason for PL/I was that two persons associated with the project, specifically Doug McIlroy and Robert Morris at Bell Labs, offered to make it work on a subset basis; they also offered to try to arrange for a follow-on contract with a vendor for a more polished version of the compiler. That is basically why we chose PL/I. We certainly debated, somewhat casually, other choices but these were the essential reasons why we picked the language.

The subset that was implemented initially as a quick-and-dirty job was called EPL for Early PL/I. Its design characteristics went briefly as follows. It had no I/O; after all, this is a system programming language and we use the system subroutines. It had no macros except the INCLUDE macro, which worked in very smoothly with the time-sharing system, CTSS, that we were using. It had no PICTURE attributes or things of that sort which represented the COBOL influence, except for structures, of course. It had no multi-tasking; we found this to be a defective idea in the sense that it wasn't thought through well enough, and we certainly didn't need it for a system programming language. It had various minor restrictions like requiring structure names to be fully qualified. No complex arithmetic, no controlled storage (you can simulate that easily), and, more importantly, no attributes such as IRREDUCIBLE, REDUCIBLE, ABNORMAL, NORMAL, USES, or SETS -- those things which allow the compiler to do an optimum job of compiling the code with advice from the program; these are sophisticated and tricky attributes, incidentally -- but the reason they're not there is that the compiler didn't intend to optimize anyway, so it would have ignored advice.

To emphasize the positive, the things that EPL did have were ON-conditions and signals; it did have recursive procedures -- in fact, the system doesn't allow any other kind easily; (if you want to work at it, you can program a nonrecursive procedure). It did have based storage and pointer variables, and it had ALLOCATE and FREE. It had structures, as I've mentioned, it had block structure, and it had varying strings, which we regret to some extent because of implementation difficulties. In other words, it was a pretty potent subset from the point of view of language facilities.

The implementation, as I said earlier, was deliberately a quick-and-dirty job. It was expected to be merely a temporary tool to be soon replaced by a polished compiler from the vendor. The team consisted of McIlroy and Morris and two to four helpers. I am going to give a detailed and candid account of the events surrounding the EPL implementation because the nature of the events together with the very high qualifications of the people involved points out clearly that the difficulties encountered were quite unusual. The original optimistic estimate for making EPL work was that it was only going to take them about six months. In spite of the dedication of the people involved, it took them over 15 or 16 months to get a compiler that was barely usable. A lot of work has gone into upgrading it in the last 18 months, since the polished compiler of the vendor never materialized and the upgrading process has not yet ended. Moreover, the EPL effort like a gruelling relay race has worn out nearly everyone who has worked on it. But to everyone's credit, the compiler works and is useful.

The language that was used to implement EPL was TMG, short for "transmogrifier", which is a language system developed elsewhere by Bob McClure. It's a clever, interpretive system specifically designed for experimental language writing or syntax analysis. However, it is not easy to learn and use and, therefore, it is hard to pick up the work of somebody else written in the language. The EPL translator was initially designed as two passes, the first one being principally a syntax analyzer and the second one basically a macro expander. The output of the second pass in turn led into an assembler which handled the specific formatting for the machine. Later a third pass was added intermediate between the first two in an attempt to optimize the object code. The quick-and-dirtiness came through when the original language subset specs had only a single diagnostic, namely, ERROR. That has been expanded so that maybe now there are half a dozen, but the only help you get is that the message appears in the neighborhood of the statement that caused the trouble. The compile rate, which was never a major issue, turned out to be a few statements per second. It has been improved a little with time, but more critically the object code that is generated has improved to a respectable ten instructions per executable statement. (There's obviously a large variance attached to these figures.)

The environment that the EPL compiler had to fit into is significant. First of all, we had adopted as a machine standard the full ASCII character set of 95 graphics plus control characters, so one of our first projects was trying to map a relationship with EBCDIC -- the IBM standard. We also intended to use the language in a machine with program segmentation hardware in which programs can refer to other sections of programs by name. Fortunately, we could use the $ sign as a delimiter to allow us to have two-component names. We also expected the compiler to generate pure procedure code which was capable of being shared by several users each with their own data section who might be simultaneously trying to execute the same procedure. We also wanted to establish as a normal standard, although not a required one, the use of recursive procedures by means of a stack for the call, save, and return sequence linkage information and automatic temporary storage. We also wanted to allow the machine to have a feature which we've called "dynamic loading" in the sense that an entire program isn't loaded per se; the first procedure is started and as it calls on other procedures, these procedures in turn are automatically fetched by the supervisor on an as-needed basis rather than on a pre-request basis. This, of course, is in conflict with any language which allows storage to be pre-declared by the INITIAL specification within any possible module that is ever used by the program. (This problem also comes up in FORTRAN.) ? fortran IV ?
We also had a feature in the machine which we called segment addressing, which is such that when you want to talk about a data segment you don't have to read it in through input/output; rather, you merely reference it and the supervisor gets it for you through the file system. In other words, we were trying to design a host system capable of supporting software

constructs which make it easier for people to write software subsystems. In this rather sophisticated environment, one of the problems was that much of the time was spent finishing the design of the compiler so as to implement the mating of the language constructs with the environment. The things that caused trouble were the SIGNAL and ON conditions, which are relatively tricky ideas and which clash head on with faults and interrupts. The call, save, and return conventions had to be mated into the standards of the system. Problems of non-local GO TO's and the releasing of temporary storage which has been invoked had to be licked. Most of these problems are implications of the language if one thinks it through, for the language has a lot of assumptions in it about what kind of an environment it is going to be in. There are also little subtleties, like when you're talking about strings of characters and operators, what is the role of control characters, i.e., codes without graphic representation such as backspace, when encountered in strings. There are also obvious difficulties in that the language doesn't discuss any protection mechanisms, a feature that every system must have to implement a supervisor-user relationship. Thus, there needed to be some additional modifications made to the compiler to make that work out. And then there are strategy problems within the implementation, such as how you're going to implement internal blocks and internal functions. These also took some time to work out and were one of the principal reasons why the compiler implementation was slow going. Further, it was done simultaneously and in parallel with the system design. I would say with hindsight that we didn't put enough effort into trying to coordinate the two. The reason we did not was that to a first approximation we felt that the language was a decoupleable project.

That was a useful thing in the early days, but as we came home toward the finish line in the design, it began to haunt us that we hadn't worked out some of these interface ideas more carefully, and we had to pay the price of redesign in various parts.

One preliminary conclusion we draw from the above experience is that PL/I went too far in specifying the exact environment. There are a lot of ideas that should be subroutines and not part of the language. I don't mean they shouldn't be thought through, but to think them through is not the same as putting them in the syntax of the compiler. In particular, things like SIGNAL and ON conditions could indeed be implemented as subroutine calls and be part of the environment of the host system. I don't think they belong in the language per se, although if one makes the language embrace a standard subroutine library, then I, of course, agree.

I'll say very little about the vendor's compiler. They estimated it would take 12 to 18 months. After approximately 24 months, we stopped expecting anything useful to appear. One of the principal reasons they failed was that there was a gross underestimation of the work, by a factor of three to five, and it was impossible to mount a larger effort by the time the underestimation became evident. Thus, the pioneering EPL has become the standard system-programming compiler.

Let me next talk about the use we made of the PL/I language. A strong point, we felt, is the ability to use long names which were more descriptive. People still get cryptic, but they're not nearly as cryptic as they were. The full ASCII character set is a strong point because we wanted to deal with a well engineered human interface. The structures and the data types, as I mentioned earlier, we consider to be one of the strongest assets (this perhaps comes as no surprise to COBOL users but this feature is very important when you're trying to design data bases). The POINTER variable and based storage concept, along with ALLOCATE and FREE, have been pivotal and crucial and have been used extensively. Some of the features like SIGNAL and ON conditions, which have cost us a lot of grief, at least in principle have been very graceful ways of smoothly and uniformly handling the overflow conditions and the like, which suddenly trap you down into the guts of the supervisor. In previous systems we have always had the quandary of how to allow the user to supply his own condition handlers in a convenient way. We're not sure that the price is perhaps too high, but the mechanism does look good. The SIGNAL mechanism also is an elegant way of handling error messages. One of the problems with an error, when it is detected at a given subroutine level, is that it's significance isn't always understood. For example, the square root routine may encounter a negative argument but only the subroutine that called it knows the significance. Maybe it means that the cubic equation solver has three roots that are not all real, but that again isn't the true significance. In fact, it may mean that the experimental data that was being analyzed was merely noisy and incorrect and that this data point should be abandoned. The signal mechanism allows each subroutine in the line to insert a message if it is wanted and allows the square root routine, that didn't know who called it, let control go back in the right way.

Finally, a last point about PL/I that is perhaps obvious, is that the conditional statements that are straight out of ALGOL are very valuable.

Overall, the general result that we got from using PL/I was a rather small number of programming errors (after a programmer learns the ropes), in fact, a sufficiently small number that one of our major sources of trouble is that a lot of bugs have been caused by mismatched declarations, getting parameters in a calling sequence inverted, getting argument types in calls mixed up, all clerical errors in which the language gives you no help and our implementation doesn't either. In fact, this is a defect in the language in the sense that the independence of the separate compilations has left a gap in the checking of types. (Sometimes programmers have used mismatched declarations for gimmicky convenience or efficiency although we have tried to avoid it because it obviously destroys machine independence.) We also found that skillful system programmers who know the machine well don't want to work in machine language because they make too many mistakes. This condition is aggravated because in modifying the machine we retrofitted a lot of involved ideas onto a somewhat ornate order code. Regardless of the reason, however, we find that programmers would rather get things done than twiddle bits.

Another major effect of the use of PL/I has been that so far we have been able to make three major strategy changes which are really vast redesigns. One of them in the management of the high-speed drum that did most of the paging. It was reworked, quite a while ago, when some insight developed which allowed a tremendous amount of bookkeeping to be eliminated. The amount of code that was involved dropped from 50,000 words to 10,000 words. This total rework was done in less than a month (although not completely checked out because the person wasn't working full time on it). A second redesign occurred in the area of a special high-strung I/O controller which has all kinds of conventions and specialized aspects. The first cut of the control program design was a little rich; it ended up involving around 65,000 words of code. After people finished debugging it and recovered their breath, they took a closer look at it and saw that by cutting out maybe 10% of the features and changing some of the interfaces and specifications they could streamline it. Two good men working very hard did the reworking in less than two months. The two months were peak effort, but they did do it. The program basically shrunk in half, down to 30,000 words and it runs about five or ten times faster in key places. This kind of redesign is invaluable. It gives one the mobility that one is after. It may be true for the use of nearly any compiler -- I'm not trying to argue that this is exclusively a PL/I attribute -- but this is the experience we're getting. Finally, we made another major change in the system strategy of handling own data sections, which we call linkage sections. We were keeping them as individual segments but we reorganized things so that they were all combined into a single segment because some of our initial design assumptions had not been correct. This reworking was done in the period of a month. The change was serial to the main line of the project development, so that it was a rather important period of time to minimize.

Now, there's another side of the coin, namely, object code performance. This aspect is illustrated in Figure 1. Remember, too, these figures represent only a preliminary view.

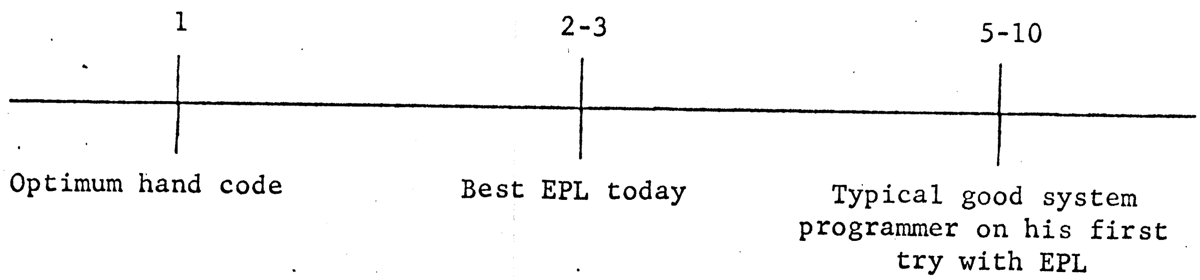

Figure 1

Figure 1 illustrates the object code execution time in the horizontal direction. (It could also be object code space since roughly speaking they are similar.) The unit 1 represents optimum machine code or hand code, where one uses all the features of the machine but stays within the specifications. The next point on the figure represents the best results obtained to date by careful juggling and tuning of EPL written programs. At the moment the comparison to hand code seems in the vicinity of 2 to 3 times worse and is largely because the compiler cannot optimize very well. A lot of redundant expressions are being calculated; this is especially true with based storage and pointers where it is easy to build up fairly elaborate expressions to access a variable and then at the next occurence repeat the calculation. Finally, and this is perhaps the one shocking note that should be taken with some caution, we find that a typical good system programmer produces on his first try, EPL generated object code which is perhaps 5 to 10 times as poor as hand code. I think this is the main problem with PL/I, because a factor of 5 or 10 at the wrong places can sink the system. The reason for the factor of 5 or 10 seems to be principally that programmers don't always realize the mechanisms they are triggering off when they write something down. The usual pattern when one writes a program, is to think of four or five ways that one can write out a part of an algorithm and to pick one of them on the basis of knowing which way works out best. What has happened is that people are too detached. For example, if you use a 1-bit string for a Boolean variable, it turns out in our particular implementation you generate a lot more machinery than if you'd used a fixed integer. Similarly, varying strings carry a fairly stiff price tag in our present implementation, (although ways are known to improve matters a little), and they must be used with caution. Occasionally too we've had mishaps where the machine independence works against us in the sense that a man declares an array of repeating 37-bit elements and the compiler dutifully does it, straddling work boundaries mercilessly. The best we've been able to do so far is to get the compiler to at least remark on the object code listing the word "IDIOTIC." There may be other reasons for the factor of 5 to 10 such as the language learning time but we do not consider them important. Other issues such as the 10 to 1 variation in ability among programmers of similar experience discussed in an article by Sackman, Erickson and Grant in the January 1968 Communications, I think can be discounted in our case. Our technical management has been thin, but we have kept careful track of the individual programmers so that mismatches with work assignments have been minimized.

With regard to remedial measures for upgrading the system one must remember that most of the code in the supervisor doesn't matter; it's not being used most of the time so the key thing is program strategy. I don't have time to discuss here how we localize the parts of the code which are the functionally important parts, but a segmented machine pays off handsomely. Meanwhile we are learning tradeoffs between the different supervisor mechanisms. In addition we are trying to develop checklists of things to avoid in the language. It turns out to be rather hard to get people to generalize so it is slow going. On a long-range basis GE is developing plans for an optimizing compiler, but it isn't going to help us right away. We are also studying on a preliminary basis smaller subsets of PL/I with perhaps modifications and changes to the language so that the implementation is more uniformly potent--or impotent, depending on how you look at it. That is the user would be constrained to a language which would implement well regardless of whether he takes one choice or another. And, finally, there are some coding tricks that might have helped if we had thought of them sooner.

One of the key problems in our use of PL/I has been that the programmer doesn't have feedback. If he had, say, a time and space estimate on each statement that he writes, given back by the compiler, and if he were in an interactive environment developing the program (i.e., he could get quick return on his compilations), he might be able to form some intuition about what he's doing. To implement such estimates is not a trivial problem, because a lot of the mechanisms that are invoked are shared, so that there needs to be a way of designating the shared mechanisms and showing why they are included.

Finally as a last resort in improving supervisor performance we can always go to machine language on any supercritical module. But this isn't a panacea, because it is easy to be swamped if one tries to put too much in machine language and moreover one has lost mobility. (Going to machine language should be compared to parachuting out of an airplane.)

I have a few general conclusions. I think that in the language area there has considerable leap-frogging. FORTRAN was the first compiler with any widespread use and it suffered because it wasn't systematic to implement and was somewhat clumsy to use. It was however a practical language. ALGOL was in a sense a reaction, but it suffered because it left out the environment and didn't come to grips very squarely with the implementation. PL/I in effect is a reaction against ALGOL's not having considered the environment, but it suffers from being designed without well-formed plans for a systematic implementation. The notion of "systematic" is important because without it the cost of implementation, the speed of the compiler and the quality of the object code may be off by factors of ten or a hundred. Nevertheless, I admire the PL/I design effort and consider it valuable because it has inspired language experts to try harder; in effect it has set as goals what is wanted. The fact that the language has not been implemented well by anyone, I consider to be an object lesson. Neverthe less techniques for mastering the problems are being found. In addition people are beginning to think of ways of accomplishing the same functional characteristics without the same internal problems. One of the ways is to try to minimize the language syntax and to think through more carefully what is the subroutine library.

Future languages will come and they'll be beneficial, too. But PL/I is here now and the alternatives are still untested. Furthermore, I think it is clear that our EPL implementation is going to squeak by, and in the long run the Multics project will be ahead because of having used it rather than one of the older languages. Now, finally, the last question, which I think is a tough one: If we had to do it all over again would we have done the same thing? I'm not totally sure of my answer; I just don't know. We certainly would have designed the language more carefully as part of the system; that was something we didn't pay enough attention to. If it was EPL or a PL/I again we would have tried to strip it down more. With hindsight we would have modified it to some extent to make sure that it could have been implemented well. If it were another language we would have tried to beef it up with things such as are in PL/I, and maybe modified it. Either course of action takes a lot of design time, and that's the dilemma: in effect, one wants one's cake and one wants to eat it, too. I think the decision probably hinges on whether or not one is trying to meet a deadline. I would probably use FORTRAN to meet a firm deadline. But if I'm trying to solve a problem with a future, I think I would use either PL/I or its functional equivalent--and the choice will have to be answered in the future.

Questions and Answers

Q: You talked about redesign as being a major part of your development, where you got a big payoff.....Could you comment on this . . . . . maybe your opinion on how well you should design to start with and how much emphasis you should place on redesign?

A: We consider our design documents in absolute terms to be mediocre, but in relative terms to be good compared to other programming efforts we know about. We worked hard on this, and we did it for a reason: one of our principal goals was to be understood. It has also saved the project over and over again because people can see what is going on, and new people can join the project. In general these design documents just weren't written and accepted; they were usually bounced around and the first proposal wasn't usually the final one. There was a deliberate self-criticism in the design with a goal being sought of functional isolation of different ideas. We felt strongly that one should try to design as carefully as possible before writing programs. Debugging incorrect ideas is a very expensive waste of both time and resources. Nevertheless it is almost impossible today to correctly foresee all the implications of a large system design. Thus some redesign is inevitable and it is here that the compiler language speeds the process. In particular: 1) you don't get distracted by bit-twiddling, and 2) you can see what you're doing and don't get lost in a sea of details. That is it allows you to keep focused on what you're trying to accomplish rather than getting caught in tedium.

Q: I have a question to ask about your comments regarding systematic implementation. Do you feel that as PL/I is implemented by a variety of manufacturers, the implementation will effectively reduce the effectiveness of standardization?

A: Are you worried about the question of whether the different manufacturers will create different languages--is that what you're saying?

Q: Well, I'm suggesting that when the language is implemented by someone that his implementation will vary sufficiently from someone else's implementation to cause one to lose the advantages of having standard syntax.

A: I don't think so. Even FORTRANs aren't the same in most machines. The minute you change the word length underneath FORTRAN you normally change many of the subtleties, including precision and the like. I think the best one ever ought to expect out of the present view of so-called language standards is a kind of a first cut at having an approximate version of the program that will work on another machine. You still have to audit and edit accordingly, and this is, I think, the best we could hope for out of PL/I. However in the case of PL/I I would expect there to be a larger class of programs for which one would be indifferent to the subtle variations and only worry about cases of drastic differences in behavior.

Q: Does that apply also to, say, a program's dependency on an operating system?

A: I think that in this area the descrepancies between PL/I implementations will be large. If you get totally wrapped up with an operating system which is peculiar to a particular set of hardware you're trapped. You're just going to have to rework it on another machine. It is a long-range goal, as yet unachieved, to minimize this problem. We've considered the problem some because one of the obvious questions that arises when you have a system which has been largely implemented in PL/I is: Could you put it on another machine? The answer is "Yes," although I think it's still at the level of a "technical challenge" even with similar hardware. To do it one would have to go through and modify and edit all the programs to make it come out just right. There are also some strategy changes required probably unless one actually built an identical machine. Nevertheless such a task is still preferable to having to start all over, writing off as a total loss the ideas and efforts of several years; this is the alternative to working in a compiler language for system programming. I don't really see how there's going to be any progress in the field until we stop killing off our system children.

Q: In light of the fact that you bemoan the lack of systemization and I get the impression perhaps it's not quite time to standardize and so forth with PL/I as it is, at the same time it strikes me from your comments that you probably, you in your project, know something more about machine characteristics that are suitable for or unsuitable for a language like PL/I if not PL/I itself. Do you have any comments to make there? That is, rather than carrying PL/I to another machine which is again incompatible, what about designing a machine which would make PL/I a useful tool, a more useful tool?

A: Well, I think it's useful now. I don't think one has to apologize for the language.

Q: I mean <u>more</u> useful, I mean <u>more</u> ideal.

A: The fact that we've gotten down to a level of only two or three times clumsier than hand code is perfectly good enough for many applications. The problem at the moment is that you can stray off the path; it's like skiing down a mountain and going off the trail into the woods unexpectedly. Some of the problems are intrinsic complications of the language, and to some extent it has to be streamlined to do a much better job. At present this issue exceeds in effect any hardware improvements to favor PL/I. As far as being accepted as an industry standard, I guess I'm a little more laissez-faire about this than most people. My own reaction is that one can judge for himself when one has a de facto standard, and treat it accordingly. It already is a de facto standard of some sort, and there will be future language standards. But they obviously will have to compete with PL/I and show that they do, at least in some sense, a better job.

Q: You indicated that the project started out as a two-year plan. I was wondering whether you'd comment on whether much of your schedule was perhaps influenced by hardware problems in addition, say, to the compiler implementation problems? What were your principal problems? And, talking about doing it over again, do you expect that you could, looking back and starting fresh and assuming no uncontrollable factors, have done it in the two-year period? Is it just this typical problem of underestimating the complexity of developing a major software system?

A: I don't think we're embarrased that it went from two to four years; that's sort of par for the course for a research project, and that's what it turned out ot be. If one really wanted to predict performance or schedules one would have to do something one has already done before. That's just what we didn't want to do. If we wanted to meet a two-year deadline, we would have had to say, "Imitate CTSS. Copy it slavishly." If we had done this though, we wouldn't have increased our understanding of computer utilities and we would have propagated many system design limitations. I think one really has to face up to the fact that if you're going to try something new whether it be a language or a system, you had better leave yourself some slack. A factor of two is pretty routine on research programming projects. We were facing three major problems all at once: a new language, new hardware, and new operating system, not to mention the fact that we had three organizations involved in three geographical locations.