The Case for PL/I as the Language for Instruction in Programming

Richard Conway

TR 83-571
August 1983
(Revised January 1984)

Department of Computer Science
Cornell University
Ithaca, New York 14853

The Case for PL/I

as the Language for Instruction in Programming

Richard Conway
Cornell University

Instruction in programming is a large and growing task for most computer science departments. Such courses have the obvious objective of instilling competence in a skill that is becoming essential in many fields. They have an equally important responsibility to foster some understanding of computing in general.

The choice of programming language for use in an introductory programming course is probably less important than the skill of the instructor or the nature of the supporting programming environment. Nevertheless, there are significant differences among candidate languages, and one ought not to handicap a course unnecessarily by a poor choice of language. The catch is, of course, in the definition of "poor", since there is no clear metric of quality in programming languages and these matters are debated with a fervor that would do justice to religion or politics.

It is widely believed that language selection should depend on the expected area of application, and the ability and maturity of the students. Hence, different languages might be appropriate for engineering and business students, or for high school and university students. There is also the consideration of immediacy of use, since there are presumably situations in which students have an immediate need to use a particular language. The objective then is <u>training</u> rather than <u>education</u>, and language selection is not an issue. The counterargument to both immediacy and choice-by-application-area is that, for most students, the introductory programming language is only the first of many they will eventually use.

Finally, there are pragmatic questions of the existence of appropriate processors for the computing equipment that is available to the course. More often than not, the selection of language is constrained by the hardware available, and one rarely has the opportunity to choose the language first and then specify appropriate equipment.

Given all these considerations, only programming language bigots believe that one language could be uniformly preferable to all others under all circumstances. Nevertheless, it could be true that for a wide variety of circumstances one language tends to dominate the other comparable, contemporary candidates. For present purposes I will summarily exclude from the discussion special-purpose languages such as Snobol, Visicalc and SPSS, functional languages such as LISP, "different" languages such as APL and Forth, lower-level languages such as C and Smalltalk, not-yet-widely-known languages such as Modula-2 and Turing, and not-yet-ready languages such as Ada. This leaves the more-or-less general-purpose, high-level, widely-used procedural languages: Fortran, Cobol, Basic, Pascal and PL/I.

In the following paragraphs I argue that for a programming course that makes any pretense to education, a carefully designed subset of PL/I dominates Fortran, Cobol and Basic, and is itself not yet dominated by any other general-purpose language. However, in considering this argument, the reader ought to be warned that I have been involved in the implementation of several PL/I-subset processors and have coauthored a number of texts based on PL/I.

A Brief History of PL/I

In 1963 a joint "Advanced Language Development Committee" was organized by the SHARE Fortran project. Although the initial intent was a six-month study to specify extensions to Fortran -- presumably to borrow structured data and I/O facilities from Cobol and block structure from Algol -- compatability with Fortran was soon abandoned and the Committee undertook to define a new language, initially called NPL. After several revisions, their December 1964 report described PL/I, more or less as it is known today. IBM released the "F-level" compiler for PL/I in mid-1966, which represented the language standard for many years.

PL/I's designers bravely intended the new language to serve the Fortran, Cobol and Algol communities. They even anticipated that systems programming could be done in such a high-level language. Although the designers sought machine independence in most respects, they also tried to make certain machine and operating system facilities (of the then-new IBM 360) accessible to programmers in a high-level language.

Usage of the new language grew steadily (at IBM installations), although it obviously never fulfilled its designers' objectives of replacing Fortran and Cobol. Not until the PL/I "Optimizer" appeared in the early seventies did potential scientific users have a PL/I implementation that was competitive with Fortran processors. Similarly, universities were initially precluded from teaching PL/I by its high processing cost, relative to the efficient instructional processors for Fortran, but this problem was alleviated by the appearance of PL/C (1971), a processor for a PL/I-subset that matched the Fortran processors in performance, and provided exceptional diagnostic assistance.

The rapid growth of instructional use of PL/I in the 1970's accompanied the emergence of "structured programming" as the dominant philosophy of programming instruction. Although the academic computer science community never much admired PL/I, it was obviously a preferable vehicle to Fortran, Cobol or Basic for inculcating the modern approach to program development. One critic called PL/I a "fatal disease" (in contrast to the "infantile disorder" of Fortran), and a another computer scientist, fond of such hyperbole, described PL/I as "more part of the problem than the solution". A much-quoted simile likened PL/I to a Swiss Army knife -- there was a blade for every task, but risk of cutting oneself on the unused blades.

In retrospect, even in the early 1960's and against very severe time pressure, PL/I's designers should have known better than to use the same symbol for assignment and equality. They should also have foreseen that the syntax of PL/I declarations was an invitation to surprise, and that the combination of implicit declaration and nested scope was an

invitation to disaster. Had these and a few other inconsistencies been avoided, or subsequently remedied, the critics would have been deprived of their most attractive targets. They could then have concentrated more energy on the great theological debate as to whether the semi-colon is a separator or a terminator, and the more substantial issue of how to manage type coercion. However, judging from the recent design of Ada, it is still not an easy task to achieve consistency and conceptual frugality, let alone any degree of elegance, in a rich, general-purpose programming language, and by comparison the design of PL/I twenty years ago was a very significant achievement.

The Case Relative to Fortran and Cobol

Fortran and Cobol are the standards of the production programming world today, and will presumably remain so for the foreseeable future. The obvious conclusion might be that one or the other of these, depending on the intended area of application, is the instructional language of choice when immediacy of use is paramount. This must be a persuasive argument, since these two languages still dominate the instructional scene in higher education today.

There are two parts to the counterargument that PL/I or Pascal is preferable to either Fortran or Cobol in most university contexts today. The first is that the language used for introductory instruction is not the only programming language the student will encounter, and it is important to make the student receptive to subsequent languages as well as the first. Even Fortran and Cobol are evolving, and eventually these languages will include facilities comparable to those already present in PL/I. (A corollary is that introducing students to more modern languages might help accelerate the glacial pace of evolution of Fortran and Cobol.)

The second point is that a programmer initially educated using PL/I or Pascal is likely to be a significantly better Fortran or Cobol programmer than one who started out in that language. It is possible to write clear, modular, well-structured programs in Fortran or Cobol, avoiding some constructions that are available and simulating some that are missing. However, this requires that the programmer have a good understanding of the concepts involved. The point is that concepts, rather than syntactic details, are the key content of a programming course, so one should choose an instructional language that assists in introducing a rich variety of ideas about programming.

For example, the concept of data abstraction is vital in programming. While neither PL/I nor Pascal facilities are ideal for this purpose, both are vastly superior to those of Fortran and Cobol. Another example is recursion -- for certain tasks this is unquestionably the clearest and easiest programming technique. Recursion is supported directly in PL/I and Pascal, but not in Fortran or Cobol. A programmer who is familiar with the concept, can, when appropriate, simulate recursion in Fortran or Cobol. However, it is unlikely that a course based on a language in which recursion is an error would do justice to this idea.

Cornell uses PL/I to teach programming to all engineering students, fully aware that they will eventually have to program in Fortran. (They are exposed to Fortran, at the end of the first course, and required to rewrite a prior PL/I program into Fortran, primarily to show how familiar concepts appear in a different syntax.) Some employers (and some faculty) were initially skeptical, but the feedback has been strongly positive. Cornell students' initial "handicap", relative to those whose initial instruction was in Fortran, is short-lived and they are judged to be better programmers in the long run.

The Case Relative to Basic

The extent of Basic instruction in universities is hard to understand. By any objective standard, the language is limited and limiting. There are some situations where the use of Basic is virtually unavoidable, since there are small machines for which it is the only high-level language available, and there are laboratory systems where the interface software is provided only in Basic.

Basic is widely taught in secondary schools, and university instructors must overcome this handicap. In this regard, Edsger Dijkstra observes (with characteristic choler):

> It is practically impossible to teach good programming to students that have had a prior exposure to Basic: as potential programmers they are mentally mutilated beyond hope of regeneration.

It does not follow that because it is difficult to recover from a start in Basic that universities must continue instruction in that language. Indeed, it would be ironic if the recent ETS action in basing advanced placement in programming entirely on Pascal were successful in upgrading programming instruction in secondary schools, only to have many universities mindlessly perpetuate the problem.

The myth is, of course, that Basic is exceptionally easy to learn or easy to use (which are not the same thing). The source of this myth-conception is probably the fact that Basic enjoys the support of an integrated, interactive environment that is much easier to use than the old-fashioned batch systems and the clumsy, incoherent interactive systems available for many other languages. However, the Basic language, per se, independent of its environment, has little to recommend it. There is no evidence that Basic is in any useful sense simpler than a well-designed subset of a less limited language.

The other source of support for Basic lies in the ease and compactness with which it can be implemented, which has made it essentially the universal high-level language on microprocessors. One would hope that this is a transitory advantage, and that as larger memories become commonplace on microprocessors, programmers will not forever be afflicted with this legacy of the early machines.

The Case Relative to Pascal

This seems to reduce the question to which of PL/I and Pascal is preferable for introductory instruction today. However, one must keep the discussion in perspective, and realize that the difference in merit between these two is much less than the difference between them and the other candidates. In fact, for the material that would normally be included in a first course in programming, PL/I and Pascal differ primarily in form rather than concept.

The unfortunate fact is that neither PL/I nor Pascal is ideal for contemporary instruction, and it comes down to choosing the one with the least harmful flaws. PL/I's shortcomings have been well advertised, although in many cases the critics seem to be focusing on the obsolete PL/I-F version and to be unfamiliar with the current state of the language. Furthermore, much of the criticism is not applicable to the carefully disciplined subsets of PL/I that are appropriate for instruction. Pascal's flaws are less publicized in the current enthusiasm for this language, but for the last decade the pages of SIGPLAN Notices have chronicled the Pascal debate. The principle points seem to be the following (with apologies to Brian Kernighan):

1) Arrays are fixed in size at the time of compilation. This lack of dynamic storage allocation complicates the programmer's task in order to simplify the implementer's. It makes string handling difficult and general purpose subroutines impossible.

2) There is no provision for **static** (or **own**) variables or for their initialization. This frequently forces the programmer to use global variables which have a larger scope than they ought to. The lack of alternative entry points for procedures makes the "packaging" of data structures and related operations clumsy.

3) The lack of facilities for separate compilation hinders the development of large programs. It also makes the use of libraries impossible.

4) The required declaration order -- constants, types, variables, procedures -- prevent related declarations from being grouped together, and the proscription against use-before-declaration again sacrifices the user's convenience for the implementer's.

5) The order of Boolean expression evaluation is not specified, resulting in convoluted code and extraneous state variables.

6) The **case** statement is very restrictive (compared to, say, PL/I's **select**) and is hampered by its lack of an **otherwise** clause.

7) The lack of **return** and **leave** constructs forces a particularly unattractive use of the abhorent **goto**.

8) The I/O facilities are severely limited. There is no sensible way of dealing with real-world files or program arguments as

part of the language. For example, variable-length variant records -- commonly used in real-world file-processing -- are not readily handled.

9) There is no escape from the limitations of the language. For example, type checking is always enforced, and I/O routines cannot be written in Pascal itself.

It should be an embarassment today to espouse a language in which the user must implement the simple "return" concept in the following manner:

```
label 100;
  ...
goto 100;
  ...
100: end.
```

The Pascal programmer's version of "leave" is similarly barbaric. While it is always possible to avoid the need for "return" and "leave", doing so does not always yield the clearest program, and a language that makes one choose between awkward construction and this distributed atrocity is not entirely laudable.

The "solution" for each of these Pascal problems is some extension to the language -- for example, the UCSD extension for string processing. Unfortunately, there is no consensus on what extensions to make or how to make them, so that today there is a plethora of extended Pascals and program portability is jeopardized. The Pascal standardization effort has been ineffective. Niklaus Wirth explains that Pascal is being criticized for inadequacies arising in use for which it was not intended. Be that as it may, more is being asked of the language today, and the Pascal community has been slow and inept in responding. Pascal was a significant achievement and, with refinement and consistent extension, could have been a superb language, but Wirth has moved on to other languages, and elected not to participate actively in the extension or standardization of Pascal.

Definition of an Appropriate Subset of PL/I

Basically, the problem is that PL/I is too big and Pascal is too small, so one must use a subset of the former or a superset of the latter. In principle, either approach could yield an attractive teaching language. In practice, neither has been entirely effective.

A subset must be extracted from PL/I by two different processes. First, omit the features and constructions that are unnecessary (or ill-conceived); then restrict the usage of the constructions that are retained. That is, require certain usage that is allowed in the complete language. For example, full PL/I permits all its loop control phrases to be combined in one glorious, incomprehensible loop. At least for instruction, it is preferable that the **while**, **until** and indexed-loop constructions be used separately (as in Pascal) and a subset can easily require this practice. As a second example, the infamous

ambiguity between assignment and equality in PL/I can be resolved by requiring that all conditions be parenthesized (thereby also eliminating a syntactic inconsistency between **while**, **until**, **when** and **if**.)

An instructional subset cannot be effectively implemented on paper -- it must be enforced by a suitable processor or the excluded parts of the language will intrude. An error with respect to the subset can accidentally be a valid construction in some excluded portion of the full language, and the effect can be disastrous. An unenforced subset would surely suffer the Swiss-army-knife syndrome.

Unfortunately, no really good instructional subset implementation of PL/I exists. The de facto standard instructional dialect is PL/C, but this was based on the obsolete PL/I-F version and has never been upgraded to include the attractive features of the Optimizer's language (for example, **select**, **until** and **leave**). Furthermore, in misguided obeisance to compatability, PL/C does not restrict usage, as described above. The ANSI standard G Subset has essentially the same shortcomings. A newer Cornell subset, PL/CS, remedies those flaws, but is much too restrictive for anything beyond a first course. (For example, it lacks structures, internal procedures, record I/O.)

The most conspicuous shortcoming of any PL/I subset, relative to Pascal, is the lack of a facility by which the user can define new types of objects, and the concomitant enforcement of consistency of use of types. However, this could be addressed by an extension to an instructional subset -- if it is not forthcoming in future releases of PL/I. An extension to PL/I for this purpose would presumably be handicapped by the traditional obligation to be compatible with old programs, but an instructional subset would have less burden in this regard and contemporary facilities could be added that would be superior to those present in Pascal.

In retrospect, the instructional dialects of PL/I have probably been <u>too faithful</u> to the parent language. There is little operational need for strict upward compatibility, since few programs are developed on one of the instructional systems and subsequently run under the Optimizing Compiler. For example, a PL/I dialect with sub-range and constant definitions, and := for assignment would be a better teaching language even though it was not "pure" PL/I.

<u>Programming Environments</u>

A programming language is used in the context of a programming <u>system</u> or <u>environment</u>, and the characteristics of the environment are at least as important as those of the language. The components of the environment are the operating system, the file system, the editor, the language translator and the execution supervisor. At one extreme, an environment consists of a loose collection of general-purpose tools; at the other extreme, it can be a tightly-integrated system, specialized to serve the needs of a particular class of user of a particular language. To some extent, Basic, APL, and Lisp have enjoyed the support of such integrated systems, which accounts in part for the enthusiasm of users of these languages.

Now, a new generation of integrated environments is emerging, with vastly improved facilities for users. The characteristics of these new environments are as much an advance over conventional interactive systems as those systems were over batch processing. At the moment, PL/I happens to enjoy a clear advantage in this respect, althought Pascal is not far behind.

The two most distinctive features of these advanced environments are language-cognizant editing, and highly interactive display of execution. When general-purpose text editors are used for program development, violations of language rules are not detected until the file is later processed by the language translator. On the other hand, an editor that is specialized to a particular language can both assist and restrict the user -- and do so immediately as program text is entered.

For example, Cornell's COPE system supports development of PL/CS programs on an IBM PC. COPE maintains the invariant that the procedure under development is always complete and correct according to the syntax of PL/CS. The user can arbitrarily modify the procedure text, but each time control returns to the system (by use of ENTER or any program function key) the system does whatever is necessary to reestablish correctness. For example, suppose the user inserts the string

```
while b
```

at some point in the procedure and then presses ENTER. Since the procedure was correct before this insertion, and the insertion is presumed to be an abbreviated form for a new construction, the system supplies whatever is necessary to complete the construction implied by the insertion. In this case, the result is the generation of a new loop at the point of insertion:

```
do while (b);
end;
```

In addition, if the variable b had not been previously declared, an appropriate declaration

```
declare (b) bit(1);
```

is supplied in the declaration section of the procedure. (All PL/CS declarations appear at the head of the procedure -- one of the usage restrictions in the dialect.) Similarly, if the user deleted or modified text, as much of the procedure as necessary is modified to restore correctness.

The response is immediate and self-explanatory. For a beginner, the action tends to be tutorial, providing constant instruction in the required syntax of the language. However, as the user becomes more experienced, the same process is viewed as an efficient way to enter program text: the user need supply only abbreviated entries and the system completes these to satisfy the syntactic requirements of the language. A user who enjoys typing can always elect to enter complete text, as one would to a general-purpose, language-innocent editor, but

most prefer to reduce the number of keystrokes (and concommitant errors) involved in entering a program.

Sometimes, of course, the user's entry is ambiguous, or his knowledge of syntax was faulty, so that the system response is not what was expected or desired. In such cases, an **undo** key provides simple and complete recovery -- the procedure is restored to the exact state in which the user last left it. If the user presses **undo** again, the procedure is restored to the state after the previous system response, eliminating the user's most recent changes. This recovery action can be repeated indefinitely, eventually restoring the procedure to its initial null state. (There is also a **redo** key to permit the user to recover from over-enthusiastic use of **undo**.)

This type of editing evolved out of the automatic error-repair pioneered by the PL/C compiler, but in the process of extension it has became something quite different. PL/C also ensures a syntactically correct program, regardless of what the user enters, but the presumption is that incorrect or incomplete entries are unintentional, so the repairs are appropriatly decorated with error messages. On the other hand, the expectation in COPE is that the user is intentionally entering highly abbreviated text, and relying on the system's ability to supply what was deliberately omitted. In effect, the result is a cooperative program generator, in which the user directs the process and the system writes the program.

These new environments are comparably supportive with respect to program execution. For example, during execution, COPE simultaneously displays four windows:

the trace window in which the text of the current procedure is shown, with a cursor indicating the statement presently being executed

the check window in which the current value of all local variables is shown, with a cursor to show the most recently changed

the output window in which the result of execution of output statements appears

the input window in which the head of the (unread) data stream is shown.

The user has the ability to control the speed and direction of execution -- that is, the **undo** facility permits reverse execution. Execution can also be interrupted at any point to change values, or the procedure itself can be modified. In many cases, the execution of a modified procedure can be resumed without having to start again from the beginning. Needless to say, testing in such an environment is quite a different experience.

Eventually such environments should be available for all programming languages, at which point their availability will not be a factor in the choice of language for instruction, but at the moment PL/I enjoys a clear advantage in this regard.

Conclusions

Three primary conclusions emerge:

1. PL/I and Pascal dominate other general-purpose languages for use in education in programming. The other candidates lack the block structure and modern control constructs that are important to effective program development.

2. Neither PL/I nor Pascal dominates the other for this purpose. A well-designed PL/I subset is not unlike Pascal in most respects, with both having offsetting shortcomings.

3. Both PL/I and Pascal are flawed and obsolete, and a successor is overdue but not yet obvious.

The only plausible argument in support of Fortran, Cobol or Basic for programming instruction is the widespread use of those languages outside of the university, but the selection of one of these says more about the objectives of the course than the merit of the language.

PL/I is too rich and Pascal is too lean for this purpose, so one must rely on a subset of the former or a superset of the latter. Either would still possess an annoying set of inadequacies, and to an unfortunate degree one is faced with the lesser-of-two-evils problem.

PL/I enjoys an advantage in implementations available for (IBM) mainframe systems, which is offset by Pascal's availability on a wide variety of microprocessor systems. On the other hand, PL/I subsets are available for the IBM PC (and for the many PC-compatible machines) which are increasingly important in instruction. Furthermore, the new IBM XT/370 makes mainframe implementations of PL/I available on a microprocessor workstation.

Both PL/I and Pascal reflect the state-of-the-art of programming of the sixties, and neither is entirely adequate to address the programming needs of the eighties. For example, concurrent programming is no longer of concern only to a few specialists who implement operating systems. It is now commonplace in data-processing, scientific applications, and embedded control systems. PL/I "tasking" allows only a crude introduction to the subject, and Pascal offers nothing at all. Newer languages such as Concurrent Pascal, Modula and Turing address the problem, but none has yet achieved wide enough acceptance to be a viable candidate for general-purpose instruction. Similarly, there is increasing interest in being able to treat at least segments of a program as mathematical objects whose correctness can be established without empirical testing. Again, new languages are exploring this issue, but none seems quite ready for widespread adoption.

An obvious question is whether Ada is potentially the successor to these obsolete languages. It certainly has the broadest base of support of all the "new" languages. But inevitably Ada is rich and complex, and Pascal advocates have trouble loving any language that large. Moreover, the official injunction against any subsetting of the language has discouraged the development of instructional versions. Nevertheless, it

is a fair guess that by, say, 1985 there will be instructional subset implementations of Ada and a vocal Ada lobby in universities.

If a computer science department were choosing today to move to either PL/I or Pascal from some other language, I suppose the choice would have to be Pascal simply because that is the standard language in such departments. However, a department that has been using PL/I (or Algol) for a decade or more should be forgiven a reluctance to endure the trauma of conversion for the dubious advantages of Pascal. At Cornell we had hoped to be able to sit out the Pascal era and move from PL/I directly to some clearly superior language. However, the arrival of that new language is overdue, and there are now some very attractive new microprocessor workstations for which no PL/I is available, so a double conversion may be necessary.

However, outside of computer science, I would argue that PL/I is still the best choice for instruction. For business students, PL/I is simply the modern alternative to Cobol. For science and engineering students, there are attractive functional advantages in PL/I, but the crucial issue is emancipation from Fortran, so I would happily settle for Pascal, if necessary.

On language issues alone, ignoring considerations of support and widespread acceptance, I find Toronto's Turing language the most attractive of the new candidates. Holt describes it as "Pascal-like", which is the politic thing to do, but it offers PL/I's power with Pascal's style, and addresses the contemporary issues of concurrency and verification. It is too bad they couldn't call it PL/II.

Acknowledgements

I would like to express my appreciation to my colleagues David Gries, Fred Schneider, and Steve Worona who helped to improve this piece. Worona basically agrees with the argument. Gries and Schneider are less fond of PL/I, but hate to see anything poorly written.

References

Archer, J., R. Conway and F. Schneider, "User Recovery and Reversal in Interactive Systems", ACM Transactions on Programming Languages and Systems, (January 1984).

Dijkstra, E. W., "How Do We Tell Truths That Might Hurt?", SIGPLAN Notices, (May 1982) Vol. 17, No. 5, pp. 13-15.

Gries, D., The Science of Programming, Springer-Verlag, 1982.

Holt, R. C., and J. R. Cordy, "The Turing Language Report", Report of the Computer Systems Research Group, University of Toronto, Nov. 8, 1983

Kernighan, B., "Why Pascal is Not My Favorite Programming Language", Computing Science Technical Report No. 100, Bell Telephone Laboratories, July 18, 1981.

Post, E., "Real Programmers Don't Use PASCAL", Research Report, Tektronix, Inc., Wilsonville, OR 97070, 1982. (Also in Datamation, August 1984.)

Teitelbaum, T. and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Communications of the ACM, (Sept. 1981) pp. 563-573.