HAS THE KING RETURNED?

Conrad Weisert
Information Disciplines, Inc.
1205 Madison Park, Chicago, Illinois 60615

Introduction

Imagine a general-purpose programming language that offers:

- better exception handling than Ada
- better string handling than Basic
- better input-output than Cobol
- better computation than Fortran
- better structured flow control than Pascal
- better macros than assembly languages
- better memory management than C

where "better" means some combination of easier to use, easier to learn, more complete, more reliable, and more fully integrated with the rest of the language. Wouldn't that language be worth looking into as a candidate vehicle for your next big software development effort? Well, that language exists, and it's not some vendor's proprietary "4th generation" wonder, but an established language supported by ANSI and international standards. It's PL/I.

PL/I? Didn't PL/I *die* a decade ago, a victim of the desktop computer revolution? Wasn't PL/I too *big* to be implemented in the leaner computing environments of the early 1980's? And hadn't it become an *old* language that ignored such modern essentials as graphic user interfaces and class hierarchy?

Yes, all of that was true, but PL/I is back.

New Product from IBM

The product is called "PL/1 Package/2" for OS/2, and so far it looks like a blockbuster.

Package/2 marks a serious return by IBM to the support of the tool it had once promoted as the "language of the future", but later all but abandoned. Package/2 is a major compiler and development environment, supporting not only the full language but also some extensions for the OS/2 Presentation Manager and modern client-server platforms. And PL/I no longer seems *big* at all in the world of Windows and C++.

Is it too late for PL/I? The answer will depend on several unpredictable factors, including IBM's marketing strategy, the growth in OS/2 use, and the response of a programming language marketplace where merit and popularity rarely coincide.

Why PL/I?

Given the momentum of C++, not to mention such higher-level GUI tools as Borland's *Object Vision* and Apple's *Hyper Card*, why should we complicate our lives by taking on another procedural programming language now? What could PL/I possibly offer us to entice us away from those "proven" performers?

That's almost the same question companies were asking a quarter-century ago about this same language! The comparison then, of course, wasn't with C++ but with the then "established" languages Fortran and Cobol. Since then, a generation of programmers has grown up on C and Basic, on MS-DOS and the Macintosh. To many young programmers PL/I is as unknown today as it was to their parents in 1968.

Through the 1970's most organizations ignored PL/I, overestimating the cost of investing in new skills and underestimating the magnitude of the potential improvements in quality and productivity. Of those who took the plunge some just used the new language as a vehicle for their traditional approach to organizing programs; they realized only modest improvements, perhaps barely sufficient to justify the cost of training and support. Others, however, seeing that new approaches were both possible and necessary, reaped huge benefits, sometimes an order of magnitude in productivit

For that minority, choosing PL/I turned out to be one of the best decisions they made in the era of third-generation computer systems. Does Package/2 offer similar rewards for similarly inclined organizations?

PL/I Shortcomings

Despite its power, PL/I is deficient in a couple of areas of critical importance to some projects.

First, it offers no built-in mechanism for defining new data *types* or classes. One can, of course, achieve some of the same effects through explicit coding, but PL/I gives us little or no built-in help with such object-oriented notions as:

- inheritance
- operator overloading
- automatic range checking on assignment

Second, it's still not possible for a function to return an aggregate result, i.e. an array or a structure. It can, of course, return a *pointer* to such an aggregate, but that usually demands messy and error-prone explicit storage management.

These are significant deficiencies, of course, for some kinds of programming. How do we balance them against the serious deficiencies of the newer languages like C++? The choice depends on the nature of the application and on its data-base architecture. It will be interesting to see how easily one can mix PL/I and C++ modules in OS/2, so as to exploit the strengths of each language.

Being in Charge

When I'm asked to cite the one aspect of PL/I that distinguishes it from other languages, I reply that it's the feeling that it's the *programmer* who is in charge, not the language or the compiler. The PL/I programmer rarely spends time devising ways of circumventing some restriction or of fooling the compiler into doing something it wasn't intended to do. Only assembly languages and APL have given me the same feeling of *control* that I get when using PL/I.

Critics of PL/I complain that it provides *too many* ways of accomplishing the same result. Perhaps, but I have yet to feel nearly so irritated or frustrated by having a choice of techniques as by finding no way at all to accomplish some perfectly reasonable result.

What "Generation" is PL/I?

Some well-known writers have recently been lumping *all* procedural languages into the so-called "third generation". That's startling to those who remember late-1960's debates comparing the old "second generation" Fortran and Cobol with the newer PL/I that supported "third-generation" notions like multi-tasking, block structure, and dynamic resource management, to say nothing of *structured programming*. While today's Fortran and Cobol are much improved, it's still misleading to put them in the same language family with languages like PL/I, Ada, and Modula-2.

James Martin has proposed an interesting criterion for admitting a language to the "fourth generation":

> A language should not be called fourth generation unless its users obtain results in one-tenth the time of Cobol, or less.
>
> Application Development without Programmers
> 1982, Prentice Hall, p. 28

But organizations that exploited the full power of PL/I and its (macro) preprocessor have reported just such an order-of-magnitude improvement over typical uses of Cobol. According to Martin's criterion, then, we're justified in calling PL/I a fourth generation language (one of the few, by the way, that isn't proprietary).

What Next?

PL/I's return is like Vladimir Horowitz coming out of retirement in 1965. Could he, a dozen years after his last public appearance, possibly regain his stature as the *king* of pianists, competing with a younger generation of musicians? We know that Horowitz went on to reign for two more decades, supreme in much mainstream repertoire, but conceding some specialties to others. PL/I is in a position to do much the same.

A threat to PL/I's comeback, however, lies in IBM's marketing strategy. PL/I may be the king of programming languages, but the existence of Package/2 remains a near secret among PC users. IBM's video tape "meet the developers" presentation is aimed at wooing confirmed PL/I programmers away from their mainframes, but says little to confirmed PC programmers. Without vigorous and sensibly targeted promotion, it's hard to imagine a groundswell of new PL/I users.

Individuals and small organizations will also be deterred by Package/2's initial price, reasonable perhaps compared with mainframe PL/I, but still several times the price of most other PC-based procedural programming language processors. Only when the price falls below $1000 will many potential users be tempted to experiment with this "new" programming language.