FUTURE DIRECTIONS IN PROGRAMMING LANGUAGES

Samuel A. DiNitto, Jr.

Rome Air Development Center
Griffiss Air Force Base, New York

Abstract: This paper explores some possibilities for the programming languages of the next century. Projections are based on what has and has not been accomplished in the last four decades and the programming tools proposed for the next decade. Influencing programming language directions at both the high levels of software development (e.g., problem decomposition) and at the lower levels (e.g., algorithm implementation) will be parallel execution. There could (finally) come a serious break with the traditional languages such as Fortran, Algol, PL-1, Pascal, C, and Ada. Very High Level Languages could take over in popularity. Within an increasing number of domains, the programming, in addition to man-machine interfaces, will be accomplished through multiple media (VHLL, speech, natural language, mouse, menus, touch screen, etc.).

I. Introduction

This paper explores some strong (at least in the author's opinion) candidates for the programming languages that we will see in the next century. For this purpose, "programming language" is defined as any language used to create a set of instructions for a computer to follow in carrying out a task, or a framework to use in solving a problem, when that solution is storable for future use. It attempts to do this by reviewing how we have arrived at today's popular programming languages, and identifying the relevant practical experience gained from that history. The history will be used with the state of today's research and expected future successes to temper the ideals for the next generation of programming languages.

This pragmatic author believes that, with regard to a programming language's success, it is more often the inertia and politics of its competitors that must be overcome rather than its relative merits. In the world of programming languages, inertia is measured in the amount of software already built in a language and still in use, the number of people trained and actively using the language, the number of popular language sensitive software development tools for the language, the requirements (both past and current) for the use of the language in specific applications, and other such culture considerations. Similarly, "politics" is concerned with such things as authority and scope of authority of those who require the use of a language, the stature of those who endorse the language, and the power (often measured as "share-of-the-market") of those responsible for the origination of the language.

Thus, "inertia" covers those items that must be overcome, and "politics" covers those that must be obtained to begin building significant inertia. As examples, FORTRAN and COBOL have much inertia and in the past have had strong political backing by Government and Industry; in the past, Pascal has received strong academic political endorsement but has developed little inertia; and Ada has seen mixed political endorsement, but is now enjoying a rapid buildup of inertia.

In synopsized form, the present state-of-affairs is simply this: The procedure oriented (or imperative) style of languages (FORTRAN, Pascal, C, ALGOL, PL-1, BASIC, Ada, etc.) still rule the roost with regard to popularity, and most of these languages (when we include all their dialects) are still increasing their inertia, although the relative share of the market is decreasing for some (e.g., the various dialects of ALGOL, PL-1, FORTRAN, etc.). The functional (or

applicative) styles, such as LISP, and the logic styles, such as Prolog, are seeing a resurgence of activity (at least in the R&D world) due to the renewed interest in artificial intelligence in the last decade and the so-called Fifth Generation and Strategic Computing thrusts. We see some successes for the application specific, or very high-level languages (VHLLs), styles in business applications, tooling, automated test equipment applications, and even AI/expert systems applications; but in the big picture, the use of such languages is not universal, even within their narrow domains, so they have not really caught on yet.

If one accepts the above view of where we are, let's now try to establish how we got here. This may allow us to determine what we can learn from a historical perspective that may be relevant for our future projections.

Most people acknowledge that it was really the language FORTRAN, in the 1956 timeframe, that seriously kicked things off for the computing world in general. In the next five years, almost everyone in the computing community had at least heard of COBOL, ALGOL, LISP, and possible NELIAC, but how about the other five dozen languages? Five years after that (i.e., 1966), in addition to the numerous dialects of the previously named languages, computer people at least heard of PL-1, SNOBOL, JOVIAL, and BASIC, but not too many heard of the other eight dozen languages available [SAMM 72, pgs 606, 607].

Let's stop here for awhile to make some observations about the languages being used to build so-called "fielded" systems, ignoring the fact that, at that time and for at least ten and more years, most software would be implemented in assembly language. Computers were expensive at that time, and the programmer who could implement software in the fewest (however measured) number of words of storage was golden. Recursion and block structure were elegant, but both, especially the former, chewed up storage and machine cycles. The need for the programmer to stay close to the machine's own instructions was evident by the above and the need to debug from so-called "core dumps" of the binary state-of-the machine's memory and registers at selected points. Finally, only FORTRAN and COBOL were being taught en masse to the people actually building the fielded systems, and I might add by some very innovative approaches like John McCracken's Autocoder for FORTRAN.

During these times, very few places had more than one computer system or had the means or inclination to fund more than one. Also, applications were becoming more and more diverse. This meant there was a need for a practical, general-purpose language to span the scientific, business, and real-time (largely for DOD applications) communities. However, the world did not leap at a very good one for the times, namely, JOVIAL, whose implementations came on the scene in the early 1960's [SAMM 69, pg. 530].

JOVIAL, for Jules' Own Version of the International Algebraic Language, built on the structure and controls in Algol-58. It added tables and arrays, capable of being packed at the bit level by programmers, or at the bit, byte, and word level by the compiler. Status, literal, Boolean, and user specified fixed-point data types were also added. The language enforced its typing rules, but on a case-by-case basis a programmer could alter them. The DEFINE facility allowed one to isolate machine dependent parameters for easy redefinition when porting to another machine. COMPOOLS of shared data, programs, and their specifications allowed the compiler to correctly integrate and/or help debug software developed by multiple programmers. Finally, if all else failed, the language allowed one to fall directly into assembly language to accomplish what could not be efficiently handled at the source level. What more could a programmer of the 1960's ask for?! What went wrong?.

Well, for one thing, while several of the large mainframers, such as IBM, UNIVAC, and CDC had some efficient compilers for JOVIAL (most directly or indirectly funded by the Air Force); while several large, complex, real-time and successful Air Force and FAA systems were implemented in JOVIAL; while even commercial applications did show up (airline reservations); and while the UK's Ministry of Defence copied the JOVIAL style and philosophy in CORAL-66, there was no perceived support or long-term commitment. After all, the newer, more modern PL-1 was being supported by the company with over two-thirds of the worldwide marketplace. Merit and track record just did not count for much. People using Fortran, COBOL, and/or assembly language were content to wait until PL-1 came around, with all the support and backing that only the large

mainframers could supply, even though by the mid-1960's the most respected software houses (e.g., SDC, CSC) were fielding and supporting JOVIAL compilers, training, and JOVIAL systems software (including a working time-sharing system).

Three points can be made at this time. First, a language wasn't going to leave the R&D community unless it made efficient use of resources. Second, without readily available training, there was little possibility of a language gaining popularity. Third, without at least perceived backing of the hardware vendor(s), a language was doomed. The limited but significant success of CORAL-66 in the UK and the promising future of Ada are built on these points.

Other points that became readily accepted by the community were that people strongly resisted switching to another programming language if they considered themselves proficient in one, and the first language you learned forced you into a style of programming and a perception of computation that was difficult to change. Anyone who doubts the above should try to teach seasoned FORTRAN IV programmers ALGOL, Pascal, or Ada (just wait till you get to recursion).

The years 1966 to the present are characterized by the introduction of many new languages, most of which died in the research community, or never gained critical vendor support. The focus with the imperative languages switched in the early 1970's from "give the programmer the power to do anything he might possibly want without encumbrances, such as strong typing or the requirement to learn and understand the whole language" (PL-1, Algol 68); to "give him a small, efficiently implemented, well-defined language and protect him from his human frailties with regard to programming and those of other programmers" (Pascal, ALGOL-W), to "give him a large, powerful efficiently implemented language with protection mechanisms" (Ada). From other perspectives, we saw attention being paid to languages (Euclid) or language features (aliasing) that respectively helped or hindered formal verification. We saw various attempts at dealing with mass storage (pointers), concurrency (semaphores, monitors, tasking), real-time (interrupts), flexibility (generic data types), etc., etc., etc. The bottom line is that, like it or not, in over thirty years, we have merely tweaked in an evolutionary way, the basic approach to programming the Von Neumann computer, and, in this author's opinion, <u>in practice</u>, we have not made that overwhelming improvement in productivity or quality that has been promised with the introduction of each new programming language in this style. If you doubt this, try to develop and conduct an experiment that will prove the advantages of one imperative language over another to the majority's satisfaction.

The case for the applicative style of language on the surface is not <u>much</u> different. The past twenty-eight years, since McCarthy's early papers, have witnessed a gradual inclusion of the more complex features that students have for years implemented as exercises (e.g., APPEND) and the borrowing of some features from the imperative style (e.g., CASE, DO-WHILE, etc.) much earlier than for the imperative styles, the various dialects of LISP have built up impressive, almost standard, software engineering environments (e.g., Allegro Common-LISP, INTERLISP, LISP Machine Environment). If we probe deeper into the LISP family, we see that, in comparison with the imperative languages, the LISP dialects have achieved much more along the lines of reusability, truly integrated support environments, and a higher level of expressability that improves both quality and productivity in terms of software development.

So why aren't we all LISP programmers? For one thing, there is still no <u>real</u> support from the big vendors, and the small hardware vendors, who do seriously support LISP, do not have a significant share of the market. For another, there is an even bigger challenge to turn FORTRAN and COBOL programmers to the applicative style than to Ada. In addition, the applicative languages don't yet make efficient use of the computer hardware that is designed to execute them, let alone the classic Von Neumann computers. Outside of possibly the best endowed computer science labs and those who <u>still</u> believe that right around the corner are all the storage and machine cycles per second we will ever need, at a fraction of our budget, this efficiency concern is still with us. While it is easier to demonstrate progress with applicative languages, this author's verdict is that progress cannot hold up to the demands for efficiency, productivity, and quality in the market place.

The problem or application oriented languages in popular use today can only

be described as "loosing ground" with respect to productivity and quality. While they tend to make some very definite headway when first employed, their development tends to stagnate rapidly and they become more or less frozen in terms of expressability and hence productivity. The problem is that they are effectively only a shorthand notation or macro to be simply translated to a routine set of assembly or higher level programming statements, and they give little or no assistance beyond that point. This doesn't have to be the case, but the further refinement of these languages would cost more sophistication and complexity in their implementations than their vendors will risk.

The logic style languages, although fifteen years old, are still in their infancy. Without the interest of Japan's Fifth Generation project and more recently the interest of the Defence Advanced Research Projects Agency (DARPA), these languages would be largely unknown. There are a few of us (this author included) who think the potential for this style has just been scratched. There are fewer yet (but, again, include this author in the few) that think the merger of the logic and applicative styles offers even greater advantages. While such a smooth and aesthetically pleasing language called SUPER (for Syracuse University Parallel Expression Reducer) exists, the hardware to efficiently execute it is not yet in the breadboard stage.

II. "Drivers" for the Next Generation of Programming Languages

If it is truly to be a next generation of languages, the new generation must catapult us out of this slow evolutionary improvement cycle we are in. The slow addition of evolutionary features, an addition usually thinly disguised by the "syntactic sugar" of a "new" language, will mean that the next century's early languages will mostly be today's languages. In 1972, Jean Sammett [SAMM 72, pg 609] pegged FORTRAN's and COBOL's life span at "..... at least five and probably ten more years." Is there any among us who doubts they will disappear before the year 2000? This author wrote his last line of FORTRAN in 1969, threw rocks all during the 1970's at those proposing that FORTRAN IV be the language for portable software, and finally gave up when in 1985 one of the research products under his control was written in FORTRAN 77 to guarantee maximum availability - even though the product analyzes Ada software! Now we are on the verge of FORTRAN 88! One doesn't become a pragmatist (or a cynic) overnight.

So, what are the drivers for a truly next generation of languages? We have been hearing this for at least fifteen years, but parallelism (both synchronous and asynchronous) must be handled directly (and safely) within the next generation languages. Even with projected successes in hardware technology, it will still fall on parallelism to provide the computing horsepower for many applications.

Certain of today's compilers for the non-explicitly parallel languages detect the obvious forms of parallelism (nested loops of independent operations), but they usually assume (because they can't always detect it) no side effects or aliasing. The parallelism is not invoked at the hint of a problem (e.g., a subroutine call).

The explicitly parallel languages allow for the programmer's declaration of the parallel sections of the program. While this allows for more parallelism than can be achieved for the nonexplicitly parallel languages, and thus potentially more efficient use of the resources, the languages count on the programmer knowing what he is doing; their compilers cannot detect all programmer induced deadlocks and races. This problem is of course compounded because such problems don't always show up in testing, and cannot always be reproduced in follow-up testing once they have occurred during operation. A truly future generation of parallel programming languages and their implementations must overcome these problems and limitations.

Object-oriented design and implemtation has created as much, if not more, enthusiasm than structured programming did in the early 1970's [Broo 87, pg 14]. While certain of today's more popular programming languages support some of the essential ingredients for object-oriented design and programming, namely, encapsulation, message passing, late binding, and inheritance, [WILS 87, pg 53 ff], they don't supply them all, or not all to the degree necessary. Usually, today's approach to object-oriented software is to design in the object-oriented style, but to implement the object orientation through "project discipline," as no current language implementation provides <u>efficient</u> support for this paradigm.

Another driver for truly new

languages is that the role of the "pure" programmer is rapidly disappearing. More and more, businessmen, engineers, scientists, etc., are developing or modifying their own software rather than deal with a programmer they probably can't communicate with, because the programmer doesn't really understand the application in enough depth. These people, while computer literate, are not software engineers capable, nor inclined, to put together large software systems, of tens to hundreds of thousands of lines of FORTRAN, COBOL, Ada, or any other similar language, requiring dozens of manyears of effort and large teams. They need powerful languages capable of expressing solutions and ignoring implementation details. They need to go beyond today's level of application specific languages which stop at reusing macros and subroutine libraries. They need reuse, but reuse of generalities in addition to specifics. They need prompting, easy interfaces, protection from obvious and subtle mistakes, etc. In short, they need reuse of knowledge. Here, the line between language and applications package may seem gray; but if a series of high-level operations is created and stored for reuse, how can one argue that's not a program created with a programming language?

On yet another scale, the software and/or system engineers need higher levels of expressability if we are to put together the large complex systems of the Twenty-First Century, in a reliable manner. Today, we have design languages, requirements languages, and even fledgling prototyping languages. However, what is missing is the mechanical translation from any one of these forms of the software to the next phase. We have enough trouble getting people to learn one new language, let alone one (or more) to deal with each phase of the software's life cycle. For years, we have argued that we must keep the human away from the implementation details with the kind of systems we need (and plan) to build; that means much more than shielding the human from the code generated by a "CASE" statement.

To amplify the above, we are also witnessing a phenomenon that will become even more of a problem in the future, namely, the proper development and maintenance of even the concept for a system. Consider the number of distinct areas of science and engineering involved with placing and maintaining a manned space station into space; the amount of change in the concept from birth to fielding such a system (three decades brings a lot of change); the number of people that will pass through such a project (loss of corporate memory); and even the advancement (favorable or unfavorable) of technology during the development of the system. How can we expect to have a "good handle" on such a concept, evaluate the thousands of alternatives for interrelated functions of systems, subsystems, sub-subsystems, etc., and communicate these concepts to the hundreds of companies that would be involved in building the software. A tool, or one would argue a general-purpose "language," is needed to create and maintain an ever evolving executable conceptual model of long-term complex systems.

III. This Author's Prognosis for the Future.

With regard to a truly new generation of programming languages in wide use to support parallelism by the year 2000, the chances are very slim. While there may be better places to start augmenting languages, (e.g., Occam), features to explicitly support parallelism will continue to be added to FORTRAN, Pascal, Ada, C, etc. It is extremely doubtful a more complete set of protection mechanisms, to prevent or detect the "overparallelization" of a program, can or will be developed, fielded, and massively supported in this century. Remember that Ada is really a 1970's vintage language and, only because of huge investments by DOD and Industry, and unabashed arm twisting by the DOD and other NATO defense organizations, is Ada now (thirteen years later) a prominent language. Thus, if the perfect parallel language was already on the drawing board, it is doubtful if it would make it by 2000. While some people argue that efficient Ada implementations (in terms of run-time) are still difficult today, we can argue those implementations are almost all for the same class of machine, the single instruction - single data architecture. The complication in getting efficient implementations of a new parallel language, and its run-time package on several different parallel architectures, is much higher than for an efficient Ada compiler strategy for the simple Von Neumann computer.

This next projection is easy. The author has no doubts there will be a set of efficiently implemented and efficiently executing implementations of object-oriented languages along the

lines of Smalltalk. Unfortunately if the history repeats itself, we will also see even more efficiently implemented and efficiently executing variants of today's imperative languages that don't quite do the job. (Like FORTRAN-77 supported structured programming, maybe there will be a FORTRAN-99 for object-oriented programming.) However, vendors appear to be latching on to varients of Smalltalk, so there is some hope we won't repeat the past too faithfully.

The application specific languages are finally going to make some real headway by the next century. With a little more commitment and some adventuresome spirit by the hardware and software vendors, there could be a rash of intelligent very high-level languages for everything from machine tooling to prototyping or even fielding and maintaining C3I systems or subsystem components.

Of course it is knowledge based systems technology that will allow this to happen. The "knowledge" built into the analogy to today's compilation process will allow for truly "typeless" language in specific application areas. It will allow for the "skeletal routines" purported in the mid-1970's as a way of leveraging productivity and quality by having generic subroutines tailorable to specific requirements, although the tailoring will be done by the intelligent compiler, and not by the "programmer," as originally conceived.

The technology is almost here to allow for a mixture of communication media between man and machine (speech, natural language, mouse, menues, etc.) in narrow domains. While a pragmatist should not project until the technology is well in hand, in this case, it will definitely be a case of "keeping up with the Jones" after the first such system is fielded; the difference in having it or not having it is not as subtle as whether or not a hardware-software combination supports bit-map displays or the hardware can support 1.2 vs. 1.0 MIPS. By the year 2000, if you don't support capabilities like an informal interface, "smart" assistance, and "intelligent" monitoring of execution, you will be closed out of certain application areas.

The role the hardware will play in supporting these projections cannot be ignored either. The cost of workstations that can handle full-up expert systems engineering tools is dropping rapidly. A card, not yet on the open market, can be added to a PC-AT to recognize 10,000 spoken words. In the next five to ten years, it is a safe bet that hardware of the class of today's best AI workstations, outfitted with the equivalent boards similar to the above, will be in the $10K range. In narrow domains, if we make efficient use of the resources, the application specific languages can really "take off" and fulfill the productivity and quality promises of the last fifteen years.

With regard to the general-purpose languages to put together large systems (the current in vogue term is Very High-Level, Wide Spectrum Languages), it is almost certain that, although direly needed, the pragmatic view is that they will not be available in the early twenty-first century. However, for the moment, even though history is against us in programming languages, let's recognize that in other disciplines, technology projections are usually overoptimistic relative to actual progress in the first five years, but pessimistic relative to actual progress made in the five-to-fifteen year bracket of the future. Taking that view, there are so many pieces of the theory and technology puzzle being worked throughout the world that an optimist would have to argue all we need is more funding and more focus to pull off these Very High-Level, Wide Spectrum Languages.

Some pieces of the problem could be solved (at least partially) with the logic-oriented style of languages. Concept modeling requirements analysis and prototyping have already been demonstrated with this style [DAY 87]. If the goals of the Fifth Generation thrust in Japan are successful, that hardware, coupled with this style, may be all that's needed as the seed corn for a wide class of applications.

The Knowledge-Based Software Assistant (KBSA) [Green 83] attempts an evolutionary approach to a revolutionary paradigm for developing and supporting large software systems. The approach is to unite a set of intelligent knowledge-based tools visible to the user (requirements assistant, specification assistant, performance assistant, project management assistant, etc.) with each other and some tools not so visible to the user (activities coordinator, project data-base builder). The unification would produce a single tool which assists in all aspects of the

software development and captures knowledge, about the specific software system being built, that normally disappears (design decisions, algorithm selection and implementation strategy, etc.). This knowledge is invaluable for that system's future maintenance and upgrades, for reducing the work necessary to build similar systems in the future, and for accommodating the inevitable personnel turnovers on a long-term project. If successful, someday (not in this century) for a specific class of very complex systems, such as air traffic control, one may only need to specify requirements for the grandson of a system previously built with this paradigm and have a KBSA do the rest. This would be the ultimate in high-level languages.

To reach the ultimate goal of the KBSA is obviously very ambitious and, while various compoents are proving out today (Project Mangement, Requirements Assistant, Specification Assistant, Performacne Assistant, and even a rudimentary framework to connect the independent components or "facets" in KBSA terms), much more work needs to be done. However, there appears to be renewed interest on both sides of the Atlantic in one of the key areas, formal (or mechanical) translation. Some other keys to the KBSA paradigm (e.g., distributed knowledge acquisition, truth maintenance) are necessary for many other applications in knowledge-based systems and are also being worked. It is hoped we can revisit the success this new paradigm in fifteen-to-twenty years

IV. Conclusion

History is against rapid change in programming languages. However, to meet the software challenges of the next century, challenges which are inseparable from and required for advancements in many fields, we must make some revolutionary changes. If we really put our minds to it and focus our attention on furthering advancements, rather than succumbing to temptation and dissipating our intellectual and financial resources by trying to accommodate each step forward in almost every existing programming language, we can orchestrate and field this very necessary revolution.

BIBLIOGRAPHY

1. [BIRD 87] Bird, Richard, "A Calculus of Functions for Program Deviation," Technical Monograph PRG-64, December 1967, Oxford University.

2. [BROO 87] Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," Computer, April 1987, pp 10-19.

3. [DAY 87] Day, William, et al, Logic Programming Applied to Requirements and Specifications, RADC-TR-87-260, December 1987.

4. [FAUG 86] Faught, William S., "Applications of AI in Engineering," Computer, Volume 19, Number 7, July 1986, pp 17-27.

5. [GENE 85] Geneserth, Michael R. and Mathew L. Ginsberg, "Logic Programming," Communications of the ACM, September 1985, Volume 28, Number 9, pp 933-941.

6. [GREE 83] Green, Cordell, et al, Report on a Knowledge-Based Software Assistant, RADC-TR-83-195, August 1983.

7. [HOAR 87] Hoare, C. A. R., "An Overview of Some Formal Methods for Program Design," Computer, September 1987, pp 85-91.

8. [HWAN 87] Hwang, Kai, et al, "Computer Architectures for AI Processing," Computer, Volume 20, Number 1, January 1987, pp 19-27.

9. [INGA 78] Ingals, Daniel, "The Smalltalk - 76 Programming System," Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, January 23-25, 1978

10. [MATT 87] Matthews, Gene, et al, "Single Chip Processor Runs LISP Environments," Computer Design, May 1, 1987, pp 69-76.

11. [SAMM 69] Sammet, Jean E., Programming Languages: History and Fundamentals, 1969, Prentice-Hall.

12. [SAMM 72] Sammet, Jean E., "Programming Languages: History and Future," Communications of the ACM, July 1972, Volume 15, Number 7, pp 601-610.

13. [TICH 87] Tichy, Walter F., "What

Can Software Engineers Learn from Artificial Intelligence?", Computer, November 1987, pp 43-54.

14. [UNGA 87] Ungar, David and David Patterson, "What Price Smalltalk," Computer, Volume 20, Number 1, January 1987, pp 67-74.

15. [WIL 87] Wilson, Ron, "Object-oriented Languages Reorient Programming Techniques," Computer Design, November 1, 1987, pp 52-62.

16. [WULF 80] Wulf, William A., "Trends in the Design and Implementation of Programming Languages," Computer, January 1980, pp 14-23.

17. [YOK 081] Yokoi, T., et al, "Logic Programming and a Dedicated High-Performance Personal Computer," Proceedings of International Conference on Fifth Generation Computer Systems, October 19-22, 1981.