Almost Perfect Artifacts Improve only in Small Ways:
APL is more French than English

Professor Alan J. Perlis
Yale University

I'm an apostate from ALGOL. Having been raised on programming languages of the ALGOL variety, I came under the influence of APL rather late in life. Like all people who enter interesting things late in life, one tends to go over one's head very quickly. I think it might be interesting to say how I came under the influence of APL, because maybe many of you have gone the same route.

I was at a meeting in Newcastle, England, where I'd been invited to give a talk, as had Don Knuth of Stanford, Ken Iverson from IBM, and a few others as well. I was sitting in the audience sandwiched between two very esteemed people in computer science and computing — Fritz Bauer, who runs computing in Bavaria from his headquarters in Munich, and Edsger Dijkstra, who runs computing all over the world from his headquarters in Holland.

Ken was showing some slides — and one of his slides had something on it that I was later to learn was an APL one-liner. And he tossed this off as an example of the expressiveness of the APL notation. I believe the one-liner was one of the standard ones for indicating the nesting level of the parentheses in an algebraic expression. But the one-liner was very short — ten characters, something like that — and having been involved with programming things like that for a long time and realizing that it took a reasonable amount of code to do, I looked at it and said, "My God, there must be something in this language." Bauer, on my left, didn't see that. What he saw or heard was Ken's remark that APL is an extremely appropriate language for teaching algebra, and he muttered under his breath to me, in words I will never forget, "As long as I am alive, APL will never be used in Munich." And Dijkstra, who was sitting on my other side, leaned toward Bauer and said, "Nor in Holland." The three of us were listening to the same lecture, but we obviously heard different things.

What attracted me, then, to APL was a feeling that perhaps through APL one might begin to acquire some of the dimensions in programming that we revere in natural language — some of the pleasures of composition; of

saying things elegantly; of being brief, poetic, artistic, that makes our natural languages so precious to us. That aspect of programming was one that I've long been interested in but have never found any level for coming close to in my experience with languages of the FORTRAN, ALGOL, PL/I school. It was clear in those languages that programming was really an exercise in plumbing. One was building an intricate object, and the main problem was just to keep your head above water. But, so difficult is it to keep your head above water with those languages that this aspect of the languages we use in programming just never surfaces. For me, in listening to Ken then — and I'd heard him before — for me, at that moment, there came what I can only call a revelation.

I heard Ken speak in 1963 at Princeton at a meeting on programming languages, and he spoke about APL. But at that time APL was not running on any computer; and he stoutly insisted that it was unnecessary that it ever run on a computer. It was for him a notation with which he could express algorithmic concepts; and for him at that time, that seemed sufficient. To those of us who were concerned with making programs work on real computers, this seemed far short of the mark. Some years later, due to a favourable conjunction of stars in the heavens — some very good programmers, the fact that Harvard didn't give Ken tenure and he went to IBM, and that the 360 was available and people thought one ought to be able to do timesharing with it — that conjunction of talented programmers, the availability of Ken, Adin Falkoff, and a few others, and the 360 gave us all APL. And I think we should all realize today that APL is what it is because of such a conjunction. The system under which it ran was so rock stable that people could actually be assured of reasonable continuity of operation at a time when timesharing was an extremely unstable activity; the language, as represented in the first 360 implementation; and a group of ardent disciples who submarined the language outside of IBM — because back in those days, IBM was not anywhere near as receptive as it should have been. But that's been IBM's history all along. Like all large corporations, it has a large amount of post facto wisdom. But that really means that we — particularly those of you who work for IBM — must always be aware of the fact that IBM is amenable to good ideas, but it takes time, lots of time, to overcome the inertia of a Goliath.

Now, to get back to this question of APL as a language. In talking to people, I am constantly amazed — but in retrospect not surprised — that when I talk about APL, it's almost as if we're talking about two different things: their

vision of APL and mine. And when I look at your symposium papers, I find that many of you are concerned with aspects of APL that don't interest me at all, and I'm sure that I'm interested in aspects that probably don't interest you at all. What does that mean? To my mind, it means that APL, as a language, is approaching a kind of completeness that we expect from a good, rich language. There are large, intelligent, useful groups of people who find the language worthwhile and have an intersection of interests in common which is almost null. That's not bad; I think it is good and inevitable. And it points out, I think, that in attempting to arrange for APL's future development, we're probably going to find the task an extraordinarily difficult one. Because those of us who use APL approach it from so many different avenues, and are concerned with so many different points about how the language should change. In a way, we are very fortunate, I think, that the language is as good as it is and is changing so slowly. One thing that those of us in programming have certainly become aware of over the years is that people don't mind programming with a bad language or an incomplete language. As long as they get some useful work from it and some pleasure, they will find any kind of rationalization to support their continuance of that language — even in the presence of much better linguistic vehicles, as witness the fact that FORTRAN thrives. I know practically no one who is interested in programming languages as such who has anything good to say about FORTRAN, other than the fact that its influence and its use continues to grow. That is a very good thing to say about any language. What's happened, of course, with FORTRAN is that it has become the lingua franca of the computing world. It is the one language that everybody understands to some level of detail — it is on every computer, in every country, made by every manufacturer — and one could learn to use FORTRAN reading books at every level of complexity, written in every language on the surface of the earth. It is universal, like the air we breathe, and I don't think it's going to be displaced for a long time to come. And one of the reasons it's not going to be displaced, and perhaps should not be displaced, is that there are always new groups of programmers coming into existence for whom FORTRAN is mother's milk. It isn't going to do us a bit of good to throw before them an APL one-liner that will do as well as 50 lines of FORTRAN. First of all, it will take the people a long time to learn how to use the one-liner and even more to write one, so that FORTRAN will continue to grow and succeed. And I don't think APL will usurp its position; there's no reason why it should. And it certainly shouldn't be a goal of people who use APL to stand forth

and say, "Why do you jackasses use these inferior linguistic vehicles when we have something here that's so precious, so elegant, which gives me so much pleasure? How can you be so blind and so foolish?" That debate you'll never win, and I don't think you ought to try.

Now, there's a great deal of talk today — since APL is such an almost perfect instrument — of making it perfect. We won't succeed in that. For the simple reason that for all of us there are so many different avenues by which APL can reach fruition or perfection. There is no single avenue. Some people say the most important issue at hand is to improve the data structures of APL. Others say what APL needs is a little bit of Franglais, which in our terms is APLGOL. "If APL only had the **while**-statement, or the **if-then-else**, or the **for**-statement, it would become such a perfect language." That's ridiculous. And it's silly to say that if APL had arrays of arrays, all of our troubles would disappears. In point of fact, what will happen is that the amount of troubles would just grow almost exponentially if that happened.

Nevertheless, it is important that one attempt to expand the language along these lines and others. And people will do so, no matter what I say or what you do. If it isn't done within IBM, it will be done elsewhere, because there is tremendous fascination in believing that you can take a language, and by making a few changes today and a few more tomorrow, bring people to the pot of gold at the end of the rainbow. What many of us forget — and we should never forget, of course — is that programming step-by-step must someday, though we don't know how, reach the point where it is universally capable of expressing our thoughts, at least insofar as they involve giving prescriptions on how to do things. Programming is thinking — not all thinking yet, and maybe never all thinking — but insofar as when we sit down at the computer we are faced with so many attractive possibilities which never occurred to us until we programmed, insofar as that happens, we are going to be dissatisfied with the programming languages we have. And that includes every language that's ever been created and those yet to come — and there will be others yet to come. So my view of how APL should alter will differ inevitably from yours. And there's no reason it shouldn't. The things that interest me in APL are not necessarily the things that interest you, and that's what makes the language, I think, so glorious.

As a professor, one of the things I'm interested in doing is teaching people how to program; but it's more than

teaching them how to program, because to teach people how to program, any programming language is sufficient. The idea that only one language or any particular language is critical to learning what it means to program is false. If that is your goal, to teach people how to build programs, BASIC is perfectly satisfactory. There are some things you can't do in it; therefore, you invent constructions for doing them; and the invention of these constructions is learning to program. Sooner or later, in all languages, we have to go to constructions which we build laboriously and arduously out of the components of the language. And we curse the fact that the language doesn't have them already. But they are not all there now, and they never will be there. The word that I associate with programming — and I'm sure everybody associates with programming — is: "frustration". Everything is possible, and nothing is easy. We sit down to write a program, and our initial, beautiful thoughts soon get bogged down in the slime of unavailable constructions. And we write procedures, or functions, or what-have-you, and more functions; we invent data structures, and more data structures; and the programs which start out so nice and elegant soon become mired down and have no structure. We find it difficult to describe to anyone else what they do. And we say, "If I programmed differently, if I used, for example, 'structured programming' this wouldn't happen ... if I had arrays of arrays this wouldn't happen ... if I had a **while**-statement this wouldn't happen ..." Nonsense. If it doesn't happen today, it will happen tomorrow, even with all those things there. Because programming, by its very nature, is a kind of bootstrapping activity. What Iverson and God give you today, you'll find insufficient tomorrow.

So, the quest for the perfect, what I call "spherical language" — the language which grows gracefully, equally, in every direction that is conceived by us today as well as by our offspring tomorrow, and those after that — is unlikely ever to become a reality. And we shouldn't be too impatient, because I think it is true that in any language that you deal with, you can say everything you really want to say about programming. Some of it will always be at the constructive level — outside the primitives of the language. But with APL, there's something extra; at least I find it so.

Some years back, we had a visit at Carnegie from a person at MIT whose name I've forgotten. He started to give us a lecture in a little office about some programming issues in LISP. He went up to the blackboard and he spoke LISP. Everything he wanted to describe, he described in terms of parentheses and CONS and CARS and CDRS. He found

himself quite capable of expressing his ideas in the language in which he programmed. Not once during the half hour or so that he lectured to us did I see the inevitable block diagram, the flow charts that show up on the blackboard with things written in semi-English. He didn't need them. And at the time I said to myself, " LISP has a very precious character, if indeed there are some people who can express programming ideas to other people in the language in which they program." I can't do that with ALGOL; never have I been able to do it with ALGOL. Whenever I've programmed in ALGOL and I've wished to make some statements about the program I was writing, I was forced to go outside the language and use English, or mathematics, or some block diagrams or what-not. In APL, I find that to a far greater degree than any other language that I've used, I can make statements about the programs that I'm writing, in APL — actually not exactly APL, but APL with some nice little extensions that I dream up at the moment but would never think of implementing. But by and large, I find that the language allows me to express myself, in the language, about the things I'm dealing with. I find that a very precious property of a programming language.

A second precious property I've found, with respect to APL, is the term that I've used in that little article that was printed in SIAM News — the word "lyrical". I find that programming APL is fun. It's charming. It's pleasant. I find that programming is no longer a chore, and one of the reasons it's not is the fact that there are always so many choices available to me. Whereas, the people in structured programming tell me if you put enough structure in programs, everybody in the room here will write the same ALGOL program or PASCAL program. Thus, it's going to be easier to read — but also dull.

God made us all different. No two of our minds work exactly alike, and one of the great powers of English is that those of us who learn to sharpen our wits on it, and use it properly, can say things differently from other people. And hence, it's a pleasure to read English when it's written by someone who has that talent. The other day I was reading a newspaper, an article by somebody in the arts who said if Shakespeare were alive today he'd be writing for TV. And I said to myself when I read that, "Not so. If Shakespeare were alive today, he'd be a programmer, and he'd be writing one-liners in APL."

If you take a problem, even a very simple one, and give it to a class of 50 people to program in APL, there's a very good chance that you're going to get 35 to 40 different

solutions. To some, that's a horrible state of affairs. To me it indicates the language really has some power to it, some value; it's just perfect for people, in a sense, to use who like to think originally, if possibly poorly, about things. This variation, this choice which is available, brings to APL programming almost what I would call a literary quality, which I have not been able to find in any other programming language. Now, people always say, "To hell with literary quality, we've got to meet deadlines; we've got to run the program on a computer. If you take 10 minutes instead of 7 minutes, that's three minutes of money that has to be paid" and so forth. That's true, and I'm not saying it's not important. But what I am saying is that when one goes to program a task in APL, at first blush, one has before oneself an enormous number of alternatives — which is not bad, but good — out of which one can satisfy any number of criteria, only one of which, I maintain, is machine efficiency.

When I first learn to program, everybody used assembly language. We programmed in machine code. Looking back at it now, I realize that something went out of programming when FORTRAN came in. What went out of programming was the pleasure, the frustration, the challenge of writing a program that was elegant and clever! There's nothing wrong with being elegant and clever; indeed, in almost every field I know of it is considered to be praiseworthy. I don't know why it shouldn't be in software. When FORTRAN came in, in effect what one did is start the program off with the statement DO or INTEGER I. I like to tell my ALGOL friends that they all have stationery, which they buy in the local stationery store, that has in the upper right-hand corner their name; in the upper left-hand corner the word **begin**; underneath, the word **integer** `i,j`; at the bottom, lower-right hand corner, **end**, followed by a place for the signature. And when one programs in ALGOL, so help me, that template is in your skull. You start off with a **begin**, and you declare `i` and `j` to be integers, and you say, "Now what am I gonna do next?" In APL, what flashes through your mind is a cascade of operations: chasing data through arrays, out of the other end of which come — limping and bruised, you know — seven numbers. After having built up arrays of rank eight and coming perilously close to a workspace full out from the other end comes these seven numbers — and they're pulled out almost painfully — and you say to yourself, "My God, that's wonderful! That's a mechanism!"

I know there are many criticisms of the one-liner syndrome. Phil Abrams has used the phrase "APL Pornography". But

as we all know, being people of the world, pornography thrives! And it thrives not because we're evil, but because we're human. And similarly, one-liners will thrive, no matter what label we stick on them, because the language APL is an invitation to create one-liners, and there is no better way to learn or appreciate that language than to write them. Indeed, APL, I believe, can only be learned by writing one-liners — only by seeing in a sense, what you can compress into a line. After a while, of course, you become more urbane, more weary of the world, and you begin to write little short things like `i=3` and so forth. But at least in the first flush of enthusiam, one appreciates the language best by writing one-liners. When we teach APL, which we've done at Yale since I arrived there, I constantly hammer on the students, "I don't want to see any loops, no **go to**s" — not because the programs they write are efficient this way, because truly one can write in many cases loop programs that run much better than one-liners, but because only by attempting to build these wringers through which you run 10,000 numbers to get two, or five — only by building them do you begin to appreciate the absolutely gorgeous harmony of APL. I want my students to come to me and say, "In this program, which is 5 lines long, I've used every APL primitive, because it fits. There was a place where decode fits, and I used it."

And indeed, we've observed that after a short time, the student programmers begin to feel that every primitive in the language serves, so to speak, an equivalent role. Transpose no longer frightens them — [dyadic] transpose is used all the time. It just fits; all of it fits. Now in changing APL — which there is every indication will happen over the next several years — we must keep, among all things we want to keep in the language, what I call this "spherical harmony". Don't ever let APL become what I've chosen to call "the dumbbell model of a language": where there are two clumps — one here, and one there — that communicate over a narrow pass band; and the programmer, when he writes his program, either spends all his time over here, or all his time over there, with very little time in between. This hasn't happened with our natural language, and it shouldn't happen with APL.

Now, when I've looked at APL, there are things about it I don't like, naturally. No language is perfect. And the things about it I don't like are the things that I and some of my students are trying to change. But they by no means represent either the things of top priority or the things of the greatest intellectual importance in changing APL. That I can't say. And I say right now, no one can say those things

about APL. It's already, as it were, too much a part of the public domain, and too many aspects of APL programs deal with what we might call the commerce of programs. No one can say that by making certain changes and those changes alone, the language will move from its current level up to a higher level.

For example, the first thing that I observed about APL that used to bother me was the fact that when I execute statements, I do create these huge arrays, and three numbers come limping out at the other end. And in the interim, large amounts of storage are used; and therefore, the first thing that occurred to me was that I had to find some way not to have to use that storage. And a very simple solution was found that enabled us to cut down on the amounts of storage drastically. It didn't change the language much, and I don't know that it's going to have any major effect on APL, either.

The second thing that occurred to us was that we need an APL machine, because APL is now running on FORTRAN machines. FORTRAN is the way it is because computers are the way they were when FORTRAN was invented. When you look at FORTRAN, what you find is a language which is ideally suited to the computer that we had in those days: one accumulator; an overflow register for handling the second part of the multiplier, and the remainder in division; and a couple of index registers; a single program counter; and an order code of maybe a hundred or so instructions. And that's what FORTRAN was built to work well on — the IBM 704. APL is a different kettle of fish, because it didn't come from the same development strain. Iverson, when he was working on APL, I believe, — I've never heard it from his lips — was not developing a language which had to fit the computers of that time. I believe he grew up on one of the Marks — Mark IV or Mark III, one of those — and there's very little relationship between APL and those machines. As a result, APL came along and was made to fit the hardware that was available commercially at that time; which hardware, by the way, had come down the path directly along the FORTRAN line. The question then comes up, "We ought to have machines on which APL can be executed much better than on these machines. What should those machines be like?" Well, again, operating on this idea that we are pushing these three numbers through a wringer, what I saw APL as — and what others, I'm sure, have seen it as — is a stream processing language, a language in which one processes streams of data. Consequently, the machine we ought to have is one which is built to execute these streams — from which one could

derive very quickly, relatively easily, a machine design. Such a machine is now being built by a couple of people at Cal Tech this year. Then we'll see if it really is as good as we'd like it to be. It probably won't be!

The third problem that came to our mind about APL was, of course, the fact that virtual memory — which came to us out of operating systems for the express purpose of enabling us to swap programs and page our data with relatively small working sets — virtual memory, which came to us down that path, is not by any means the right way to organize memory for an array language like APL. How should one handle large arrays? Well, it occurred to us, and I'm sure it has occurred to others, that a way of solving this problem is to study the scheduling of data through these APL one-liners — again, using this stream model of the numbers going through the wringer and coming out the other end. In what order are the data needed? And how do we bring them in from backup memory, so that using relatively small buffers we can always have the data in memory that are needed for the execution of the part of the expression which we happen to be executing? We've had some very good results which indicate that using this kind of model, the amount of memory one needs for data storage is a small fraction of what we've become accustomed to thinking is needed in virtual memory machines.

Back of all this development, in my mind and those of my students, has been the fact that we've got to get BASIC out of the public school system. BASIC is really harmful for young people. It's all right for old-timers. But for young people in the 11th and 12th grades of high school, in junior colleges, and in universities, the belief that by writing BASIC programs they are appreciating the beauty of programming at the exercise level that they do, and can do, in the amount of time available to them — that idea is pernicious. It is very dangerous, to boot. We are creating a set of semiliterates, whom you people at IBM, unfortunately, will have to try to retrain. It is very difficult, believe me, to take someone who has been exposed to BASIC — and therefore believes that he now has the key — and teach him how to think in terms of APL; it's just very difficult. It's almost as though you're asking them to turn in one lobe of their brain for another — and they're not prepared to do so. So in the back of our minds has been this problem: How do you get an APL machine at such a low cost that it can be used in the public school systems of this country, so that people can begin to appreciate the lyrical nature of APL? Because we have learned — and I think everybody has learned this —

once you've learned APL, you know BASIC, you know FORTRAN, you know ALGOL, indeed, I think you know all programming languages. You don't know how you know them, but you know them. The mastery of the complexity of putting these things together, these tinker toys — that mastery carries with it implications that cover all programming. Whereas, the low level at which you operate intellectually when you program in BASIC does not allow you to appreciate the aesthetics which are involved when you switch over to APL. Indeed, what it appears to be is one arduous puzzle that you're not prepared to work on.

I've heard from so many people who program in ALGOL — or PASCAL now in the universities — that they are not willing to learn APL because of the threshold investment in their time that it takes. They are not willing to spend the time to think the way I think you have to learn to think if you're going to program in APL — to learn to think, so to speak, of arrays not as mathematical objects but as carriers. Because when you program in APL, one of the things I think you realize is that arrays, by-and-large, are used by you as control carriers through which you blast sequences of operations. And the use of rank is merely a convenient method of carrying small arrays on the backs of larger arrays. It is very difficult to get people to think that way when they've been raised on languages like BASIC and ALGOL, and as a professor, it pains me. But the solution, of course, is not the 5100; it's too expensive. The solution is to come out with some computer at the $1000 or $2000 level, at most, with a reasonable amount of memory and some kind of backup diskettes or what-not on which people can execute APL reasonably efficiently. And to do that is going to require an APL machine. It's going to require some kind of APL compiler, so that you can bind those things you're going to do over and over again, and get them done very fast. And it's going to require some sensible handling of the backup memory.

Those are the problems that we have tried to solve. They are by no means the most important problems in dealing with APL. Because APL is an article in the commerce of our world of programming; and we all know that what that means is people being able to talk to each other, exchange programs, being guaranteed of reasonable efficiency, being guaranteed that the same language will work on a variety of machines — and those things are as important, or even more important, in your world of commerce. But to get people into that world, to get people into that world from universities and high schools, it's imperative that they

learn APL before they become contaminated by BASIC. And I say that very seriously.

Well, there are some other issues that have arisen recently in my mind, and I' m sure that they probably have arisen in your minds, too. We are faced today with an impending revolution in computing. That revolution is, for lack of a better word or phrase, very large-scale integrated circuitry. Now many things have been promised us over the years by the hardware people, and by and large they have delivered them. One thing that I take for granted and that guides me is that hardware drives our field — not software, but hardware. It' s what we are given by the hardware people that determines the set of opportunities available to us. Well, they are going to give us in the next five years or so, on a small piece of real estate, an enormous number of circuit elements — without them knowing at all what those circuit elements should do, how they should be combined, what kind of programming languages will run on those machines — with enormous resources available to them. Well, as you might all guess, APL is just the ideal language, or closer than any other language, for using that real estate. Consequently, one of the problems that we all face, those of us in software, is how we bridge the gap between the software needs that occur at our end of the spectrum, as it were — providing languages that people can use — how we bridge the gap between that and the hardware down at the other end which the physicists and the electrical engineers are going to provide us with — solid-state devices that can do so much potentially, but nothing unless we tell them what they should do. I can' t imagine anything less satisfying than to be given a machine which is capable of so much and find that it' s nothing more or less than the IBM 704 or the IBM 360, replicated on a chip. Not bad, but certainly I think we have higher hopes than that. So that in the years to come, as far as I can see, the major thrust on APL is going to come from that direction — that APL will be a very good candidate for a language to accommodate to this huge increase in real estate that' s going to be available to us on these silicon chips.

Transcription of a talk given by Professor Perlis at the APL' 78 Conference held at Foothill College, Los Altos, CA. on 1978-03-29.

created: 2011-09-19 18:45
updated: 2020-03-15 09:05