The Use of English as a Programming Language

By Jean E. Sammet*

The purpose of this talk is to make a personal plea, backed up by some practical comments, for the use of English or anyone else's natural language as a programming language. This seems to be a suitable subject for the conference, since whatever definition of pragmatics is decided upon, it certainly seems to be tied in with the users of any programming language and what the language means to them.

When I say that I wish to use English—or any natural language—as a programming language, I very definitely include mathematics or any other scientific notation. It should be fairly clear that after having spent most of my time for the past three years trying to get a computer to do formal mathematics, I have not the slightest intention of removing this capability, which we are trying to supply with FORMAC, and others are trying to do with other systems. If someone wants to use the equation "$y = ax^2 + bx + c$" I expect him to be able to write that, and not "y equals ax squared plus bx plus c".

Using English definitely involves the requirement for the computer (or more accurately its programming system) to query the user about any possible ambiguity. Furthermore, if there are numerous possible syntactic interpretations of a sentence, the computer would be expected to use its presumably vast store of information to determine the correct one, and again to query the user if there is any doubt. It is important to note that it should be possible to "learn" the idiosyncrasies of the individuals using it and thus reduce its query time.

There are several reasons *why* I think this is a desirable step. The first is an economic one. Right now, there are too few programmers for the machine capacity we have. I suspect this will continue to be true for a number of years. If one accepts this premise, then there are two ways to cure it. One is to make the professional programmers themselves more productive by giving them better tools—better assemblers, compilers, time-sharing systems, etc. I don't think this will eliminate, or even substantially reduce the bottleneck. Even if it could be made to work, I believe it would be less desirable than the other alternative, which is to eliminate the need for professional programmers for handling applications, and make the computer available to everyone who has a problem he wishes to solve.

It is *not* true that you have to know about the insides of a computer to make it do useful work for you. Even today, very few programmers know anything about the actual hardware, and very few application programmers know anything about the compilers and operating systems they are using. This is completely analagous to the current situation in which people use telephones and drive cars without knowing how they work. If they get into trouble they call for expert help. The main point is that millions of people successfully use equipment whose workings they don't understand at all. Obviously, they must learn what functions the computer is capable of performing, just as people must learn the purpose and usage of different appliances.

The second main reason for wishing to put every person in communication with the computer is to speed up and make easier his ability to get his problem solved. Anybody who has ever done any applications programming knows how hard it is to get the problem defined, and how often it keeps changing. It was for just such reasons (i.e., to permit the person with the problem to write his own program) that systems such as FORTRAN were developed. I simply want to extend this concept to its furthest limits. The only way a person can truly concentrate on his problem and solve it without constraints imposed on him by the communication problem are if he is able to communicate directly with the computer without having to learn some specialized intermediate language.

The argument is sometimes made that programming languages (i.e., formal artificial languages) will force a person to think more logically. I dispute this. I think more time is spent worrying over the format of a DO statement, or a **begin** - **end** bracketing than in whether one should be looping or bracketing at all.

After a person has compiled his ALGOL, FORTRAN, COBOL or other type program enough times, he has removed all the syntactic errors, and only then can he spend his time finding the errors in his logic. If he could have stated the problem in the way most natural to him in the first place, he would have saved a lot of time.

A third reason why this is desirable is the obvious fact that every application area has its own jargon, and the less a user has to jump out of this the better off he is. Only by encompassing all jargon areas—which would mean all of English—can we achieve this.

It is the *how* this is to be done that is of major interest and presents the major problem. As it appears now, the use of true conversational time-sharing systems are a key factor in this activity. The reason for this is based on the philosophy that an individual should be able to converse with a computer in the same way that he converses with another person. If the person does not understand the speaker he asks questions until the point is clarified. Naturally, there is always the danger of unknown misunderstanding. That is, the computer (or the listener) may think he understands the speaker and proceed ahead in the wrong direction. This is a penalty we pay every day for the lack of clarity in natural language. It does not seem any worse to have this happen on a machine, except perhaps some wasted time; however, by the time this proposal is actually operational, machine time will probably be so cheap it won't matter.

There is a major amount of language development that needs to be done. As an oversimplification, there appear to be two ways in which this work can progress. One is the top down approach, which is equivalent to working with as much of natural language as can be handled at a given point in time. This is basically the approach taken at Harvard and Mitre. In this situation, some of the "natural language" sentences may not be understood by the system, but the user is free to use any correct language he wishes. The second approach is what I call the bottom-up approach, which involves the creation of artificial languages which come closer and closer to natural English. The difference here is that each step upward is made with a language which we can translate. Thus, if the user stays within the restricted language he is sure of being understood by the system, but he does not have the freedom of the full range of natural language. Both approaches are worthwhile, and work should continue with them. We ourselves are concentrating on the second approach.

The basic concept of allowing a person to communicate with a computer in his natural language will surely take many many years, and may exceed the lifetime of some of us. This does not mean that it is not a goal worth striving for.

DISCUSSION

Naur: I do not disagree nearly as much as you might have expected. You have adopted my principle that we *could* accept the complete language without restriction. This is one extreme. If your main problem is putting a lot of idle computers in our affluent society to work, then I agree that this would be an excellent scheme (laughter).

Morton: I disagree with the speaker's attitude that the user need have no feeling of knowing what a computer really is. My opinion is not that you have to know about the way it works, but

* IBM, Cambridge, Mass.

that you have to know that it is a computer you are talking to, and not a human.

Computers don't ask "what do you mean by that?" as freely as humans do; they usually make some arbitrary interpretation of what you meant, and continue their business by doing the wrong thing. For example, if your data are on some erasable medium, like magnetic tape, and the interpretation is to destroy them, the computer will go ahead and do it; a human most likely wouldn't do this (laughter).

Sammet: Only one comment. I have had secretaries, and competent ones at that, misunderstand my directions and work for days doing exactly the wrong thing.

Abrahams: It seems to me that programming involves as much the manipulation of the expression of concepts and ideas as it does the words of a particular language. In fact, when you train programmers in FORTRAN, it is not only in how to talk it, but also in how to think in it. This distinction is important.

Another advantage of programming languages we overlook until we try to design new ones is that they provide precise ways to express algorithms; they are concise, we try to make them easy to understand, and the fact that we have evolved them shows that they are not something you just think up off the top of your head.

Finally, let me give an example of a fairly hard problem which I just don't see how I can communicate to a machine, especially if I don't know very much. Suppose I am a chess player and want to tell the program to play chess. I have several problems. For one thing I tend to give the machine lots of advice, and it might even understand every word I said, but it would have no idea of how to incorporate these ideas into instructions. Furthermore, if I gave it a new heuristic, it would have no idea of how to relate it to the rest of the program. If even a programmer doesn't know how to do it, how would I as a chess player understand what made it difficult for the machine to react to my input?

Sammet: With all due respect, I am concerned about getting that work done which normally needs doing in our economic, or perhaps not so economic, society, and I would contend that chess is not this kind of work.

I don't object to conciseness; I wouldn't *require* anyone to use English. I only want them to be *allowed* to use English, if they prefer it, to a language you choose to call concise.

As for the question of thinking in FORTRAN rather than writing in it, I do not consider such a question germane. Having to think through a problem remains with us whether we use FORTRAN or English. Whether this is good or bad, it is what happens.

Young: First, I believe the idea of natural language communication is fine, but there are two things which must be kept in mind. One is that what is normally associated with programming, however it is done, is less of an expenditure than the systems analysis preceding it, or the running of the problem, which follows. The second is basically a problem of knowing how to organize your program so that the facilities will be used efficiently. This is the task which you are asking to be transferred from the programmer to the system itself; I leave it as an exercise for the programmer (laughter).

Green: We don't have to know exactly how the machine operates; but we must have some model in mind of how what we want will be achieved. Otherwise we cannot intelligently sequence the orders necessary to achieve what we want.

For example, my original image of the 650 was a kind of Disneyland with drums revolving like Ferris-wheels, and trolleys carrying marbles around, but I had a model of a mechanism in mind and I designed my programs by visualizing the changes of state the sequence of operations were creating in that mechanism. Now, I think the language itself is a very minor factor compared to the need to have an understanding of some model of a mechanism whose states we want to change.

Sammet: I am not trying to eliminate a person's understanding of what a computer is capable of doing; I am trying to eliminate his having some very artificial way of communicating to the computer what he wants it to do and knows it can do. The main reason I made such a strong point was that many people imply that you must know what the flip-flops and gates are doing. I think that is silly.

Orchard-Hays: While I have a great deal of admiration for such a project, of such a magnitude, as undertaking to make English available, and while I think a great many good ideas can come out of such a thing, I think some of the arguments in support of it are rather far-fetched. First of all, saying that the bottleneck in computing is the lack of programmers is like saying that the bottleneck in building cities is the lack of people. Then there is an old adage that "a problem well-stated is half-solved." Many of the problems we are undertaking are difficult, and until we understand them we will not solve them.

Putting all the people in the United States in contact with computers will not help.

Strachey: I believe you've overlooked the absolutely central importance of notation in the history of the development of mathematics. You'll find a dramatic difficulty in trying to do our arithmetic in Roman numerals; this shows that more than a mere notation is embodied in arithmetic and Arabic symbols.

To quote Whitehead in one of his discussions on the development of mathematics: "It is a well-known and profoundly untrue adage that we should cultivate the habit of thinking about what we do; quite the reverse is the case. Most of the advances of mathematics have been made by relegating difficult and complicated operations to the mechanical applications of rules (like multiplication, division, and using a computer). These operations are like cavalry charges in a battle. They are extremely expensive and should be reserved for crucial moments." (laughter)

Nixon: English is not a language. It is a family of languages. We speak one English in our homes, mathematician's English to mathematicians, and programmers' English here. The reason we use all these languages is that a natural language is extremely flexible, but so imprecise that it is *too* flexible. If English is used, we will have to define each term time and time again until those meanings we want at the moment are understood.

Kurki-Suonio: The main reason I find it convenient to use English words and phrases for strictly defined programming purposes is that English is *not* my native language (laughter). English provides a huge store of symbols to which any meaning can be assigned (more laughter) without referring to my former knowledge.

Burrows: We [at Mitre] are working on some of these areas from the top down. I think also, that voice is going to be essential because no one will be willing to use a typewriter to communicate in the verbose manner of English.

Zemanek: Shannon has pointed out that the normal English sentence has about 75 percent redundancy. Humans make good use of the redundancy; whether a computer can is another matter.

Floyd: It may well be that within a few years linguists will emerge with genuinely operational syntaxes of substantial fragments of English; so far they are not close to this goal. Even when this is done, there is still the problem of setting up effective correspondences between the meanings of English sentences and the sequences of operations needed. This is intrinsically more complicated. Phrase structures bring compactness into programming languages by leaving close together the elements which are related syntactically; but this is not the whole story in deriving meanings of English sentences and transforming them into operations in programs; there is a meaning structure left which is almost unrelated to the phrase structure. But even after that is mastered, there is a kind of beyond-the-COBOL English you would not consider to be free association with the computer. For example, you might want to say "convert payroll file", or "balance the books", or "enter the number in the little square."

I am not disparaging the idea of using English; rather I would point to the tremendous knowledge of the world, including the knowledge of the temporal change of knowledge, which is built into English words. And each word must be implemented by a procedure which somehow contains its meaning and consistently interlocks with other procedures for other words. I can't think of any task, intellectual or not, which has ever been carried out which approaches this magnitude.

Yngve: As programming languages have developed, they have encompassed more and more features of natural languages. This is an historical trend, and, by and large, a good one, mainly because it makes the programming language seem more natural and easier to learn.

Another point in favor of supporting this trend is the following. You talk to a child of two in English; a foreigner, who also has a smaller knowledge of the language, can talk to you in English. A feature of natural language is that it can accommodate to such varying intersections of knowledge among communicants. This feature should also be in programming languages.

I essentially agree with the paper just presented. Most of the objections raised can be answered, and many are not satisified with such answers only because they are afraid of the time-scale and difficulties implied. But it is clear that Jean has thought of these objections; she did say it would take a long time, and is difficult. Maybe we only think so because we don't know how to do it; it may turn out to be not so hard.

Microprogramming, Emulators and Programming Languages

By Julien Green*

The problem we have been concerned with is that of converting language to action—or intellectual energy to mechanical energy. The medium that we use for this purpose is language and therefore we are preoccupied with the subject of language. In the areas of language investigation we have concentrated first on formalizing syntax and then on semantics. I believe semantics has received an unfair share of the energy of too many people—mainly misdirected to trying to reduce semantics to an algebraic manipulation of symbols, without enough attention to reality. By reality I mean the devices or mechanisms which are going to use these symbols. We have been overly concerned with what the symbols mean to one class of users, namely, the human or the senders of the symbols, without sufficient concern about the receivers of the symbols, in this case the machines.

Perhaps it is time we devoted a little more effort to understanding how the machines treat the symbols. The relationship between the machines and languages should have been a major topic of discussion at this conference. As a matter of fact, I consider the needs expressed in some of the earlier sessions as a clear indication that we have really delayed too long in devoting our attention to this area of pragmatics.

I would particularly like to point out one area in computer systems which is becoming increasingly important and which requires the type of theoretical understanding I have pointed to in order to be properly developed in the future. This area is microprogramming, and its chief utilization at this time is in the production of the machine language interpreters.

I will first describe the situation as it exists at this time. The IBM family of machines called series 360 and the RCA Spectra 70 series all accept essentially the same machine language. Let us examine how this is accomplished. In general, a computer has a set of built-in operations quite different than those in the "machine language instruction", but there exists an interpretive system in each computer which bridges this gap. The elements of machine language instruction interpretation are the data linkage, the control linkage to determine which of the built-in operations are performed, and the sequencing to the next instruction. Usually the machine language is a design compromise between the interpretive system and the built-in operations. What we have always had has been a machine with built-in operations to change its state and an interpretive system to provide a more useful language in which to describe a desired change of state. Another interesting facet of the present situation is the development of read-only storage to the point where it is now practical to program the interpretive system, or a large part of it, in read-only storage. In any case what you now have is a series of different machines in which the interpretive systems have all been designed to handle the same language. It is important to stress that it is the interpretive system in these different machines which allows them to accept a compatible language.

The interesting ramification of the read-only storage microprogramming is that it is simpler to produce a new interpretive system, and thus you see the development of emulators. An emulator is a system for simulating one machine on another where at least part of the simulation is performed by replacement of or addition to the machine's interpretive system. A simulator is usually understood to be an interpretive system written in the language of the machine performing the simulation. What I am trying to say should make it clear that in a sense the model 30 of the IBM 360 series is really a 360 emulator in the same sense that additional read only storage with another interpretive system enables it to be a 1401 emulator.

Let us explore this relationship between machine and language a little further. Just consider the case where a system 360 or Spectra 70 machine has a program which is an IBM 7090 simulator. In turn there may be a 7090 program which is a 1401 or 1620 simulator. There is, for example, a 1620 interpretive system called COGO. Now consider the possibility of running the whole sequence which I have described, namely a COGO program by an interpretive system in 1620 machine language, which is being interpreted by a 7090 program, which is being interpreted by a 360 program, which is being interpreted by a program in the read only storage of a particular machine. And that is as far as I can carry it at the moment because I have not determined how the device of using an interpretive system is ultimately terminated. In any case, for the whole structure which I have postulated, what is the language? What is the machine? Where is the boundary between hardware and software?

To try and clarify this subject and to set a background for meaningful discussion I would like to give a set of definitions. The attempt here will be to clarify the relationship between language and machine.

1. Language—a set of symbols with the rules for the formation of meaningful sequences of these symbols.
2. Automation—a mechanism which has a state and the property of being able to change state.
3. Operation—a sequence of symbols in the language describing a change of state.
4. Interpreter—the mechanism within the automation which affects the change of state defined by an operation.
5. State—explicitly, the value of all objects which are the subject of discourse of a particular algorithm and implicitly the value of the objects used by the interpreter which are not available as objects to the language.
6. Algorithm—a sequence of operations describing a change of state in the automation.
7. Microprogramming—programming the interpreter.

* National Computer Analysts, Inc., Princeton, N.J.