A New Approach to the Programming Problem

William Orchard-Hays
C-E-I-R, Inc.
Arlington, Virginia.

The computing and data processing field as we know it today, that is, employing high-speed EDP equipment, although but a few years old, has grown accustomed to major advances, changes and reworkings of systems in quick succession. However, the increasing complexity of programming and of applications, together with the rapid appearance of several powerful hardware systems, has now reached such proportions as to cause considerable uneasiness and confusion. It appears sometimes that the effective lead time for preparing for a new machine has not only gone to zero but become negative. As to problems, it has become a cliche' that they are always wanted "yesterday".

It would be hard to deny that the programming profession has brought some of this on itself although the total problem is much broader than what is usually considered programming. It appears sometimes that the equipment users have not kept pace with the equipment makers in imagination, sensible distribution of effort and just plain pride of workmanship. Yet, a realistic evaluation of the work done on large machines in the past three or four years will elicit admiration from anyone capable of understanding it. A realistic evaluation of the work done on large machines in the past three or four years will elicit admiration from anyone capable of understanding it. But mingled with the understanding admiration is likely to be a trace of fear that man cannot yet effectively utilise the powerful machines which he is building. Some of the most ambitious, thoroughly planned and well funded projects have been little short of fiascoes. And machines five times as powerful are on the threshold, with other machines twenty times as powerful as that following.

A brief look backward is sometimes helpful. In earlier days of machine computing, say as far back as 1952-3, there was a keen interest in getting jobs done and a lively anticipation of doing these jobs better on the equipment soon to be available. The dark ages of the 602 and 604 had given way to the transitional CPC with occasional excursions on the EDVAC, SWAC or UNIVAC. People were steeped in the care and precision of EAM procedures, still believed in the doctrine of economy of space and time, and still accepted a challenge to do some operation in a better way. Machines were looked on as tools, and it was assumed that people would be required to make effective use of them -- people who understood them. It was also accepted that computing departments existed to serve some higher purpose and that it was the function of the programmer or machine specialist to provide means to make machines useful to those with problems.

It would be hard to say that these notions have ever been deliberately scrapped but, somehow, they no longer seem to describe current attitudes. Perhaps it is just that the problems have gotten harder faster than we have gotten smarter. We can hardly be accused of idleness. Programs have proliferated in ever increasing sizes until today one person can scarcely be aware of all the routines available for one computer, let alone understand their use. At the same time, the number of new jobs presented which require original programming has grown at least as fast. This in itself, of course, is encouraging; we do not wish to work ourselves out of a job. But neither, I think, do we wish to attempt to re-define every field of technology. I am aware that much lip service, and much real effort, is devoted to providing better programming aids and "automatic" systems, presumably for those who have a need to exploit machines but who have neither time nor inclination to become professional programmers. Few would claim an outstanding success here. Perhaps the trouble is that we are attempting to bridge an ever-widening gap with the same double step that was effective in the days of the CPC, namely the general programmer and the system programmer.

The standard solution to the problem of "getting on the air" with a big project has been to proliferate ever larger programming groups - by hiring, training, renting, pirating or any other means including dumping a bunch of people in to sink or swim. This method is becoming increasingly ineffective. A large number of people doing a mediocre job only adds to management's problems. This is not to say that the people are mediocre. Rather they often lack direction.
However, that there is a definite limit to the number of potentially top-flight professional programmers - dedicated, intensely interested, highly skilled and capable of creative work - is apparent to anyone in the industry. This limit is indeed small, I would guess in the hundreds.

This problem has not gone unnoticed, of course. Besides a number of elaborate processors and operating systems, there has been talk of a universal problem-oriented language, of a universal machine-oriented language, and even of some kind of a universal language for both purposes. People evidently take these seriously since there are many active committees in existence. But just what meaning do such phrases have?

Let us look at the problem-oriented language first. Before we can begin, we must assume that it is somehow possible to foresee every kind of term that will ever be used and every kind of action that will ever be taken in defining and solving a problem. We can, you may say, assume synonyms and isomorphisms so that it is not necessary to pre-define every actual combination of characters and symbols. But this assumption implies a recognition and/or memory capability which far exceeds anything we now know outside of the human brain. Moreover, there is the deeper implication that all classes of quantities and operations are now known. To say that the operation of defining an operation will be permitted is only evading the issue since this implies that we know how many innovated operations will interact with all existing operations. Given any defined language, it is not very difficult to show that there are reasonable problems which may require instructions, i.e. notions, outside the language. Hence no problem language can be universal.

Similar arguments can be made regarding a machine-oriented language. Consider a simple example. One of the tasks to which such a language must be adequate is the translation of any program to a form usable on a machine and, at the same time, responsive to the needs of the problem formulator. Suppose this person is studying improved methods of handling his problems and wishes to determine how much information is required in translating a program. Then he must be able to write a program which outputs information about every action of the processor. Hence the processor must accept modifications to and interruptions of itself. But then it is no longer merely the processor of the original language. Hence he must be using a more extensive language.

There are other and more powerful arguments against the feasibility of a universal machine-oriented language. This is betrayed by the fact that such supposed languages have a bad habit of turning into rather specific problem-oriented languages where the problem is to describe a machine, whereas what is desired is the ability to use a machine. Yet the notion of a universal machine-oriented language system does have meaning, it seems to me, in a somewhat different sense.

In order to develop this argument, it is necessary to consider a few basic principles. I will restrict myself to four, all of which have to do with the dynamic nature of computing.

1. The introduction of new ideas and new applications is an inherent part of EDP technology, both in the manufacture and in the use of equipment. Some of these new ideas and applications have unanticipated but far-reaching logical implications.

It seems to me that a failure to recognize this principle has caused us to get farther and farther behind. We are forever coming up with excellent solutions to problems which we muddled through a year previously, while at the same time trying to cope with current problems which have new requirements. Furthermore, we should certainly be aware by now that every major procedure, when perfected, is looked on by the analyst as an available technique or operation and that he will seek new degrees of complexity in his next project. No fruitful line of development is ever complete and very few computer program systems will remain static if they are found useful. This is verified by many years of consistent experience in nearly all installations. Compatibility is a greatly overrated virtue. It is desired most urgently when there is a need to reprogram, say for a new machine, a horrendous complex of codes which we have lost control of and no longer understand. We don’t really want the compatibility, which stifles advancing the art, but rather we fear to expose the muddle into which our own work has gotten.

2. Since the handling of information involves the generation and manipulation of information about information, the growth of data handling requirements is self-compounding unless carefully constrained.
This principle is insidious, but true. It might be called the Parkinson's Law of data processing. It is interesting to note that human beings, in doing clerical work or calculations, impose the necessary constraints in the form of what we call common sense and judgement. The dullest sort of person is able to exhibit these qualities in far greater measure than we often admit, but the brightest genius has difficulty in precisely defining them. Which leads to the third principle.

3. Nothing can be mechanized which cannot be precisely defined. This is as true of second, third and higher orders of control as of primary mechanization.

I have come to the conclusion that many people who ought to know better don't really believe this principle. Sometimes I try to ignore it myself, always to my eventual grief. It is too tempting to solve easy problems, pushing the hard ones back with a sort of mental note that "we'll do something" when the time comes. If pressed we may mumble something about statistical methods, which sounds impressive. Now I would not want to appear to minimize the worth of faith, self-confidence, and drive. But it is foolish to undertake a piece of work without having first estimated the effort required. In data processing, we have sometimes been trapped into doing this because of a failure to understand the true complexity of what we are asking for. Thus complex, far-reaching and important decisions and plans are left to those who are unqualified to make them, technically, administratively or policy-wise.

The inability to precisely define all aspects of a technology does not bother us too greatly in most endeavors. A car still requires a driver, a plane a pilot. We think of an assembly line as highly mechanized but it works only because many people are working on it and making little spot judgements all the time, sometimes big ones. But who can work at the rate of 40,000 additions per second, even in a supervisory capacity? Of course, the very essence of the stored-program concept is to allow complete pre-planning, which amounts to canned supervision, of the whole sequence of operations required by some process. There are two catches to this. The first is our second principle, i.e. the bookkeeping soon outweights the data processing operations and becomes a bigger data processing problem than the original one. The second catch is that it is impossible to pre-plan the proper action for every eventuality; in fact, it is questionable whether the term "proper" has any meaning in this context except as dictated by particular policy. All this is simply a roundabout way of saying that human beings cannot be taken out of the daily activities for which they are responsible and which exist in the first place by their decision. The introduction of new machinery - no matter how revolutionary, powerful or potentially useful - does not change this basic fact of life. Rather than relieving people of effort or requiring less in the way of disciplined endeavors, this new equipment is more demanding if its use is to be profitable.

Let us admit boldly, or humbly if you prefer, that there is a lot of confusion in data processing work today. Let us further admit that there is a great deal of difficulty and uncertainty in the projects to which DP equipment is often applied. If there weren't, most of us would be out of work. Now while it may be possible to cut down on the confusion, it appears realistic to me to assume that the problem difficulties will increase, certainly not decrease. These problems are not the exclusive domain of any one discipline. Certainly they do not all belong to the field of programming. Programming has its share of tough problems and also its share of skill and helpful techniques to be contributed. What makes the programmer invaluable is that he knows how to make a computer perform. What makes him inadequate is that he doesn't always have the script. After all, problems originate in people's minds and techniques of operation reside in other people's minds. We need to remember, and to believe, that the really valuable residue of a completed job consists of the ideas, insights and techniques recorded in human brains, in short the experience gained. This leads to the fourth principle.

4. A wide variety of skills are involved in present-day EDP applications thus multiplying the problems of communication.

Communication evidently means different things to different people -- which seems odd. But one must recognize that vast efforts are underway to increase communication between systems, sub-systems, or just plain code and systems. It is not clear to me that communication between inanimate objects can exist but I admit my knowledge of psychology is limited. At any rate, I am more concerned with communication between humans and by humans with machines.
Now there are at least three important conditions for communication:

1. The parties concerned must have rapport in the area under discussion;
2. The quantity and rate of information sent at one time must be within the capacity of the receiver to assimilate;
3. The language and medium used must be known and compatible to both parties.

A lack of any of these can be compensated for by sufficient determination, repetition, patience and time. In EDP, however, efficiency must be given great weight. We cannot always afford to compensate for poor communication.

What should the system programmer be doing? He should not be building systems for himself. He should be providing standard parts and routines with which systems can be built. Of course, his consultation will be needed but the whole system should be literally built by the whole spectrum of people who are going to rely on it. How can the programmer anticipate the kind of system and languages that the next project will require? He must be prepared to build any definable system or at least to supervise its building. That is his specialty. The same thing is true of language. What we need is not a universal language but the ability to define and utilize any language that suits our purposes.

Consider a moment. Systems of 10,000 - 15,000 instructions are routine, 25,000 - 50,000 instructions are not uncommon. Some plans now underway will likely require 50,000 - 75,000 instructions. Where do we stop? Some of these systems are supposed to be all things to all people but on closer examination it turns out that they generally do about one thing well. Maybe they compile algebraic statements, or maybe they assemble and execute handwritten programs or maybe they perform certain standard D.P. functions. But they are only straight-line cuts out into a whole field to be harvested. Furthermore, they are difficult to understand and require a staff of specialists to service each one. Far from facilitating communication, they vastly complicate it.

Suppose then we begin by trying to extract from all systems those parts which are common. We will, without question, have to establish certain rules and conventions, for example, the card form for instructions to a monitor (any monitor) and the first and last words of each tape record, etc. In other words, we have to establish standards. But these standards must not be restrictive to the user. Their only purpose is to allow standardized parts from which any system can be started. We must not forget that knowledgeable people must be integrated with systems if versatility is to be maintained.
In addition to a package of standard parts, there is something more needed. This is a standard philosophy of using them. There is a somewhat subtle distinction here. We do not want to make all systems be just alike, otherwise we may as well program a standard system. The idea is to have standard techniques to cause a system to grow in whatever way is needed at the time. The word "grow" is not used lightly. Certain characteristics of an organism must be provided. To me, the notion of a universal computer-oriented language has meaning only in this dynamic sense.

There is one further requirement. The method must not lead to unnecessarily cumbersome systems, that is, this must not be inherent. With the present state of the art, at least, this will require some rather stringent conventions regarding symbology. Hopefully, they can be relaxed as experience is gained. I do not see this as a great problem, however. People are pretty good at picking up rules of this kind and at getting used to cryptic notations. They may grumble but their real problems usually lie elsewhere. If they understand the structure of the system and its attendant language, the actual encoding is not too important.

What are some of the features that might be expected to be available with such a method? I think the following is a minimum list.

(1) A core of highly polished utility subroutines, a standard skeleton monitor, and a basic assembler, all capable of expansion in well-defined ways and maintained by a single small group (per installation) of highly skilled professional programmers.

(2) The capability of defining hierarchal sets of macro-operations tailored to the job and understood to the depth required by the various echelons or categories of professional people involved. Since they would be involved in defining them, they should easily understand them.

(3) The ability to make changes at any level provided that at least the next lower level were understood, and to add levels as need dictates.

(4) The ability to debug in detail at object time without detailed pre-planning.

(5) Complete revision of a system by revising definitions and restarting from the basic system, with a minimum of elapsed time.

(6) Partially automatic documentation of a system since higher levels of operations will require fewer lines of programming.

(7) Ability to experiment with systems as such, and to extract production systems at will.

(8) Alleviation of the training problem by spreading the load now imposed on programming over a wider variety of skills.

Let me re-emphasize that such a technique as just outlined is going to be very demanding of the people using it. Responsibility is a necessary counterpart to responsiveness. The ability to exercise higher level control implies the ability to make bigger, more disastrous errors. With proper safeguards, however, this is somewhat compensated for by the ability to quickly reconstruct a previous situation.

To sum up the foregoing very briefly, no one can become expert in many fields simultaneously. Programming must provide access to machines for others, via the programmer but not by the programmer on every occasion. To expect to find individuals who can understand all the goals, restraints, techniques and policies of a large project and still be capable of translating these into a mechanized system is unrealistic. To hope that a system can be defined by one group and turned over to another for implementing is just as unrealistic. Everyone must play his proper role from the outset and communication must exist between members of a team, at least by pairs. After all, systems exist to serve some human purpose, not to be self-sufficient automatons. The trick for making the hardware a useful member of a team is to construct systems which are self-adaptive, but externally responsive in a problem oriented sense. The know-how to accomplish this exists.

As will be seen, there are implications of so-called self-organizing systems here. I believe that the reduced cost per computation, promised by the bigger, faster and more expensive machines around the corner, can only be achieved in practice by some such innovation. This is
not, however, a new idea, as suggested at the outset. It is simply the old idea of breaking down a technology grown complex into simpler pieces understandable separately by a wider cross-section of the population. The skills so highly developed and so cherished by the expert programmer are as needed and as important as ever. But his field has also become so specialized that he must concentrate on particular areas to be effective. He must use his skill to open up new facilities to others so that they too can work more effectively. Only by a multi-link chain of people between problem and machine can the growth of EDP be kept on a workable basis such as gave it its initial impetus.