Computer Science Department, Courant Institute

SETL Newsletter #33 May 15, 1971

What is Programming?

J. T. Schwartz

I will elaborate a series of answers to this pregnant question.

I. To start with, programming is the activity which builds the interface between man on the one hand, and computers on the other. Certain of its characteristics will then be determined by man, others by the computer. The goal of programming is the construction of advanced function. This requires the perfection of complex programs. Therefore

II. Programming is the process of constructing complex objects. In a previous newsletter, certain basic laws affecting such processes of construction were outlined. To repeat: compound objects are built by successive correct choices of a sequence of elements $E_1,\ldots,E_n$. Each element $E_j$ must be chosen in a logical context summarizing all those aspects of other elements which are relevant to the choice of $E_j$. We call the collection of all these influences the local context of $E_j$, and call any reasonable numerical measure of this collection the context complexity of $E_j$. It may then be observed that the chance of choosing $E_j$ correctly falls off very rapidly as its context complexity increases, and effectively becomes zero at a not-very-large threshold T. This observation allows us to define the class of constructible objects: an object is constructible if it can be built by choosing elements successively, each in a context of complexity less than T. A function is programmable if it can be realized by a program which is constructible.

To construct a large object successfully, one must therefore combine many subelements. The rules according to which elements may be combined are of course part of the logical context of every element. These rules must therefore be simple. But a simple set of rules allowing the indefinitely iterated combination of simple elements into a large totality defines some sort of "algebra". Therefore

III. Programming constructs compound objects from simpler elements by combining elements according to the rules of some "algebra".

In order to program, therefore, one must be aware of some such algebra, which must be capable of generating objects representing useful processes. Before they can be used, such algebras must be found. We conclude therefore that in a deeper sense

IV. Programming is the discovery of algebraic principles allowing the iterated combination of elements into compound objects representing useful processes.

Next, observe that, although the maximum threshold T of tolerable complexity postulated above will vary from person to person, for no one is it very large. In this regard a group of people is no better than a single person. Therefore, an object not constructible in the above sense can really never be constructed directly, either by individuals or by large teams. And it is very unlikely that such an object will be formed spontaneously by the action of a random process, even if this process acts repeatedly over long periods of time. Objects irreducibly unconstructible must therefore remain nonexistent. The barrier to their existence should be as firm as those set for mathematics by theorems of the type of Gödel.

There is, however, a way in which we can hope to find a way around the obstacle revealed by these pessimistic reflections. To see this, observe that the maximum context complexity of the elements of a compound object is by no means independent of the representation of the object. What in one representation may appear as a densely interconnected mass will in another representation appear as an object, perhaps still large, but consisting construcibly of items no group of which are impenetrably related.

To discover this second representation of a programming problem is to break the problem's back, since this discovery allows one to build what formerly were obscurely integral objects using systematic incremental techniques, that is, to proceed by the progressive accumulation of tables of information possessing no overwhelming degree of internal interconnectedness. In a still higher sense, therefore,

V. Programming is the discovery of viewpoints or logical transformations which uncover hidden algebras in terms of which compound objects representing useful processes may be built. That is, programming is simplication, and like mathematics, is a hunt for lucky simplifications.

It is worth emphasizing that to discover these simplifications is the essential goal of experimental, as distinct from applied, programming. If in a strictly research situation we build a highly compound object, we do so only in the hope that immersion in the realities of a particular construction process may put us in mind of principles allowing this process to be simplified.

The transformation of a constructible compound object into that more highly interwoven form in which it directly represents some interesting function plainly amounts to a kind of compilation. (The practical possibility of carrying out such transformations is of course the contribution of the machine to the process of programming, which, in the preceding remarks, we have viewed almost exclusively from the human side of the man-machine interface.) We may therefore say that

VI. Programming is the discovery of algebras allowing the construction of objects worth compiling, and is the programming of compilers for these objects.

Elements which programmers are to combine need to be simple externally. But, as long as their internal complexity can be hidden, they need not be simple internally. Indeed, when objects having simple external description but themselves embodying powerful function can be allowed within an organized algebra, the programmer's reach is multiplied. Hence

VII. <u>Programming is the discovery of highly functional logical entity types possessing simple external descriptions and thus capable of being integrated into an algebra useful for the construction of still higher functions; and is the discovery of the 'internal' algebras which allow the construction of entities of these types</u>.

The above remarks predicate an indirect method for creating functioning machine-level process representations. Our reflections concerning context complexity suggest that in the construction of highly compound objects such an indirect approach is inevitable. However, since this approach is, to begin with, fixed upon simplification and standardization as goals, in following it we run the risk of ignoring alternative constructions which might realize a given function in a particularly efficient way. Efficiency-oriented departures from a standardized approach are traditionally the perogative of skilled human programmers. The mind, ranging analytically, can incorporate very useful variations into a basic approach; as long, that is, as the additional complications which such departures cause do not carry one over the threshold T of allowable context complexity. The programming range which we contemplate will however involve transformations of form so repeated and elaborate as to exclude the possibility of external meddling with the compiled versions of objects. Given that we will have to allow efficiency-enhancing variations to enter into the compilation process, it follows that in the programming range we contemplate it will be found necessary to systematize these variations, and to build a <u>program</u> capable of weaving them into the compiled version of an initial text. Such a program must of course be able to analyze programs in sophisticated global ways. The programmer may assist this optimizer by adding, to a text to be compiled, disjointed declarations which summarize and transmit significant conclusions concerning the text, but his role may not safely be allowed to exceed this limit. We may in this regard say that

VIII. Programming is optimization, i.e., is the programming of optimizers able to analyze and improve other programs, and is the discovery of principles which allow the simplification of such optimizers.

The use of the indirect technique suggested above, involving the optimizing compilation of sequences of constructible objects, will eventually allow functions to be programmed which lie utterly beyond the scope of more primitive direct methods. Nevertheless, just as Gödel's theorem assures us that certain rather simple questions lie quite out of the range which the method of mathematical proof can reach, so we may also take it that certain functions which might be of great use are not programmable, in that no constructible object can represent them, even after compilation. It is therefore of interest to consider whether the construction of artificial intelligences is at all possible. Might it not be that, among all those objects constructible within the maximum complexity threshold T of the human mind, none exists which can represent all the capacities of the mind?

In coming to grips with this question, one must first of all realize that it concerns innate and not learned capacities. That which is learned is drawn from an accumulation of separately encountered facts, presented in no particular order or relationship. No inextricably interwoven object is immediately represented in the pile of fragments presented as input to the learning process. If facts within the mind are interwoven in uncompilably complex ways, they can be so only because the mind is innately capable of establishing exceedingly complex connections. If the ability to learn can be programmed, the teaching process will be trivial. That which we seek to duplicate is therefore as fully present in the neolithic savage as in the savant.

But might not this innate facility, in spite of the somewhat restrictive definition which the above remarks give it, still be unprogrammable? It might. But I doubt that it is. Hard evidence in this area is still missing. To argue from what has not been done, or from the collapse of inflated initial projections, is an absurdity, given that the computer is still less than twenty-five years old. It seems to me that the fragmentary evidence which does exist ought to incline one rather strongly against such arguments. Substantial progress toward the programming of mental function has been made in a few cases. For example, the parser-compiler type of program captures a striking part of the ability to learn languages. Note that, in accordance with the general principles stated above, it is the discovery of an underlying algebra, specifically the algebra of pattern combination in the manner embodied in BNF grammars, which enables us to construct such programs.

One may conjecture that mental faculties which, like the ability to learn languages, are generalized and involve explicit learning will prove to be more readily constructible than faculties, such as visual pattern analysis, which are more rigidly fixed. Learning at the level of language learning is surely of late evolutionary arrival, and one may therefore surmise that this faculty has not had the time to grow as complex as have others. In view of the general pattern which evolution exhibits in regard to physical organs, we may take another hint from this observation. Speech and higher reasoning, rapidly evolved, may possibly employ specially adapted versions of faculties which antedate them. If this is true, then successful duplication of the mind's language-handling faculty may provide clues valuable for the analysis of still other mental functions.

The optimistic remarks of the preceding paragraph, if they can be trusted, lead one to try putting the question of artificial intelligence quantitatively. The programmability of a complex function is, as we have seen above, defined by the battery of simplifying transformations which determine one's programming technique. How many as yet undiscovered simplification principles remain to be found before artificial intelligences will, in this sense, become programmable? If and when these principles become available, how large a body of compilable text will be required to define the intelligence? I emphasize again that the text in question is that which organizes the intelligence's capacity to learn, not that possibly larger body of text which defines the total mass of facts available to it. That is, an intelligence is defined by those highly integral programs which determine the principles according to which it organizes more disjointed information tables subsequently fed to it. It would be rash to try to answer the questions just raised. Nevertheless, putting them serves, when one notes the extent to which a simple yet well organized programming system like LISP makes it possible to define quite striking language processing faculties by quite a small body of text, to buttress optimism. Putting these questions also serves to emphasize the central importance, for the eventual construction of artificial intelligences, of progress in programming technique. They also tell us what to look for: transformations which allow originally integral functions to be represented incrementally and in this sense to become learnable. Thus, for example, we may recognize that the organization of at least part of the language-analysis function around an explicit Backus algebra of syntactic patterns is a very significant step, the sort of thing that we must energetically seek to extend. Other functions can be cited for which organizing 'algebras' are desirable and might be possible. An associational 'feature noticing' function of a generalized sort would be useful in a wide variety of situations, for example in optimization by the method of 'special cases', where such a mechanism might permit the easy addition of new optimizations. At a more technical level, a language of memory management, allowing certain central problems of concrete algorithmics to be treated systematically, could enhance our ability to produce efficient versions of concrete algorithms rapidly.

In connection with this last remark we may raise yet another quantitative question concerning artificial intelligence. The capacity of an intelligence is measured both by the level of function which its responses embody and by the speed with which these responses can be generated. Assuming that it becomes possible to construct an intelligence, how fast will this intelligence be able to think? This question touches upon all those questions of efficiency which the concentration on abstract programming issues characterizing our preceding remarks has caused us to neglect. Its answer will of course be determined both by the basic capacities of the hardware available at a future date, and by the extent to which optimization is able to overcome the natural tendency to inefficiency of a highly compiled programming style. Till now, almost all the most dramatic increases in program speed have come from basic hardware speed-ups. In a few cases, as with the development of the fast Fourier transform, fast sorts, hashing and list-organized search techniques, and in the imporvement of certain little-used combinatorial algorithms programming has made similar contributions to efficiency. The domination of efficiency by hardware should continue for at least a while longer, as clock-cycles diminish twoard 10 nanoseconds, and especially as improved manufacturing processes weaken the I/O barrier by making greatly expanded electronic memories available. In this regard programming may for a while have the largely subsidiary role of choosing algorithms that bypass potential combinatorial disasters. A more systematic, but perhaps less immediately significant contribution of programming to efficiency will probably come through the continued development of optimization methods, especially those which, like cross-subroutine optimizations, aim at preventing the efficiency losses which a naive and highly compiled programming technique would imply.

Efficiency loss through the use of such techniques is in fact far from being a crucial problem. It has generally been true that, once able to organize a given programming area clearly, one has also been able to invent systematic optimizations which permit indirect programming techniques attain an efficiency comparing not badly with the results obtained by the use of much more expensive and eventually quite impractical manual techniques. In regard to the programming of intelligences, it may also be remarked that, once we are able to create a faculty, we may expect to be able to improve its efficiency substantially by providing it not in the most general form possible but in a specialized, 'reflex-like' rather than fully 'adaptable' form.

As the simplifying techniques needed to organize complex functions are progressively revealed through the progress of programming, the significance for efficiency of those elementary subprocesses exercised most constantly by the compiled form of programs written using these techniques will become plain. By realizing such 'inner' subprocesses in hardware, one improves their efficiency through the elimination of unnecessary generality and by that use of large-scale parallelism which gives such great advantages to hardware realizations. An example of the type of situation we have in mind is currently seen in the tendency to simplify programming by speaking in terms of extremely large 'virtual' memories. Such an approach makes certain simple 'memory mapping' operations of constant use, and has led to the construction of these functions in hardware. Similar future influences of programming concept on hardware design are to be expected.

Artificial intelligences, if realized, will take programming as one of their first tasks, and it is interesting to try to guess the effect that this might have on programming. One of the great advantages of such intelligences will be their enormously large complexity tolerance, as compared to the capacity of the natural mind. In connection with the remarks made above we surmise that this will greatly extend the class of programmable functions, though in what way is not clear. Certainly, however, they should be capable of optimizing programs to a degree impossible to the natural mind, and in this way can contribute substantially to their own development in efficiency.