WHY PROGRAMMING IS A GOOD MEDIUM FOR EXPRESSING POORLY UNDERSTOOD AND SLOPPILY-FORMULATED IDEAS

Marvin Minsky

MIT

This is a slightly revised version of a chapter published in *Design and Planning II -- Computers in Design and Communication*, (Martin Krampen and Peter Seitz, eds.), Visual Committee Books, Hastings House Publishers, New York, 1967.

There is a popular, widespread belief that computers can do only what they are programmed to do. This false belief is based on a confusion between form and content. A rigid grammar need not make for precision in describing processes. The programmer must be very precise in following the computer grammar, but the content he wants to be expressed remains free. The grammar is rigid because of the programmer who uses it, not because of the computer. The programmer does not even have to be exact in his own ideas-he may have a range of acceptable computer answers in mind and may be content if the computer's answers do not step out of this range. The programmer does not have to fixate the computer with particular processes. In a range of uncertainty he may ask the computer to generate new procedures, or he may recommend rules of selection and give the computer advice about which choices to make. Thus, computers do not have to be programmed with extremely clear and precise formulations of what is to be executed, or how to do it.

=====

The argument presented here is not specifically about "design," but about the general question of what we can get computers to help us do. For a number of reasons, it is customary to underestimate the possibilities. To begin, I want to warn against the pitfall of accepting the apparently "moderate" positions taken by many people who believe they understand the situation. Science-fiction writers, scientists of all descriptions, economic forecasters, psychologists, and even logicians tell us often, and make it a convincing tale, that computers will never really think. *"We must not fall into anthropomorphic ways of thinking about machines; they do only what their programs say; they can't be original or creative."* We have all heard these views, and most of us accept them.

It is easy to understand why a humanist will want to rhapsodize about the obscurity of thought processes, for there is an easy *non sequitur* between that obscurity and the desired an anthropomorphic uniqueness. But this isn't the *non sequitur* important here. The fallacy under discussion is the widespread superstition that we can't write a computer program to do something unless one has an extremely clear, precise formulation of what is to be done, and exactly how to do it. This superstition is propagated at least as much by scientists‹and even by "computer scientists"‹as by humanists.

What we are told, about the limitations of computers, usually takes this general form: "A

computer cannot create. It can do only exactly what it is told. Unless a process is formulated with perfect precision, you cannot make a computer do it." Now this is perfectly true in one sense, and it is absolutely false in another. Before explaining why, it is interesting to note that - long before computers - the same was said of the Devil: he could only appear to be creative.

In the September 1966 issue of *Scientific American*, I discussed three programs: one is the checkers program of Samuel, which plays at the master level. Another is the ANALOGY program of Evans, which does rather well on certain intelligence-test problems of recognizing analogous relations between geometric figures. The third is the program "STUDENT" of Bobrow, which takes high school algebra "story" problems given in English:

Mary is twice as old as Ann was when Mary was as old as Ann is now. If Mary is 24 years old, how old is Ann?

and solves some, but not all of them. In that article I was concerned with problems of going further, to extend such work in the direction of more versatile general intelligence. But for my purpose here, they can serve as adequate examples even in their present state, for while limited in what they can handle, they already do enough to confound the old comfortable superstitions.

The old view is that a program is "nothing but" a set of rigid rules for exactly what to do in each situation. This is indeed a useful point of view for reassuring beginners at programming, or for analyzing the programs written by beginners. However, for more advanced processes, while "perfectly" true in one sense, it would be as correct to say that "houses are nothing but arrangements of construction materials" or "books are merely long strings of words." Indeed, a review of my Scientific American article (in Computer Reviews 8, 1, Jan. 1967) asserts that these programs are made of "dictionary lookup routines, sequences of search and comparison functions, and sort-merge type operations."

LOGIC AND CONSISTENCY

Let me begin by discussing one of the skeptical attitudes that is derived from some statements of good logicians and bad philosophers. We are told that certain theorems about proving the self-consistency of logical systems have the consequence: "It is impossible to mechanize completely the process of discovery and therefore humans will always have the advantage over machines." I am sure almost every reader has heard of Godel's theorem, and most have heard of this alleged consequence. But this is not a logical consequence, for it is based on a careless technical oversight. If applicable at all to the problem-solving question, Godel's theorem applies strictly only to perfectly self-consistent logical systems. People are not this consistent, and there is no reason whatever why we should feel constrained to build our machines along such lines. Instead we can, and already do, build machines that can tolerate contradictory factual assertions. To do this, we have to add selection rules for resolving contradictions, priority hierarchies for choosing between incompatible statements, and the like. A simple example of such a

scheme is described in Bertram Raphael's 1964 MIT PhD dissertation. Here is an example of a dialog with that program:

☒ EVERY BOY IS A PERSON
(I UNDERSTAND)
☒ A FINGER IS PART OF A HAND
(I UNDERSTAND)
☒ EACH PERSON HAS TWO HANDS
THE ABOVE SENTENCE IS AMBIGUOUS €€ PLEASE REPHRASE IT)
☒ THERE ARE TWO HANDS ON EACH PERSON
(I UNDERSTAND)
☒ HOW MANY FINGERS DOES JOHN HAVE?
(I DON'T KNOW WHETHER FINGER IS PART OF JOHN)
☒ JOHN IS A BOY
(I UNDERSTAND)
☒ HOW MANY FINGERS DOES JOHN HAVE?
(HOW MANY FINGERS PER HAND?)
☒ EVERY HAND HAS 5 FINGERS)
(I UNDERSTAND)
☒ HOW MANY FINGERS DOES JOHN HAVE?
(THE ANSWER IS 10)
☒ HOW MANY AUTOMOBILES DOES JOHN HAVE?
THE ABOVE SENTENCE IS AMBIGUOUS. PLEASE RE-PHRASE IT)
☒ WHO IS PRESIDENT OF THE UNITED STATES?)
(STATEMENT FORM NOT RECOGNIZED)
☒ THE B0Y IS JUST TO THE LEFT OF THE TABLE
(G02840 IS A TABLE)
(I UNDERSTAND)
☒ THE LAMP IS JUST TO THE LEFT OF THE TABLE
(G02841 IS A LAMP)
(THE ABOVE STATEMENT IS IMPOSSIBLE)
☒ THE TABLE IS TO THE RIGHT OF THE CHAIR
(G02842 IS A CHAIR)
(I UNDERSTAND)
☒ WHAT IS THE RELATIVE POSITION OF A PERSON?
(THE LEFT-TO-RIGHT ORDER IS AS FOLLOWS)
(CHAIR BOY TABLE))

Raphael's knowledge-collecting, question-answering system shows how it gives higher priority to particular facts when they conflict with general principles. This "Exception Principle" is, of course, a rather simple-minded way to resolve contradictions, and it can be only used when there is a clear difference in level of generality between the statements. When they are on the same level, the program simply rejects the later statement, as seen here:

☒ THERE ARE 5 FINGERS ON EVERY HAND)

(I UNDERSTAND)
☒ THERE ARE TWO HANDS ON A PERSON
(I UNDERSTAND)
☒ A BOY IS A PERSON
(I UNDERSTAND)
☒ TOM IS A BOY
(I UNDERSTAND)
☒ DICK IS A BOY
(I UNDERSTAND)
☒ HARRY IS A BOY
(I UNDERSTAND)
☒ TOM HAS NINE FINGERS
(THE ABOVE SENTENCE IS AMBIGUOUS €€ BUT I ASSUME (HAS) MEANS (HAS AS PARTS))
(I UNDERSTAND)
☒ DICK HAS ONE HAND
(THE ABOVE SENTENCE IS AMBIGUOUS €€ BUT I ASSUME (HAS) MEANS (HAS AS PARTS))
(I UNDERSTAND)
☒ HOW MANY FINGERS DOES TOM HAVE?
(THE ANSWER IS 9)
☒ HON MANY FINGERS DOES DICK HAVE?
(THE ANSWER IS 5)
☒ HOW MANY FINGERS DOES HARRY HAVE?
(THE ANSWER IS 10)
☒ HOW MANY FINGERS DOES JOE HAVE?
(I DON¹T KNOW WHETHER FINGER IS PART OF JOE)

But of course Raphael could have written some other priority rule. Incidentally, the program's statement, "The above sentence is ambiguous..." concerns the possibility that the word "has" might mean either "has as a part" or "owns. " Raphael's program usually guesses correctly by a study of whether the entities in question are already known to own things, or to be parts of things, etc. I will describe this later in more detail. Raphael's demonstration that such "contextual" decisions can be programmed, illustrates a more general point, or rather, shows a different and healthier attitude toward programs than the "nothing but" approach. We will therefore try to explain some of these better ways to think about programs.

(1) A PROGRAM AS A SEQUENCE OF INSTRUCTIONS TO BE OBEYED.

The most common and simple-minded view is that a computer program is a sequence of clear-cut operations to be performed on some data. Let's take a simple example of a program: suppose that X is a number given as input:

```
START:   SET R = 1
LOOP:  SET S = X/R
```

SET T=R+S
SET R = T/2
SET U = R x R
SET V = X - U
IF |V| > .001, GO TO LOOP
ELSE PRINT V

This program (attributed to Isaac Newton) will compute an approximation to the square root of X. At each line in the program it is perfectly clear what is the current status of things, and what is to be done next. Of course, it requires a mathematical analysis, to show that the sequence of situations so developed will eventually lead to an acceptable value of R, and terminate the computation.' So there is no great harm in thinking of these statements as a sequence of steps. The "GO TO" instruction might be considered to be a super-abbreviation for several more copies of the "main loop." Of course, until the analysis is made, one should think of the program as an expression of a useful but poorly understood idea.

(2) A PROGRAM AS A COURT OF LAW

In Raphael's program, it has to be decided whether "has" means "owns" or "has as parts" when a statement of the form "x has y" is encountered:

☒ JOHN IS A PERSON)
(I UNDERSTAND)
☒ DICK IS A PERSON)
(I UNDERSTAND)
☒ A CHAIN IS PART OF A BICYCLE
(I UNDERSTAND)
☒ THE POWER-SAW HAS A CHAIN
(THE ABOVE SENTENCE IS AMBIGUOUS €€ BUT I ASSUME (HAS) MEANS (HAS AS PARTS))
(I UNDERSTAND)
☒ JOHN OWNS A CHAIN
(I UNDERSTAND)
☒ DICK HAS A CHAIR
(THE ABOVE SENTENCE IS AMBIGUOUS €€ BUT I ASSUME (HAS) MEANS (HAS AS PARTS))
☒ THE CUCKOO-CLOCK HAS A CHAIN
(THE ABOVE SENTENCE IS AMBIGUOUS €€ PLEASE REPHRASE IT)

The problem, when recognized, is transmitted to a part of the program that is able to review all that has happened before. This sub-program makes its decision on the following basis:

(1) Is y already known to be part of some other thing? Or is y a member of some set whose members are known to be parts of something?

(2) Is y known to be owned by something, or is it a member of some set whose members are known to be owned by something?
(3) If exactly one of (1) or (2) is true, make the choice in the corresponding direction. If neither holds, give up and ask for more information. If both are true, then consider the further possibilities at (4) below. (Thus the program uses evidence about how previously acquired information has been incorporated into its "model" of the world.)
(4) If we get to this point, then y is known already to be involved in being part of something and in being owned and we need a finer test.

Let U1 and U2 be the "something" or the "some set" that we know exists, respectively, in the answers to questions (1) and (2). These depend on- y. We now ask: is x a member of, or a subject of U1 or U2? If neither, we give up. If one, we choose the corresponding result-"part of" or "owns." If both, we again give up and ask for more information. As Raphael says:

"These criteria are simple, yet they are sufficient to enable the program to make quite reasonable decisions about the intended purpose in various sentences of the ambiguous word "has." Of course, the program can be fooled into making mistakes, e.g., in case the sentence, "Dick has a chain," had been presented before the sentence, "John owns a chain," in the above dialogue. However, a human being exposed to a new word in a similar situation would make a similar error. The point here is that it is feasible to automatically resolve ambiguities in sentence meaning by referring to the descriptions of the words in the sentence-descriptions which can automatically be created through proper prior exposure to unambiguous sentences."

Thus, the program is instructed to attempt to search though its collection of prior knowledge, to find whether x and y are related, if at all, more closely in one or the other way. This "part" of the program is best conceived of as a little trial court, or as an evidence-collecting and evidence-weighing procedure. It is not good to think of it as a procedure directly within a pre-specified sequence of problem solving, but rather as an appeal court to consult when the program encounters an inconsistency or ambiguity. Now when we write a large program, with many such courts, each capable if necessary of calling upon others for help, it becomes meaningless to think of the program as a "sequence." Even though the programmer himself has stated the "legal" principles which permit such "appeals," he may have only a very incomplete understanding of when and where in the course of the program's operation these procedures will call on each other. And for a particular "court," he has only a sketchy idea of only some of the circumstances that will cause it to be called upon. In short, once past the beginner level, programmers do not simply write 'sequences of instructions'. Instead, they write for the individuals of little societies or processes. For try as we may, we rarely can fully envision, in advance, all the details of their interactions. For that, after all, is why we need computers.

(3) A PROGRAM AS A COLLECTION OF STATEMENTS OF ADVICE

The great illusion shared not only by all terrified humanists but also by most computer "experts," that programming is an inherently precise and rigid medium of expression, is

based on an elementary confusion between form and content. If poets were required to write in units of fourteen lines, it wouldn't make them more precise; if composers had to use all twelve tones, it wouldn't constrain the overall forms; if designers had to use only fourth order surfaces no -one would notice it much! It is humorous, then, to find such unanimity about how the rather stiff grammar of (the older) programming language makes for precision in describing processes. It's perfectly true that you have to be very precise in your computer grammar (syntax) to get your program to run at all. No spelling or punctuation errors are allowed! But it's perfectly false that this makes you have a precise idea of what your program will do. In FORTRAN, if you want your program to call upon some already written procedure, you have to use one of the fixed forms like "GO TO." You can't say "USE," or "PROCEED ON TO," etc., so the syntax is stiff. But, you can "GO TO" almost anything, so the content is free.

A worse fallacy is to assume that such stiffness is because of the computer! It's because of the programmers who specified the language! In Bobrow's STUDENT program, you could type once and for all, if you wish, "USE ALWAYS MEANS GO TO" and in simple situations it would then allow you to use "USE" instead of "GO T0." This is, of course, a trivial example of flexibility, but it is a point that most people don't appreciate: FORTRAN's stiffness is, if anything, *derived* from the stiffness superstition, not an instance of some stiffness fact!

For an example of a modern system with more flexibility, a programming language called PILOT, developed by Warren Teitelman (Ph.D. dissertation, MIT, 1966), allows the programmer to make modifications both in his programs and in the language itself, by external statements in the (current version of) the language. We can often think of these as "advice" rather than as "program," because they are written at odd times, and are usually conditionally applied in default situations, or as a consequence of previous advice. **An example is the following** typed in while developing a program to solve problems like the well-known "missionaries and cannibals" dilemma - the one with the boat that holds only two people, etc:

Tell progress, if m is a member of side-1 and m is a member of side-2 and (countq side-1 m) is not equal to (countq side-1 c), then quit. (An earlier collection of advice statements to the input system has been used to produce the reasonably humanoid input syntax.)

The program is a heuristic search that tries various arrangements and moves, and prefers those that make "progress" toward getting the people across the river. Teitelman writes the basic program first. But the missionaries get eaten, and the above "advice" says to "modify the progress-measuring part of the program to reject moves that leave unequal numbers of missionaries and cannibals on the sides of the river." As Teitelman says:

This gives the eating conditions to PROGRESS. It is not sufficient to simply count and compare, because when all of the cannibals are on one side with no missionaries, they do outnumber the missionaries 3 to 0. However, nobody gets eaten.

The point, however, is not in the relaxation of syntax restrictions, but in the advice-like character of the modification just made in the program. The "tell progress" statement can be made without knowing very much about how "progress" works already or where it lies

in the "program." It may already be affected by other advice, and one might not have a clear idea of when the new advice will be used and when it will be ignored. Some other function may have been modified so that, in certain situations, "progress" won't get to evaluate the situation at all, and someone might get eaten anyway. If that happened, the outsider would try to guess why.

He would have the options (1) of thoroughly understanding the existing program and "really fixing" the trouble, or (2) of entering anew advice statement describing what he imagines to be the defective situation and telling the program not to move the missionary into the position of being eaten. When a program grows in power by an evolution of partially-understood patches and fixes, the programmer begins to lose track of internal details and can no longer predict what will happen‹and begins to hope instead of know, watching the program as though it were an individual of unpredictable behavior.

This is already true in some big programs, but as we enter the era of multiple-console computers, it will soon be much more acute. With time-sharing, large heuristic programs will be developed and modified by several programmers, each testing them on different examples from different consoles and inserting advice independently. The program will grow in effectiveness, but no one of the programmers will understand it all. (Of course, this won't always be successful-the interactions might make it get worse, and no one might be able to fix it again!) Now we see the real trouble with statements like "it only does what its programmer told it to do." There isn't any one programmer.

LATITUDE OF EXPRESSION AND SPECIFICITY OF IDEAS

Finally we come to the question of what to do when we want to write a program but our idea of what is to be done, or how to do it, is incompletely specified. The *non sequitur* that put everyone off about this problem is very simple:

Major Premise: If I write a program it will do something particular, for every program does something definite.
Minor Premise: My idea is vague. I don't have any particular result in mind.
Conclusion: Ergo, the program won't do what I want.

So, everyone thinks, programs aren't expressive of vague ideas.

There are really two fallacies. First, it isn't enough to say that one doesn't have a particular result in mind. Instead, one has an (ill-defined) range of acceptable performances, and would be delighted if the machine's performance lies in the range. The wider the range, then, the wider is one's latitude in specifying the program. This isn't necessarily nullified, even when one writes down particular words or instructions, for one is still free to regard that program as an instance. In this sense, one could consider a particular written-down story as an instance of the concept that still may remain indefinite in the author's mind.

This may sound like an evasion, and in part it is. The second fallacy turns around the

assertion that I have to write down a particular process. In each domain of uncertainty 1 am at liberty to specify (instead of particular procedures) procedure-generators, selection rules, courts of advice concerning choices, etc. So the behavior can have wide ranges-it need never twice follow the same lines, it can be made to cover roughly the same latitude of tolerance that lies in the author's mind.

At this point there might be a final objection: does it lie exactly over this range? Remember, I'm not saying that programming is an easy way to express poorly defined ideas! To take advantage of the unsurpassed flexibility of this medium requires tremendous skill-technical, intellectual, and esthetic. To constrain the behavior of a program precisely to a range may be very hard, just as a writer will need some skill to express just a certain degree of ambiguity. A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. That is the argument we have heard from our humanists and most of our computer scientists. Computer programs are good, they say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it.