Context and Creation in the Learning of Computer Programming

CHRIS HANCOCK*

*Chris Hancock is a project associate at Harvard's Educational Technology Center (ETC), a federally funded research center which investigates the uses of computers in education. Some of the ideas discussed here were developed in collaboration with other members of ETC's programming research group.

Teaching and learning computer programming at the introductory level have both turned out to be difficult. Many students in high schools and colleges have had unrewarding experiences in programming courses and come away thinking "I'm not the kind of person who can do this."

It's hard for those of us with a lot of programming background to appreciate the difficulties that novice programmers confront. Perhaps it's partly because we've worked with the basic concepts of programming for long enough that they now seem trivial. But another reason may be that there are some important ideas in programming that no-one made explicit to us when we learned to program. We picked these ideas up without realizing what we were learning, and even now we don't have words for some of our most fundamental knowledge.

In recent years some progress has been made in identifying the kinds of knowledge that underlie the ability to program. Two ideas, the *mental model* and the *programming plan*, have proven valuable in teaching at the introductory level. I'll present these two ideas, and then I'll try to show where they fall short. What needs to be recognized is that computer programming is not just about computers — it's also about the world.

Mental models

Suppose you were to take a group of typical beginning programming students and ask them to predict the output of this program (or its equivalent in the language they're learning — I'll use BASIC and Pascal in this article for the sake of discussion, but the ideas apply to most languages):

```
10  X = 20
20  FOR Y = 1 TO 6
30  IF (Y/2) = INT(Y/2) THEN GOTO 50
40  X = X - 3
50  PRINT X + Y
60  NEXT Y
```

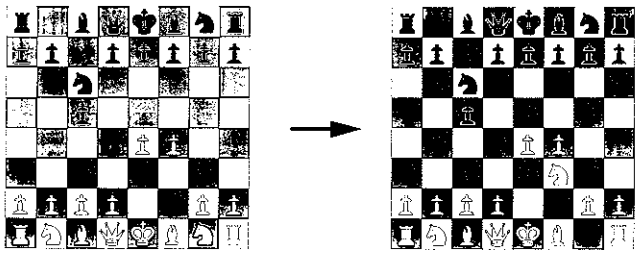
You might be surprised at the results. Beginners find problems like this very difficult, because they require one to *hand-execute* the program — that is, to simulate, step by step, what the computer will do when the program is run.

Hand-executing nonsense programs like the one above may not be very important, but hand-execution is an essential skill for checking and debugging one's own programs, and for reading programs that others have written. Why should such a mechanical skill pose so much difficulty? Consensus is growing in the research and teaching communities that students who can't hand-execute code need a better *mental model* of the internal workings of a computer — that is, a way of imagining the state of the computer and the changes in state produced by each step in a program. If you know BASIC, look at the little program above and try running it in your mind. How did you do it? Did you picture something? Did you imagine annotations to the program text on the page? One way or another, you had to remember the state of the computer — the values of the variables, the line currently being executed, and the output accumulating on the screen. We can define your mental model of BASIC as your way of representing the state of the computer at each step of a program. Your model may be visual, non-visual or a combination of the two. But in any case it is your key to understanding BASIC.

Another example will help to show how important a model is for understanding languages. Newspapers sometimes publish transcripts of chess games. They look like this:

```
1  P—K4     P—QB4
2  P—KB4    N—QB3
3  N—KB3    P—K3
                 {and so on}
```

Suppose you wanted to understand one of these games. Assuming you knew the rules of chess and chess notation, what would you do? The most obvious approach would be to get out a chess set and follow the game through, move by move. If you didn't have a chess set handy you might draw a picture instead. Or, if you were ambitious you might try to follow the game by picturing a chessboard in your mind. The point is this: to make sense of the transcript you have to refer, one way or another, to a chess set — be it real, drawn or imagined. The chess set is vital because it allows you to assign *meaning* to notations like "N—KB3". In the context of this game, the meaning consists of the transformation of position (a) into position (b).



(a) (b)

Figure 1

The theoretical way to describe this is to say that a chess set constitutes a model for an operational semantics of the language of chess notation (an operational semantics is one that maps utterances of a language to transformations on a model) The psychological fact that goes along with this is that in order to understand chess notation (and chess itself) you need to have recourse to a chess set.

Imagine trying to teach someone chess without showing them a chess set It's unthinkable! The unfortunate pupil would be sure to find chess notation utterly opaque, governed by arbitrary and unfathomable rules. Yet if we are not careful, this is exactly the situation that we will create for students of programming. What we need is a model that does for programming languages what a chess set does for chess notation.

But while a chess set is the universally acknowledged model for chess notation, there isn't a standard model for, say, BASIC. We have to design our own Maybe the most obvious thing the model should reflect is the values of variables For example, we might say that at one point in the program the meaning of "X = X + 3" is the transformation of the value of the variable X. But actually, a collection of variable names with corresponding values is not enough. Such a collection cannot illustrate, for example, the effect of a GOTO statement or a PRINT statement Our research group at ETC recently designed a model that is adequate for a core subset of BASIC, ignoring things like data files and peeks and pokes (see Figure 2)

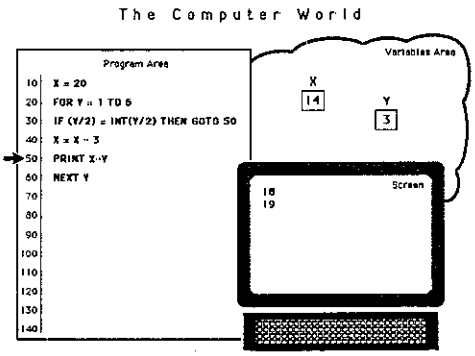


Figure 2

As it happens, there are real chess sets in the world But our computer world is just an abstraction It doesn't — nor should it — illustrate what physically happens inside a computer. This is both theoretically and pedagogically sound. Theoreticaly sound, because a complete semantics of our BASIC subset can be defined using our model; pedagogically sound, because you don't have to understand how a BASIC interpreter is implemented in order to understand BASIC. In principle, one could define a semantics of BASIC in terms of a raw memory map of the computer, but this would be needlessly complicated. The goal is to come up with a model that is complete and consistent, and also as simple as possible Some teachers have felt compelled to tell students about symbol tables, memory allocations and the stack because these are "what really happens" (although these too are abstractions above the physical state of the machine). While this information can be interesting for students it is not an essential part of understanding the meaning of BASIC

I may be giving the impression that most teachers of programming don't give their students any kind of model of the programming language. This is certainly not the case. However, I think the importance of a complete, consistent and simple model is not generally appreciated

Most programming languages are like BASIC in that they come with no explicit model. However, some languages do include models (partial ones, at least), and this makes them especially valuable for beginning programmers LOGO is perhaps the classic example The turtle's movements and drawings on the screen constitute an explicit model for LOGO's drawing commands. *Karel the Robot,* designed as a way for novices to get into Pascal, also puts its semantic model right on the screen. Fortunately, products like these are becoming more common

I have said that a model for the programming language is essential to the ability to hand-execute programs. Hand-execution is important in itself, but a good mental model has other benefits too. For instance, a model is a very useful thinking tool for composing programs. When we play chess, most of our thinking is expressed not in notational terms but in terms of the chess set itself ("if she goes *there,* then I can move my rook *there,* oh, but she might push that pawn . . ."). Likewise when we compose a program we often think not in the programming language itself but in terms of our mental model ("first I'll change all the negative numbers in the profits array to zeroes, then I'll sort the other array so that . . .") Certainly there's plenty more to programming than just having a mental model for the language. But the model is fundamental.

Programming plans

In principle, perhaps, a novice with a complete mental model for BASIC, who knows the semantics of every BASIC command, has enough knowledge to write any BASIC program But of course this never happens, just as no one plays good chess the day they learn the rules. Programming and chess are both complex endeavours in which expertise takes a long time to develop

But what is expertise made of? A recurring theme in recent psychological studies of expertise in various domains is that in addition to knowing the fundamental facts, experts are able to think in terms of higher-order units, known variously as "chunks," "plans," "schemas," "structures," or "patterns." A good chess player knows

some standard openings, and also knows recurring patterns like knight forks and pins A good rock guitarist improvises not by inventing each note on the fly, but by drawing on a large repertoire of "licks" — sequences of notes that can be used and re-used in many situations. Experts may be more or less conscious of having such higher-order knowledge Some chunks are more acknowledged than others Some are shared among experts in a particular domain while others are idiosyncratic

In programming, most of the research on these higher-order units has been done by a group at Yale headed by Elliot Soloway, who calls them "programming plans." A plan can be something as simple as a prompt/input sequence, or a statement to increment a variable (X = X + 1). More interesting examples are the maximum plan, the lag variable plan, or the sentinel-based loop (see Figure 3) If you are an experienced programmer, take a look at the last few programs you've written. How many lines in these programs can you recognize as parts of plans that you use in many different situations? My guess is: most of them.

Maximum Plan

```
max := 0;
↻  if newvalue > max then max := newvalue
```

Lag Variable Plan

```
previous := {some initial value}
↻  current := {something}

   previous := current
```

Sentinel-Based Loop Plan

```
{get next value for X}
while X <> {sentinel value} do
      begin

         {process X}

      {get next value of X}
      end;
```

Three typical programming plans, shown here in Pascal
The circular arrow indicates a loop of some kind
Figure 3

Clearly plans save time If you're working with a stream of data that has a sentinel value at the end, it's quicker simply to plug in your sentinel-based loop plan, rather than to figure an approach out from scratch But another important function of plans is that they let you think about more of the problem at once, because you don't have to hold so many details in your mind You can confidently think "I'll loop through those numbers" and leave the step-by-step details for when you write the code

A natural analogy is to subroutines in programming, which also save programming time and allow you to broaden your perspective on the problem. Plans are like "subroutines for the mind" up to a point, but there's an important difference All you need know about a subroutine is its external specifications; but to really know a programming plan and use it to full advantage you have to understand how it works, in detail. For example, suppose you had to write a program that takes a list of numbers, terminated by a sentinel, and reports the largest increase and the largest decrease between any two successive numbers. You would have to combine all three of the plans in Figure 3 To do this successfully you would have to decide what order to combine the lines in, which variables could be shared among plans, what extra code (such as IF statements) to put in to enable the plans to do their jobs, and so on. This would be very difficult unless you knew the plans well enough to be able both to hand-execute them and to explain in detail why they are constructed the way they are.

Many students use code constructs like the ones in Figure 3 without fully understanding them. Instead they use them as formulas. What these students have is not a plan but a *template*. For example, the student who can use a FOR loop to print the numbers 1 through 10, but can't adapt this to have each number followed by its square, knows the FOR loop only as part of a template. "Template-bound" programmers tend to have a lot of difficulty when they encounter a programming problem that does not closely resemble the problems that they are accustomed to.

At this point we should clarify our terminology, because there is a potential ambiguity in the word "plan." By "plan" do we mean a pattern of lines of code, like the "sentinel loop plan?" Strictly speaking we do not, because it's only a plan if the programmer understands it in a deep way — otherwise it's a template. Let's use the word *construct* as a neutral word to describe the pattern of lines of code, and use "plan" or "template," as appropriate, to describe the corresponding knowledge in the programmer.

Problem-seeing

The distinction between the code and the programmers's knowledge becomes even more important when we realize that plans and templates are not just knowledge about how to use constructs: they are also knowledge about how to look at problems. Suppose, for example, that I ask you to write a program to print a line of stars, as long as the user likes, like this:

```
HOW MANY STARS? 5
*****
```

The key to this is to think of the line of stars not as a unit, but as having internal structure, namely that it consists of the same thing (a star) repeated over and over again, and the number of repetitions is equal to the number the user entered. That insight suggests that you might use the FOR loop construct to solve this problem — provided you can figure out what goes inside the loop. The *quotient problem* (by which I mean the sub-problem that results if you break the original problem according to your first insight) is to

print a single star in such a way that the cursor remains on the same line, one position to the right of the star. Line 40 of this solution solves the quotient problem

```
10  PRINT "HOW MANY STARS? "
20  INPUT H
30  FOR I = 1 TO H
40  PRINT "*";
50  NEXT I
60  PRINT
```

As part of our research at ETC, we asked some high school students to solve this problem. When we gave the problem to Alice (a tenth-grade student in her second semester of BASIC programming) we got some peculiar results. Alice described her thoughts to us as she worked on the program. "I'm thinking about whether I should use a FOR X = 1 TO, a FOR/NEXT... I think I'll probably have to if I want to use INPUT." Her first effort looked like this:

```
10  FOR X = 1
20  PRINT "HOW MANY STARS?"
30  INPUT N
40  X = (*) * N
50  NEXT X
```

This is a rather stange-looking program. Line 40 won't work, of course, and line 10 has a syntax error. But why did she use a FOR loop at all, if it was only to go around once? The experimenter asked Alice about the loop and she explained, "I have one piece of information I want to put in, just 5, so when it asks me how many stars I want, I will just tell it 5, and it will go through this process and then it will stop because I only have one piece of data. I've never used one piece of data."

This comment would be incomprehensible, except that fortunately we have some background information about the kind of programs that Alice's programming class worked on. Many of their assignments involved reading through a DATA list and printing a table. A simple one might go like this:

```
10  REM THIS PROGRAM MAKES A TABLE
    OF STUDENT GRADES
20  PRINT "    STUDENT     GRADE"
30  PRINT "    -------     -----"
40  FOR X = 1 TO 5
50  READ S$,G
60  PRINT S$,G
70  NEXT X
80  DATA MARY,66,FRED,88,BILL,47,JANE,
    90,ED,75
```

My interpretation is that Alice was accustomed to solving such problems using a template along these lines:

"process-an-item" template

```
FOR X = 1 TO {number of items}
{read an item in}
{print an item out}
NEXT X
DATA {items go here}
```

What Alice did with our line-of-stars problem was try to apply this template to it! She matched the user's input (5) to the "read an item in" category (except that she would use INPUT instead of READ/DATA) and she matched the line of stars to the "print an item out" category. And since this exchange was to happen only once, there was only one item. This explains her cryptic (to us) remark, "I've never used one piece of data."

What happened next was even more interesting. The experimenter (who did not have my interpretation in mind at the time) decided to switch Alice back to an easier problem — a program that would take no input from the user, but simply print five stars. The stars were to be in a column rather than on one line.

```
*
*
*
*
*
```

With some coaching, Alice removed the useless one-time-around FOR loop from her program. With a little more coaching, she realized that she could use a FOR loop to repeat the stars. She wrote this correct program:

```
10  FOR X = 1 TO 5
20  PRINT "*"
30  NEXT X
```

She commented that it was the input statement that had confused her before. Why did she say that? I think that as long as there had to be input, she could only see the problem in terms of her "item processing" template. When the problem no longer involved input, she was able to look at the problem in new ways.

Encouraged, the experimenter asked Alice to return to the original line-of-stars problem. Amazingly, Alice had no idea how to proceed, despite her intervening success. She said, "The thing that's confusing me is there is only one item here. In the other one there were five items, because there was no queston." To our eyes, the problems are very similar: in one the stars go down; in the other they go across. But Alice's eyes were looking for *items* (that is, lines of input or output) and so the two problems looked completely different to her. The fact that the line-of-stars problem needed input served to lock her into trying to apply her item-processing template to it.

One might reasonably object at this point that perhaps Alice had never used PRINT with a semicolon, in which case she would never think of breaking the line up. Fair enough. But it is still very striking how differently she described the two problems and she gave no hint of even considering the quotient problem (unlike another student, clearly at the plan level in this case, who said "I could do it with a FOR loop if I could just get one star to come out at a time on the same line"). I remain impressed by the way a programming template can actually constrain one's view of a problem, obscuring patterns that one might otherwise see in it.

To summarize this section so far: plans and templates

are not just knowledge about code constructs — they are also ways of seeing problems (If you find this idea as intriguing as I do, you might want to look at Robert Lawler's book *Computer experience and cognitive development*. Lawler's concept of a "micro-view" has much in common with plans and templates) The important differences *between* plans and templates in this respect is that plans give the programmer a conscious option of how to look at the problem, while templates work more automatically, precluding alternative ways of seeing the problem

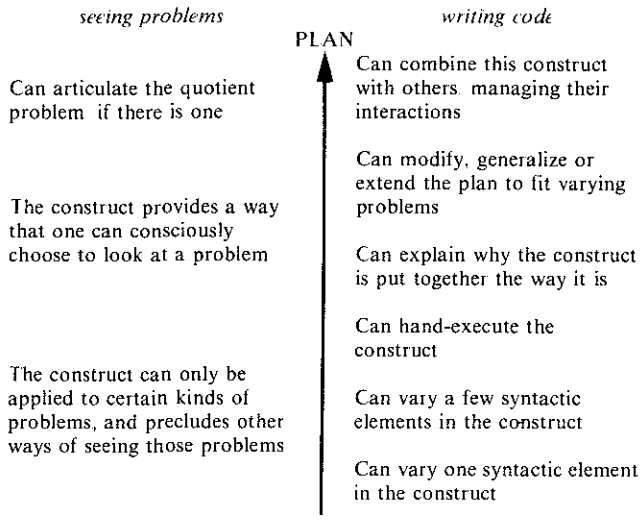
Problem-seeing is a neglected aspect of computer programming. If it is an issue when we try to make stars come out on a screen, imagine how much more of an issue it must be when we try to write a program that actually does something useful Amidst the complexity and confusion of a real-life problem, a programmer has to be able to find enough regularity and structure to define a computer program that will be simultaneously as simple and as useful as possible This is no small task. I once worked with some adults in a intermediate-level programming course. At the end of the course each student was to write a large program to solve any problem they liked About half of the students could not think of anything to write about! This certainly wasn't because there was nothing in their jobs or personal interests that would make an interesting program. Rather, these students had still not learned to apply their programming knowledge to the real world.

Teaching about programming plans

Should we explicitly teach students programming constructs like the sentinel-based loop and the lag variable? From what we have just seen, there appears to be a danger that if we do, students will swallow the constructs whole, learning them as templates instead of as plans. Because template-style understanding of programming constructs falls so far short of our own expert understanding, we naturally assume that it is bad But I no longer believe that it is Certainly we wouldn't want students to finish our courses knowing only templates, but I wonder if the template might not be a natural stage on the way to the plan As an intermediate stage, it has some useful functions: it lets students use a construct to write interesting programs without having to understand them completely, it gives them time to become familiar with the surface details of the new construct (e g the syntax of FOR), and it allows them to appreciate the usefulness of the new construct The template can thus be a good base from which to develop a deeper understanding Figure 4 shows a progression from the most rigid template to the most flexible plan

I think that there is potential for the conscious, constructive use of templates in programming classes I have one warning, though: when students learn a template, it should be with the clear expectation that they will later come to understand it more fully. It's dangerously easy to learn a new construct at the template level, and mistakenly believe that you now know it.

Behind the question of how to teach higher-level constructs lurks a more fundamental question, a question that we may be scared to face: to what extent is *invention* an integral part of programming, and how much invention should we expect of our students? In my view, invention is an integral part of programming. It plays a vital role in



An approximate progression of understanding from templates to plans.
Figure 4

students' understanding. For example, we can contrast plans and templates by saying a plan is *understood as an invention,* while a template is not I don't mean that the student has to have invented the plan himself or herself (although that would be one way to achieve this level of understanding), but rather, that

- the student understands that the plan was invented by a person;
- the student understands that there are alternative ways of seeing the problem and alternative ways of coding a solution to it; and
- the student understands the role that each part of the plan plays in making it work.

To understand something as an invention you have to have done some inventing yourself That's why getting students to invent should be a high priority in programming classes. I don't have an easy prescription for achieving this — I think it's partly a matter of the teacher's art. The teacher can try to make a class environment where invention is possible. The teacher can try to spot the inventive thought in each student's efforts, even (or rather, especially) in the student's mistakes, and work to make contact with it and encourage it to grow. If students have had the experience of inventing programs they will be in a position to learn templates and come to understand them as plans.

Pragmatics

Earlier in this article I discussed the similarities between the programming language BASIC and the language of chess notation I stressed the fact that they have the same kind of

semantics — an operational one. You may have objected that the languages are very different, too. "After all, we write programs to make a computer do something, but chess notation is a record of a game played between two people!" Very true. This important difference is not in the semantics of the two languages but in the contexts in which they are used, and in the purposes to which they are put. The system of contexts and purposes for a language is called the *pragmatics* of that language. I believe that the concept of pragmatics can help us to understand students' difficulties better, and help us improve our teaching in many ways.

To begin with, some very elementary student errors can be understood as arising from a pragmatic misunderstanding. Here, for example, is an erroneous program typical of some novice programmers:

```
10   C = (F - 32) * 5/9
20   F = 56
30   PRINT C
```

This program reflects the pragmatic theory that when we write a BASIC program we are "telling the computer." First we tell it the formula for converting Fahrenheit to Celsius; then we tell it the temperature in Fahrenheit; then (once it "knows" those two facts) we ask it to show us the the Celsius equivalent. This is not an unreasonable hypothesis about the pragmatics of BASIC, especially in view of the fact the BASIC is called a language, and a language is usually a way for people to tell each other things. But the actual pragmatics of BASIC is not telling, but causing: we arrange the statements in our program to cause certain changes in the internal state of the computer; these changes will in turn cause the computer to serve the purpose of the person who uses the program. Furthermore, programs that produce the effects we want have to be *invented*. The fact the programs (and programming plans) are inventions is another fundamental pragmatic fact about programming — a fact that sometimes escapes novice programmers.

Pragmatic knowledge also serves as a basis for other kinds of programming knowledge. For example, knowing the context of use of a language construct can help to "anchor" one's knowledge of its syntax and semantics. We noticed that many students in Alice's class didn't know whether to use READ or INPUT, even when we provided a sample run of the program. These students had studied both commands in class, so why should they have so much difficulty? The root problem, I believe, is that these students were not sufficiently aware that a program can often be used by someone other than the person who wrote it. They had certainly never written such programs themselves. Without the pragmatic distinction between the programmer and the user, the semantic differences between READ and INPUT were neither meaningful or memorable to these students.

I teach an introductory programming course (for adults) in which students spend the first two weeks using a simple database package. I ask them to create a database that is related to their own work or interests. I think this approach has many benefits but a primary one is that it lets the students actually experience the role of user, before they have even begun to program. The experience prepares them to *imagine* a user for the programs they write later in the course. In addition, during the early part of the course I occasionally ask students to use a program specification sheet (with three sections entitled "purpose," "interaction," and "examples") — either to fill one out or to work from one. This form shows no code and thus represents the user's point of view of a program.

Knowing the context in which a program will be used supports the writing of the program in many ways. Consider the common practice of embedding assignments in a story, like this one (for Pascal students):

> In the principal's office at Local High School they keep a computerized file recording how many times each student has been absent. You are to write a program to help them keep the file up to date. The file is a file of Pascal records, of this type:
>
> ```
> type abs_rec = record
>                  name : string[25];
>                  num_absences : integer;
>                  end;
> ```
>
> Every morning someone in the office will run your program. The program should let them enter the names of that day's absent students, and automatically update the file so that the number of absences for each of that day's absentees is increased by one.

This assignment puts a challenging problem in the manipulation of files and records in an interesting hypothetical context. The context helps the student understand the problem (it would be considerably harder even to *describe* the problem without a context for it), and *it also guides many of the design decisions in the writing of the program*. Should the program check to see if an entered name is actually on the list? If so, what should it do if the name isn't found? What should the program do if the user enters a blank name? Should the program read all of the data into memory at the beginning or work with one file component at a time? These questions can all be answered by referring to the context: What kind of interaction would be convenient for the secretary? How big is the database likely to be, and how powerful is the computer?

As teachers we sometimes feel forced to set arbitrary-sounding policies. "Always idiot-proof your programs." "Remember to handle special cases (like the empty string, for example)." How much better to let these principles follow naturally from the context of the problem.

Unfortunately, reasoning from the imaginary context can lead to trouble if it is taken too far. What if there's a new student in the school — shouldn't there be a way to add new students to the master list? What if they mistakenly enter a student as absent, and want to correct the file? What if the secretary enters "Bill Green," but in the file it's "William Green?" Now the problem is getting out of hand, and the reason is that the assignment is meant to be realistic, but not completely realistic. The proper way for the student to read it is with some suspension of disbelief. The

wise student balances some reasoning about the imaginary context with knowledge of the limitations of his or her ability and of what kind of program is expected in the programming class (and when in doubt asks the teacher what the program is supposed to do) Now that is a sophisticated way to have to read an assignment!

Actually I think students are quite capable of making this reading After all, it takes equally sophisticated pragmatic reasoning to understand *Miami Vice*. But they can only do it if they have seen or participated in a *real* context of the program use, so that they can try to discriminate what is realistic in the assignment from what isn't. Students who can't begin to make this discrimination (and I believe that there are *many* students in this predicament) can only have a vague sense of arbitrariness and falseness in what they are asked to do, or a sense that they can't quite grasp the enterprise of programming.

This is why I believe that we should work harder to create better *concepts for programming* for our students Perhaps students could write small programs for each other to use, or use programming as part of their work in other subjects like science and social studies. Perhaps programming classes could undertake real projects for the school office or for the community These kinds of activities would help to bring programming alive. They would also help to stimulate the inventive thinking that is so important in learning to program.

Not all programs are written to do something useful One alternative purpose for programs is simply to sharpen one's programming skill. It's quite reasonable to ask a student to write a program to, say, read in ten numbers and then re-display them in the opposite order. Such programs are exercises and there is no ambiguity about how they should behave. But while pure exercise justifies writing one program, it doesn't justify the whole enterprise of programming Students must also write more meaningful programs

Another purpose for programs is not to do something useful but to create an entertaining or expressive effect. This is the usual pragmatic basis of LOGO programming, and it can work well with youngsters. Most teenagers and adults, however, are unlikely to find self-expression in programs that are not connected to the outside world. Those students who, for one reason or another, *can* enjoy programming without a context of use are the ones who have fared well with our present teaching approach. Others have fallen by the wayside.

Conclusion

In many people's minds, the word "programming" conjures up an image of a solitary hacker hunched before a flickering screen, absorbed in the closed world of his computer If programming is really so self-contained, so cut off from the world, then I wonder whether we should try to teach it to so many people.

It's true that a computer is a self-contained system. (The LOGO environment, for example, is described as a "microworld".) There's no question that in order to write programs one has to understand and gain mastery over the logic of this system. That's what mental models are about and that's what higher-level code constructs are about. *But the essence of programming is making a creative connection between the computer system and the outside world, in order to make the computer do something meaningful.* As I have tried to show, learning to program must also mean learning to look at problems in new ways, and learning to think about the role that computers can play in helping people to do the things they do. By giving this aspect of programming its due attention, I believe that we can not only teach it better but make it a more valuable experience for our students.

Selected references

R Lawler, *Computer experience and cognitive development* Ellis Horwood Press, 1985

D N. Perkins, C. Hancock, R. Hobbs, F Martin, and R. Simmons Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research* 2(1), 1986

E Soloway and K Ehrlich. Empirical Studies of Programming Knowledge *IEEE Transactions on Software Engineering* Sept 1984

Continued from page 17

References

Bartlett, F C [1932] *Remembering, a study in experimental and social psychology.* Cambridge: The University Press

Bloor, David [1981] Hamilton and Peacock on the essence of algebra. In: H Mehrtens, H. Bos and I Schneider (eds) *Social history of nineteenth century mathematics* Boston: Birkhäuser

Brunschwicg, Léon [1971] Dual aspects of the philosophy of mathematics In: F. LeLionnais (ed) *Great currents of mathematical thought.* Volume 2, New York: Dover Publications English trans. of the 1962 edition of *Les grands courants de la pensée mathématique* Paris: Librairie Scientifique et Technique

Henry, Jules [1960] A cross-cultural outline of education *Current Anthropology,* 1, 4

Keller, Evelyn Fox [1985] *Reflections on gender and science* New Haven: Yale University Press

Simon, Herbert A. [1981] *The sciences of the artificial* Second edition Cambridge, Mass: The MIT Press

Smith, David Eugene [1900] *The teaching of elementary mathematics* New York: Macmillan

Tahta, Dick [1986] In Calypso s arms *For the Learning of Mathematics* 6, 1