J. EDUCATIONAL COMPUTING RESEARCH, Vol. 2(1), 1986

SOME DIFFICULTIES OF LEARNING TO PROGRAM

BENEDICT DU BOULAY
University of Sussex

ABSTRACT

This article is a brief introduction to some of the issues that teachers of programming may find helpful. It starts by presenting a fairly idiosyncratic view of teaching programming which makes use of mechanistic analogies and points out some of the pitfalls. The article goes on to examine certain errors based on the misapplication of analogies as well as certain interaction errors. The main emphasis is on the notional machine both at the general level of understanding (and misunderstanding) the relationship of the terminal to the computer as such, as well as at the more specific level of understanding assignment. Notation and mistakes that poorly-designed languages can induce novices to commit are discussed.

AREAS OF DIFFICULTY

Learning to program is not easy. The difficulties can be separated into five areas with a certain degree of overlap. First there is the general problem of *orientation*, finding out what programming is for, what kinds of problem can be tackled and what the eventual advantages might be of expending effort in learning the skill. Second, there are difficulties associated with understanding the general properties of the machine that one is learning to control, *the notional machine*, and realizing how the behavior of the physical machine relates to this notional machine. For example, it is often unclear to a new user whether the information on the terminal screen is a record of prior interactions between the user and the computer or a window onto some part of the machine's innards. Third, there are problems associated with the *notation* of the various formal languages that have to be learned, both mastering the syntax and the underlying semantics. The semantics may be viewed as an elaboration of the properties and behavior of the notional machine, crudely sketched above. Fourth, associated with notation

are the difficulties of acquiring standard *structures*, cliches or plans [1] that can be used to achieve small-scale goals, such as computing a sum using a loop. Finally, there is the issue of mastering the *pragmatics* of programming, where a student needs to learn the skill of how to specify, develop, test, and debug a program using whatever tools are available. None of these issues are entirely separable from the others and much of the "shock" [2] of the first few encounters between the learner and the system are compounded by the student's attempt to deal with all these different kinds of difficulty at once.

KINDS OF MISTAKES

Orthogonal to the areas of difficulty noted above are the kinds of mistakes that novices make about ideas in each of these areas. There seem to be three important types that teachers of programming should look out for. First there are errors due to the *misapplication of analogy*. These are errors that arise when the learner tries to extract more structure or relationships from an analogy than are warranted. For example, students often believe that since a variable is like a "box" it can hold more than a single value [3]. A second kind of error is concerned with *overgeneralizations* without any sense of misapplying an analogy. An example here might be the student surrounding the text following a REM statement in Basic with quotes because the text following a PRINT statement is quoted, or separating the actual parameters in a Pascal procedure call with semicolons because the formal parameters can be so separated in the procedure definition (such errors need not be limited to syntactic overgeneralizations). A third sort of error arises through inexpert handling of complexity in general, and *interactions* in particular. Thus we find different sub-parts of a program improperly interleaved [4], or the perceptual shape of the program on the screen interfering with a correct appreciation of what its text actually denotes.

This article is a brief introduction to some of the issues that teachers of programming may find helpful. It starts by presenting a fairly idiosyncratic view of teaching programming which makes use of mechanistic analogies and points out some of the pitfalls. The article goes on to examine certain errors based on the misapplication of analogies as well as certain interaction errors. The main emphasis is on the notional machine both at the general level of, say, understanding (and misunderstanding) the relationship of the terminal to the computer as such, as well as at the more specific level of, for example, understanding assignment. There is also some discussion of notation and of the sorts of mistakes that poorly-designed languages can induce novices to commit.

THE NOTIONAL MACHINE

An Engineering Analogy for Computing

Learning to program is like learning to use a toy construction set, such as a Meccano, to build mechanisms, but as if inside a darkened room with only very limited ways of seeing the innards of one's creation working. To someone

unfamiliar with engineering many of the pieces inside such a set look very odd, and while the learner may have some general idea what the set as a whole is for–to build cars and cranes etc.–she or he may not realize how apparently dissimilar mechanisms can be decomposed into similar small chunks and how the shapes of the pieces in the set are suited to the construction of such standard chunks.

A running program is a kind of mechanism and it takes quite a long time to learn the relation between a program on the page and the mechanism it describes. It's just as hard as trying to understand how a car engine works from a diagram in a text book. Only some familiarity with wheels, cranks, gears, bearings, etc. and "getting one's hands grubby," gives the power to imagine the working mechanism described by the diagram and crucially, to relate a broken engine's failure to work to malfunctions in its unseen innards.

Even after becoming familiar with some of the pieces that make up a program, e.g., PRINT or IF, the novice may not see a program as a working mechanism in the same way as an experienced programmer [5]. This ability to see a program as a whole, understand its main parts and their relation is a skill which grows only gradually. Two comparisons are often made. One is to foreign language learning with its halting progression from disconnected words and small phrases to fluent speech. The other is to the way an experienced chess player can "read" a chessboard, where a novice sees only a jumble of individual chessmen.

A Tool-Building Tool

The computer plus its programming system is also a mechanism, but a mechanism which can be used to build other mechanisms (i.e., programs). Even if no effort is made to present a view of what is going on "inside" the learners will form their own. Without help, this view may be rather impoverished, relying on coincidence, and may be insufficient to explain much of the observed behavior. When things are going well, this may not matter, but understanding errors often requires knowledge of the sort "the machine was trying to do this . . . but came across that . . . and so did not know how to continue."

This view of the computer system as a machine has to develop at the right level of detail. Some learners are given instruction in the binary system, gates, the central processor unit, and other low-level detail, which they find impossible to relate to the machine they have contact with. The learner needs a very unsophisticated explanation of what is inside, but she or he does need some explanation. Teaching schemes based on "match box" computers and their like seem quite successful, especially in equipping the novice to solve novel problems, even if they make little difference to their ability to solve stock exercises [6, 7].

What Does the Screen Show?

Pressing ENTER or RETURN is an issue which causes difficulty at first but which soon becomes second nature to the students. However underlying it is the

question of whether what one sees on the screen is a record of what one has typed, a place to display the effects of certain commands, or a kind of window into the "inside" of the computer. An advantage of certain simple systems, such as the Sinclair Spectrum, which display only the current command line is that pressing RETURN has an obvious effect on what one has typed. It disappears, in a sense "into" the machine, and some effect is produced. The notion that one can change the line, using character deleting keys or whatever, up until one presses ENTER makes sense where executing a command causes it to disappear from the screen. However there are disadvantages in that, if each command is issued for immediate execution, there is no visual record of previous commands, so one cannot point out the one-to-one correspondence between objects drawn on the screen and commands previously typed.

Typing and other mistakes will occur when entering single commands. At first the novice will be unsure of how to respond to the subsequent error message. The beginner will wonder whether the command has been partially understood and partially carried out so that the sensible response would be to rephrase that part of the command that seems to have caused the error. Should the command be completely reformulated? Has some irrevocable internal change been produced?

More often than not the system will simply disregard the command which it could not interpret and issue a new prompt inviting the user to try again. Learning to treat errors as producing no internal change of state has its dangers later. It reduces the chance that the novice will spot when a real change of state has occurred, perhaps by accident, as when pressing an ESCAPE key rather than RETURN or ENTER. One can tell the novice that observing the type of prompt helps, but attention can easily be distracted from this small signal of internal state.

Modern bit-mapped graphic display screens may make life easier for the novice because different areas of the screen have quite different relationships with the underlying notional machine. One part may be a record of what has been done, another shows a file, yet another displays the results of some running program and so on. These different roles may more easily be contrasted since they can be shown in parallel.

Managing the Notional Machine

There is a crucial distinction between the different roles a computer can play which is often missed by the learner. One is as a machine which is executing one of the learner's programs. In this mode the computer effectively becomes the mechanism described by the program. Pressing the RUN button, or otherwise making the program run, turns the computer into a machine executing the given stored program.

In its other mode, the computer is used as a kind of "manager" of programs– to do the reading in, storing, and editing. In this mode, input from the keyboard

will be concerned to do something to a program, e.g., add a line to it, make some change, print it out, store it on disk, or delete it. The conventions or rules for how one tells the computer to carry out these actions will be fixed by command language of the system in question (and with modern systems they may even be iconic).

Confusing the issue further, there are other conventions which make up the rules of the programming language, such as Basic, in which one describes programs. These latter conventions are usually different from those used to type to the machine in its managerial role. For example, few languages allow us to include an edit command as part of a program.

It is important not to underestimate the problems which arise from the above, namely misunderstandings about "who" is being addressed and which set of conventions (i.e., which language) is appropriate. Learning a programming language involves not only learning that language, but also the language for managing programs as well as the language for editing programs, e.g., use of the cursor keys in a screen editor.

It is common for programmers at all levels of skill to forget who they are addressing and issue the wrong kind of command. Students often find it hard to escape from the muddle that this causes, partly because they take longer to recognize that something silly is going on, partly because they may have expectations that the system will do what they mean rather than what they say, and finally because they may not be familiar with the various methods providing for escaping from such situations, such as BREAK, ESCAPE, or control characters.

Beginners sometimes take over-drastic action to restore the computer to a standard state, e.g., switching it off and on again, or deleting a complete program where only some part of it is wrong. Some people give up the attempt to learn programming because they fail to master the use of the computer in its managerial role rather than because they cannot write programs.

Once the student starts making programs there will be the need to make a further distinction, i.e., between typing in the text of a program, editing that text, listing the program on the screen, or some other output device and running the program. Novices are sometimes directed to make their initial programs out of sequences of PRINT statements. This can cause confusion because the effect of listing the program containing the PRINT commands is not all that different from the effect of that program when it is run. So listing the program and running it produce very nearly the same kind of output, thus blurring the distinction between these two ideas.

The notion of the system making sense of the program according to its own very rigid rules is a crucial idea for the learner to grasp. Both our own use, as teachers, of anthropomorphic language, e.g., "it was trying to . . . ," "it thought you meant . . . ," and the use of English words and names (e.g., PRINT) in programming languages can mislead. These can suggest that the system has the normal human ability of being able to infer what is meant from what is said.

Early teaching examples of simple programs based on sequences of instructions for human execution (e.g., how to boil an egg) can also give the wrong impression in two ways. First, the language for the instructions is usually unconstrained and, secondly, the instructions usually gloss many issues which it is perfectly reasonable to leave implicit. Novices are often surprised at the level of detail in which tasks need to be programmed.

I do not wish to suggest that anthropomorphic terms are a bad thing. Treating the computer as a highly limited rational entity is a perfectly reasonable way to proceed. Indeed enlisting the students' tolerant pity for the impoverished intellect of the object can be a good strategy for dispelling any fears. Where the system produces good error messages it is often beneficial to get the learner to make errors deliberately so that she can see how the system works.

The Analogy with English

The use of English words in programming languages can mislead in further ways than just suggesting more intelligence than the system possesses. The English word is usually chosen precisely because it suggests something about its effect. The hope is that using words like PRINT, READ, or INPUT makes things easy. This is often far from the case. First of all such words have often come to have a very specialized meaning within computing which is far removed from their everyday connotation. Secondly, many of these words have several meanings in English and the learner can easily latch onto the wrong one. For example, people sometimes use "then" in the sense of what next: "I went to the shop and then I bought a paper." The boolean operator "and" is sometimes used in this sense as well, as a way of joining a sequence of actions together: "Wash your hands and set the table." "Repeat" misleads beginners who expect that there must be something already in existence to be repeated, and so on. Exasperation with a programming system can occasionally be caused by the mismatch between the designer's and the user's understanding of what is implied by a particular name.

Where Is the Program?

Typing in a program brings with it the idea of a program being some kind of object and of it being somewhere. The Sinclair Spectrum, for example, provides physical analogy in that, on pressing the ENTER button, the line just typed pops up to the top half of the screen where the program being built is shown. This uses the top half of the screen as a kind of window into the memory of the computer.

In more complex systems the notion of a place for holding code before it is placed in main memory is usually extended from the single line of the Spectrum to a buffer of arbitrary size. Editing an existing program means that there are likely to be three different versions of a similar object. One will be the program

actually in memory, the other will be a copy of it in the edit buffer incorporating whatever changes have been made, and the third will be the version held in a permanent file (this is likely to be the source of the version in memory). Most editors provide two methods of exit. One simply abandons the contents of the edit buffer and leaves the contents of memory or filing system unaffected. The other overwrites memory or the original file with the contents of the edit buffer. A problem for beginners is that although entry to an editor, and the consequent change of state of the machine, is usually well signalled by a change of prompt or a visual representation of the buffer, exit from the editor is a much quieter business. The large difference between the two exit routes is not usually marked in any visually distinctive manner. This is a manifestation of a very general problem in learning to program: the fact that many important internal changes of state of the machine are not externally signalled in a clear way.

Compiled Languages

The problems mentioned in the previous section, namely use of editors and keeping up with the system's changes of state are compounded when the novice's first language is compiled, such as Pascal, rather than interpreted.

Much technical detail has to be mastered, both to do with the language itself and the system for managing programs before even a simple first program can be run. In general only complete programs can be run, making it impossible to find out easily what a single command, procedure, or function will do on its own. This makes it hard for the novice to learn how to build programs out of smaller parts that he has become familiar with. One partial way around this problem is to provide program templates in which the novice is asked to insert or adjust some small segment.

Novices can be confused about the status of the compiled version of the program and expect that this will automatically keep in step with changes in the source code. A related error is believing that the source code has to be re-compiled each time the program is run, rather than just re-executing an existing compiled version of the program (if one does exist). Use of compiled languages brings other problems for the novice, especially in the diagnosis of errors, since there are usually few debugging aids and the run-time system often has little access to the source code to help construct meaningful error messages.

INAPPROPRIATE ANALOGIES FOR ASSIGNMENT, VARIABLES AND ARRAYS

This section looks at assignment as one small aspect of the notional machine and describes some of the ways that people misapply analogies as well as deal badly with interactions, such as the sequencing between commands. Many of the errors are taken from the work of Soloway and his colleagues at Yale [1, 3, 4, 8].

In the first place, assignment often has a kind of mathematical flavor derived from the names of variables, e.g., A or X, from the quantities assigned, e.g., 3 + B, and from the symbol used to denote assignment, e.g., =, := or –>. The mathematical flavor on its own may be enough to evoke negative reactions.

There is an asymmetry in assignment which confuses some learners. Thus in BASIC we can say

```
LET A = 2
```

but not

```
LET 2 = A
```

which looks entirely plausible.

A common exercise for beginners in Pascal is to interchange the values of two variables. This requires the use of a third, temporary variable, so

```
TEMP := A;
   A := B;
   B := TEMP;
```

Many students get these assignments in the wrong order and express individual assignments back to front. Difficulty in expressing the overall order of the assignments may be due to a lack of regard for the sequential nature of the three commands. They look a little like three equations which are simultaneous statements about the properties of A, B, and TEMP rather than a recipe for achieving a certain internal state (certain programming languages, not considered here, do indeed, make it possible to write equations which may be considered to operate in parallel).

Beginners are often puzzled by such assignments as:

```
LET A = A + 1
```

precisely because they have not understood the asymmetry and the sequential nature of the execution of even this single assignment. The "A"s on each side of the "=" sign are not being treated in the same way. One stands for a location and the other for a value. This difficulty is bound up with lack of understanding both of exactly what a variable is and of the rules for interpretation of statements in the language. Some people get round the difficulty in such statements as:

```
LET A = A + 1
```

by employing a second variable:

```
LET B = A + 1
LET A = B
```

The most common analogy for a variable is to some kind of box or drawer with a label on it. Without further explanation this can be confusing. In particular it is important to stress that each box can hold only one item at a time. An alternative analogy which underlines this idea is that of a slate on which the value is written. Unless this idea is stressed some beginners may not realize that assignment overwrites the existing value. They may think of a variable as either a little self-contained stack or a self-contained list which has the ability to "remember" the history of assignments made to it. After all, most boxes hold more than one thing. They expect that the value overwritten is still available somewhere and can be retrieved.

Another misconception concerns the after-effects of assigning one variable to another:

```
LET A = B
```

Some novices see this as linking "A" and "B" together in some way so that whatever happens to "A" in future also happens to "B." For those who continue in programming there are, of course, situations like this, so the idea is not entirely fanciful. For instance, one of the difficulties when advanced students learn about pointers is understanding that side effects can be produced. So if A and B are pointers and one is assigned to the other (in Pascal)

```
A := B;
```

following down the pointer "B" and changing some part of the structure pointed at also changes the (identical) structure pointed at by "A." This conflicts with what they will have learned to be the case when "A" and "B" hold other kind of data.

The above is an example of a hard issue that occurs in many places when teaching computing, namely the difference between identity and equality. In other words knowing when two objects are in reality the same object as distinct from merely having correspondingly similar parts. A related issue,which is equally hard to grasp concerns objects which look identical on the terminal screen, or printed on paper, but which are really quite different objects. A favorite example is a string of characters such as 456 and the corresponding number.

A third misconception about assignment concerns temporal scope: the fact that the value does not fade away and hangs around until either explicitly changed, the contents of memory are erased or the machine switched off. This may be linked to the idea of a variable remembering its previous values.

A common mistake, when using a variable to keep a running total, is to forget to initialize the total to zero. This omission is reasonable when following the box

analogy. After all, if one has not put anything into a box, it's empty, which is sort of like zero. So adding the first item is like adding it to zero. Some systems exacerbate this misunderstanding by assigning default values to unassigned variables.

A further misunderstanding concerns the assignment of values derived from an expression

```
LET A = 7 + 4
```

A student may understand "A" to hold 7 + 4 as an unevaluated expression rather than 11. In some Basics "A" is only allowed to hold numbers, so this misunderstanding can be tackled by stressing the idea that a variable can hold only one number.

A very common idea is that assigning from one variable to another involves removing the contents of one box and placing it in the other. Thus the end result of

```
LET A = 2
LET B = A
```

is seen as "A" empty and "B" containing 2. Underlying many computer operations is the notion of making a copy and there then being two objects with independent existences. One of the problems for the learner is distinguishing an operation which implies copying, and so independence, from one of sharing and so dependence.

Names are another source of trouble in assignment and in learning to program in general. The policy of using meaningful (to the human) variable names can suggest that the names are meaningful to the computer as well and that there is some link between the human meaning of a name and its corresponding value.

It is a useful teaching strategy to occasionally introduce the use of nonsense names for variables and other entities. This helps to dispel expectations that the machine understands English. However, it may prove surprising to someone who has only understood the system by thinking that it "understood" the appropriate words, so it is as well to be prepared for the resulting confusion.

Arrays

Many of the difficulties concerning assignment of numerical values carry across to assignments involving arrays. In addition, arrays themselves introduce their own problems. Beginners muddle up the array as a whole with a single cell within that array. They also often confuse the subscript which denotes one of the cells with the value stored in that cell.

Arrays are often illustrated as sets of contiguous boxes. Sometimes these are shown in a horizontal line, sometimes as a vertical line. While it does not

particularly matter when dealing with one-dimensional arrays, drawing one-dimensional arrays horizontally can cause problems when later dealing with two or more dimensions. This is because it may conflict with standard conventions and so make it harder for the beginner to grasp which subscript deals with which dimension. As it is, students often find it hard to distinguish rows and columns and which subscript should be varied and which held constant in order to progress along a row or down a column.

The confusion between the subscript of an array cell and the value stored there can be deepened by the more general problem of understanding assignment and that symbols on the left of the assignment refer to locations and those on the right to values. See, for example, the following which uses square brackets to enclose array subscripts:

```
A[3] := A[4] + A[2];
```

and more difficult:

```
A[3] := A[3] + 5;
```

This last case is hard for all the same reasons that understanding

```
a := a + 5;
```

is hard with the added complication that the beginner may wonder whether it is the subscript which is being added, rather than the value at A[5]. Luckily many of these early difficulties do not prove intractable, but "arrays" do represent a real hurdle for the novice programmer. At a later stage further problems arise because of the way that arrays are used to refer to data indirectly, e.g., arrays holding subscripts as their values. This in its turn is an example of a generally hard problem which programmers at all levels find difficult, namely issues concerned with "indirection," i.e., use of pointers, etc.

Beginners get confused by the kind of language used to talk about arrays, for example, the words "read" and "write." We often talk about reading data in and out of arrays both when we mean simply assigning to, or copying from, as well when true input/output is involved.

A stumbling block for many beginners is building a program to input ten values, insert them in sequence into an array, process the array in some way, and then print out a result. Coordinating the necessary loop, array indexing and input/outout and array assignments often proves surprisingly difficult.

There is an interesting input error related to misunderstanding subscription and variables. Some learners write for example (in Pascal) [8]:

```
read(x);
while x < 999 do
begin
    sum := sum + x;
    x := x + 1
end;
```

Here incrementing in the penultimate line (x := x + 1) is understood as meaning read the next value of "x"–as if the values for it already existed in an array and the code meant that it takes on the next value in that array.

INTERACTION ERRORS

Flow of Control

Most students soon learn that a program designates a sequence of events and that these events usually take place on the same order as the instructions are set out in the program. What sometimes gets forgotten is that each instruction operates in the environment created by the previous instructions. Some beginners seem to imagine that the effects of each instruction are somehow saved up until the end of the program, at which point they all happen. Others fail to appreciate the ubiquity of the default rule about flow of control–namely that the next instruction is always executed unless the program instructs otherwise. Some seem to have fond hope that the system will of itself jump around and ignore sections of code which are not wanted under some circumstances. For example, consider the following program designed to print tables for an input number greater than 0.

```
20  INPUT I
30  IF I < 1 THEN GOTO 70
40  FOR J = 1 TO 10
50  PRINT J, "TIMES", I, "IS", J x I
60  NEXT J
70  PRINT "YOUR NUMBER IS TOO SMALL"
```

Here line 70 gets executed whatever the value of I. It's as if mentioning line 70 in the GOTO is thought of as "insulating" this line from entry except via that GOTO.

This kind of "drop through" error is quite common but can be helped by having the student single-step the program if the system supports this. The error can be viewed as an example of a more general and very common difficulty, namely dealing with a programming problem by subdividing it into subproblems and then not taking enough account of the consequent interactions between the

subproblems. In the example, the subproblems are "deal with numbers under 1" and "deal with all the other numbers."

There has been a great deal of discussion of the GOTO in the computer literature. Some maintain that like sugar it should be banished from the scene entirely, others suggest that it does not harm in small doses, and others again cheerfully sprinkle their code with it. There is some evidence that they make programs harder to debug (because it is harder to find one's way back through a program, against the stream of the GOTO's as it were) [9]. It is also rather harder to use indenting to help emphasize the structure of a program when using IF . . . GOTO as opposed to IF . . . THEN . . . ELSE.

Novices have trouble with IF . . . THEN and with IF . . . THEN . . . ELSE as they do with IF . . . GOTO (or syntactic variants). They forget that action 2 in

```
IF <boolean expression> THEN <action 1>;
<action 2>
```

in the above is executed whether or not action 1 is: again the same "drop through" error.

Loops cause beginners all kinds of trouble. FOR loops are troublesome because beginners often fail to understand that behind the scenes the loop control variable is being incremented on each cycle of the loop. This is another example of that ubiquitous issue of hidden, internal changes causing problems. Learners also fail to see that an expression involving the control variable will have a different value on each cycle of the loop.

Soloway and his colleagues have collected many examples of undergraduates' difficulties with WHILE and REPEAT loops in Pascal. Many of these center around a program to read in a sequence of values terminated by 99999 and compute the average of the values, not including the 99999. Some of the difficulties concerned the interaction between reading data and looping. For example, a solution might be produced where the first item of data is read outside the loop and so is not included in the total. Many of the problems concerned the ordering of statements inside the loop so that it either did not start or terminate correctly, e.g., by including the 99999 in the average.

Finally, some students treated the WHILE loop as if it generated some kind of interrupt. They expected that the loop could terminate at the very instant that the controlling condition changed value (imagining incorrectly, but reasonably, that the system constantly keeps a check on that value) rather than the next time it was evaluated as the loop cycled.

Some people find it very hard to grasp the idea of what goes on when a program reads in data, say from the keyboard (i.e., INPUT in Basic). In most languages the syntax disguises that a kind of assignment is involved and that the variable mentioned in the statement has its value changed (or initialized) by the statement. Beginners' programs sometimes contain redundant READ or INPUT

statements as if they acted as a declaration of intent to the use of the variable named later on. They do not appreciate that execution of the READ stops further execution of the program at that point and that it cannot continue until something is typed in by the user. They do not understand how this statement transfers responsibility from the system to the person at the keyboard to ensure that something happens. I have seen many beginners getting puzzled and complaining that nothing is happening—even eventually breaking out of the program altogether—because they have not understood that the system was waiting for them rather than vice versa.

Perceptual Interactions Between Form and Content

Many syntactic errors in Pascal are associated with the semicolon [10]. Beginners misuse the semicolon in various ways, by omitting it when it should be included, by inserting it when it should be omitted and by muddling up its use with that of the comma.

Semicolons are often omitted in places where the layout of the program appears to make them perceptually redundant, e.g., in the positions surrounded by | |symbols in the following examples.

```
program myprog |;|
var x, y : integer;

functions double(i : integer) : integer;
begin
   ~~~~~~~~
   ~~~~~~~~
end |;|

begin {main program}
   ~~~~~~~~
   ~~~~~~~~
end.
```

In some ways the better the program is laid out, the less need there seems for some of the semicolons. A similar effect seems to apply to the BEGIN at the beginning of the main program which is also often omitted, especially when no procedures or functions are declared.

Redundant, but harmless, semicolons are often placed just prior to an "end" or an "until":

```
begin
   x := 3;
   y := 5;
end
```

Some teachers argue that putting a semicolon in at this position is good practice in that otherwise if an extra line is subsequently inserted before the "end," it is easy to forget to include the terminating semicolon on the previous line.

Extra semicolons in certain positions can disrupt the logic of a program to cause either compile errors or peculiar (and often hard to understand) run time behavior. A favorite position for an extra semicolon is between the "then" part of a conditional and the "else" part:

```
If x > 4
then writeln('x is greater than 4') |;|
else writeln('x is less than or equal to 4');
```

The semicolon surrounded by | | symbols looks perfectly harmless sitting at the end of a line just like other semicolons. The compiler however treats it as the terminator for the "if" statement and then wonders what to do with a new statement starting with "else." Often the error message is not very explicit about what the novice has done wrong.

As mentioned earlier, careful layout of a program can hide misplaced semicolons, whether they be omitted or inserted. As an example of the latter, consider the following program extract:

```
for i := 1 to 10 do |;|
begin
   myarray1[i] := 0;
   myarray2[i] := 0;
end;
```

Perceptually the scope of the FOR loop includes the compound statement from BEGIN down to END. The intention of the loop is to initialize ten elements of two arrays to zero. Because of the extra semicolon, marked, the effect is to execute the FOR loop ten times doing nothing at all and then execute the compound statement once, using whatever value i has (usually ten) once the loop has terminated. This is the kind of behavior that makes novices despair of ever learning to communicate with the computer.

CONCLUSIONS

Programming is a complex skill to learn where even languages designed for the novice such as Basic and Pascal contain many traps for the unwary. Two issues stand out in the examples I have cited. First is the need to present the beginner with some model or description of the machine she or he is learning to operate via the given programming language. It is then possible to relate some of the troublesome hidden side-effects to events happening in this model, as it is these

hidden, and visually unmarked, actions which often cause trouble for beginners. However, inventing a consistent story that describes events at the right level of detail is not easy. Very often an analogy introduced at one point does not fit later on, so producing extra confusion in addition to any misapplication of the analogy at the point where it was appropriate.

Occasionally teachers have complete control, not only over the content of the teaching syllabus, but also over both the syntax and semantics of the programming language, as well as over the associated command language.When this is the case it is possible to ensure that the system behaves in a way which is both self-consistent as well as consistent with what the learners are being told by their teaching notes. Most teachers are not in this fortunate position and have to put up with, for example, terrible error messages from their computer whose content, if not impossible to understand, may well be at odds with the story about the machine that they have told the students.

The second important issue concerns the way that learners form a view of how the programming language works and what is going on inside the computer. Very often they form quite reasonable theories of how the system works, given their limited experience, except that their theories are incorrect. They may derive from chance associations with the meaning of English words used in the programming or command language, from overgeneralizations from one part of the language to another, or from the application of inappropriate anlogies. I have often been surprised at the bizarre theories about how the computer executes programs held even by students who have successfully "learned to program."

ACKNOWLEDGMENTS

This article is an abridged version of a longer work commissioned by the Scottish Microelectronic Development Programme. I thank my colleagues, Peter Gray, Roly Lishman, and Josie Taylor for helpful comments and the anonymous referee for convincing me that abridging the original was a worthwhile task.

REFERENCES

1. E. Soloway and K. Ehrlich, Empirical Studies of Programming Knowledge, *IEEE Transactions on Software Engineering*, *SE-10*:5, pp. 595-609, 1984.
2. C. B. Kreitsberg and L. Swanson, A Cognitive Model for Structuring an Introductory Programming Curriculum, *AFIPS Conference Proceedings*, *43*, pp. 307-311, 1974.
3. M. H. Burstein, "Learning by Reasoning," unpublished doctoral thesis, Department of Computer Science, Yale University, New Haven, CT, 1985.
4. J. C. Spohrer, E. Soloway, and E. Pope, Where the Bugs Are, *CHI-85 Conference Proceedings*, San Francisco, 1985.
5. B. Shneiderman, Measuring Computer Program Quality and Comprehension, *International Journal of Man-Machine Studies*, *9*, pp. 465-478, 1977.

6. R. E. Mayer, Comprehension as Affected by Structure of Problem Representation, *Memory and Cognition*, *4*, pp. 249-255, 1976.
7. J. Statz and L. Miller, The Egg Series: Using Simple Computer Models, *The Mathematics Teacher*, *71*, pp. 459-467, 1978.
8. E. Soloway, E. Rubin, B. Woolf, and J. Bonar, MENO-II: An AI-Based Programming Tutor, *Journal of Computer-based Instruction*, *10*, pp. 20-34, 1983.
9. M. E. Sime, A. T. Arblaster, and T. R. G. Green, Structuring the Programmer's Task, *Journal of Occupational Psychology*, *50*, pp. 205-216, 1977.
10. G. D. Ripley and F. C. Druseikis, A Statistical Analysis of Syntax Errors, *Computer Languages*, *3*, pp. 227-240, 1978.

Direct reprint requests to:

Dr. Benedict duBoulay
Cognitive Studies Program
University of Sussex
Falmer
Brighton BN1 2QN
United Kingdom