Int. J. Human-Computer Studies (1999) **51**, 265-277
Article No. ijhc.1981.0309
Available online at http://www.idealibrary.com on IDEAL®

The black box inside the glass box: presenting computing concepts to novices

BENEDICT DU BOULAY
Department of Computing Science, University of Aberdeen

TIM O'SHEA
Institute of Educational Technology, The Open University

AND

JOHN MONK
Faculty of Technology, The Open University

Simplicity and visibility are two important characteristics of programming languages for novices. Novices start programming with very little idea of the properties of the notional machine implied by the language they are learning. To help them learn these properties, the notional machine should be simple. That is, it should consist of a small number of parts that interact in ways that can be easily understood, possibly by analogy to other mechanisms with which the novice is more familiar. A notional machine is the idealized model of the computer implied by the constructs of the programming language. Visibility is concerned with methods for viewing selected parts and processes of this notional machine in action. We introduce the term "commentary" which is the system's dynamic characterization of the notional machine, expressed in either text or pictures on the user's terminal. We examine the simplicity and visibility of three systems, each designed to provide programming experience to different populations of novices.

Introduction

One of the difficulties of teaching a novice how to program is to describe, at the right level of detail, the machine he is learning to control. Novices are usually ignorant about what the machine can be instructed to do and about how it manages to do it. Basing one's teaching on the idea of a notional machine is an effective strategy for tackling this difficulty. The notional machine is an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed. That is, the properties of the notional machine are language, rather than hardware, dependent. For example, a novice learning BASIC will be learning how to work a BASIC machine; and this is quite different from a LISP machine (say), both in terms of the mechanisms he must learn to understand and the class of problems that can be solved easily with those mechanisms. For a strategy based on a notional machine to be effective, the notional machine must conform to two important principles. First, the notional machine employed should be conceptually simple, and second, methods should be provided for the novice to observe certain of its workings in action.

For example, Mayer (1979) represents the workings of a BASIC machine in terms of a small set of "transactions", where a transaction consists of an "operation", an "object" and a "location". Each BASIC instruction is described in terms of one or more of these transactions. The objects and locations are the components of the notional machine, and the operations are the ways in which these components can be used. Transactions are neither a description of the hardware on which the BASIC is implemented, nor a formal definition of the semantics of the programming language. Rather, they characterize what BASIC instructions do in terms of a simplified, functional model of a computer. A transaction, and hence the underlying notional machine, can be explained using analogies and concrete apparatus.

The transactions characterize a mechanism that has sufficient structure to explain the sequence of events while a BASIC program is running, but is simple enough to be grasped by a novice and avoids over-technical descriptions that would be confusing and irrelevant. Mayer's transactions are "black boxes" whose own internal workings do not need to be explained. Now the idea of a transaction is not limited to movements of data, say, in response to an instruction. It can also be used to describe the notional machine at other levels including the movement of a complete program from backing store to the workspace.

Functional simplicity can be achieved by limiting the set of transactions and by ensuring that each instruction does not need too many transactions to describe its action. This aspect of the simplicity of the notional machine must be distinguished from two other aspects of simplicity in a first programming language, namely *logical* simplicity and *syntactic* simplicity. Logical simplicity implies that problems of interest to the novice can be tackled by simple programs, i.e. the tools are suited to the job. Syntactic simplicity is achieved by ensuring that the rules for writing instructions are uniform, with few special cases to remember, and have well chosen names.

Functional simplicity of notional machine is not enough on its own to guarantee learnability or ease of use since the majority of the machine's actions will usually be hidden, and will have to be inferred by the novice unless special steps are taken. This problem can be acute. For instance the manuals accompanying certain makes of pocket calculator make no attempt to explain the reason why given sequences of button presses carry out the given computations. The user must follow the manual's instructions blindly because it is difficult for him to imagine what kind of underlying machine could be inside that demands these particular sequences of presses. During the course of a calculation, he has to guess the current state of the device using his recollection of what buttons he has pressed since the device's previous recognizable state (e.g. all registers cleared) because the device gives little or no external indications of its internal state.

Computers larger than calculators usually provide the user with more information about their internal states, but even so people still get confused about which environment they are in. Florentin & Smith (1978) give examples of the sorts of clue that experienced programmers use to infer the state of the machine they are using, and they comment on the unintelligibility of some novices' communications with a computer where they had lost track of what state it was in. Now languages for novices can be implemented in such a way that some form of *commentary* is available. This commentary is the "glass box" through which the novice can see the "black boxes" working. It functions rather like the cut-away models of machines to be found in technical museums, and indicates the more important events going on inside. What counts as important depends on both the programming language and on the task in which the novice is engaged. If the novice is transferring a program from his workspace to backing store (say) he ought to be able to see a diagram showing the movement of the program as a unit. Whereas if he is tracing the execution of a BASIC program, step by step, he may wish to see a more detailed representation of the current transaction. In neither case, however, does he need to be aware of such fine hardware distinctions as that between the arithmetic unit and the central processor.

Adherence to this approach requires that programming should be learned initially on an interactive system, since batch entry and turn-around delays only increase the remoteness of the notional machine. It also means that the implementation of the programming language and the preparation of teaching materials must be developed as a whole, so that each refers to the machine in the same terms. The commentary, whether pictorial or written, should be at a level of detail appropriate to the novice's task and to his level of understanding of the underlying concepts. Its terminology and diagrams should be properly matched to the other explanations that are provided, for example, in teaching materials. Error messages are a crucial part of the commentary and they form an important window into the machine. So care is needed to ensure that they describe errors in terms of the components of the notional machine known to the novice. Thus if it is decided that the novice need not be aware of a stack, say, then stack overflow error messages should be reworded in terms of some other notion he has been told about, such as some less precise notion of "store size".

A conceptually simple notional machine does not necessarily imply either a low level language (such as a simulated assembler) or a weak high level language (such as BASIC). There is no reason why a high level language such as PROLOG, with its pattern-matching and backtracking facilities, could not be implemented to present the novice with a simple and visible notional machine. PROLOG (Pereira, Pereira & Warren, 1979) is a declarative language in which programs are written as a series of sentences that specify goals and sub-goals, rather than as an algorithm. The order in which the sub-goals are attempted is largely left up to the system to decide. Pattern-matching is used to select the next sub-goal to be attempted. One of the "transactions" of PROLOG would be concerned with the pattern-matching mechanism, and this would be treated as a given and not explained in terms of some more primitive machinery. Increasing the visibility of PROLOG would involve some method of dynamically displaying the stack of sub-goals and the process of scanning through the sentences of the program searching for one which matches the pattern of a sub-goal. One question that needs further investigation is whether this type of language has advantages for novices. For example, Miller (1975) found that novices were more at home with "qualificational" rather than "conditional" languages. They preferred instructions of the sort "put all red things in box 1" to the conditional form, "if thing is red then put it in box 1". That is a statement of the goal rather than an algorithm for achieving it. This preference probably derives from the way instructions are usually given in English, for instance in workshop manuals, and underlines the fact that instructing a computer is an "unnatural" activity and not at all like instructing a person. But caution must be exercised in attempts to make programming languages that look like English, lest the novice be fooled into believing that he is communicating with a machine with human capabilities and knowledge (Plum, 1977). The argument for PROLOG is not that it is like English syntactically—it is very different—but that the specification of plans in terms of goals rather than in terms of an algorithm has precedents in English.

We have found that some highly experienced programmers find special languages for novices quite distasteful, and regard pedagogically simple explanations as not telling the "truth". We believe that this arises from their attempts to use such languages in ways that they were not designed for, and from their concern for elegance and generality rather than human factors. For example, the simple line editors of LOGO and BASIC are fine for novices, working at teletypes, because their action is easy to understand, but very frustrating for those who have mastered more powerful facilities. In just the same way we might expect to find that racing drivers are impatient with slow, family saloon cars, though the latter are much better than racing-cars for the learner driver. Experienced programmers forget just how many of the implicit conventions about programming they have absorbed in their exposure to a variety of programming languages. For example, the work of Mayer (1976), Miller (1974) and Sime, Arblaster & Green (1977*a*) has demonstrated how much novices need to learn about flow of control specified by conditionals.

There is a certain amount of empirical evidence to support the intuitively plausible case for simplicity and visibility in a first language, though one difficulty about testing programming ability has been both the small samples used and the large variation in subjects' ability. Mayer (1975) has conducted a series of experiments which show that novices, given a description or a model of their first language's notional machine, learned programming more effectively than those without these aids. The benefit was greater when the problems subsequently given to the novices were unlike the training tasks and demanded more understanding of programming. However, presenting the model only at the post-testing stage did not help the students to answer these questions. So the students had to have their initial exposure to programming using the aids. Both Miller (1974) and Sime *et al.* (1977*a*), who wanted to test novices' ability to write and debug certain kinds of program, were able to dispense with a long, prior training session by describing and providing a concrete model of the notional machine employed.

One of the advantages of simulated assembly languages as first languages is that they are easy to explain in terms of a simple model of the computer. Their disadvantage is that they are hard for novices to apply to any but the simplest (mathematical) problems. As far as we are aware, only one experiment has compared the ease with which a high and a low level language is learned by novices. The experiment was conducted using high school children, aged between 12 and 15 years, and compared their progress while learning LOGO then SIMPER, SIMPER then LOGO and LOGO together with SIMPER (Weyer & Cannara, 1975). SIMPER is a simulated decimal assembly language. Rather surprisingly, pupils progressed best who learned both languages together. Although there was a certain amount of confusion between the languages, each illuminated aspects of the other and this advantage outweighed the effects of interference between the conventions of the two languages.

Towards simplicity

One path towards simplicity is by limiting the repertoire of both input and output to the notional machine. The former is achieved by having a small language with few constructs and the latter by restricting the machine's possible actions. This will almost certainly reduce the generality of the language but, if the domain of action is well-chosen, will provide a congenial environment in which to learn programming, especially if the range of machine action is clear to the novice. One example of this approach is provided by that subset of LOGO concerned with controlling a "Turtle". A Turtle is a small vehicle with an attached pen that leaves a trail on the floor as it moves in response to instructions to go forward, go backward or turn. By sequencing drawing instructions appropriately, the novice can make the Turtle draw pleasing patterns. Sometimes a Turtle simulated on a display screen is used to provide faster and more accurately drawn pictures than can be produced by the mechanical Turtle. With a repertoire of about 10 instructions and through the use of procedure/sub-procedure calls, it is possible to initiate a rich and interesting, but clearly delimited, set of machine actions, namely line drawings.

However, both simplicity and visibility of the notional machine can be spoiled by poor language design or implementation. Underlying functional simplicity can be masked by surface syntactic features which can give rise to ambiguities and can blur important distinctions. For example in BASIC the same symbol " = " is used to do a variety of different jobs, including assignment and testing for equality, so that the meaning of the symbol is heavily context dependent, thus offending against Weinberg's (1971) design criterion of "uniformity". Gannon & Horning (1975) found that students confused the assignment operator ":=" with the test for equality " = " in a comparison of two languages that differed in the way that assignment was denoted, as well as in other features. Gannon and Horning argued that some quite different symbol should be used, such as ← used in APL. Weinberg (1971) gives examples of lack of uniformity such as the restrictions in FORTRAN on the form of index arithmetic expressions. Such special cases increase the amount that has to be learnt by the novice by complicating the properties of the notional machine. In many implementations of LOGO the distinction between invoking a procedure and passing its name as an argument is glossed and the choice made by the interpreter is context dependent, thus rendering the language more complex than it need be. This problem can be solved in LOGO by demanding a more uniform parsing scheme, but it makes the language syntactically less like English. This in itself is no bad thing since it emphasizes the difference between a dialogue with a LOGO machine and a dialogue with a human.

Sometimes a superficial simplicity is achieved at the expense of expressive power by limiting the repertoire of inputs to the machine in an unhelpful way. Thus the naming restrictions and lack of control constructs in BASIC seem to give the novice less to learn, but quickly trap him in programs that are hard to read and understand. Although LOGO has its defects, its procedural nature and unrestricted naming allow interesting results to be obtained quickly by novices writing simple programs. Such logical simplicity is achieved by matching the language to problems of interest to the novice. What are suitable, simple problems in the given domain should be solvable by simple programs, so there can be no universal best first language. For example, SOLO with its data-base and pattern-matching is much more suitable than BASIC, say, for students of cognitive psychology because simple inferencing can be modelled easily without the overheads of string manipulation or equivalent necessary BASIC mechanism. Of course, writing programs to do number crunching in SOLO would be horrendous. Some attempts have been made to extend BASIC so that it can do simple text-processing (e.g. Brown, 1972; Raskin, 1974) in order to teach arts students programming. This domain seems more suitable than one in which problems about averages or tax computations might have to be set.

Towards visibility

Various attempts have been made to make aspects of a notional machine more accessible to the novice. A Slot Box (Hillis, 1975) can be used to illustrate flow of control through a program using the movements of a Turtle. The slot box is an input device consisting of a row of slots, representing a simple linear address structure, into which are placed tokens representing instructions to move the Turtle. The tokens are about the size and shape of credit cards and, as well as having an instruction name printed on them, have a computer-readable code designating that instruction. A program consists of an arrangement of these cards in the sequence of slots in the box. This program can be run and flow of control is indicated by the successive illumination of the bulbs beside the slots and is further externalized in the sequence of Turtle movements. Program editing is achieved simply by rearranging the tokens in the slots. A Turtle, consisting of a light spot on a display screen, is now implemented as part of an introductory course in Pascal (Bowles, 1979).

Another fruitful line of attack is by implementing a language in such a way that either pictorial or written traces can be displayed that comment on the actions taken by the notional machine during a program run. This requires the facility for single-stepping a program, so that the effect of each instruction can be examined.

Two kinds of information are useful to the programmer, sequential and circumstantial (Green & Arblaster, 1980). The former is concerned with the sequence of events as they occur in the program, while the latter is concerned with the particular circumstances which lead the program to be in a given state. More work has been done to provide automatic access to sequential information than to circumstantial information. For example, BIP is an impressive system developed at Stanford (Barr, Beard & Atkinson, 1976) that displays the effects of a BASIC program entered by the student. Sequential flow of control is illustrated by a pointer following the instructions as they are executed, and assignment is specifically commented upon on the display screen so that the student can see the current values of variables. Similar systems have been developed for FORTRAN (Shapiro & Witmer, 1974), for an operating system (Tracz, 1974) and for an assembly language (Schweppe, 1973), though BIP functions as programming teacher as well as a simulator for a BASIC machine.

BIP's multiplicity of roles, e.g. as a tutor, a BASIC interpreter, an editor and a commentator on BASIC programs, can lead to difficulties for the novice who must keep in mind with which of these notional machines he is dealing. It is commonly found (see, for example, Huckle, 1980; Florentin & Smith, 1978) that novices get confused when the computer changes state in this way. The novice finds it difficult both to discriminate between the various states of the machine and to remember the particular language conventions associated with each state. One way of reducing the latter difficulty is to give the programming language, the command language and the editing language the same syntax. This will not help the novice distinguish which environment he is in, but it will reduce the load of language learning to a single set of conventions. ELOGO (see below) goes some way in this direction by constraining the novice to carry out what would otherwise be command language instructions, such as program storage and retrieval, from within ELOGO using the same syntax as the rest of the language.

Novices make inferences about the notional machine from the names of the instructions, so it is important to choose names that have the appropriate connotation. Kennedy (1975) gives an example where novices, learning to use a computerized hospital filing system, confused the everyday meaning of some of the command names with the specialized meaning implemented in the system. This led them to use the filing system incorrectly. The users' understanding of the functional components of the filing system did not match reality. Some words, commonly used for instruction names such as LOAD and STORE, have much more specific connotations for experienced programmers than for novices. LOAD and STORE are usually used to label two actions where each is the inverse of the other. LOAD is used to mean move some object into a place where it can be worked on (e.g. a file into a workspace or a number into an accumulator) whereas STORE means the reverse movement. Both LOAD and STORE usually involve copying data destructively into the target location, *and* leaving behind a copy at the source location. These specialized meanings have only a very loose connection with the everyday use of these two words, and need special explanation if introduced as part of a first language.

Three examples

In this section we describe three languages developed specifically for novices. In each case the language was implemented under different constraints and for a different population of novices. SOLO is an interactive data-base language used by Open University students learning cognitive psychology and artificial intelligence. Our second example concerns a microprocessor based assembly language used in another Open University course for managers in industry. Our third example is ELOGO, an Edinburgh dialect of LOGO, designed as a language for school pupils, aged about 12 years, to explore mathematics.

SOLO

Students of Cognitive Psychology at the Open University learn SOLO, a language for manipulating a relational data-base (Eisenstadt, 1978). Here the "black boxes" are such mechanisms as inserting a symbolic description into the data-base, or pattern-matching against descriptions already in the data-base. In terms of the amount of machine code they involve, the instructions for carrying out these actions are more complex than BASIC instructions but their effects and properties are no harder to grasp. They function as primitive, undecomposable parts of the SOLO notional machine in just the same way as Mayer's transactions do for the BASIC machine. Eisenstadt (1979) found that Open University students were able to learn SOLO quite effectively, despite the fact that they had no previous programming experience and were working mainly on their own at remote terminals where help was not usually available.

Functional simplicity was achieved in SOLO by restricting the scope of the data-base searching mechanism and by "locking up" certain language features until the novice had progressed beyond a given point. Syntactic simplicity was increased by arranging that when a student typed the "IF" part of a conditional he was automatically prompted for both the "THEN" and the "ELSE" part. Sime, Arblaster & Green (1977*b*) showed that this was a successful method of reducing errors in conditionals. Some automatic spelling correction is carried out on the user's input.

Only a certain amount could be done to ensure the visibility of the language, because most of the users would be working at teletypes rather than at display screens, and teletypes are limited in the extent to which they can reproduce diagrams. However, data-base items to be displayed are presented at the terminal in a form that both suggests the meaning of the item and is in agreement with the teaching material, as follows:

```
FIDO
|
|
|
|---ISA--→ DOG
|
|
|
|---HAS--→ FLEAS
|
|
|
|---LIKES--→ BEER


DOG
|
|
|
|---ISA--→ ANIMAL
```

SELF SIMULATION ON A MICROPROCESSOR

The general principles of simplicity and visibility can be applied to much simpler programming systems. Recently the Open University produced a course for managers in industry that aims to familiarize them with some aspects of microprocessors (Open University, 1979). Part of the course provides "hands-on" experience of a small microcomputer. It was designed to give people some feel for what tasks a microcomputer can do and how it can be made to do them. The clients for this course are not expected to have any prior experience and the course is designed to help them appreciate the problems faced by their own development teams, and give them some understanding of their jargon. Cost played a part in determining the facilities of the microcomputer, which is sent through the post. The practical work is just one part of a course detailing the effects of the introduction of microprocessors in industry.

The small size of the system, based on an Intel 8049, precludes a comprehensive introduction to programming, but this is not needed by the customers of the course. The system provides only 10 instructions: LOAD, STORE, ADD, DECREMENT, JUMP, JUMP IF ZERO, INPUT, OUTPUT, CALL, EXCLUSIVE OR. Each instruction has its own button bearing an abbreviated form of the instruction name. Most of the store is taken up by an interpreter for these instructions, which are a subset of the microprocessor's own instruction set. Thus the main task of the device is to simulate a simplified version of itself. Only a small amount of space is left for user's programs and consists of 63 locations for programs and eight registers for data. This architecture forces a strong data/program distinction, though this is not necessarily a disadvantage in this context.

Instructions may be entered in one of three modes, binary, denary or mnemonic. There are buttons with mnemonics for each instruction that allow them to be keyed in easily (see the buttons on the bottom right-hand side of Fig. 1). A light indicates which mode is currently being used. The contents of a location can be observed as either binary or denary codes.

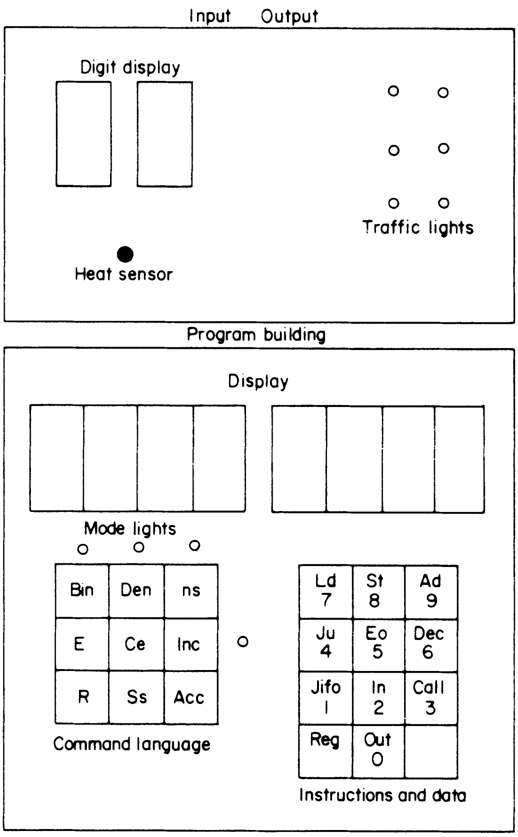

FIG. 1. Microcomputer layout.

The other buttons, on the bottom left, form the "command language" and enable instruction entry, examination of program locations and registers, and running or single stepping programs. These are the instructions for dealing with a program (e.g. editing it) as opposed to the instructions that comprise a program. This distinction has proved hard for novices (see, for example, Weyer & Cannara, 1975) and the separation of the two sets of buttons was designed to underline the difference.

The system contains a number of pre-defined subroutines that can be called by a user's program and whose instructions can be examined by him, though the code for the interpreter itself is inaccessible to the user. These subroutines are used as a "kit of parts" in a variety of applications and illustrate the idea of modularity of programs. The applications described in the manual include a program to display temperature as a two digit number read from a temperature sensitive input device, and a program to control six coloured light bulbs in a way similar to traffic lights. The timing subroutine can be made to work over periods of minutes and this underlines the fact that a micro-processor, obeying thousands of instructions per second, can be made to react over a much longer timescale.

An attempt is made to distinguish between those input and output devices that can be accessed by a user's program and those whose job is solely to allow him to communicate his program to the machine and examine or debug it. This is a similar distinction to that between the command language and the programming language. The two kinds of devices are physically separated in the layout, except for the program readable sense switch, which is among the buttons for the instructions.

The eight-digit display can be set to show either the address or contents of a program location, or the contents of the program counter. The multiple use of the same display leads to ambiguity since the novice cannot tell by looking at it whether it is showing the contents of the program counter or some other address. A similar lack of information was described earlier in the case of a pocket calculator. It could be solved by increasing the cost and complexity of the device and providing further display lights. The small number of digits in the display restricts the scope of error messages and other comments considerably, and most of them are just strings of symbols that have to be looked up in a table in order to decipher them. One solution to this would be to have the comments generated by having the words of the message cycle through the display, rather like messages on advertising hoardings consisting of an array of light bulbs. This solution would, however, demand more store space than was available.

The teaching materials introduce the notion of the "state" of the machine and the novice is invited to navigate around a state-transition network by pressing keys and comparing the observed behaviour of the machine with the descriptions on the nodes of the network. The transitions in the diagram correspond roughly with Mayer's (1979) transactions.

The notional machine is functionally simple, but this simplicity is achieved at the expense of having a complicated program interpreting the user's key presses. Attempts are made at visibility, e.g. the mode lights, the examinable code of the standard subroutines and the single stepping facility. Visibility could be improved by adding more "command language" buttons, e.g. to display the contents of the program counter, and by improving the error messages.

Despite the restrictions, the user can be introduced to a large range of computing ideas including: planning, coding, running and debugging programs, flow of control, subroutines, stack, conditional jumps, input and output of information, internal representation of data, addresses and contents. An important notion which is not covered is naming of both data and programs.

ELOGO

ELOGO is a procedural, interactive language with facilities for drawing using a turtle and for symbol manipulation using integers, words and lists as data-types (McArthur, 1974). The language is used to help children and adults develop their mathematical thinking by writing programs and by running pre-defined programs with interesting (usually pictorial) mathematical effects (du Boulay, 1980; Howe, O'Shea & Plane, 1980).

Special attention was paid to the problems faced by novices, including both their difficulties with typing and their unfamiliarity with the purposes and powers of a programming language. The initial introduction to programming is via a button-box and turtle, where each button stands for an instruction, see Fig. 2. The design of the buttons themselves gives clues to the language syntax. FORWARD and LEFT can be followed by one of the five number buttons, and DEFINE can be followed by one of the blank buttons.

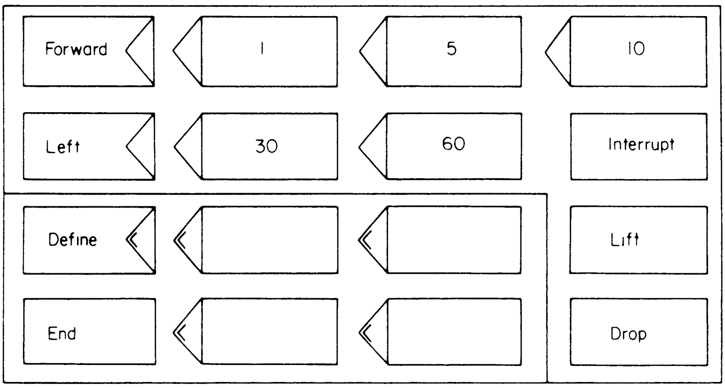

FIG. 2. Button box.

The labels on the buttons correspond to what the novice must type when he progresses to using a teletype. The blank buttons can be labelled by the novice and can be used to run or store a procedure of his own definition. A user-defined procedure can call other user-defined procedures or can call itself recursively. The simple notional machine implied by the button box and Turtle system is used as a foundation to build the user's understanding of the more complex, complete ELOGO system inplemented on a main-frame computer. Our experience has been that this introduction allays the fears that many novices have about computers in general and also their worries about not being able to type. The button box succeeds in introducing, in a straightforward and non-threatening way, many important computing concepts including: commands, arguments, stored procedure, sub-procedure call and recursion.

Once the novice has mastered the button box system he moves on to use the full ELOGO system. This contains many more facilities, some of which he probably will not need and will not be told about, such as an inferencing mechanism used to teach Artificial Intelligence (Bundy, 1978). Each novice is introduced to only those parts of the language that he needs to solve his programming problems.

In ELOGO the main task for the novice is the interactive definition, testing and debugging of procedures. The novice decomposes a complex programming task into simpler sub-tasks. This decomposition is mirrored in the structure of the procedure/sub-procedure calls of his program. Sub-tasks may need further decomposition and so sub-procedures may need to call sub-sub-procedures, and so on. A procedure, once defined, can be debugged, filed and run as an individual unit, so the sub-procedures of a program can be thoroughly tested before testing a procedure which calls them (although, chronologically, the super-procedure may have been defined prior to the sub-procedures).

Drawing a picture with the Turtle quite naturally decomposes into the sub-tasks of drawing the different parts of the picture, and each part can be drawn by its own, named procedure. In this context the novice can see the sense of the modular programming methodology, outlined above, and can attempt to apply it later on when dealing with, for example, more complex list-processing problems which involve decomposition into functions as well as procedures.

Efforts were made to keep the notional machine functionally simple. For example, the basic programming unit is the procedure, and the filing system was arranged so that individual procedures could be easily stored, loaded and borrowed from other users. A simple line editor is used (like that for BASIC) and this has proved adequate for novices working at teletypes.

Visibility was sought by ensuring that most of the important hidden actions, such as storing a procedure, were concluded with a written comment from the system. The teaching materials, a primer (du Boulay & O'Shea, 1976), were developed along with the language implementation. This meant that the comments from the system, such as error messages, could be worded using the same analogies as those used in the primer. Attention was paid to the debugging facilities which allow various forms of procedure tracing and single-stepping through a procedure.

Conclusion

Novices should be introduced to programming through languages that embody simple notional machines with the facilities for making certain of the actions of the notional machine open to view. This will almost certainly mean that much of the code in the language implementation will be used to make life easy for the novice, rather than to run his programs. This loss of efficiency is a worthwhile price to pay in order to make programming more pleasantly accessible to a wide cross-section of people.

We thank Thomas Green, David Hall, Jim Howe, Fran Plane, Peter Ross and Hugh Smith for helpful comments on a draft of this paper. Benedict du Boulay acknowledges the support of a Social Science Research Council grant while at the Department of Artificial Intelligence, University of Edinburgh.

References

BARR, A., BEARD, M. & ATKINSON, R. C. (1976). The computer as tutorial laboratory: the Stanford BIP Project. *International Journal of Man–Machine Studies*, **8**, 567–596.

BOWLES, K. L. (1979). *Microcomputer Problem Solving Using Pascal.* New York: Springer-Verlag.

BROWN, P. J. (1972). SCAN: A simple conversational programming language for text analysis. *Computers and the Humanities*, **6**, 223–227.

BUNDY, A. (Ed.) (1978). *Artificial Intelligence: An Introductory Course.* Edinburgh: Edinburgh University Press.

DU BOULAY, J. B. H. & O'SHEA, T. (1976). How to work the LOGO machine: a primer for ELOGO. *D.A.I. Occasional Paper No. 4*, Department of Artificial Intelligence, University of Edinburgh.

DU BOULAY, J. B. H. (1980). Teaching teachers mathematics through programming. *International Journal of Mathematics Education in Science and Technology*, **11**, (3) 347–360.

EISENSTADT, M. (1978). *Cognitive Psychology: Artificial Intelligence Project.* Milton Keynes: Open University Press.

EISENSTADT, M. (1979). A friendly software environment for psychology students. *A.I.S.B. Quarterly*, **34**.

FLORENTIN, J. J. & SMITH, B. C. (1978). Guessing the current state of the computer. *Proceedings of the Workshop on Computing Skills and Adaptive Systems*, Liverpool.

GANNON, J. J. & HORNING, J. D. (1975). Language design for programming reliability. *IEEE Transactions on Software Engineering*, **SE–1**,(2) 179–191.

GREEN, T. R. G. & ARBLASTER, A. T. (1980). *As You'd Like It: Contributions to Easier Computing.* MRC Social and Applied Psychology Unit, Sheffield, Memo No. 373.

HILLIS, D. (1975). *Slot Machine: Hardware Manual.* Massachusetts Institute of Technology. Cambridge, Massachusetts: LOGO Working Paper No. 39.

HOWE, J. A. M., O'SHEA, T. & PLANE, F. (1980). Teaching mathematics through LOGO programming: an evaluation study. In (Lewis, R. & Tagg, E. D., Eds) *Computer Assisted Learning: Scope, Progress and Limits.* Amsterdam: North-Holland.

HUCKLE, B. A. (1980). Designing a command language for inexperienced computer users. In (Buck, D., Ed.) *Command Language Directions.* Amsterdam: North-Holland.

KENNEDY, T. C. S. (1975). Some behavioural factors affecting the training of naive users of an interactive computer system. *International Journal of Man–Machine Studies,* **7**, 817–834.

MAYER, R. E. (1975). Some conditions of meaningful learning for computer programming: advance organizers and subject control of frame order. *Journal of Educational Psychology,* **68**, (2) 143–150.

MAYER, R. E. (1976). Comprehension as affected by structure of problem representation. *Memory and Cognition,* **43**, 249–255.

MAYER, R. E. (1979). A psychology of learning BASIC. *Communications of the ACM,* **22**, (11) 589–593.

MCARTHUR, C. D. (1974). EMAS LOGO: user's guide and reference manual. *DAI Occasional Paper No. 1.* Department of Artificial Intelligence, University of Edinburgh.

MILLER, L. A. (1974). Programming by non-programmers. *International Journal of Man–Machine Studies,* **6**, 237–260.

MILLER, L. A. (1975). Naive programmer problems with specification of flow of control. *Proceedings of National Computer Conference, AFIPS,* 44.

OPEN UNIVERSITY (1979). *Microprocessors And Product Development, A Course For Industry.* Milton Keynes: Open University Press.

PEREIRA, L. M., PEREIRA, C. N. & WARREN, D. H. D. (1979). User's guide to DECsystem-10 PROLOG. *D.A.I. Occasional Paper No. 15.* Department of Artificial Intelligence, University of Edinburgh.

PLUM, T. (1977). Fooling the user of a programming language. *Software Practice and Experience,* **7**, 215–221.

RASKIN, J. (1974). FLOW: a teaching language for computer programming in the humanities. *Computers and the Humanities,* **8**, 231–237.

SCHWEPPE, E. J. (1973). Dynamic instructional models of computer organisation and programming languages. *SIGCSE Bulletin,* **5**, (1) 26–31.

SHAPIRO, S. C. & WITMER, D. P. (1974). Interactive visual simulation for beginning programming students. *SIGCSE Bulletin,* **6**, (1) 11–14.

SIME, M. E., ARBLASTER, A. T. & GREEN, T. R. G. (1977*a*). Structuring the programmer's task. *Journal of Occupational Psychology,* **50**, 205–216.

SIME, M. E., ARBLASTER, A. T. & GREEN, T. R. G. (1977*b*). Reducing programming errors in nested conditionals by prescribing a writing procedure. *International Journal of Man–Machine Studies,* **9**, 119–126.

TRACZ, W. (1974). The use of ATOPSS for presenting elementary operating system concepts. *SIGCSE Bulletin,* **6**, (1) 74–78.

WEINBERG, G. M. (1971). *The Psychology Of Computer Programming.* New York: Van Nostrand Reinhold.

WEYER, S. A. & CANNARA, A. B. (1975). Children learning computer programming: experiments with languages curricula and programming devices. *Technical Report No. 250,* Institute for Mathematical Studies in the Social Sciences, Stanford University, California.