BEGINNERS NEED POWERFUL SYSTEMS

Aaron Sloman

gnitive Science Research Paper

rial no: CSRP 012

e University of Sussex
gnitive Studies Programme
hool of Social Sciences
lmer
ighton BN1 9QN

Revised April 1983

BEGINNERS NEED POWERFUL SYSTEMS

Aaron Sloman,
Cognitive Studies Programme,
School of Social Sciences,
Univesity of Sussex,
Brighton,
BN1 9QN

Introduction

Computing systems are often very hard for most people to approach. To some extent this is because of the inherent difficulty of giving machines the ability to communicate in ways which people find natural: not enough is understood about speech and how it is decoded, about the structure and functions of natural languages, about how the average person thinks about problems and processes. However some of the problems of man-machine communication arise out of two additional factors:

(a) Owing to the high costs of hardware in the past, systems have had to be designed to maximise machine efficiency.

(b) Most computer languages and software environments have been designed by experts for experts. (E.g. most programming languages are far too algebraic, to be approachable by the average user, and non-numerical applications too often have to be reduced explicitly to numerical representations.)

The first obstacle to real progress is rapidly diminishing. Very soon it should be possible to buy powerful 32-bit microcomputers with large amounts of memory very cheaply. These will give individual users the power currently obtainable only from much larger and more expensive machines. So problems of memory and processor time will no longer be an excuse for not making systems more approachable by ordinary people.

The second obstacle will remain, unless we can be far more imaginative and adventurous about designing languages, programming environments, operating systems, etc. But old thinking habits die hard, and computer scientists are often more interested in how to design systems about which one can prove theorems than in how to make systems more approachable for ordinary people. Work in Artificial Intelligence is an exception, but until now AI research e.g. into natural language communication, has often been hampered by a lack of computing power, and a lack of understanding of how to make systems easy for novices to use. Nevertheless, some of the languages and program development environments created to support AI research are more suitable for beginners than the more common ones. At Sussex University we are already using such a system, called POP-11, derived from POP2, for teaching absolute beginners. The same system is also used for advanced research. It is sufficiently powerful that most of the POP-11 system, including a sophisticated screen editor, is written in itself. (This makes it potentially very portable.) In fact our POP-11 system is embedded in a larger system, which we call POPLOG, since in includes PROLOG, and also LISP. These languages a powerful screen editor, and a large collection of library programs for teaching and other purposes, are all written in

POP-11.

With the new more powerful machines which will be increasingly available we shall be able, and we should try, to provide young programmers with a training which is more suited to the design of serious programs. This requires a language, or more precisely a programming environment, which encourages the development of clear, well structured, programs, and which makes systematic testing, debugging, and maintenance as easy as possible. Increasingly it is programmer time, not machine time, which is expensive.

Unless there is a co-ordinated effort at national or even international levels to look ahead to the needs of the future, there is a real risk that many schools will be stuck with an out of date approach to computer education, namely teaching based on BASIC. This can do permanent damage. I've seen evidence that some children are being turned right off computing by what they currently experience in school. How many of them will subsequently never take another opportunity to give computing a second chance? This and the bad habits taught to the minority who enjoy BASIC hacking leave me wondering whether putting micros into schools at present is doing far more harm than good.

Even if not everyone now can have the most desirable tools, at least more people should be thinking about them, developing and testing prototypes. A top-down approach is desirable. We should not be arguing now about which of the existing alternative languages is best. Rather we need to agree on features that should be present in languages and programming environments used for educational purposes, and then perhaps develop such languages, if no existing languages are found to be adequate. In particular, I want t, stress the importance of the environment, that is the editing facilities, the "help[1] facilities, the tracing and debugging facilities, the documentation facilities, the library facilities, the ability to share programs, the ability for users to collaborate on programs, and so on. The environment is at least as important as the language, if programming is not to be a real chore. A good environment will encourage the discipline of documenting and cleaning up programs. A poor environment will not.

What do beginners need?

What kinds of tools should be provided for teaching programming? At present (early 1982) there is probably nothing better than BASIC that is widely available on a variety of cheap machines. The environment that usually comes with BASIC has some good features which account for the popularity of the language. In particular it is extremely reactive, and very unfussy - e.g. variables don't need to be declared. But it has many faults which arise out of its having been designed (a) to minimise machine costs, (b) with mainly mathematical applications in mind. There has been so much public criticism of BASIC that I shall not enlarge on this. What sorts of languages and program development tools should we then aim at?

To provide an appreciation of the sorts of programs and applications that are already possible, and will be more widely available in the near future, beginners need a powerful language with a wide variety of features. Big building blocks make it easier to build more interesting and more useful structures. The language should not

only be easy to use for the first few hours. The language, and the general hardware and software environment should go on providing facilities which make learning, program design, testing and development as easy as possible for an increasingly complex and varied range of programs. This will make the use of computers fun and fruitful for far more people than at present.

Here are some proposals, based on experience with a number of different systems, and several years experience teaching non-numerate students and some children. These proposals can be put into effect using current hardware and software technology.

(a) The environment is as important as the language.
As in BASIC, the language should provide access to an editor, and the editor should have built in knowledge of the main language constructs, to minimise errors, and reduce the amount of typing. I.e. the editor should know about the formats for procedure definitions, conditionals, loops, list brackets, etc. A full range of editing commands should be provided, including searching, formatting, block moves, global substitutions, etc. Ideally the editor should make use of a screen with pointing device so that it is easy to make changes without a great deal of laborious typing. It would also be desirable to give spoken commands, in some contexts. The editor should be available for ordinary text processing, so that it is easy to write comments in programs, documentation, and project reports. (Training in documenting programs should be an essential part of any programming course. Many programmers are unable to explain clearly to anyone else what their programs do or how to use them.)

(b) Good syntax.....
It is widely acknowledged that a language should have far better syntax for conditionals and loops than most BASIC systems. Similarly the language should allow long variable names and procedure names, to make programs more readable, less error prone, and easier to develop and debug. Moreover, unlike LISP, for example, the syntax should be fairly redundant, so that programs are readable and the parser can give helpful messages if there are syntactic errors. Ideally the language (like POP-11, and some other AI languages) should allow the syntax to be extended by the user, so that the original language doesn't have to anticipate all possible types of applications in detail. This could be used both by teachers, who may wish to provide new commands for particular applications, and by more advanced students. For instance one of the POP-11 library programs demonstrates how sentences can be parsed. By defining "---" as a macro for use with that program, we allow students to type

```
--- this is a test sentence
```

instead of:

```
s([this is a test sentence]) ==>
```

(c) Use an interpreter or incremental compiler.
There should not be a slow edit-compile-link-load-test-edit cycle. For all but the smallest programs this requires the ability to be able to alter procedures at run time, and to type in commands in the language, to test procedures on different examples, without having to anticipate the commands and prepare a file which has to be compiled and run. This interactive testing should not require

patching machine code. Alterations to procedures must be possible at the source-language level. BASIC and some Artificial Intelligence languages provide this facility by interpreting source code represented as modifiable data-structures within the run time system. Others (e.g. POP2 POP-11, and some versions of LISP and PROLOG) provide an incremental compiler. Both approaches require the lexical analyser and parser, as well as the symbol-table, to be part of the run-time system. Allowing old procedures to be re-defined may also require the use of a "garbage collector" to reclaim the space occupied by old procedures, and also requires procedure calls to be indirect, so that no re-linking is necessary. In the old days, when memory and processor time were expensive, these requirements were serious obstacles. Now they are not.

(d) General-purpose data structures.
The language should have several more data-types than BASIC, e.g. lists and vectors, whose fields can contain arbitrary entities. Thus it should be easy to construct trees and networks with arbitrary components in them (numbers, lists, words, strings, etc.). For beginners there should be one very flexible general purpose data-type, for instance lists, with a built-in collection of useful procedures and good syntax for constructing instances. Such structures are needed for building programs which manipulate non-numerical data. For instance, programs which analyse sentences need to be able to treat a sentence as a list of words and create a list of lists indicating the decomposition of the sentence.

(e) General purpose data-manipulating procedures.
It should be possible to provide a lot of general-purpose procedures for manipulating data-structures. For instance, general procedures (e.g. intersection, union, membership tests, etc.) which treat lists or vectors as sets should be available and new ones easily definable. They should be general in the sense of being applicable no matter what sorts of objects are in the lists. This is easy in the AI languages, like LISP, POP and PROLOG. It is hard in BASIC since there are so few data-types available, and the improvements offered by 'more structured' versions don't address this issue at all. It is hard in PASCAL and similar languages because the rigid type structure rules out general procedure definitions. ALGOL68 provides facilities for getting round this, but they are not easy to use.

(f) Control of memory.
Storage allocation should be dynamic, and an automatic garbage collector should be available to reclaim inaccessible space, so that it is easy to write programs which build lots of temporary structures while exploring some problem. Beginners (and others) should not have to be bothered with the tedious algorithms required for reallocating memory. Garbage collection enables space to be automatically reclaimed when procedures are recompiled interactively. Even with falling memory prices, this will remain necessary for a friendly programming environment. PASCAL and other widely used languages don't normally provide garbage collectors. The need hardly arises in BASIC as it's so difficult to get to the point where you need a garbage collector! (If you are locked in a room all the time, you don't need public transport either.)

Recursion and local variables.
The language should permit recursive procedures with local variables, so that it is easy to write programs which explore trees and networks, and implement naturally recursive algorithms, for ^stance searching algorithms. Proper local variables generally acilitate the design of well-structured programs. Allowing mutual ecursion makes it possible to break down complex programs into odules which perform well defined sub-tasks no matter which other odules call them.

Lructure-building syntax.
>od syntax should be provided (as in POP-11 and LISP) for •nstructing lists and trees by typing in templates, making it necessary to go through 15 calls of the list link constructor to eate a 15-element list, for example. E.g. compare the syntax of st construction in POP-11

```
[A list C in a list J of words]
```

:h the syntax of a more conventional language:

```
CONS("A[#],CONS("list",CONS(CONS("in",CONS([a",CONS("list[,NIL))),
     CONS("of",CONS("words",NIL)))))
```

earner should not have to type the latter. For that matter ther should an advanced programmer! The beginner should not even i to know that when she builds and manipulates lists she is ntially making use of binary trees, any more than the person who ns to use a television set needs to learn about transistors, or car driver needs to learn about spark-plugs. We have nstrated at Sussex that students can learn to do quite advanced manipulation by thinking of lists only at a high level of ^action. The need to know more about the implementation arises when links are changed in a previously constructed list. By /ing list elements to accessed by numerical indices we also e the cognitive demands. Compare:

```
list(4)

front(back(back(back(list)))
```

e LISP version:

```
(cadddr list)
```

H directed structure decomposition.
n-matching utilities should be provided so that structures can sily checked and decomposed, without having to write awkward f nested function calls. Here is the POP-11 syntax for digging he second and third elements of a list named LIST, and ing the elements to X and Y respectively:

```
LIST -> C= ?X ?Y -=D
```

[l] means [ignore one item], and '==• means ignore any number ms, while •?' prefixes a variable which is to be given as he matching elememnt. Compare this with the more conventional

```
X := FRONT(BACK(LIST)); Y:= FRONT(BACK(BACK(LIST)));
```

Dortantly, suppose the programmer wants to make a list of all

the words between "I" and "you" in a given List, and assign the new List to VERBPHRASE. Here is the pattern-directed structure decomposition in POP-11:

```
LIST -> [-- I ??VERBPHRASE you --] 1
```

The "--" symboLs mean: ignore arbitrariLy many List eLements. The prefix "??" says that the next word is a variabLe to be given as its vaLue a List of aLL the corresponding eLements in the originaL List (the one on the Left of "->").

Compare this with the more conventionaL:

```
VERBPHRASE := NIL;
UNTIL FRONT(LIST) = "I" DO
        LIST := BACK(LIST)
ENDUNTIL;
LIST := BACKCLIST);
FOR ELEMENT IN LIST DO
    IF ELEMENT = "you" THEN
        RETURN(REVERSE(VERBPHRASE))
    ELSE
        VERBPHRASE := CONS(ELEMENT,VERBPHRASE)
    ENDIF
ENDFOR;
ERR0R("you" missing from List)
```

Programs based on patterns are much easier to understand, much Less error prone, and can be much more compact. For instance, having a pattern embedded in a pattern can be equivaLent to nested Loops in the more conventionaL Languages, e.g.

```
LIST -> t == CSENTENCE -= I ??VERBPHRASE YOU -= 1 -= 1
```

Here LIST is a List of Lists, and it is to be searched for a List starting with the word "sentence". That List is then to be searched for a set of words occurring between "I" and "YOU", and the List of aLL those words assigned to VERBPHRASE. Try writing that in your favourite Language.

Without cLaiming that the POP-11 syntax is the best that couLd be devised (and we are open to suggestions for improvements) we do cLaim that the use of this essentiaLLy graphicaL representation is much more naturaL and approachabLe than the kind of representation which most of the more aLgebraic Languages force on the programmer. (Of course POP-11 aLLows the conventionaL styLe of programming when that is required.)

(j) 'Object-oriented' programming.
For more advanced programming, data-cLasses and cLass hierarchies shouLd be definabLe, with inheritance of properties from cLasses, or super-cLasses, to instances, to faciLitate moduLar design of compLex programs. (These ideas derive from SIMULA-67 and SMALLTALK.) I don't have concrete evidence, but I suspect that most Learners wiLL not find it naturaL to express aLL programming ideas in this sort of framework, as required by SMALLTALK. It is good just to start with a coLLection of generaLLy useabLe data-types and generaLLy usefuL procedures for operating on them. Whether an instruction is presented as appLying a procedure to a structure, or sending a

message to a structure may not make very much difference for more sophisticated users. However I suspect that for the beginner the Latter may be very confusing. Few peopLe wouLd naturaLLy interpret 3 + 5 as an asymmetricaL request to 3 to add 5 to itself.

(JO Procedures ajs data.
Procedures shouLd be objects, the vaLues of variables, so that they can be created by programs, stored as data, and seLected by programs as required for probLem soLving tasks. (This ruLes out many conventionaL programming Languages, incLuding PASCAL, since they have a rigid separation of program from data). AdmittedLy, the need for this kind of power is not LikeLy to be feLt by a beginner programmer. But for many tasks where decisions have to be taken at RUN time, and the program needs considerabLe flexibility in its decision-making strategies, it is important not just to be able to embed conditionaL instructions in procedures, but aLso to be abLe conditionaLLy to change procedures avaiLabLe at run time to different parts of the program. The creation of new procedures at run time requires that the parser or cbmpiLer shouLd be avaiLabLe as a subroutine for user programs. The provision of good debugging tooLs may aLso require that debugging aids shouLd be abLe to treat procedures as objects which can be modified, so as to aLter their behaviour temporariLy for debugging purposes. Making a procedure produce trace printing is an exampLe.

U) Using the buiLt jn> parser/compi Ler for teaching
Another advantage of run-time avaiLabiLity of the parser, etc., is that teachers can write teaching programs which at appropriate points ask the system to read in and execute student commands. This makes it possible to use teaching programs which give students a great deal, of opportunity for initiative. If their •answers[1] are essentiaLLy programs which can be executed, then the teacher does not need to anticipate all possibLe answers. Students then feeL more in controL of their own Learning, and can, consequentLy, Learn more. The aLternative wouLd be for the teacher to write a command interpreter for constructs in the Language. But this is wasted effort if there is a compiLer. The Sussex University POPLOG system makes use of this kind of power for teaching purposes, in the ways described beLow.

^ Faci Lities for interacting during £ •break*•
More advanced students wi CL create programs which go wrong after constructing quite compLex databases, networks, Lists, etc. The fuLL power of the parser/interpreter/compiLer shouLd be avaiLabLe during error breaks, or user-defined breaks, to facilitate the testing and debugging of such programs. Most conventionaL Languages provide at best an interactive debugging aid which enables one to access values of variables, set break points, and single step through procedures. They do not provide the full power of the original programming language, so that the user can write new procedures after an error occurs to examine complex data-structures and print out selected results in a useful format, for example. Ideally it should be possible in an error break to redefine a faulty procedure and continue the computation from just before the error. This is possible in interpreted LISP systems. In special cases it is possible in POP-11.

(n) Terminal interrupts.
The user should be able to interrupt and suspend processing, run arbitary procedures during the break, then continue from the point of interrupt. This also aids development, testing debugging. BASIC provides this, unlike PASCAL, FORTRAN, ALGOL and other widely used languages. The trouble is that what you can do with BASIC when you have interrupted your program is as limited as what you can do in the original program. For instance, you can't easily define a procedure to crawl around a network printing out all the nodes which fail to satisfy some test, since in general that requires recursion.

(o) User defined 'interrupt' or 'exception' handling.
The nature of what happens at an interrupt or error should not be rigidly fixed, but should be user definable, and should be dynamically alterable. It should be easy for teachers to make the error messages different for beginners. Similarly some demonstration packages for use by beginners should be able to trap errors, and prevent confusing messages being printed out. (This facility would not be directly used by the more naive students.)

(p) Parallel programming.
There should be facilities for simulating running sets of programs in parallel so that it is easy to write various simulations of interacting systems. Often it is more natural and modular to specify two procedures, and talk about how they are to be synchronised, than to define a single serial procedure to do the same thing.

(q) Demons.
It should be easy to define demons which will run when certain conditions are met, e.g. when a particular variable has a specified value, or when a certain sort of data-structure is accessed or altered. The concept 'whenever' needs to be added to 'if', 'while', 'until', etc.

(r) Inference mechanism.
For many purposes it is useful to be able to give a program factual information, including inference rules, in a logical language of some sort, and leave it to the computer to have to draw the conclusions, instead of the programmer having to translate the inference rules into a procedural form. Many interesting programming problems require the program to search for a solution by systematically exploring alternatives. For this purpose it is often useful if the language has built-in facilities for such back-track searching. These facilitites are provided in PROLOG and related languages. PROLOG is becoming more widely available, though the most common implementations do not yet provide a rich environment of the sort described here. Moreover, although such 'logic based' languages are powerful and natural for a limited class of applications, e.g. the manipulation of a database of information some of which is represented explicitly and some of which is to be inferred on the basis of rules, or the analysis and transformation of linear or tree like structures, it is not clear that they are good for other problems, where a depth-first searching structure is not the most natural process organisation. The Sussex University POPLOG system combines PROLOG with POP-11, to provide the best of both worlds. [Hardy and Mellish 1982].

(s) File and terminal handling
For more advanced uses, there should be a variety of standard ways of reading in files, e.g. as bit streams, as byte-streams, as character-streams, as text-item streams, etc.

(t) Extendable syntax and data types.
For more advanced users the language should be extendable in a variety of ways. For instance users should be able to define their own additional data types (as in PASCAL and POP2) for greater modularity, and they should be able to define their own infix operations and "macros" to extend the syntax for special applications (as in LISP and POP2). Teachers would then find it easy to make available simplified commands for users with special needs, as was pointed out above. This would also make it easier to transport programs written in one dialect of the language to a machine running another dialect. Definitions of the relevant syntax extensions can simply be attached to the beginning of the program.

(u) Graphics.
For many beginners it is important in the early stages to have a good visual display of what the program is doing. For this and other reasons it is useful if good tools are available for writing programs which draw pictures on a screen, as in most LOGO systems. Graphical facilities are also useful for programs written by teachers to help students.

(v) Providing a total environment.
The language should give full and easy access to all operating system facilities such as reading and creating files, interrogating the system, examining directories, sending messages across a network (if there is one), etc.

(w) Help facilities
There should be a good on-line 'help' system. At the very least the user should be able to ask for the definition of a system procedure. There should also be ways of getting reminders of the syntax, and of some definitions of cncepts and techniques. An interactive help system will often be much more convenient than a printed manual, requiring tedious searches through tables of contents or index entries. Ideally the online documentation to cater for a variety of levels of expertise, providing both full explanations of concepts and facilities for those who don't know about them and terse reminders for those who have met them previously. A tree-structured menu-driven help system is sometimes used, but experience shows that this can be unhelpful if the decomposition into a tree does not match the conceptual structure of all users (i.e. when presented with a set of options at a high level it may be hard to decide which branch to take to get to what one wants). A help system requires many access routes. Ideally it should have an 'intelligent' interface which selects appropriate information on the basis of a dialogue in which the user's needs are clarified.

(x) A good shareable library.
Beginners should not have to construct their programs from the small building blocks typically provided as part of the definition of a programming language. Ideally they should have easy access to a library of procedures which they can use as subroutines in their programs, so that they will quickly reach the stage of designing

programs which do something interesting. Ideally such a library should be shared by a collection of users, so that if a new item is added it is immediately available for others. This requirement is more easily met by a single time-shared machine than by a collection of stand-alone microprocessors. Good networking facilities are required for the latter.

(y) Intelligent programming aids.
As program-understanding systems become available, it should be possible for the techniques to be built into 'monitors' which help users test and debug their programs. When making changes, the user should be advised automatically of potential inconsistencies. It should be possible for users to specify conditions which the monitor would check. This should be up to the user, instead of a collection of constraints being rigidly enforced by the language definition, as in typed languages.

(z) Facilities for sharing and communication
One of the great benefits of a time-shared or networked system is that different users can share each other's files, system enhancements, bug-fixes, or library extensions, are immediately available to all users, and users can communicate easily using a 'mail' facility. Many users of such systems have found it extremely useful to be able to communicate by computer. For instance, students in difficulty can leave mail messages for the tutor, instead of having to go and see her. We have found that many students make good use of this facility, and it suits tutors better, as they can deal with the problems at times which suit them. Computers in schools should be linked into networks, preferably networks extending across schools, so that the experts in one place can help people in another. National networks have proved very useful in this way to scientists and engineers, both in Britain and in the USA.

(aa) A good collection of intuitively meaningful packages for students to play with, use as subroutines in building programs, then later re-implement themselves. The success of Logo depends in part on the fact that motion and pictures are familiar and interesting to everybody (unlike quadratic equations, bits bytes or statistics, for instance). But this idea of building on familiar and entertaining 'micro-worlds' can easily be generalised. For instance, students can play with programs which 'think' about the movements of people or animals from one place to another. A database of information can represent the state of the world and be changed by executing procedures. For example, one of our teaching packages (originally written as a project by an undergraduate, then adopted by Max Clowes for teaching) introduces the world of the puzzle in which a man has to get a fox a chicken and some grain across a river, using a boat which will hold the man and at most one other item. Commands are available like

```
PUTIN([FOX]);
GETIN();
CROSSRIVER();
```

This sequence will produce a 'mishap' message announcing that the chicken has eaten the grain. Students can ask for the current database to be printed out at any stage. The syntax is slightly ugly, but it does introduce the idea of applying a procedure, the idea of a list of words, etc. But the syntax is not the main point.

Another micro-world involves simulating a hand moving blocks about on a table, making towers for example. However, in this case students are led through the process of building the programs which simulate the movements themselves.

In both cases it is very important that the simulation is not just doing something on the screen or on a real table with a real robot arm, but in a representation of the world, in the computer. For this introduces the idea of a computer which thinks about something, makes plans, tries out plans and strategies. Not only does this introduce very powerful ideas for thinking about how computers can be used, and abused, it also leads to new ways of thinking about how the human mind works. Having a computer control external things which it cannot perceive does not have these advantages. For this reason we have made our turtle create pictures not by doing something on a screen, but by 'painting' a two-dimensional array inside the computer. This can be printed out, but it can also be used as input to other programs which analyse the picture, and produce some sort of description of their structure. Again, students start by using library programs which do the analysis, then later they write their own programs to do this.

Other packages we make available include a program which enables students to type in a simple grammar, e.g. for a subset of English, and then explore its properties either by seeing which sentences it will parse, and what sort of analysis of the structure of sentences it produces, or by having the program generate at random sentences which accord with the grammar. The latter generally shows that the grammar allows far more constructions than the student intended, and trying to control this by extending the grammar can lead to real linguistic insights. Again, the fact that the computer is manipulating the rules for the grammar, presented in a form which is easy for people to read also, helps to indicate some of the power of computing, and raises challenging questions about how people understand language.

I cannot claim that we have come near to exhausting the potential of this approach. If students come to us for only one term, there is no time for the majority to explore more than four or five such packages, though the very best students manage to achieve far more than even the average ones. There is probably no right set of packages -- different teachers, different environments, different age-groups, can all be accommodated with different sorts of toys. Indeed, it is probably important for teachers not just to import someone else's packages, but to have direct experience of building their own, so that they have a deep understanding of the problems the students will encounter when they try to construct their simpler versions, or when using the package produces unexpected behaviour.

The presentation of these teaching packages in the POPLOG system makes very heavy use of features of the environment described above. The editor is used by the students to read a library file which gives instructions. It is also used to type in commands. The incremental compiler is used to compile appropriate library files, and also to compile and execute commands typed in by the student. Powerful list manipulating and pattern matching facilities make it relatively easy for the teachers to build interesting programs, and for the students to do likewise. The mail system and other

facilities allow users to help one another, to share files, to request help from the teacher, etc.

The development of computing languages has shown a steady move towards more and more sophisticated virtual machines, vastly improving the ease of communication between person and computer. There is no reason to believe that existing programming languages have achieved anything close to the maximum level of comprehensibility. Much research is needed into the reasons why it is so hard for all but a small proportion of the population to become good at writing programs. Perhaps this will lead to a collection of new constructs for programming languages. In particular, there is no reason why at Least for the non-expert programming languages should not be far more closely modelled on natural languages. Why shouldn't the programmer be able to say:

Make a list of all the words between "I" and "you" and call it VERBPHRASE.

insteady of having to use some artificial and unfamiliar language? Of course natural languages are often ambiguous and imprecise, and that means that communication has to be two-way, with the hearer free to ask for clarification and disambiguation. Similarly, programming could be much more of an intelligible dialogue with the computer. For professional programmers writing programs to be used by others, it is very likely that artificial languages will remain superior. But for beginners and for those who wish to use computers as non-specialists, it should be possible to use a more familiar and natural language. It is not easy to give a computer the ability to understand unrestricted English, but work is progressing, and it shouldn't be too long before much more friendly systems are possible than anything currently available. Of course, they will make heavy demands on processors and memory, but that should not be a problem.

One of the implications of these ideas is that beginners need very sophisticated systems, not simple systems. It might be thought that different systems are needed for advanced users and sophisticated programmers. As a teacher, I have found I prefer the language I use for writing complex programs to be essentially the same as the one I use for teaching, even if the students only learn about a subset of the language. Some of these features would not make much difference to absolute beginners, but the more advanced pupils (aged about 12 upwards) would be able to benefit. Moreover, teachers would find such a language much better for writing teaching programs than BASIC and other common languages. And our experience shows that even absolute beginners benefit from having good list-processing facilities from day one.

This collection of desirable features will probably not make much sense to someone whose experience so far is restricted to primitive languages like BASIC, FORTRAN, PASCAL, and assembler languages. I sometimes find that pupils with such experience have more difficulty learning to use powerful techniques, like recursion, than beginners, who have no previous experience of computing. Going the other way, from a good language to a poorly structured one is easier, though very annoying!

I am worried that our schools will produce pupils who have got used to a style of programming which is most unsuited for the design of complex programs, and that such pupils will find it hard to absorb more

powerful concepts and techniques, like people used to Roman numerals resisting the move to Arabic notation. (After all, they'd probably argue, 'I II III' is clearly superior to '1 2 3'. Never mind the problems of expressing '562,349' in Roman numerals!

I am also worried that instead of helping lots of children to approach computing, the present practice of putting very limited microprocessors, with primitive languages, into schools will not only teach the gifted few bad programming habits, but will permanently turn the ordinary majority against computing. They will find it boring, frustrating and difficult, and, having been permanently turned off computing, may therefore not take up the opportunities to learn to use much better systems which will be available in a few years. Perhaps it would be better to keep micros out of schools until far more approachable systems can be provided! I wonder how many micro-computers eagerly sought as Christmas presents not long ago now lie unused in cupboards because they are so difficult and frustrating for the ordinary user? Alternatively they may be used solely for the purpose of running game-playing programs of a type which may teach some manual and visual skills, and little else.

The software knowledge to design and implement better languages is now available. Machines with the power to make such languages cheaply available will soon be on the market. A co-ordinated initiative, with co-operation between government, manufacturers, and schools could put really powerful and effective computing resources at the fingertips of all children within a few years.

But I fear this will not happen because too many people are too content with what they already know and love. There is plenty of evidence already that in the world of software it is not necessarily the fittest which survives.

ACKNOWLEDGEMENT
I thank Masoud Yazdani for helpful comments on an earlier draft.

NOTE:
Readers may find it useful to consult

S. Papert, Mindstorms, Harvester Press, 1980.

S. Hardy, 'The Poplog programming system' Cognitive Science Research Papers, No. 3, 1982, University of Sussex.

C.S. Mellish and S. Hardy 'Integrating Prolog in the Poplog environment' Cognitive Science Research Papers, No. 10, 1983, University of Sussex.