Chapter 1.2

The Nature of Programming

T. R. G. Green

MRC Applied Psychology Unit, 15 Chaucer Road, Cambridge CB2 2EF, UK

Abstract

'Programming' is an exceedingly diverse activity, and many questions confront anyone who tries to say how programmers should work. This chapter attempts to describe some of the issues where cognitive psychology is relevant, without offering answers. Different programming cultures stress different virtues, on one hand neatness and well-definedness, on the other hand openness and effectiveness. The neat-scruffy differences show themselves both in individual styles and in claims that languages should be small, or that languages should contain all the necessary tools; although the argument ultimately depends on judgements of utilities, there are many cognitive issues not yet answered. The cultural differences are supported by various social mechanisms whose study would be of interest in its own right. Programming environments show similar divisions into neat or scruffy, but they also differ in being high or low-level. Low-level text-based environments are still commonest but structure-based editors, usually built on syntactic structure, are becoming more common. More recently, editors based on semantics have been investigated. Each higher level poses more cognitive questions, most of which remain unanswered. Developments in visual representations of programs also raise urgent questions of what information should be displayed and how, as do the increasing numbers of specialist languages. The chapter ends with a sketch of some of the most promising recent developments, indicating yet again a few of the cognitive issues that are raised.

Psychology of Programming
ISBN 0-12-350772-3

2.2 Individual style

The contrast between neats and scruffies certainly depends in part on the needs of the task and on the local culture, but it also depends in part on individual preference. Some people enjoy writing cryptic code, while others enjoy writing self-evident code. The mystique of cryptic programming, in which programs become fiendish puzzles for outrageously intelligent technocrats, was seriously dented in the 1970s by a series of polemics propounding the ideals of good style, such as Kernighan and Plauger (1974). This book contained a great deal of common sense; some of it was apparently wrong when tested in the laboratory – but all the same, it added up to a convincing case for straightforward, simple programming. Ingenuity for its own sake became less popular.

Despite that shift away from the baroque, individual style is still detectable in programming. What do we know about it? Next to nothing. Aptitude tests, widely investigated as a personnel selection tool, tell us nothing about programming style, and personality factors seem to have been unhelpful in investigations. One of the few successful investigations into personality factors in this area, perhaps even the only one, is the demonstration that risk aversion plays a part in success with certain types of programming environment, notably structure-based editors (Neal, 1987).

2.3 The urge to economy

Another type of contrast separates big, rambling programming languages from small, economical ones. Small languages using uniform principles are 'obviously' easier to learn and use, some say; others deny that, asserting instead that if a programmer needs facilities for, say, string handling, they should be provided, rather than having to be built from scratch each time. This is one of those disputes where each side has got hold of a truth, but equally each side refuses to see that the other side also has a truth. In consequence, I have heard some spirited arguments– and some incredibly silly claims. In general, neats tend to go for small languages, scruffies for big ones.

The basis of the 'small is better' argument is perfectly clear and intuitively appealing. Unfortunately it leads to absurdities. For example, in one version of Prolog the standard way to perform subtraction is by inverse addition. Addition is performed by writing

```
SUM (X, Y, Z)
```

with the effect that if `X` and `Y` were 'bound' (i.e. had already been given values) and `Z` was unbound, `Z` would be set to `X + Y`. No problem so far. But subtraction – setting `Z` to `X − Y`, say – is performed by writing

```
SUM (Y, Z, X)
```

This has two difficulties. It is hard for the programmer to work out how the terms should be arranged to generate the right sum, and it is hard for the reader to discover which is meant, addition or subtraction; that can only be discovered by working out whether the variables are bound or unbound at the time of execution. Here we see an interrelationship with the question of mental models of program behaviour: a possible reply, in the case of Prolog, would be that the programmer only 'ought'

to need to know that the relation `Y + Z = X` holds, without needing to know how that relation is evaluated. Empirical research casts considerable doubt on that claim, however. In any case similar absurdities arise in other contexts; Green (1980, p. 285) points out that 'small is better' predicts that it would be better to use the single logical operator `P/Q`, meaning 'at least one of `P` and `Q` is false', rather than the usual untidy collection of `AND`, `OR`, etc. But if you do that, then straightforward expressions turn into monsters:

`(p or q) and r` *becomes* `(((p/p)/(q/q))/r)/(((p/p)/(q/q))/r)`

This is neither comprehensible nor beautiful.

2.4 'Natural' programming

The inventors of new techniques of programming frequently assert that their technique is better because it is 'natural'. They rarely say what they mean by that, let alone discuss why their invention is more natural than other available techniques! Are these claims well founded? Is there a programming language that is natural? In my opinion, no. One day, maybe, but not yet.

For example, one school of thought supports logic-based programming, on the grounds that our natural mental model is supposed to concentrate on logical relationships rather than on the order of executing actions. The best-known logic-based language is Prolog, which has been extensively studied as a possible teaching language for children and for students. But, in reality, many difficulties arise in Prolog, some of which reflect the simple fact that logic may be logical, but it is not natural; it is not how people think. Empirical research indicates that Prolog novices find execution-based models of computation easier than logic-based models (see chapter 2.5), and that even experts use both types of model, not just logic.

Virtually identical claims have been put forward for object-oriented programming: it is supposed to simulate in some important fashion how we naturally conceive the world – in this case, however, we are supposed to conceive the world not as logical relationships, but as objects and classes of objects, communicating by doing things to each other ('sending messages'). Yet the interrelationships between real-world objects are far more subtle and various than the interrelationships that are available in any programming system, and typically a set of objects (in the programming sense) has to be defined with very great care and skill to achieve the right effect. Both anecdotal evidence and observational data (Détienne, 1990) show that programmers have difficulty in deciding which logical entities shall be represented as objects and which as attributes of objects. Object-oriented programming may be effective, but it is certainly not artless: and in that case, it is not natural.

2.5 The social maintenance of culture

The differences between programming cultures are neither accidental nor short lived. Yet in many cases they seem to be independent of the language itself; for instance, Pascal textbooks usually use rather long identifiers, while C textbooks use rather short ones. How are these cultural differences maintained? There are several mechanisms.

Firstly, the pedagogic traditions are very different. A cursory examination of textbooks dealing with neat languages will reveal the prevalence of certain tutorial examples, such as the 'eight queens' problem and sorting and searching problems, which are very clean and well defined. Each problem is usually solved as a complete enterprise, rather than being the basis of evolutionary growth, and there is frequently some analytical reasoning about the invariants of the program or about its performance with respect to size of data set. Students are exhorted not to start coding until they have fully analysed the problem, nor to approach the computer until the coding details are fully worked out. Ideally, the code should run correctly first time. Code that does not solve the problem is an 'error'. The 'scruffies' work with a larger problem repertoire, and frequently prefer to give space to domain-specific problems, such as interacting with the operating system, or algorithms for natural-language parsing, rather than to analytical reasoning. Students are expected to think in code, and to get the feel of hands-on experience as soon as possible; programs are frequently developed by incrementally modifying other programs, and understanding is gained by making 'fruitful mistakes', so code that does not solve the problem is seen not as an error but as a useful step on the path.

Secondly, there is considerable social pressure to conform to the local culture. Where Pascal is fashionable, it is common to sneer publicly at Basic ('encourages hacking, rots the brain') and Fortran (a 'dinosaur'). Pascal programs written in the style of Basic, with many 'go to' commands and with no proper use of procedure structure, incur real opprobrium. But Prolog experts sneer at programs making too much use of cut-and-fail as 'Pascal written in Prolog', asserting that programs should be conceived declaratively not procedurally. And so on for other languages. Adherents of one language frequently claim that learning another language first will ruin the learner's chances of ever being able to program 'properly'.

Thirdly, in certain circumstances there are clear demands for a particular culture. Development laboratories need freedom to experiment; but in commercial software production of large systems that are intended to have a long life, each program has to interact with other programs, written and maintained by other people, over long periods of time, and the impetus must be towards standardization and simplification – similar solutions to similar problems, standard coding and documentation styles, and formalization of change procedures. The non-conformist, either a would-be standardizer in a development laboratory or a carefree explorer in a production team, would experience very strong pressure to change.

While differences in tradition are inevitable, and in any case are probably quite healthy, the arguments would often be assisted by some proper data. Can any pedagogic advantage be demonstrated for any of the different traditions? Is it really true that choice of first programming language seriously affects subsequent programming ability? Yet again, the evidence is lacking.

3 The environment

Much of the work described in this book deals with writing and reading programs using the simplest of technology, just pen and paper or a simple text editor. In truth, that is how most programming is still done. Yet computer scientists have been remarkably fecund in their creation of alternative programming environments, and many interesting psychological problems arise. The design of an environment

presumably reflects someone's view of how to simplify some important types of programming task, but unfortunately the reports do not usually go into much detail about what tasks are intended to be supported. Furthermore, virtually no research has been reported on the advantages of different programming environments, which means that yet again we can do little beyond point out some of the questions that come up.

3.1 Neats and scruffies again

The characteristics of neat and scruffy environments reappear, of course, in their surrounding environments. The neat environment is closed and well defined; it is designed to encourage methodical, well-documented working, in which the coding stage is not started until the program is well-designed. Just as a single document completely defines the language, so the environment is well defined, with much emphasis on standardization across different locations.

The scruffy environment supports 'evolutionary' construction: users can slap a bit on here and a bit on there as their ideas develop. Xerox's Interlisp is a superb example. Not only can programs be built incrementally but Interlisp's 'advise' functions also allow second thoughts to be stuck in without having to recompile or even understand the original programs. Even the system programs can be modified by advise functions, allowing the environment itself to be modified without having to recompile all the source code. There is a tool for making programs try to run even in the presence of programming errors; this tool, called DWIM ('Do What I mean') completely defies the neat doctrine that a single document fully defines the language, because the behaviour of the program now depends entirely on how DWIM interprets it. The environment is open and extensible, so that from month to month new bits appear. The technology of electronic mail and of networks allows the new versions of the environment to be created frequently and mailed to users. Additions to the library of packages can easily be made by any competent user. Discovering what facilities are actually available in all these shifting sands can be quite difficult for newcomers: the simplest way to find something is to ask an expert. Smalltalk-80 is another extensible environment, which can be manipulated by programmers to suit their own needs or preferences. Expert advice on the available resources is needed very often by baffled newcomers to Smalltalk, just as in Interlisp.

3.2 The question of level

Programming languages (even low-level ones) contain a great deal of structure. It is possible to build and manipulate programs using high-level operations that reflect this structure. There are many differing opinions about high-level programming environments, reflecting differing beliefs about how programmers do, might or should think about programs and their construction, interpretation and manipulation.

Text-based editors remain the favourite choice, despite all subsequent developments. Present-day editors for use in programming environments provide some help with formatting programs, and editors specialized for programming use provide help with particular awkwardnesses – for example, editors for use in Lisp, with its many parentheses, provide some means to indicate which parentheses are paired together. A few editors, notably Emacs, can be customized to provide templates for lengthy and frequently used constructions, such as loops. In other respects the text-based

To start writing a function the user selects DEFINITION from the menu, which automatically generates the following code:

```
(DEFUN <NAME> (<PARAMS ...>)
       <FN-BODY ...>)
```

The angle brackets, <...>, surrounds slots that must be filled appropriately. Suppose the user next types the name FACTORIAL, giving:

```
(DEFUN FACTORIAL (<PARAMS ...  >)
       <FN-BODY ...>)
```

Next the user chooses to insert an IF-construct into the <FN-BODY> slot:

```
(DEFUN FACTORIAL (<PARAMS ...  >)
       (IF     <IF-TEST>
               <THEN-CASE>
       <ELSE-CASE>
       <FN-BODY ...>)
```

This goes on until the function is complete.

Figure 2: How Struedi, a Lisp editor, is used.

editors have changed little. Their users like them partly because they are simple to understand and do not add much of an extra learning load. More importantly, they do not constrain their users in any way at all. Programs can be built in any order – top-down, bottom-up, middle-out; pieces of code can be moved around the design at will; code can give way to remarks like 'I must finish this bit later'.

The major alternative is the editor based on syntactic structure, allowing users to manipulate their programs at the level of statements or expressions within statements. A fairly typical example among many others is Struedi (Köhne and Weber, 1987; Figure 2), designed to help novices learn to use Lisp without being bothered by syntax problems. It is based around templates of the language constructs, which can be filled in from a menu of various possibilities.

Although its design philosophy is not unusual, Struedi was built with exceptional care for human factors, and its authors even report a small experiment which suggested that Struedi helped with semantics learning, presumably by eliminating overheads spent on the syntax problems of Lisp (for a similar result, see Sime *et al.*, 1977). This sort of result is very encouraging to those who believe that learners suffer unnecessarily at the hands of badly designed languages presented in environments that give them little help.

Struedi is a neat language. It imposes an order of development on the programmer, and it insists that all changes to the program are made by modifying the source code, so that there is one and only one text defining the current state of the program. The code cannot be run until all errors have been removed from the text, so what the

text says is a true and faithful indication of the program's behaviour. The editor is based on a 'parse tree', a syntactic analysis of the text, which means that like several other systems mentioned here, it is based on control flow. It is debatable whether that is the information the programmer most needs.

A different approach was taken with 'KBEmacs' (Waters, 1985), a knowledgeable program editor in which the user can call on a library of programming clichés to help construct the program. A *cliché* is a typical program fragment, such as 'file enumeration', and the user can call it up by asking for it by name. However, the cliché contains more than code: it also specifies *roles*, such as the input role defining the file or other structure, the *empty_test* which determines when to stop, the *element accessor* which accesses the current element in the structure, etc. In the example shown by Waters (p.1308), the user issues a short sequence of commands, starting:

```
Define a simple_report procedure UNIT_REPAIR_REPORT.
Fill the enumerator with a chain_enumeration of UNITS and REPAIRS.
etc
```

A total of six such lines is sufficient to generate more than fifty lines of Ada, forming a complete and correct procedure definition.

KBEmacs is not meant to be used as a novice's tool, but as an aid to expert programmers. The intention is to formalize their knowledge so that it can be used by the computer. Every expert programmer is expected to know that a 'simple report' cliché will have a role which enumerates elements, and to be acquainted with the cliché of 'chain enumeration'. All the programmer has to learn is the vocabulary – what the clichés and roles are called. KBEmacs preserves the cliché structure while it builds code, so that subsequent editing can refer to roles ('Fill the enumerator with <something else>').

Waters makes two points that are important for an assessment of KBEmacs as a partner to humans. First, unlike many systems it does not force the user to preserve syntactic correctness, nor does it impose a pure top-down development. According to Waters (p. 1318) it 'supports an evolutionary model of programming wherein key decisions about what a program is to do can be deferred to a later time as opposed to a model where only lower level implementation decisions can be deferred'. Secondly, clichés operate in the domain of programming plans, rather than of text or parse trees, so it is fundamentally unlike the systems mentioned above.

Unfortunately, Waters adds, attempting to externalize programming knowledge would be a 'lengthy' undertaking, since programmers probably know thousands of clichés, whereas the 1985 demonstration version of KBEmacs knew only a few dozen and was already some 40k lines of Lisp code. It was also 'fraught with bugs' and ran very slowly. Nevertheless it is an exciting indicator of possible lines of progress.

3.3 What to display

In Chapter 2.2 ('Programming Languages as Information Structures') I review the main findings on the problems of visual representations of programs. As will be seen, many questions are left unanswered by present research. One of the major questions is just what should be displayed, given today's powerful screen-based environments. Some visual programming systems go little further than to translate a standard

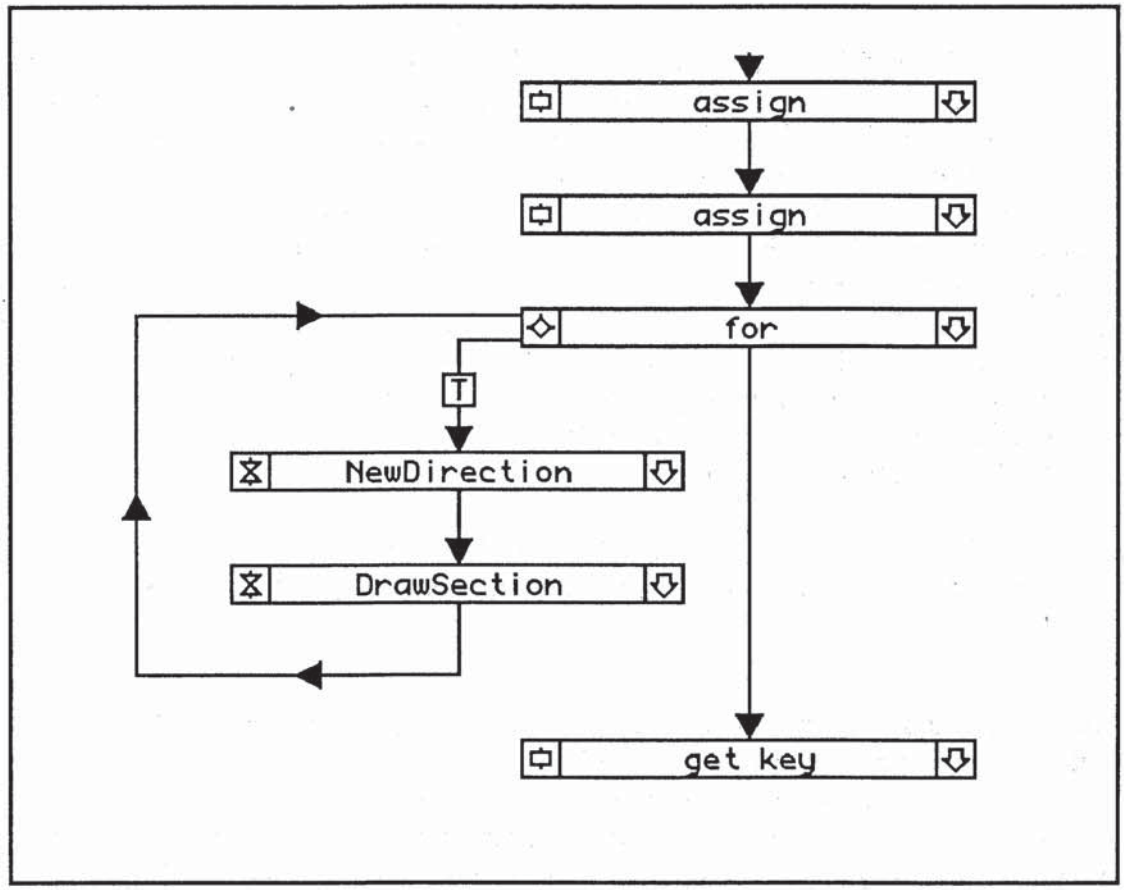

Figure 3: The VIP (Visual Interactive Programming) system, by Mainstasy Inc., provides an automated flowchart. A menu of tools allows the programmer to construct and manipulate the diagrams. Each box can be 'opened' to give more detail: the 'for' box, for instance, contains information about the details of an iterative loop.

programming language into a visual representation. A good example is Mainstay's VIP (Visual Interactive Programming), which presents the programming constructs of Pascal as labelled boxes, with arrows to mark the sequence of operation. A menu of tools provides for adding to and manipulating the diagram. VIP, as Figure 3 shows, is little more than an automated flowchart; this is evident in the level of abstraction, the vocabulary of 'assign', 'for', etc, and the choice of 'sequence of operation' as the main type of information to represent. Criticisms of structured flowcharts, such as those summarized in Chapter 2.2, include poor modularity and rigid control structures (see Green (1982) for a full account of research findings to that date). Such criticisms will probably apply to VIP with little change.

Other systems present information that is harder to represent in text. For example, good textual conventions for indicating parallel or concurrent execution are hard to devise: however hard we try to ignore the essentially linear nature of text, the mere fact that line A precedes line B insidiously murmurs to us that A is executed first. Putting special parentheses around the two lines to indicate that 'There is no constraint on order of execution within these bounds' is not an adequate solution. An example of a purely textual language designed to represent concurrent processes in that way is Occam, which uses the keywords `SEQ` and `PAR` to indicate sequential and parallel execution.

This Occam fragment shows one possible control structure for asking for two quantities to be input and then outputting the result of a computation:

```
SEQ
  PAR
    Width  := askWidth
    Length := askLength
  printArea (Width, Length))
  beep
```

The `SEQ` directs the computer to perform the subsequent commands in the sequence specified, that is, the paragraph starting with `PAR` (within which the two commands can be performed in any convenient order), then '`printArea`', then '`beep`'. Occam cannot show that the nature of the constraints is different for `printArea` (which logically cannot be executed until its data is ready) and for beep (which is executable at any time, but which the programmer has chosen to have executed only when all else is complete).

Parallel or concurrent execution is nevertheless relatively easy to represent diagrammatically. There is no built-in linear ordering in a diagram, it can simply be decreed that every box on the page is allowed to commence execution whenever the scheduler sees fit. 'Prograph', by The Gunakara Sun Systems, is such a language. The connecting lines in Prograph diagrams show *data flow*, not order of execution, and each box can be executed as soon as all its data is available. Different box shapes indicate different types of box: some supply constants as data, some take input from other sources, some invoke primitive operations, etc.

Programs are organized as 'methods', and the display is organized in terms of windows, each window representing a method. Figure 4 shows a simple method, using box shapes equivalent to subroutine calls – i.e., they denote methods defined in other windows. Because Prograph is a concurrent language, there is no constraint in this program on which of the top two boxes is executed first, 'ask width' or 'ask length'. However, 'print area' cannot be executed until both those have been executed, because it takes data from them. Thus Prograph's notation seems superior to Occam's in these respects.

But, given a visual display, the central question is *what to display*. VIP presents a simple translation of standard text into graphic form, as many other systems have done; for evaluations, see Chapter 2.2. Prograph presents material that would be hard to display as text in a simple form. Likewise, Brayshaw and Eisenstadt's 'Transparent Prolog Machine' and Böcker and co-workers' 'Kaestle' system try to *complement* the text: TPM animates the execution of Prolog programs, which is hard to deduce from the Prolog text, and Kaestle displays animated views of data structures in Lisp programs, likewise hard to deduce from the text (Brayshaw and Eisenstadt, 1988; Böcker *et al.*, 1986).

3.4 Browsing through programs

As programs get larger, programmers find it harder to locate the information they need. The techniques that have been developed to help them 'browse' include a wide range of very straightforward techniques, answering direct questions like 'where

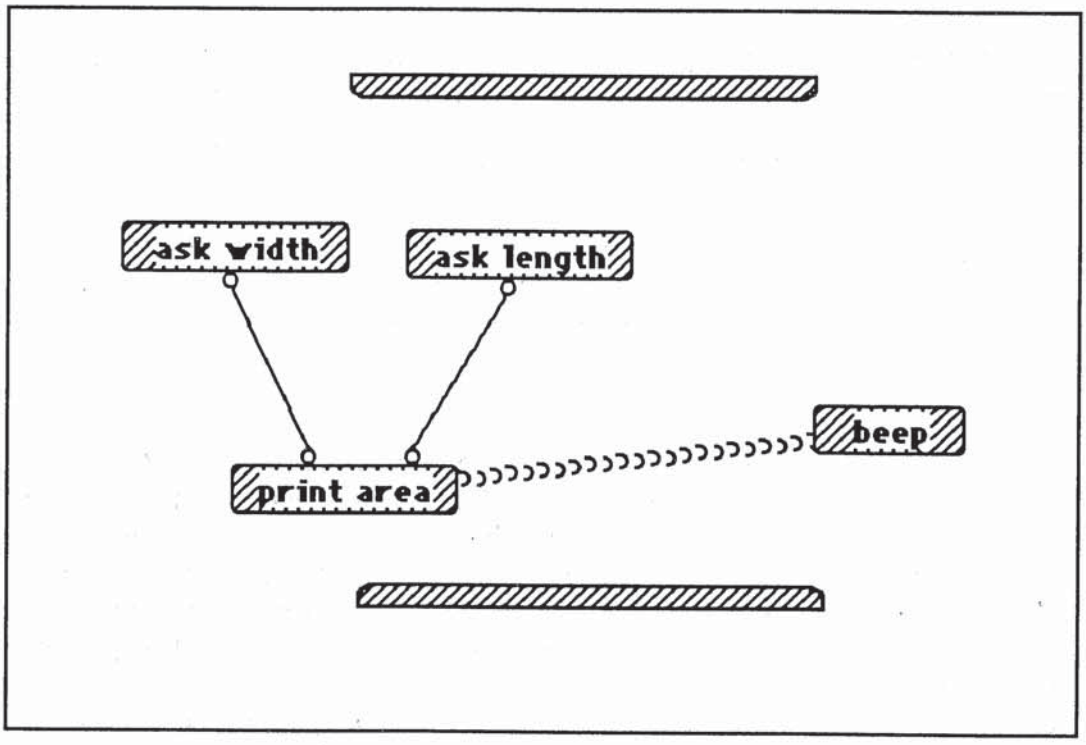

Figure 4: This is one 'method' from a Prograph program, itself calling further methods. Solid lines show data flow (from top to bottom); the 'horseshoe' line shows a user-defined constraint that 'print area' must be executed before 'beep'. In the absence of constraints any method may be executed as soon as its data, if any, is ready; thus this program does not specify which of 'ask width' and 'ask length' is to be performed first.

is this identifier used?' There are also some more interesting techniques based on psychological conjectures, either about the search patterns of people browsing large programs or else about the way information is mentally represented.

In the first category come browsing methods built on 'hypertext'. The conjecture is that with conventional systems programmers frequently need to interrupt their search for information about one aspect of their program to find out what something else does. Keeping track of where they were, and finding their way back, becomes a serious overhead. Hypertext systems provide built-in links, so that by say pointing to an identifier and querying it, the programmer is immediately shown the definition of the associated subroutine or data structure. Experience shows that hypertext users lose track of where they came from, so good systems help the programmer keep track of the search path and find the way back; for example, the recent 'Hybrow' system (Seabrook and Shneiderman, 1989) provides special support to help in managing the many information windows that make up the search trail.

Quite different in spirit are browsers that try to filter out irrelevant information. Among many such, the 'fish-eye view' (Furnas, 1986) deserves mention. Instead of showing the programmer a simple slab of C text, as much as will fit on the display, the fish-eye view in his example lists only the lines that seem to be important in providing context around the line that is currently being edited. Another example from that paper is a 'fish-eye calendar' where the current day is shown in detail, the current week in less detail, and the current month in still less detail. A similar approach, less well worked out, is taken in certain structure-based editors.

Browsers like these are obviously built in response to conjectures about what information programmers want and how they would like to receive it. There has been very little serious work on that issue. The simple conjecture of the previous section is that the greatest effort should go into making it easy for programmers to get at the information that is not easily extracted from the text, which implies that the fish-eye view might give its users less help than the hypertext browser. But that is yet another unresolved issue.

4 Sidestreams

So far we have concentrated on examples taken from what I see as the main line of progress in programming languages and environments. Around this line have developed huge numbers of experimental or special-purpose languages and environments, driven by the needs of the moment or by a spirit of exploration, and some interesting experiments in combining paradigms that appear at first sight to be incompatible. No survey would be complete without mentioning at least a few of these and considering whether they raise special problems, outside those raised by the more conventional languages.

4.1 Specialist languages

Some unusual and interesting designs for languages have developed from the need to control specific pieces of equipment. One of the earliest was Forth (Brodie and Forth, Inc., 1987), developed originally to control astronomical observation equipment. Since the control equipment was driven by primitive microprocessors, the overheads of conventional high-level languages would have been unacceptable, and in any case it was necessary to stay very close to the machine level. Nevertheless pure assembly code was unworkable – it took too long to write and debug a program, and changing the program was too hard. Forth was a compromise between the virtues of assembly code and the virtues of high-level languages. It has become a popular language for several types of work, including robotics, and has continued to develop – there has even been an object-oriented Forth. Recently the effectiveness of Forth as a compromise between high and low levels was underlined by the release of PostScript (Adobe Systems Inc., 1985), a device-independent page-description language now widely used to control laser printers. PostScript is comprehensible to humans and can be used as a programming language, yet it also serves as a communication medium between computers: word processors and graphics programs generate PostScript programs which are executed by the microprocessor in the laser writer or other output device.

The main idea behind Forth, continued in modified form in PostScript, is thoroughgoing submission to the reverse Polish notation (RPN). In RPN, operations are expressed as $AB + C *$ rather than as $(A + B) * C$. Certain types of hand calculator do much the same. The difficulty about RPN is how to express control structure, i.e. loops and conditionals, and how to cope with arrays. Forth boldly adopts an RPN solution to all these. For instance, the following code defines a 'word' (subroutine) that will print a warning message if its argument is greater than 220:

```
: ?TOO-HOT   ( temperature -- )
    1  >  IF   ." Danger -- Reduce Heat!  " THEN ;
```

Notice the terse syntax: : is the keyword that introduces the definition, ." prints the following symbols up to the closing quotation marks. Notice also the RPN conditional: the keyword, IF, follows the predicate, and the closing keyword THEN follows the action arm. The Pascal version would of course be:

```
IF temp > 220 THEN writeln('Danger ..').
```

But the most striking aspect of Forth is the free use of the execution stack. The programmer is at liberty to leave values on the stack for as long as desired. When ?TOO-HOT is executed it expects to find a value on the stack. In most high-level languages, a formal parameter would be declared and on entry to the subroutine an automatically compiled operation would remove the parameter from the stack and put it into the parameter variable. Forth leaves all that to the programmer: less complexity in the system, faster speed of execution, but more risk of error. 'Scruffy heaven' is its philosophy on all counts. Questions raised by Forth are: Is it readable? Can we parse for structure? Is it error-prone? Is it easy to learn because of having fewer new concepts?

Forth might look a bit old-fashioned today, but it must be understood that a whole heap of historical, organizational and economic factors determine choices of industrial languages. The same is true of languages for numerical control and for programming robots. Visser (1988) describes several types of language in use for computer control of automatic tools, one of which (MODICON) is based on relay closures, represented either as schematic relays or as Boolean expressions, with intermediate variables holding subexpressions. Serious and lengthy programs are developed in this language.

Now for a very different group of specialist languages, developed as sequence controllers. Sequence control languages typically contain no conditional structures but they provide repetition of patterns and of groups of patterns, and some means to define new patterns. Languages of this type have been developed for use in diverse areas. Advanced knitting machines aimed at the domestic market are programmed using pattern-description languages capable of creating a wide range of effects, yet intended for use by people with no background in informatics. In contrast, sequence controllers for musical effects (synthesizers, drum machines, tape editing for video, etc.) are likely to be used in a 'hi-tech' background. Many of these controllers are meant for professional use, but a few years ago an inexpensive drum machine for a domestic computer (the 'SpecDrum', distributed by Cheetah Marketing Ltd, for the Sinclair Spectrum) already provided remarkable capability. The user could store sixteen 'songs', each consisting of up to 255 sections, each of which could be repeated up to 255 times. In each section, one of a list of patterns was performed, a pattern consisting of some number of bars of however many beats was convenient. Six different types of drum sounds could be inserted into the bars, both on and off beat, and the tempo of the song could if required be over-ridden by an individual pattern.

Sequence control looks simple, but here are new questions for the list. First, what degree of abstraction is tolerable – could the SpecDrum user have coped with

Language	Example Query for 'Find the names of employees in department 50'
SQL:	SELECT NAME FROM EMP WHERE DEPTNO = 50

QBE:

EMP	NAME	DEPTNO	SALARY
	p. Brown	50	

Figure 5: Two designs for query languages (from Reisner, 1988). Although much simpler than programming languages, these languages share some of the characteristics. The advantages and disadvantages of different designs are still not well understood.

subpatterns, and subsubpatterns? Second, if the system is to be readily usable, must the degree of abstraction be predetermined (songs, sections, patterns), or can it be extended by the user at will? (Remember that if the degree of abstraction is to be extensible, the user needs not only a method to define and call patterns, but also a method to define new levels of subpatterns...). Achieving such complexity of structure without making the system impossible to use in a domestic setting will tax the designers!

The last examples of specialist languages that I shall mention are query languages for database searching. Query languages are somewhat outside the major scope of this book, yet ultimately when we understand how to design programming languages and other notations we should be able to deal with query languages on the same basis. Two well-known query languages are SQL and QBE: Figure 5 (from Reisner, 1988) illustrates the same query in each representation. Ideally, we should be able to predict the pros and cons for each design. Unlike most other areas, there have been controlled experiments on query languages, very well reviewed by Reisner (1988): yet she concludes (p. 267), 'It is clear that at this stage of our knowledge only a few of the issues have even been identified, much less studied. Clear guidelines to aid good design do not exist'.

4.2 Mixed paradigms

Many problems seem to be peculiarly intractable to any single programming paradigm. If approached procedurally, it becomes clear that part of the problem is best approached declaratively, and vice versa. Why not use a mixed paradigm, and treat each aspect of the problem on its merits?

An interesting early development was a scheme for transforming a typical procedural program to satisfy a number of 'sequence relations' (Middleton, 1980). Examples of the type of relation covered are: 'whenever the number of times that A has been executed is a multiple of N, do B'; 'stop as soon as the number of times C has been executed is M' (where the action C might occur in more than one location in the original program); 'between actions D and E, do F at least once'

(e.g. between opening a file and terminating, make sure it has been closed – whatever path is taken). So a complete program could have two parts, a standard Pascal-like part plus a declarative part specifying sequence constraints, and the compiler would automatically combine them. Middleton points out that meeting these and other requirements in conventional sequential terms causes a huge proliferation of administrative variables, keeping track of how often various events have been performed. A toy example shows a very simple seven-line program, comprising a loop and a conditional, being transformed to meet four sequence relations; the final program is thirty-four lines of vicious-looking code, and it seems undeniable that the mixed scheme is more comprehensible and easier to modify.

The Middleton scheme has not been tried as a working system, but another approach has: the combination of Pop-11 (a procedural language with Lisp-like data structures but a more expressive syntax) and Prolog, to form PopLog. In principle, a recalcitrant problem can be solved partly in Pop-11 and partly in Prolog, with data communicated between the two systems. Although this scheme has been available for some years now, as have similar schemes based on Lisp rather than Pop-11, this combination has made little impact outside its own band of aficionados. The reasons are not clear, for it would appear to yield precisely the information that the user wants in the most digestible form.

5 Where next?

We shall all be programmers soon. That, at least, is the impression one receives from the vast spread of programming possibilities, creating new environments such as the knowledge-based and visual systems illustrated above; new models of programming such as logic-based and constraint-based programming; and new applications, to science, learning, and education, to the office world, the domestic world, and the leisure world. Some idea of where programming is going can be got by looking at a few recent developments. This cannot possibly be representative; for instance, I am specifically excluding all educational programming systems, which will receive a brief treatment in a later chapter, and making no mention of program generators, fourth-generation languages, very high-level application-specific languages, or the prospects in languages for embedded systems such as domestic microprocessors. My aim in this closing section is to pick out what may well be seminal developments in programming styles.

5.1 Spreadsheets

The spreadsheet is now a familiar computational tool. It is so simple to use that in many people's eyes it hardly counts as programming, but merely as a declarative statement of relationships between numbers and formulae. The computation of area (used to illustrate Prograph above) can be trivially expressed in a spreadsheet, by putting the two values in two cells and putting the multiplication formula in a third cell. Other domains are now sometimes packaged with a spreadsheet-like interface, and this style may become even more widespread. For instance, Spenke and Beilken (1989) describe Perplex, which interestingly 'combines the power of logic programming with the popular user interface of spreadsheets'; the user creates predicates by successively refining first-solution attempts, with immediate knowledge of results, as

with conventional spreadsheets. 'There is no new formalism or language the user has to learn in order to define new predicates. Programming general solutions is almost as easy as solving a single, concrete problem. The user need not even know in advance that he is writing a program.'

5.2 Knowledge engineering

'Knowledge engineering' describes the explicit computer-based representation of human knowledge of how to perform skilled tasks, such as bidding in contract bridge, choosing suitable crops for farmers, or identifying the cause of an emergency in a nuclear reactor; most commonly such knowledge is represented as an 'expert system', which contains the knowledge used by experts and can be consulted for advice in place of an expert. Vast claims have been made for them, giving the impression that can readily be built by persons with no specialist programming experience, that real expertise can be captured, and that they can be used by clients with very little expertise as a genuine alternative to consulting a live expert. In actual fact, they are at present useful but quite limited. They will probably become more common and more useful in the next decade, but despite the early claims it is unlikely that systems of any complexity will be built by non-programmers or easily used by non-specialists.

The area has been dominated by representational techniques using 'production rule' systems, programming languages built around the IF-THEN construct. A commercially available example is 'Xi Plus' which uses a relatively comprehensible, English-like syntax, in the form:

```
if temperature < 55
then room is cool
```

A set of such rules, plus a set of facts established for the current case (e.g. temperature = 51), forms a 'knowledge base'. Figure 6 gives as an example a fragment of a knowledge base for choosing house plants for a given room. The rules can be used in two ways – either by supplying some facts about the room in question, and then asking Xi to report what conclusions follow ('forward chaining') or else by asking specifically to find out whether a given conclusion is true ('backward chaining'). In forward chaining the expert system examines all its rules: any rule whose IF part is satisfied by the existing facts can be executed, and the statements following the THEN part are added to the list of current facts. In backward chaining, the expert system first examines rules whose THEN part can establish the desired goal: if these rules have an IF part which only uses currently known facts, well and good; otherwise, the system tries to find a rule whose THEN part can satisfy that IF, and so on. These two schemes produce very different behaviour, and Xi Plus is unusual in allowing not only both modes, but also a smooth transition from one to the other depending on circumstances.

Expert systems, even more than other forms of programming, aim to put expert knowledge into a tractable and comprehensible form. One favoured technique is to allow the client who uses them to ask 'why?' questions: 'Why is ivy the best plant?' receives the reply 'Because the light is poor and the room is cool'. Essentially, this is a form of directed browsing over the execution history of the program. Whether this

```
if temperature < 55
then room is cool

if temperature >= 55
   and temperature < 65
then room is warm

if temperature >= 65
then room is hot

if light is poor
   and room is cool
then best plant is ivy

.........................
.........................

if light is sunny
   and room is hot
then best plant is collection of cacti

question light is sunny, bright, poor
text How good is the light in your room?

question temperature = 45 to 75
text What is the temperature of your room in
degrees Fahrenheit?

query best plant
```

Figure 6: Fragment of an expert system built in Xi plus. The IF-THEN rules form a knowledge base. In response to the instruction 'query best plant', the system looks for rules that can establish a best plant. On finding rule 4, which applies in poor light, it looks for facts or rules about light; the instruction 'question light is ...' tells it to ask the user 'How good is the light in your room?' and to use the reply as a newly established fact. The reply 'poor' will allow it to go on to '...and room is cool'; other replies will cause it to discard the ivy rule and look for another possibility (not shown here).

```
IF
    When:
       Every day at:  5:00 AM
    Header contains:
    Msg body contains:  <string>
    Folder name is :  In-Tray
    Msg length is:
    Msg is:  Read AND NOT Tagged
    Date sent is:
       More than 2 days ago
THEN
    Move-to Old-Mail
```

```
WHEN daily(5:00 AM)
   FOREACH msg IN "In-Tray"
   IF read(msg) AND
      NOT (tagged(msg) AND
         date-sent(msg) > 2 days ago
THEN
   move-to-folder(msg,"Old-Mail")
```

Figure 7: Example of forms-based version (left) and language-based version (right) of the same program (from Jeffries and Rosenberg, 1987). The task is at 5 a.m. to take all the old messages (that is, ones that have been read and have not been tagged for keeping) out of the 'in-tray' and put them into an 'old-mail' folder.

is the best way to help the client understand how the decision was reached remains an open question.

Also at issue is whether such systems will really become favoured vehicles for non-programming specialists to communicate expertise to others. One of the few studies of difficulty in this type of system (Ward and Sleeman, 1988) reports a wide variety of minor difficulties with the notational details, suggesting that the style of language has to evolve further. More importantly, the programmers were observed to meet problems requiring 'deep and careful thought', especially to do with such issues as 'How should rule premises and rule conclusions be organized, and how should rules inter-relate?'

5.3 Forms programming

Another application of the IF-THEN format is in 'programming by form filling', which is intended to provide a simple method for non-programmers to build programs in limited domains. The limited possibilities of the domain can be exploited to replace conventional procedural language style with multichoice techniques. The example (Figure 7) concerns a programmable handler for electronic mail.

Form-filling methods give users fewer ways to go wrong than conventional languages, so it is not surprising that non-programmers do better; more surprising, however, is the fact that experienced programmers also scored slightly better with the form-filling methods. The authors point out that these systems are 'close to the prototypical tasks the user wants to perform', so that the translation distance between the goal and the required actions is considerably reduced. This is an interesting explanation, although there has been no report of a test such as presenting the

two versions of Figure 7 to users and asking which one is 'closer to the task'. The sceptic might suggest that one version does a better job of reminding users about what conditions to include and how the syntax works! More use should, perhaps, be made of techniques which have such a property.

5.4 Everyday programming

The most challenging task is to get everything right at once: a programming language that is easy for beginners, has enough power for experts, comes with an environment which meets the user's needs, and is attractive to use. While we are a long way from achieving that, some interesting possibilities are emerging, among them HyperCard for the Macintosh. HyperCard reverses the normal order of things, in which the programmer writes a program which might, rather painfully, put some graphics on the screen. Instead, HyperCard provides ample tools for users to build graphics as in a normal painting-style application, and then allows fragments of program to be attached to the resulting picture. The fragments are usually attached to 'buttons' which, when pressed, cause them to be executed. One single screen is called a 'card', and a HyperCard program in fact consists of any number of cards, each of which can show a different picture. A simple example is shown in Figure 8; the user is expected to use the mouse and keyboard to write numbers into the two fields, and then to 'press' the button to cause the computation to be performed.

A limited form of inheritance caters for the frequent need to produce a set of cards with the same basic design, such as all having a button in the same place. All cards have a 'background', which can be shared by any number of cards; and so a button which is placed on the background will be present on each card. Both scripts and graphics can be inherited from the background, and further levels of inheritance are also provided for scripts.

Apple, the creators of HyperCard, clearly intended it to become as near an everyday programming language as they could manage, and it is instructive to consider some of the steps they took. First, they tried to give it wide *everyday applicability*, not so much by building in computing power as by providing means for HyperCard to control domestic gadgetry. With the right attachments, it can control compact disc players and video recorders, synthesize sounds, run a MIDI interface to audio equipment, and even dial telephone numbers; in the office it can provide calendars, reminders and organizers. Next, the system is meant to be *foolproof*. There are no variations depending on the particular hardware, the system has been very thoroughly debugged, it is hard (but not quite impossible!) to create catastrophic errors which delete important material, and the error reporting is almost jargon free. They also emphasized *simplicity*. Difficult concepts have been avoided. (Most data structuring methods have been classified, rightly or wrongly, as 'difficult', so there is no form of linked list and only a rudimentary form of array.) The programming language has been kept free of special symbols and funny-looking words, and although the result is rather verbose at least the user can usually remember what the syntax rules are. The use of inheritance techniques is intended to *reduce drudgery*, although this has not been entirely successful – more powerful tools are needed. Finally, and perhaps most innovative, is the principle that *program fragments are attached to screen objects*, not the other way about.

HyperCard has the faults of its virtues. The language has little expressive power, and it is verbose and tiresome to proficient programmers. By attaching program frag-

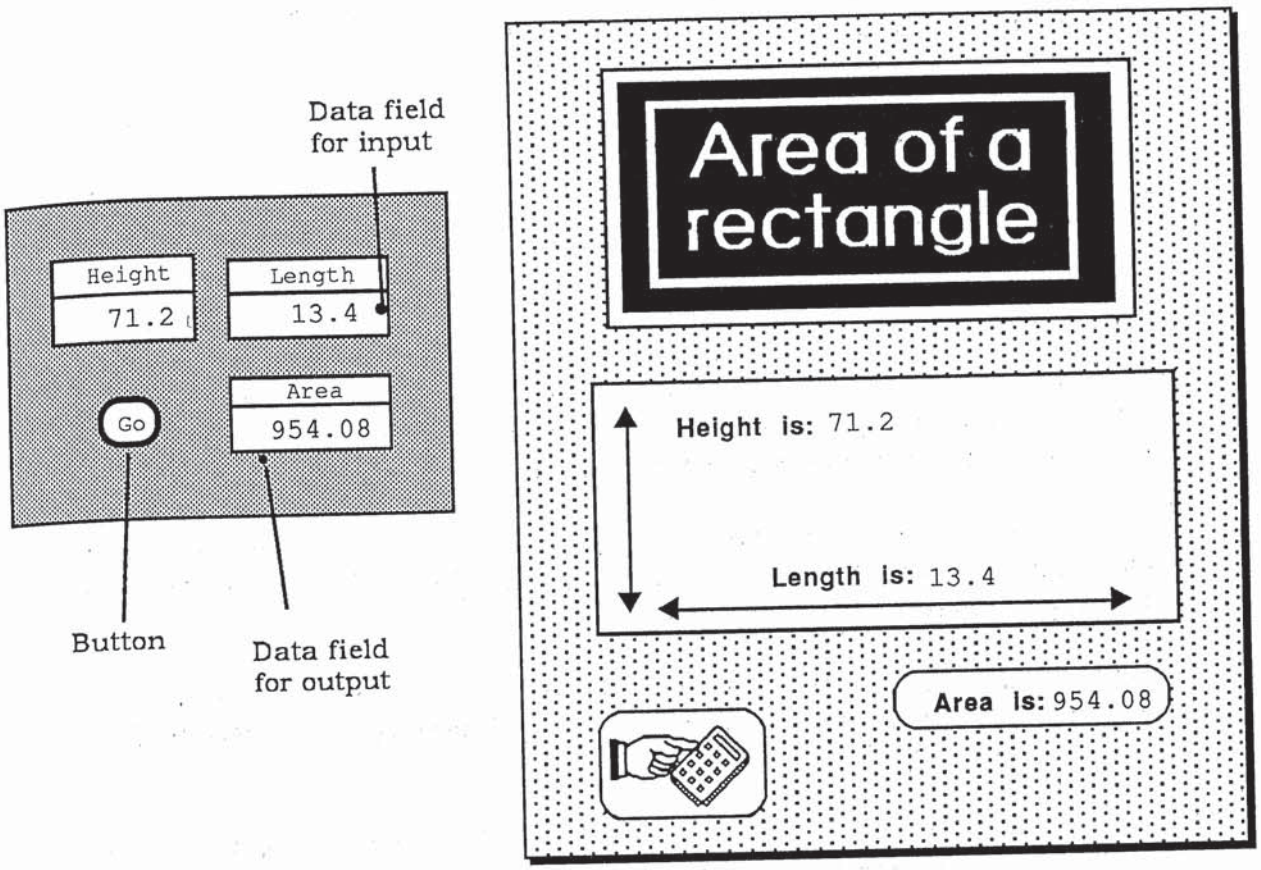

Figure 8: HyperCard programs are attached to graphic objects, typically buttons, on 'cards'. To the user they look like active diagrams. Here are two different possible visual arrangements for the same simple example, in which users enter data in 'fields' and press a 'button' to start the computation. In this example, the 'script' (program) for the button 'Go' might be:

```
on mouseUp
    put card field "length" * card field "height" into card field
"answer"
end mouseUp
```

Scripts are executed when a given event occurs. This script will be executed when the mouse button is released ('mouseUp') within the screen area of the button. Neither the script of the button nor the names of the fields are visible in the normal mode: the programmer has to 'look behind' them. More complex programs may relate several different cards: e.g. successive cards may handle area computations for a variety of different shapes.

ments to screen objects the overall visibility of the program has been very seriously reduced (you can't easily see the scripts for two buttons at the same time) and programmers have to commit themselves too early to decisions they may later revise. It will be interesting to see what the next attempt looks like.

6 Conclusions

The development of computing has been based largely on guesses about what people would find easy to use. After this quick tour, what do we know – or better, what do we now realise that we don't know, but would like to?

(1) We still think too readily of programs as just being for compilation. We should think of them also as being for communication from ourselves to others, and as vehicles for expressing our own thoughts to ourselves. So we should think more about reading versus writing, capture of ideas versus display of ideas, etc.

(2) Existing environments are better at displaying program information than at providing means to manipulate it effectively. We need more research on manipulation of complex structures.

(3) Not enough is known about the advantages of the techniques currently being explored, such as program animation, mixed paradigms, etc. Surely these deserve active investigation.

(4) There has not been enough conscious effort to find ways to display alternative representations of information or structure; instead, many systems simply translate textual information into an equivalent form.

(5) Where new types of information structure have been devised (e.g. representations based on programmer's knowledge, such as KBEmacs), there is often not enough existing psychological research to be useful.

(6) The relationship between the structures of programming languages and the tools for operating on programs has not been well explored. Research on each tends to occur without considering the other. Should we not try to discover, at a generic level, what support programmers need (or at least, programmers of a particular experience level), and then when designing tools and languages try to ensure that nothing is omitted?

(7) Is there a 'solution' to such profound differences as the argument between the 'neats' and the 'scruffies'? Probably not. Perhaps we should aim for an agreement to co-exist, with different approaches suiting different personalities of different contexts. This demands more research into individual differences – not merely in aptitude.

In short, we know something about the psychology of learning to program, of understanding programs, etc., but only in a very restricted range of languages and environments. It is important to extend our studies to wider vistas.

Acknowledgements

I should like to thank Heather Stark, Barbara Kitchenham and Marian Petre for their helpful comments on an earlier version of this chapter.

References

Adobe Systems Inc. (1985). *PostScript Language Reference Manual.* Reading, MA: Addison-Wesley.

Böcker, H. D., Fischer, G. and Nieper, H. (1986). The enhancement of understanding through visual representations. *Proceedings CHI'86 Conference on Computer-Human Interaction.* New York: ACM.

Brayshaw, M. and Eisenstadt, M. (1988). Adding data and procedure abstraction to the Transparent Prolog Machine (TPM). *In* R. A. Kowalski and K. A. Bowen (Eds), *Logic Programming: Proceedings of the 5th International Conference and Symposium.* MIT Press.

Brodie, L. and Forth, Inc. (1987). *Starting Forth.* Englewood Cliffs: Prentice-Hall.

Détienne, F. (1990). Difficulties in designing with an object-oriented programming language. To be presented at *INTERACT '90 Conference on Computer-Human Factors.* Cambridge, England.

Furnas, G.W. (1986). Generalized fisheye views. *Proceedings CHI'86 Conference on Computer-Human Interaction.* New York: ACM.

Green, T.R.G. (1980). Programming as a cognitive activity. *In* H.T. Smith and T.R.G. Green (Eds), *Human Interaction with Computers.* London: Academic Press.

Green, T.R.G. (1982). Pictures of programs and other processes, or how to do things with lines. *Behaviour and Information Technology,* **1**, 3-36.

Jeffries, R. and Rosenberg, J. (1987). Comparing a form-based user interface for constructing a mail program. *In* J.M. Carroll and P.P. Tanner (Eds) 'CHI+GI 1987', *Proceedings ACM Conference on Human Factors in Computing Systems and Graphics Interface.* New York: ACM

Kernighan, B.W. and Plauger, P.J. (1974). *The Elements of Programming Style.* New-York: McGraw-Hill.

Köhne, A. and Weber, G. (1987). Struedi: a Lisp-structure editor for novice programmers. *In* H.-J. Bullinger and B. Shackel (Eds), *Human-Computer Interaction - INTERACT '87.* New York: Elsevier.

Middleton, A.G. (1980). A program transformation system for implementing sequence relationships. Technical Report 8001, Dept. of Computer Science, Memorial University of Newfoundland, St John's, Newfoundland, Canada.

Neal, L. R. (1987). User modelling for syntax-directed editors. *In* H.-J. Bullinger and B. Shackel (Eds), *Human-Computer Interaction - INTERACT '87.* New York: Elsevier.

Reisner, P. (1988). Query languages. *In* M. Helander (Ed.), *Handbook of Human-Computer Interaction.* New York: Elsevier.

Seabrook, R.H.C. and Shneiderman, B. (1989). The user interface in a hypertext, multi-window program browser. *Interacting with Computers,* **1**, 299-337.

Sime, M. E., Arblaster, A. T. and Green, T. R. G. (1977). Reducing errors in computer conditionals by prescribing a writing procedure. *International Journal of Man-Machine Studies,* **9**, 119-126.

Spenke, M. and Beilken, C. (1989). A spreadsheet interface for logic programming. *Proceedings CHI'86 Conference on Computer-Human Interaction.* New York: ACM.

Visser, W. (1988). *Langages de prommation dédiés: quelques exemples dans le domaine des automates programmables industriels.* Technical Report, INRIA, Rocquencourt, France.

Ward, R.D. and Sleeman, D. (1988). Learning to use the S.1 knowledge engineering tool. *Knowledge Engineering Review,* **2**, 4

Waters, R. C. (1985). The programmer's apprentice: a session with KBEmacs. *IEEE Transactions on Software Engineering,* **11**, 1296-1320.