Chapter 2.2

Programming Languages as Information Structures

T.R.G. Green

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

Abstract

This chapter describes three 'implicit theories' of programming, which have at different times governed the styles of programming language design. Not surprisingly, the experiments performed by empirical researchers have also been deeply influenced by the prevailing theory of the day. (1) The first view regarded programming as transcription from an internally-held representation of the program. This leads to Fortran-like languages, and the early empirical studies of programming correspondingly emphasised particular language features and their incidence of errors. (2) The second view stressed program comprehension and required programs to be demonstrably correct, leading to Pascal-like languages and the pseudo-psychological theory of 'structured programming'. Empirical research during this era showed that giving programmers easy access to information they needed (whether structured or not) was what really mattered. Visual programming began during this era, but most empirical studies failed to demonstrate any inherent advantage to visual presentation. Where advantages were found, they appeared to fit the same fundamental principle. (3) Today's view is that program design is exploratory, and that designs are created opportunistically and incrementally. Supporting opportunistic design means putting a minimum of unnecessary demands on working memory, permitting designers and

Psychology of Programming
ISBN 0-12-350772-3

programmers to postpone decisions until they are ready for them, allowing easy additions or changes to existing code, etc. Empirical research on change processes and reuse of code has demonstrated differences between language designs, but has a long way to go. The chapter ends with suggestions for 'lowering the cognitive barriers to programming' by more careful design of languages.

1 Introduction

Even a decade ago, a survey distinguished about 150 documentation techniques (Jones, 1979). There are probably far more now – and far more programming languages. They vary a great deal in every possible respect. Even for experts, certain details of language design – what I shall call the 'information structure' – are likely to affect the speed and accuracy of using the language or the documentation technique. Any readers who belittle the relevance of notational structure, and prefer to believe that 'anyone can get used to anything', are recommended to earn their living doing arithmetic with roman numbers for a while. So, whereas other chapters (e.g. Chapters 1.3, 2.4 and 3.2) describe what we know about the *mental processes* of programming, this chapter describes research on how *external factors* influence the processes.

1.1 Information structure

We can consider the user of a programming language as someone who has to find out information from a program, add new information to it, and occasionally reorganize it; and the details of the programming language will influence how easy it is to do such tasks. The framework we shall adopt is that a program is a kind of information structure, just as a library is a kind of information structure. Libraries can be organized in many ways, and different choices will facilitate different tasks. Most libraries determine the shelf positions of books exclusively by subject indexing, but a few use a different principle: in Cambridge University Library, for instance, shelf positions are determined partly by subject matter but also by the physical size of the book (so books of different size categories are stored separately). Both these systems can be used to shelve the same books – in other words, they have equivalent power; what is different is the way the information about the books is structured. In consequence, different sets of user tasks are supported by the systems. Browsing by subject in Cambridge University Library is notoriously unrewarding – but *if* you happened to want unrelated books of the same size, it might be just what you wanted.

One of the themes of the chapter is that an information structure can make some information more accessible, but usually at the cost of making other information less accessible. Programs have in common with libraries and other information structures that *the structure of the information should match the structure of the task*.

Given this outlook, I shall not make much distinction between a true programming language, on the one hand, and a documentation format, on the other. The principles governing access to information, ease of change, etc., will presumably be the same.

In subsequent sections we shall see how the view of the programmer's task has altered over the years. The design of programming languages has altered in sympathy with different views of the task, and the style of programming research has likewise altered.

1.2 The state of evidence

The aim of this chapter is to bring together the existing empirical evidence, rather than to present responses from practitioners, however revealing. Much of the available evidence was gathered in pursuit of some other objective than comparisons of notational structure, especially in studies of the problems of novice programmers. It is still relevant but needs caution: learners and experts are likely to have different notational needs. One would expect that while the problems of learners are often to do with recognizing and recalling components, with fitting them together and with perceiving relationships, the problems of experienced users would be 'high-performance' problems – finding information quickly, and modifying programs without unnecessary effort. (Comparisons of student and professional performance by Holt *et al.*, 1987, support this view.)

Rather than painfully distinguish all through the chapter between the problems likely to affect learners and the problems likely to affect experts, I have simply tried to clarify concepts and sources of difficulty. Anyone evaluating a programming language for actual use will have to consider many criteria; among them should be sources of difficulty *for the intended users*.

2 The programmer's task

2.1 Types of task

Consider, therefore, what tasks are required of programmers. Present-day views have been heavily influenced by the work of Sumiga and Siddiqi, Visser, and Guindon (see Chapter 3.3) showing clearly that the design of large programs by professionals corresponds to 'opportunistic planning', an activity in which a design is constructed piecemeal with frequent redesign episodes. Much use is made of an external record, whether on paper or a VDU, and new portions are inserted as they come to mind. As far as we can tell at present, this picture applies in any activity which has a large design component, regardless of the notation or of the stage in the development process.

Now, this is *not* the 'approved' style of development. Software designers like to speak of the 'waterfall' model, in which higher-level requirements are dealt with before starting on lower-level processes; or of other similar methodologies. Perhaps such disciplined approaches would be better, but they seem to be infeasible as a general technique, because the consequences of higher-level decisions cannot always be worked out fully until lower level ones are developed. Be that as it may, this chapter will concentrate on what it seems that people really do, not on what they might do if they were perfect.

Typical activities therefore include: comprehending or 'parsing' the partially developed design, to remind oneself what has been written so far and how it works; making modifications, of any scale, large or small; inserting new components into what exists already; looking ahead to foresee consequences of design decisions (this is often done in an ancillary formalism of some sort – see, for example, self-observational studies by Naur, 1983); and recording degree of commitment to particular design choices.

Although typical experimental studies do not, unfortunately, record such a wide variety of activities, the importance of discriminating between them has been demonstrated, at least for the comprehension case. Comprehension has been shown to be a complex task (see Chapter 1.3), of which one facet is conveniently labelled 'deprogramming', meaning that after a portion of the mental representation of the problem has been translated into code (or specifications language, or some other notation), it is then translated back again into mental representation language, as a check. In many notations there is an apparent asymmetry of effort: it seems to be easier to develop the code than to recover its meaning. (Spreadsheets, for example, are created quickly and easily, but discovering precisely what a spreadsheet does can be difficult.) In other notations, the asymmetry is much less, and 'deprogramming' is relatively straightforward.

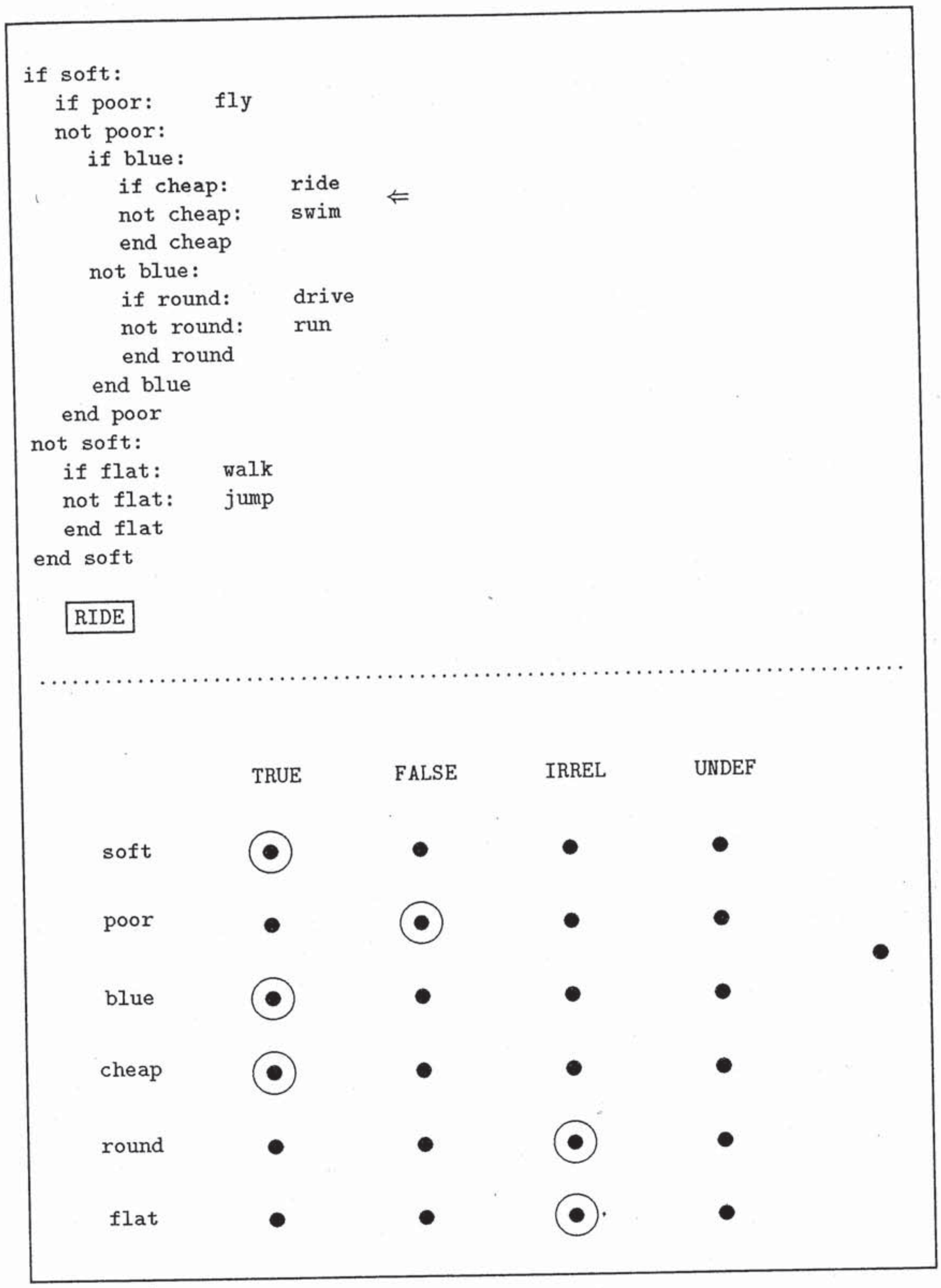
To appreciate how deprogramming was identified as a problem we need to look ahead a bit and mention some studies; we shall return to them in their proper place. The asymmetry of effort between writing and comprehending was first postulated by Sime *et al.* (1977), after they had shown that novices debugged their own programs much faster in one miniature language than in two others. The languages were based on conditional structures, and it was claimed that the deprogramming problem lay in working 'backwards' through the program, to discover what set of circumstances ('taxon') caused a particular outcome. Later, Green (1977) strengthened that conjecture, by showing that professional programmers could trace one type of conditional program backwards much faster than another, even when the forwards tracing times were not significantly different. The clincher was given by Curtis *et al.* (1988), who showed a statistical separation between different types of question in the same experiment, and thereby demonstrated that the concept had discriminant validity. Their study also showed that understanding data flow was a third type of task, again statistically separable from the previous two.

What all this shows, therefore, is that detailed aspects of information structure can have large effects on particular parts of the process of creating designs and programs. Other parts of the process may be almost entirely unaffected. To discover the virtues or difficulties inherent in particular notational structures it is therefore necessary to consider many different tasks.

2.2 How information structure affects behaviour

Differences in information structure are sure to affect the programmer, but not necessarily directly. Green (1989) has used the idea of opportunistic planning to develop an analysis of 'cognitive dimensions' of notations, with suggestions as to how variations along each dimension affect typical programming tasks. (More accurately, what affects the usability is the combination of a notation and the environment in which it is used: a notation suitable for 'pen and paper' might be less suitable for a top-down structure-based editor.) We cannot investigate these dimensions in any detail here, but we can give a few examples.

The first is *viscosity*, or the degree of resistance to local changes. The degree of viscosity clearly depends on what change is being made, for we can see that it is often very easy to make a minor change to a relatively loose structure, like Basic, but a



```
if soft:
  if poor:     fly
  not poor:
    if blue:
      if cheap:    ride      ⇐
      not cheap:   swim
      end cheap
    not blue:
      if round:    drive
      not round:   run
      end round
    end blue
  end poor
not soft:
  if flat:     walk
  not flat:    jump
  end flat
end soft

  RIDE
```

	TRUE	FALSE	IRREL	UNDEF	
soft	[✓ ●]	●	●	●	
poor	●	[✓ ●]	●	●	●
blue	[✓ ●]	●	●	●	
cheap	[✓ ●]	●	●	●	
round	●	●	[✓ ●]	●	
flat	●	●	[✓ ●]	●	

Figure 1: An experimental task from Green (1977), simulating the postulated mental activity of 'deprogramming'. Given the program (*upper panel*), the subject's task was to report the conditions under which the designated action 'ride' will be the first action performed. Responses were made using a stylus, touching studs on a response display (*lower panel*). Here, the full response has been set up; touching the far right-hand spot will complete it. Subjects who were professional programmers found this task much easier in this type of miniature language than in the types based either on GOTOs or on conventional nested structures.

large change tends to create 'knock-on' effects, further changes that are consequence of the original one. The object-oriented programming systems are particularly ef fective in reducing knock-on effects, and to that degree are an improvement on th dimension of viscosity. A second example is *premature commitment*, where the prc grammer is forced to make a decision before the consequences can be foreseen. Thi can happen where the notation and the editor, or some other construction systen are incompatible, or where the system has been designed around a different vie of programmers' tasks. Hoc (1981, 1988) has demonstrated the problems that aris with editing systems that enforce commitment to outline program designs befor the programmer is fully ready. It is noteworthy that many designs for structure based editors, despite being intended to help programmers, seem likely to deman premature commitments. The third example is *role expressiveness*, the ease of 'de programming', discovering what the parts of an existing program are and what i the role or purpose of each part. An excellent study by Pennington (1987), describe by Gilmore (in Chapter 1.5), has demonstrated notational differences in role expres siveness, by showing that role-based abstractions are more easily constructed abou Fortran programs than about Cobol ones.

The suggestion is that programmers will choose their style of working accordin to the particular combination of information structure and editing tools. If th system is viscous, they will attempt to avoid local changes and will therefore avoi exploratory programming. If the system has poor role expressiveness, at least th less experienced programmers will avoid exploratory programming, since they wi have trouble in recognizing program components. If the system enforces prematur commitment, the effect will depend on the viscosity; with low viscosity, prematur commitment is only a small risk; as viscosity rises, postponing commitment become more urgent.

2.3 Implicit theories of low-level programming

Clearly, if we take opportunistic planning seriously, the demands on the notation ar subtle and various. Both language designers and language researchers have taken long time to reach today's views, and no doubt further complexities are still to come But to understand the research tradition it is necessary to realize that the generall accepted view of programming has changed over the years, and with it the focus c empirical research has also shifted. Language designs attempt to meet the designer' view of programming, and so the design presents us with an 'implicit theory' of wha tasks the programmer must accomplish.

Three main stages can be distinguished in the implicit theories apparent i programming languages. First, the implicit theory was that programming was a errorless transcription. So long as a program performed the correct computation its comprehensibility was immaterial; no thought at all seems to have been given t problems of the modifying programs. In the second stage, language designers gav more attention to programs as constructions that had to be comprehended by others or possibly by programmers themselves at some later date. Languages conformin to this view still paid little attention to the modification of programs; they wer presented as solutions to well-defined problems, and much was said about the desir ability of knowing that the solution was correct before starting to code the program In the third stage, where we are at present, many designers accept the 'evolution ary' style of programming, in which the activity of program design is one of repeate modification – frequently starting from a seed which was an already-existing program that solved a related problem.

Since the prevailing view has affected the type of research question that empiricists asked, we shall review the research under appropriate headings.

3 The errorless transcription view of programming

3.1 The implicit theory

We will take the Fortran-Basic tradition as a model of how programming languages were originally designed. The implicit theory could be described as the 'one-way, error-free' view. Fortran I was a very great success in its day, showing that the implicit theory was clearly at least partially correct – in its day.

The one-way view of programming is very simple. It sees programming as *errorless transcription from a previously worked-out representation* (possibly held in the head, possibly on paper). The mental representation of a program is apparently viewed as a sequence of steps – step 1, step 2, step 3, ... – and each step is individually translated into its coded representation in the target programming language. Nothing more is involved. Consider some of the design features of this tradition:

* The Fortran programming system (punched cards) and the Basic line-numbering system encouraged programmers to create their programs in the order of the text – i.e. line 1 of the final text was also the first line to be punched in. Thus, *the program was fully developed at the start of coding, needing only to be transcribed.*

* Fortran and Basic have very few guards against typing errors, which can readily create a new text that is syntactically acceptable but not, of course, the intended program. By implication, *programmers do not make typing errors.*

* Neither Fortran nor Basic originally supported any use of perceptual cues to help indicate structure. Possible cues would have included indented FOR-loops, demarcated subroutines, bold face or capitals to indicate particular lexical classes, etc. The implication is that *programmers can comprehend the program text without assistance.*

* The use of GOTOs as the sole method to determine control flow encourages small changes but makes large changes extremely tedious. The implication here is *programmers do not need to modify their first version, except trivially.*

Fine, you think. All that used to be true, but we have moved on. But have we? Consider the structure of spreadsheets (a form of programming language), of Prolog, and of the production system languages used in expert systems. In many important respects these three systems continue the tradition of the 'one-way' implicit theory. None of them guard against typing errors, none of them supply perceptual cues, none of them limit the complexity of program structures that have later to be understood. Prolog and production systems allow the text to be generated in the order the programmer prefers, rather than starting at line 1, but spreadsheets encourage starting at the top left-hand corner, whether convenient or not. One advance that has been made is that subsequent modifications are sometimes easier. Not always: some types of modification are really long winded, in each of these three languages. Many other systems could be pointed to. In short, the implicit theory of programming as transcription is still alive.

3.2 Research on language features

Empirical research arising from this view of programming is likely to focus on the individual features or syntax constructions of programming languages. From the 1970s onwards there have been several such studies. Youngs (1974) reported frequencies of errors for a number of different statement types (comments, assignments, iterations, GOTOs, conditionals, etc.) in several languages. These results were achieved simply by looking at programs that had been written and looking for where the bugs occurred. Although his study was more comprehensive than most similar studies, it is difficult to see what has been learnt from a collection of error frequencies that can usefully be generalized to the problem of language design. Slightly more can be learnt by comparing different designs for the same language feature, such as logical versus arithmetic IF statements in Fortran (Shneiderman, 1976) or nesting versus GOTO styles of conditional (Sime *et al.*, 1973, 1977; Mayer, 1976).

Many of the early studies asked simplistic questions: 'Are logical conditionals [always] better than arithmetic ones?', 'Are flowcharts [always] better than code?', 'Are nested conditionals better than GOTOs?', etc. It is easy to see today that in general the answer is going to be 'X is better than Y for some things, and worse for others'. But that is because we can clearly see now that programming is a complex set of skills, not a unitary process.

In so far as a unitary process can be expected, it might be 'readiness to build a program rather than use repeated operations'. In a thorough set of investigations, Wandke and his colleagues have investigated changes in programming readiness during the acquisition of expertise in very simple situations. In much of their work, subjects practised searching a database for various targets, using a miniature search language. Typically, the language contained five commands plus one command to define a macro-instruction. A carefully designed study (Wandke, 1988) showed that subjects were reluctant to define macros for conditional constructions even when very large numbers of keystrokes in subsequent search operations would have been saved. The inclusion of control structure dramatically increased cognitive effort; simply counting keystrokes gives no indication of the real effort involved.

In a further study from the same group, Wetzenstein-Ollenschlaeger and Schult (1988) also put subjects in a situation where it would save effort to build macros, and after each of four sessions tested subjects' knowledge of semantics and syntax and of the interrelationships between the commands in the miniature language. The task structure had a core command sequence, which had to be repeatedly typed out in full unless a procedure was defined for it. Numbers of subjects using procedures rose with experience. Subjects using procedures scored slightly higher on knowledge of syntax and semantics, though not much. But they scored a great deal higher on knowledge of interrelationships between commands (about 70% against 11 to 50%). The interesting fact is that the bulk of this difference came not from the core task, but from the other parts of the task – the commands that were *not* proceduralized. Readiness to use procedures, in short, depended on understanding the whole of the language, not just the immediately relevant parts.

4 The 'demonstrable correctness' view of programming claims

4.1 The implicit theory

With the rise of 'structured programming' in the 1970s a new implicit theory emerged: programs had to be clearly seen to be correct. Comprehensibility was equated with formal structural simplicity, which favoured hierarchical composition, and harsh words were used of earlier programming languages ('Basic causes brain damage', 'Fortran programmers can never learn sound programming principles'). Programs were to be built from a small number of structures which could be related to each other in simple ways. This was the hey-day of the 'neats' (see Chapter 1.2).

Pascal, a powerful influence, rejected GOTOs in favour of a small repertoire of nestable loop and conditional structures, and also contributed a technique for defining hierarchically composed data structures. After Pascal, further developments down the 'neat' line rejected variables and iterative constructions in favour of pure compositions of functions. The argument in all cases was that the correctness of programs was easier to perceive.

4.2 Research on program structures and the structured programming claims

The second phase of empirical research on programming can be seen as confronting the structured programming dogma with empirical data which showed that the story was more complex. One line of research concentrated on the recommended style of program construction. The structured-programming methodology of 'stepwise refinement' instructed the programmer to decompose difficult problems into easier problems, and then into still-easier problems. For this reason, hierarchically structured notations were preferred, and much opprobrium was poured on users of notations that allowed 'unstructured' control flow. Doubts were soon raised as to whether research on problem solving really supported the enforced use of hierarchical structures – among other questioners, Green (1980) contrasted the computer scientists' assertions with evidence that showed that problem solvers did *not* customarily use such simple methods as stepwise refinement.

These doubts were matched by empirical evidence. Sime *et al.* (1977) compared novice programming in three types of notation, one of which was unstructured, one of which was structured in the usual way, and one of which was structured in the same way but also contained additional information. The additional information was logically redundant, since it could be deduced from what was already present. They showed that both the nested notations were improvements over the unstructured notation, as predicted by the structured programming school, but that the additional information greatly helped novices to find solutions to short programming problems using conditional structures. In particular, as mentioned above, it greatly aided 'deprogramming'. Thus the structured programming approach had grasped only some of the truth. Another study (Arblaster *et al.*, 1979) explored the difference between structured and unstructured notations. Was it vital to have a hierarchical structure, as claimed by structured programmers, or would other types of structure also be effective? Their results showed that of three types of structuring (hierarchical, decision-table-like, and compromise), all were better than a condition with no structure but that the hierarchical structure was not markedly better than the other two.

Once again it appeared that the structured programming school had only grasped some of the truth.

An important qualification was made by Vessey and Weber (1984), showing that it was possible to differentiate between effects due to problem solving and effects due to coding. They held the problem-solving component constant by presenting problems in a neutral language, and extended Sime and co-workers' three languages to include unindented and indented forms. (Sime *et al.* had studied indented nested languages versus unindented GOTO languages.) Apparently novices' performance was determined much more by indentation than Sime *et al.* had supposed, and the relative advantages of nested conditionals were less clear cut.

Laboratory-based research using 'micro-languages' may not readily generalize, of course, but at least it was shown that these results generalized to professional programmers: Green (1977) showed that the hierarchical structure with added information allowed professional programmers a small but highly significant speed advantage in answering certain types of questions about programming. This result shows that the 'coding' explanation advanced by Vessey and Weber is not entirely sufficient.

A more important aspect of Green's study was that the task of 'program comprehension', which had previously been treated rather casually, was given a deeper analysis. Many previous studies had asked subjects to execute the program mentally and to report on what output was achieved for given input. Green added the inverse, or 'deprogramming', task of asking what input was required to achieve given output (see Figure 1). It was this second task that differentiated between the different notational structures. The results obtained earlier by Sime *et al.* (1977), which had shown that novice programmers debugged their programs faster in the language with additional information, were therefore ascribed to differences in the ease of deprogramming the different notations.

4.3 Generalizing to other computational models

The explanations advanced by most researchers studying notational design have been phrased in terms of information-processing demands. In principle, therefore, they could be applied to any programming paradigm, not only to procedural programming languages. Gilmore and Green (1984) set out to demonstrate a parallel of Green's procedural-language results, this time using a declarative programming system. They based their study on what looked like a very obvious claim: if it was easier to answer procedural-type questions than declarative ones, given a procedural program, then it should be easier to answer declarative-type questions than procedural ones, given a declarative program.

Looking at their study in more detail, Gilmore and Green argued that Green's 'deprogramming' task was equivalent to asking subjects to convert from a procedural representation into a declarative representation. The additional information given in one of Green's notations assisted the subjects because it contained cues to the declarative form. The other two notations contained no such cues and hence answering the declarative-type questions was harder. *Mutatis mutandis*, similar results should be obtained by starting with a declarative notation: procedural questions would be harder than declarative ones, since they would require information in a structure not provided by the notation. The extra difficulty would be lessened if additional information were given, containing cues to the procedural form. So they compared the difficulty of answering both declarative questions and procedural questions, working from declarative programs with and without cues to procedural information.

The results were, however, less clear cut than was expected: the anticipated effects were present but weak, except when the subjects were working from memory, in which case quite effective results were obtained. Why should this be? We do not, at present, know: but at least two factors could contribute. The first is that the 'natural' mental representation for the type of problem used may often be procedural rather than declarative. This has been suggested by work such as that of Hoc (see Chapter 2.3) which indicate that novices frequently conceive programs as a series of steps rather than as what-if structures. The second factor may be simply that the cues used in Green's study were more perceptual in nature, while the Gilmore and Green study used cues that were more symbolic in nature. The importance of perceptual cues to program structure will be taken up below.

4.4 Program visualization: diagrammatic notations

Many attempts have been made to replace text-based programming notations by diagrammatic notations, in the hope of improving comprehension. Reviewing them will take a rather long section because a subsidiary argument must be included: whether one must use genuine diagrams, proper little pictures; or whether perceptual enhancements to program text will be adequate. Is there evidence that diagrammatic notations genuinely offer something that symbolic notations cannot match? If so, what; and how do we balance the trade-off against the disadvantages of having no text editor, no easy electronic mail, and all the other infra-structure of text? This is a crucial issue, at present little understood. It is disturbing to observe that many workers have not even asked the question, and instead assert uncritically that 'Pictures are more powerful than words ... Pictures aid understanding and remembering ... Pictures do not have language barriers. When properly designed, they are understood by people regardless of what language they speak' (Shu, 1988, pp. 7-9).

If the more dewy-eyed propositions of program visualization enthusiasts are accepted, then we can expect diagrams to be better for all purposes (and enhanced textual presentations to offer no advantages). The alternative position, more in keeping with the outlook of this chapter, is that the critical factor determining comprehensibility of notations is *accessibility of information*. Certain types of information are likely to be accessed more easily from diagrams. In many cases, however, perceptual enhancements to text-based presentations will also improve access.

(We should also note that all the questions that might affect choices in a real world have been brusquely suppressed. What about the sheer physical size of a diagrammatic representation of a complex program, or the memory requirements, the editing facilities, the speed of display, compilation and execution? We leave all those aside so that we can concentrate on the prior question: are diagrams any good?)

Although developments in program visualization are a lively area, and have given rise to various reviews and taxonomies (Myers, 1986; Shu, 1988; Chang, 1990) and some special issues of appropriate journals, we have at present nothing remotely resembling an adequate body of empirical research. There is clear evidence that both diagrams and enhanced text presentations can improve performance, but we do not at present know the types of task that are most improved. We shall consider diagrammatic notations in this section, and enhanced symbolic notations in the following one.

Despite the wide variety of diagrammatic representations that have been proposed, almost all the existing empirical studies deal with comprehensibility o flowcharts, a notation now widely believed to be rather poor. Early studies tendec to support that opinion. Thus Shneiderman *et al.* (1977) found little advantage for novices in having a flowchart as supplementary documentation, over a variety of tasks; Atwood and Ramsay (1978) found that a program design language was actually better than a flowchart for software design by graduate computer science students; Brooke and Duncan (1980a,b), who improved on the experimental techniques previously used, again found that flowcharts were of little help in debugging (though they did help in tracing execution flow) and that although they helped tc localize the area where a bug was, they were insufficient to identify the bug.

Gilmore and Smith (1984) compared listings, flowcharts and structure diagrams as aids to debugging, and found no overall differences. Unlike most authors they were able to pursue the data analysis at a more detailed level, and found that subjects were using two main debugging strategies: some attempted to extract as much information as possible from program breakpoints, while others tried to form a mental model of the program. It appeared that among the subjects choosing the modelling strategy, listings – i.e. straight text, with no diagram at all – gave the fastest performance, while among the breakpoint-using subjects, flowcharts gave fastest performance. Thus, diagrammatic notations are likely to be good for certain purposes only. (Although even this picture was also affected by some individual differences between subjects and problems.)

Curtis *et al.* (1988) report systematic comparisons of several notations across several tasks using professional programmers. They compared three types of elements (flowchart symbols, pseudo-code, and natural language) arranged in three types of diagram (sequential listing, branching flowchart, or hierarchical construction), and studied the comprehension, coding, debugging, and modification of smallish programs. In the comprehension experiment they examined mental execution of programs; 'backwards' or 'deprogramming' problems, asking what input conditions must be met to achieve specified program behaviour; and comprehension of dataflow. Analysis of results from this large and well-designed study was unusually thorough and rigorous. Results indicated that the pseudo-code (or PDL, program design language) was best overall, although flowchart-like representations gave the best performance on tasks 'that accentuated the importance of tracing the control flow rather than grasping higher-order relationships'. The least-effective formats were those that employed natural language; 'in particular, sequentially presented natural language, the format of ordinary text, was especially ineffective on their tasks'. In general, the choice of element types carried more weight than the spatial arrangement. They also noted that individual differences accounted for more of the variance than any other variable! Later work (Boehm-Davis *et al.*, 1987) has shown that the reason for the superiority of the PDL may be that it lessens the 'translation distance' from the documentation format to the program code.

Despite these negative findings concerning flowcharts, Cunniff and Taylor (1987) and their co-workers have achieved reasonable success. Disturbed by student difficulties with introductory Pascal, they have devised a graphical language, FPL, which is a structured flowchart 'informationally equivalent' to Pascal. In one experiment they compared speed and accuracy of novices (drawn from a course teaching both FPL and Pascal in tandem) on recognition of simple structures, flow of control, and input/output, and evaluation (hand simulation) of simple program fragments. FPL was clearly superior. Note, however, that they did *not* use the 'deprogramming' tasks.

A further study (Cunniff *et al.*, 1989) has investigated novices' program construction. At present, although the data is scant, it seems that certain typical semantic errors are just as common in FPL as in Pascal (cf. the study by Spohrer *et al.* (1989) of novices' Pascal bug frequencies), but that certain other bugs appear to be rarer in FPL. The authors' explanation is a clear statement of the importance of making information accessible: FPL's 'spatial arrangement allows the user to think more clearly about the placement of such assignments [e.g. missing initializations]. This is especially helpful in the performance of updating and incrementing counters and running totals. The user is able to 'see' where the looping begins and ends, resulting in a more clear-cut definition of the correct location for updating' (p. 428). This explanation is, in fact, a statement that users must 'deprogram' their programs. Had the authors' previous study included a deprogramming task, it would have been very helpful to their argument.

FPL is an example of the structure diagram, of which a wide variety of different forms were critiqued by Green (1982). Some problems can be foreseen without empirical investigation. Take, for instance, the Nassi-Shneiderman diagram, which indicates that process B is a subcomponent of process A by writing the icon for B *inside* the icon for A: clearly the user will have to write smaller and smaller as the nesting gets deeper! Also some notations lent themselves better to subsequent modification than other notations, a vital requirement, as we saw above. However, there was – and still is – little empirical evidence available by which to make comparisons.

Recently, the hegemony of control-flow studies of traditional procedural notations has been challenged. Boehm-Davis and Fregly (1985), arguing that concurrent programming systems raised special problems, compared Petri nets to pseudocode and resource-sharing documentation. Swigger and Brazile (1989) compared times and errors for experienced subjects making modifications of two kinds (procedural and data oriented) to a rule-based expert system, supported by one of two types of diagram, entity relationship or Petri net, respectively expressing data relationships and order dependencies. Both these studies found that Petri nets, with their strong expression of control flow, were less successful, but both also concluded that the program size affected the issue and that further investigation of large programs would be needed. Regrettably, no determined effort has been reported to locate which tasks are best supported by each of these various types of documentation.

Bonar's work on BridgeTalk (Bonar and Liffick, 1990) offers a different alternative to control-flow, by working at the level of programming plans. (See Chapters 2.4, 3.1 and 3.2 for a description of programming plans.) Programs in BridgeTalk are constructed by slotting together plan components, and a simple set of subcomponents is used to show the data flow. The interleaved character of Pascal (not to mention of flowcharts), in which the initialization subcomponent of a plan may be located arbitrarily far from the 'focal line', is thereby replaced by a simple regime in which all the subcomponents are kept together, even at the price of simplifying the semantics. Other interesting decisions were made, which are not strictly at the level of information structure and will not be described here. This is virtually the only serious investigation of the *design* of a notation, rather than the comparison of existing notations; but, being motivated by educational objectives, it was not designed to yield general-purpose conclusions. Nevertheless, it brings out, better than the flowchart studies reviewed above, an important notational principle: *strongly related subcomponents should be kept together, not dispersed.*

Unfortunately there are many more diagrammatic possibilities being touted by their supporters, than there are investigations.

4.5 Diagrams versus enhanced text

Instead of drawing diagrams and creating a new notation, one can continue to use the existing notation but use enhanced typography to make *perceptual cues reflect the notational structure.* There are obvious practical advantages, such as existing compilers, in sticking with existing notations. There are also more scientific advantages: it will help to clarify which claims about program visualization are true, and which are 'eyewash'.

A whole string of studies has now shown that ordinary program text can be made more comprehensible by supplying perceptual cues to important types of information. Payne *et al.* (1984) showed that a simple but hard-to-use command language for string editing was easier to use when commands were in upper case, argument strings in lower case. Isa *et al.* (1985) found that a meta-language for expressing syntax rules was more usable when structural cues were included. Gilmore (1986) showed that an improved version of Lisp 'pretty printing' helped his subjects both in writing and in comprehending programs, and Saariluoma and Sajaniemi (1989) showed that the components of spreadsheet programs were better recognized when the structure of the layout closely matched the internal structure of the spreadsheet, so that the perceived structure could be used to chunk the cells meaningfully.

In general, what ought to be useful is *improved access to the information that is (a) required, and (b) obscured.* The 'deprogramming' problem (Figure 1) illustrates one type of obscured information; 'programming plan' information is another type, since plans consist of statements that are dispersed in different places in the program. Gilmore and Green (1988) used colour cues, using the same colour for all components of the same plan, and showed that this one perceptual cue could improve learners' recognition of plan-based errors; their experiment also showed that control-flow error recognition was improved by indenting, and that cues to plans did not improve control-flow debugging, nor cues to control flow improve plan debugging. Thus the information-access hypothesis was well supported.

The Gilmore and Green results were obtained for small Pascal programs; Basic learners did not show the plan effect, but a later study by Davies (1990) showed that Basic programmers who had been taught structured programming techniques gave results similar to the Pascal programmers, while other Basic programmers thought mainly in terms of control flow – and were therefore presumably not seeking plan-like information. Van Laar (1989) has used the same technique to show that colour can supplement indentation in showing control flow in Pascal programs, with some net performance gain for learners answering a variety of comprehension questions. It would be extremely interesting to combine some of these techniques with the 'fish-eye' display (Furnas, 1986; see Chapter 1.2).

So the evidence is that the claims for improved comprehension from diagrams may be as well supported by non-diagrammatic techniques. The best conclusion at the moment, in the comparison between genuinely diagrammatic notations versus

```
 10 program prob12;
 20 vars depth, days, rainfall : integer;
 30   average : real;
 40 begin
 50   for days := 1 to 40 do
 60   begin
 70     depth := 0;
 80     writeln ('Noah, please enter todays rainfall');
 90     readln (rainfall);
100     rainfall := rainfall + depth;
110   end;
120   average := depth/40;
130   writeln('Average is', average);
140 end.
```

Figure 2: A small Pascal program from Gilmore and Green (1988). Shading represents the coloured highlighting of two plan structures, for forming a total in the variable depth and for inputting a value with a prompt. This program contains a 'plan' error at line 100 (it should read `depth := rainfall + depth`) and an 'interaction' error, combining plan and control-flow components, at line 70 (this line should be outside the For loop).

perceptual enhancements to text, seems to be that both are effective. There is no reason to suppose at present that diagrammatic notations are inherently superior. But it must be stressed that far too few of the diagrammatic possibilities have been properly investigated.

5 Programming as exploration

5.1 The implicit theory

We come now to the present view, which sees programming as opportunistic exploration or evolution, in which different alternatives are tried out and their consequences are considered. A favoured procedure is to find an old piece of program that 'almost' solves the present problem, and then to modify it bit by bit until the solution has been created. Recent languages and programming environments, led perhaps by Smalltalk, have put much emphasis both on re-usability and on the process of gradual, incremental change.

5.2 Research on change processes

We need to ask ourselves what features of notational and environmental design will support the process programming by placing fewest demands on the programmer. For learners, the tactical details of writing and changing code will be important. Gray and Anderson (1987) report the analysis of 'change episodes', points where a programmer (an advanced novice, in their study) altered the code that had been written. They argued that the most frequent changes would be made to those notational structures where the most planning was required, and further, that notational structures which could take a wider variety of forms would require more planning than structures that were more rigid. An example of such a plastic structure is the Lisp conditional structure. 'All conditionals have a Left parenthesis, a predicate, one or more clauses, and a Right parenthesis. Within this rigid structure the number, type, and the order of clauses can vary and it was these attributes that were involved in 15 change episodes. Because of the variability in how it can be used in Lisp, the goal-structure for the conditional ... must contain a large proportion of planning goals. The presence of planning goals is reflected by the frequency with which the conditional is involved in change-episodes' (p. 192). The authors contrast this with another form, the conditional clause, which is more rigid and which was involved only in the statistically expected number of change episodes.

As the authors point out, this study should not be over-interpreted, since it used only fifteen subjects and only a single notation: moreover, the fact that conditional structures cause problems was entirely predictable from earlier research. What they have done is to predict those problems from theoretical grounds, and to supply the firm prediction that notations with a more rigid, conventionalized structure would create fewer planning goals and therefore make programming easier.

Green *et al.* (1987) report a study scrutinizing performance in a variety of languages instead of only one, but paying correspondingly less attention to the fine details of individual behaviour. Using professional programmers working in their language of choice, they posed very simple problems (to reduce problem-solving behaviour to a minimum) and took detailed records of keystrokes as solutions were constructed. The key variable studied was the frequency of a 'backward move' to insert new material into part of the text that had already been written. The languages compared were Pascal, Basic and Prolog. Very similar solutions were produced in Pascal and Basic, but about four times as many backward moves were made in Pascal as in Basic, with Prolog occupying an intermediate position.

The most interesting aspects, from our viewpoint of notational design, came from considering the interplay of knowledge structures and code design. These authors accepted the 'programming plan' model of programmers' knowledge as a working hypothesis (see Chapters 3.1 and 3.2) in which plans contain a 'focal line' which achieves the goal of the plan. Sometimes a plan contains a precondition also, for example the counting plan, which contains a focal line `x := x + 1` and a precondition `x := 0`. They postulated that these preconditions would be forgotten more frequently than other plan components. Their data showed that the Pascal programmers more frequently went back to insert preconditions than the Basic programmers, and that the Prolog equivalent (the 'base case') was hardly ever the target of such a retrospective addition to the text. Could that be because base cases are spatially located right beside their associated focal line in the main case?

Davies (1989) extended this paradigm by examining the backward moves made by experts and intermediate learners in generating Pascal and Basic programs. The level of expertise was a more important discriminator than the programming language, but the most remarkable result was that intermediates jumped *within* plans while experts jumped *between* plans. Very similar results were found in a free-recall study. This takes us to questions of how knowledge representation interacts with notational structure, which we cannot consider here.

Research in this area is currently well behind the technology. Object-oriented programming languages are specifically designed with an information structure intended to support processes of change and modification, and especially of software reuse; yet we have very little evidence on whether they are successful – let alone on which of their many variations! Lange and Moher (1989) report a prolonged study of a single subject, who indeed frequently reused code, but – despite the intentions behind object-oriented programming languages – did so by copying text, rather than by using the special facilities of method-inheritance provided by the system. In general, 'the subject ... avoided techniques requiring deep understanding of code details or symbolic execution whenever possible'. While disappointing, no doubt, to the language designers, results like this do highlight the problems to be solved.

6 Lowering the barriers to programming

Studying how to improve programming notation has led us from considering whether some language features are intrinsically hard, through the problems of comprehension, into the question of how the notational structure interacts with the cognitive processes of planning and the 'fit' between the notational structure and the knowledge structure. As a side issue on the way we have examined the claim that visual notations are especially revealing, and have found little evidence to support it.

There is an urgent need for research on the interrelationship of notations and environments. To date, surprisingly little comparative work has been reported on environments, and virtually none that treats the problem of matching notation to environment. As it becomes easier to construct environments to support programming and programming-like activities, the possibilities will be explored and our knowledge will grow.

Knowledge structures and their interaction with notational design also need to be investigated in far greater detail than to date. Other chapters in this volume describe how our understanding of these topics is being investigated. No doubt, future studies will report further details of how notational designs can increase or decrease the need for local planning during coding, but also how the programmer's knowledge base can similarly affect the issue, in line with the work by Gray and Anderson reported above.

Where do we stand now? It seems that existing notations too often act to raise barriers against programming: barriers to learners, preventing understanding, and likewise to experts, preventing the fast access to information and modification that they need. There have been a few attempts to state design requirements for good languages, drawing on cognitive principles. Lewis and Olson (1987) suggest that, for the learner, one of the most important requirements is to suppress the 'inner world' of programming, the world of variable declarations, loops and input/output. The spreadsheet may be the model of the future, as they see it. (Although the information structure of ordinary spreadsheets has some problems – see Green, 1989).

Fitter and Green (1979) analysed the purported advantages of diagrammatic notations. Ten years later, I believe that their conclusions about notational design still hold good. Curtis (1989, p. 96) restated them so succinctly that I shall use his words:

> Fitter and Green (1979) argued that the primary problems in specification formats are the tractability and visibility of structure. Useful notations contained not only symbolic information, but also perceptual cues. For instance, maps use spatial location, histograms use variation in size, and Venn diagrams use spatial containment. They listed five attributes of a good notational scheme. These attributes are:
>
> (1) Relevance – highlights information useful to the reader.

(2) Restriction – the syntax prohibits the creation of disallowable expressions. [I.e. hard to understand, or likely to be confused with closely-related forms.]

(3) Redundant recoding – both perceptual and symbolic characteristics highlight information.

(4) Revelation – perceptually mimics the solution structure, and

(5) Revisability – easily revised when changes are made.

Of course, there is more to be said. But if programming notations met just those requirements, it would make a start.

Acknowledgements

My grateful thanks to Jean-Michel Hoc, Dorothy Graham, and especially Marian Petre, who all made detailed comments on an earlier version of this chapter.

References

Arblaster, A. T., Sime, M. E. and Green, T. R. G. (1979). Jumping to some purpose. *The Computer Journal,* **22,** 105-109.

Atwood, M. E. and Ramsay, H. R. (1978). *Cognitive structures in the comprehension and memory of computer programs: an investigation of computer debugging.* Alexandria, VA: U.S. Army Research Institute, TR78A21.

Boehm-Davis, D. A. and Fregly, A. M. (1985). Documentation of concurrent programs. *Human Factors,* **27,** 423-432.

Boehm-Davis, D. A., Sheppard, S. B. and Bailey, J. W. (1987). Program design languages: how much detail should they include? *International Journal of Man-Machine Studies,* **27,** 337-347.

Bonar, J. and Liffick, B. W. (1990). A visual programming language for novices. *In* S.-K. Chang (Ed.), *Principles of Visual Programming Systems.* Englewood Cliffs: Prentice-Hall.

Brooke, J. B. and Duncan, K. D. (1980a). An experimental study of flowcharts as an aid to identification of procedural faults. *Ergonomics,* **23,** 387-399.

Brooke, J. B. and Duncan, K. D. (1980b). Experimental studies of flowchart use at different stages of program debugging. *Ergonomics,* **23,** 1057-1091.

Chang, S.-K. (Ed.). (1990). *Principles of Visual Programming Systems.* Englewood Cliffs: Prentice-Hall.

Cunniff, N. and Taylor, R. P. (1987). Graphical versus textual representation: an empirical study of novices' program comprehension. *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Cunniff, N. , Taylor, R. P. and Black, J. B. (1989). Does programming language affect the type of conceptual bugs in beginners' programs? A comparison of FPL and Pascal. *In* E. Soloway and J. C. Spohrer (Eds), *Studying the Novice Programmer.* Hillsdale, NJ: Erlbaum.

Curtis, B. (1989). Five paradigms in the psychology of programming. *In* M. Helander (Ed.), *Handbook of Human-Computer Interaction.* Amsterdam: Elsevier (North-Holland).

Curtis, B., Sheppard, S., Kruesi-Bailey, E., Bailey, J. and Boehm-Davis, D. (1988). Experimental evaluation of software documentation formats. *Journal of Systems and Software,* **9**, 1-41

Davies, S. P. (1989). Skill levels and strategic differences in plan comprehension and implementation in programming. *In* A. Sutcliffe and L. Macaulay (Eds), *People and Computers,* vol. V. Cambridge: Cambridge University Press.

Davies, S. P. (1990). The nature and development of programming plans. *International Journal of Man-Machine Studies,* **32**, 461-481

Fitter, M. J. and Green, T. R. G. (1979). When do diagrams make good computer languages? *International Journal of Man-Machine Studies,* **11**, 235-261.

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

Gilmore, D. J. (1986). Structural visibility and program comprehension. *In* M. D. Harrison and A. F. Monk (Eds), *People and Computers: Designing for Usability.* Cambridge: Cambridge University Press.

Gilmore, D. J. and Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies,* **21**, 31-48.

Gilmore, D. J. and Green, T. R. G. (1988). Programming plans and programming expertise. *Quarterly Journal of Experimental Psychology,* **40A**, 423-442.

Gilmore, D. J. and Smith, H. T. (1984). An investigation of the utility of flowcharts during computer program debugging. *International Journal of Man-Machine Studies,* **20**, 331-372.

Gray, W. and Anderson, J. R. (1987). Change-episodes in coding: when and how do programmers change their code? *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Green, T. R. G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology,* **50**, 93-109.

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

Green, T. R. G. (1989). Cognitive dimensions of notations. *In* A. Sutcliffe and L. Macaulay (Eds), *People and Computers.* vol. V. Cambridge: Cambridge University Press.

Green, T. R. G., Bellamy, R. K. E. and Parker, J. M. (1987). Parsing-gnisrap: a model of device use. *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Hoc, J.-M. (1981). Planning and direction of problem solving in structured programming: an empirical comparison between two methods. *International Journal of Man-Machine Studies,* **15,** 363-383.

Hoc, J.-M. (1988). Towards effective computer aids to planning in computer programming. *In* G.C. van der Veer, T.R.G. Green, J.-M. Hoc and D.M. Murray (Eds), *Working with Computers: Theory Versus Outcome.* London: Academic Press.

Holt, R. W., Boehm-Davis, D. A. and Shultz, A. C. (1987). Mental representations of programs for student and professional programmers. *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, N.J.: Ablex.

Isa, B. S., Evey, R. J., McVey, B. W. and Neal, A. S. (1985). An empirical comparison of two metalanguages. *International Journal of Man-Machine Studies,* **23,** 215-229.

Jones, C. (1979). A survey of programming design and specification techniques. *Specifications for Reliable Software.* IEEE Computer Society.

Lange, B. M. and Moher, T. G. (1989). Some strategies of reuse in an object-oriented programming environment. *Proceedings of the CHI'89 Conference on Computer-Human Interaction, 69-73.* New York: ACM, pp. 69-73

Lewis, C. and Olson, G.M. (1987). Can principles of cognition lower the barriers to programming? *In* G. M. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Mayer, R. E. (1976). Comprehension as affected by structure of problem representation. *Memory and Cognition,* **44,** 249-255.

Myers, B. A. (1986). Visual programming, programming by example, and program visualization: a taxonomy. *Proceedings of the CHI'86 Conference on Computer-Human Interaction.* New York: ACM.

Naur, P. (1983). Program development studies based on diaries. *In* T.R.G. Green, S.J. Payne and G.C. van der Veer (Eds), *The Psychology of Computer Use.* London: Academic Press.

Payne, S. J., Sime, M. E. and Green, T. R. G. (1984). Perceptual structure cueing in a simple command language. *International Journal of Man-Machine Studies,* **21,** 19-29.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology,* **19,** 295-341.

Saariluoma, P. and Sajaniemi, J. (1989). Visual information chunking in spreadsheet calculation. *International Journal of Man-Machine Studies,* **31,** 475-488.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Information and Computer Sciences,* **52,** 123-143.

Shneiderman, B., Mayer, R. E., McKay, D. and Heller, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM,* **20,** 373-381.

Shu, N. C. (1988). *Visual Programming.* New York: Van Nostrand Reinhold.

Sime, M. E., Green, T. R. G. and Guest, D. J. (1973). Psychological evaluation of two conditional constructions used in computer languages. *International Journal of Man-Machine Studies,* **5**, 105-113.

Sime, M. E., Green, T. R. G. and Guest, D. J. (1977). Scope marking in computer conditionals – a psychological evaluation. *International Journal of Man-Machine Studies,* **9**, 107-118.

Spohrer, J. C., Soloway, E. and Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. *In* E. Soloway and J. C. Spohrer (Eds), *Studying the Novice Programmer.* Hillsdale, NJ: Erlbaum.

Swigger, K. M. and Brazile, R. P. (1989). Experimental comparison of design/ documentation formats for expert systems. *International Journal of Man-Machine Studies,* **31**, 47-60.

Van Laar, D. (1989). Evaluating a colour coding programming support tool. *In* A. Sutcliffe and L. Macaulay (Eds), *People and Computers V,* Cambridge: Cambridge University Press

Vessey, I. and Weber, R. (1984). Conditional statements and program coding: an experimental evaluation. *International Journal of Man-Machine Studies,* **21**, 161-190.

Youngs, E. A. (1974). Human errors in programming. *International Journal of Man-Machine Studies,* **6**, 361-376.

Wandke, H. (1988). User-defined macros in HCI: when are they applied? Paper delivered at *Macinter 2 Conference on Man-Computer Interaction.* Unpublished technical report from Sektion Psychologie der Humboldt-Universität zu Berlin, Oranienburger Str. 18, DDR-1020 Berlin, GDR.

Wetzenstein-Ollenschlaeger, E. and Schult, S. (1988). The influence of knowledge on the definition of procedures. Paper delivered at *Macinter 2 Conference on Man-Computer Interaction.* Unpublished technical report from Sektion Psychologie der Humboldt-Universität zu Berlin, Oranienburger Str. 18, DDR-1020 Berlin, GDR.