This is a report on recent research by myself and my colleagues, Max Sime, Stephen Payne and David Gilmore. Where I say "we" and "our" it refers to all of us. I hope I have not misinterpreted their ideas too much.

Previous research at this Unit into the causes of difficulty in comprehending programs led us to the conclusion that it is useful to regard programs in the same light as other forms of presentation of complex information, and to ask how easy is it to extract necessary information from them. This viewpoint emphasises the role of structure: the program structure must be easily perceived, and it must make it easy to perform the user's task given the usual human abilities and disabilities. Structure must be well-specified, visible, and appropriate.

In the first section of this paper I shall briefly outline the course of our work on program comprehension, in order to establish our views on structure. The following sections describe recent research at this Unit into the causes of difficulty in learning and using text editors. We believe that the notations of command languages and of programming languages need to satisfy very similar requirements as regards visible and appropriate structure. The final section offers some conclusions, necessarily tentative.

1. COMPREHENDING SMALL PROGRAMS: POOR NOTATION CAUSES PROBLEMS

The early programming work at this Unit is often linked with the use of the 'Hungry Hare', a simple card-sorter with some lights on it; this purpose-built laboratory device has aroused both admiration (du Boulay and O'Shea and Monk, 1981) and disdain (Sheil, 1981). Early experiments established that students with no programming experience found it easier to write simple conditional programs using a particular form of nested syntax than using an unconstrained GOTO language (Sime, Green and Guest, 1973, 1977); in particular, with one syntax (called Nest-INE) they cleared up bugs more easily. Subsequent experiments attempted to establish an explanation of these results in a form that was sufficiently general and powerful to extend to other constructions in programming. Unfortunately these experiments are often regarded simply as comparisons of conditional designs, rather than comparisons of classes of information structure in which the conditional serves as a representative of many similar structures.

There are strong indications that the effects are caused by differences in the ease of extracting information from programs, and in particular the ease of extracting 'circumstantial' information: that is, discovering the circumstances which cause a conditional program to behave in a specified way. Among these clues, Green (1977) found that when professional programmers answered circumstantial questions about programs, the different syntactic designs greatly affected response times; whereas when they answered sequential questions (the inverse of circumstantial - that is, given the circumstances they found the action) the different syntaxes made little difference; a similar result, obtained with a quite different paradigm, was described by Green (1980). Another important indication was that when we returned
to asking novices to write programs for the Hungry Hare, using the worst of our syntax designs, the GOTO or Jump language, we found that their programs were more likely to be correct first time if they were constrained to use GOTOs to simulate nested conditionals instead of using them haphazardly - BUT they were no better at correcting their mistakes than the unconstrained group (see Arblaster et al., 1980, for summary).

The obvious explanation is the one we have proposed: conventional, procedural, programming language designs favour the extraction of sequential information, but circumstantial information can be made more available by using a well-structured design to make the information-gathering operations simpler, and making the information more visible with cues (here provided by Nest-INE). Can other explanations be found? Our experiments clearly showed that 'good structure' on its own was insufficient, and indeed studies by Van der Veer and Van de Wolde (1983) and unpublished studies by ourselves have found cases where 'good structure' is counterproductive. In his review paper, Sheil (1981) suggests that variations in program length, caused by the different syntax designs, may have caused these effects, but since the same programs were used for both types of question in the studies described by Green (1977) and Green (1980) I am at a loss to understand his reasoning. It is equally hard to understand how his explanation would address the results on correction of errors using constrained and unconstrained GOTOs.

Pursuing our explanation leads to the following position. (1) The activities of programming clearly involve both reasoning about programs, and discovering the facts upon which reasoning is based. (2) Discovering the facts frequently means extracting information from written programs (or other notations, such as specifications, manuals, post-mortem dumps, etc.). (3) Different classes of information structure highlight different types of information; in particular, procedural programming languages highlight sequential information, and declarative ones highlight circumstantial information. (4) The availability of information is therefore limited by the difficulty of extracting it from the given structure, and when there is a mismatch - e.g. when requiring circumstantial information from a procedural program - performance will be worse; this will create errors, slow up performance, impair reasoning, encourage programmers to guess rather than make certain, etc. (5) The problems can be ameliorated by adding cues; what the Nest-INE syntax did was to add cues to help in the extracting of circumstantial information. (6) Simple models of information extraction, using such mental operations as searching for labels, negating predicates, performing parsing manipulations, etc., can not only account for the results but can also extend them to many other familiar aspects of programming language design, such as parameter passing.

1.1 EXPERIMENT I: CUES AND STRUCTURE

In an experiment by Gilmore and Green (1984), we have gone some way to justify some of these grand claims by a direct test of the 'mismatch hypothesis'. A short algorithm was coded in four ways: either procedural or declarative and either with cues or without cues. Subjects (non-programmers, to avoid effects of prior experience) were asked questions that required either sequential information or circumstantial information. Some of the questions were answered with the program in front of them, some were answered from memory of the program. Full details of the study cannot be given here, but Table 1 shows a sample of the results, taken from the recall stage. The figures show quite clearly that cues improved performance in mismatch conditions, as predicted. Moreover the differences between uncued match and mismatch conditions were in complete agreement with our predictions, although in the cued conditions our predictions were slightly upset.

Although our explanation is not perfect, it is intuitively 'obvious' and it successfully predicts effects unknown to any competing theory. (Examples of competing theories of program comprehensibility that cannot explain these results include the 'syntax-semantics' model of Shneiderman and Mayer, 1979; the structured programming