Chapter 2.1

Expert Programmers and Programming Languages

Marian Petre

Instituut voor Perceptie Onderzoek/IPO, Postbus 513, 5600 MB Eindhoven, The Netherlands

Abstract

This chapter contrasts cursorily the aspirations of general-purpose programming language designers with some evidence about expert problem solving and programming behaviour. The contrast is summarized in a rough wish list of what experts want from general-purpose programming languages. The programmers' wish list differs from the aspirations of language designers less in detail than in emphasis: whereas the designers emphasize well-foundedness and correctness, the expert programmers emphasize utility, control and efficiency. It is argued that a programming language is a tool, not a panacea; tools make easy the tasks for which they are designed, but the outcome depends on the intention and expertise of the wielder.

Psychology of Programming
ISBN 0-12-350772-3

1 Introduction: the language-user/language-designer schism, and why practitioners complain about new languages

Programming languages, assessed as they tend to be in personal and mystical terms, attract a tenacious confusion of myth, of variable quality. The accumulation of programming language myth reflects the evolution of computing. Programming languages are artefacts developed in mixed environments: technological, social and philosophical. Assumptions – often unspoken, sometimes accidental – incorporated in those original environments may survive the environmental constraints that nurtured them and may become irrelevant, inappropriate, or even misguided when the environment changes. Evolution implies changes in truth; myths convenient or pertinent at their inception may obscure issues in their continuation.

Programming began in the hands of engineers and hackers who created improvements as needed to provide obvious advantages of speed and power. Other criteria for languages were personal. The point was to build a tool and exploit it; 'software' was a way to build up layers of tool enhancements.

Software has since moved into the hands of theoreticians whose criteria are different from the hackers'. Theoreticians intend to improve languages by design rather than just by demand and tinkering, and they aspire to make languages that conform to formal, usually classical, models (e.g. functional languages are based, with varying mathematical purity, on the lambda-calculus). This evolution produced a schism between those who use languages (engineers and hackers) and those who design them (theoreticians). There is a shift of orientation from utility to 'well-foundedness'.

Early programming languages were clumsy and occasionally dangerous because they were not well-founded. Modern languages often fail in practice because emphasis on theory displaces attention to use. This re-orientation is not inherently bad, although it can result in impractical languages. The re-orientation is, however, incomplete and hence misleading; critical decisions (e.g. choice of formal model, patterns of notation or implementation not covered by the model) remain ill-founded. For example, Kowalski champions logic programming on the disputable grounds that 'Symbolic logic was first designed as a formalization of natural language and human reasoning' and hence 'Logic reconciles the requirement that the...language be *natural and easy to use* with the advantage of its being machine-intelligible'. (Emphasis added; Kowalski, 1982.) Yet there is evidence to undermine these 'foundations' (e.g. Taylor and du Boulay, 1986; Fung, 1987).

The job now is to reconcile the factions. Foundations must be ascertained and verified in the context of utility. Models must be chosen that both afford the refinement of high-level reasoning and accommodate real-world constraints.

This chapter contrasts cursorily the aspirations of the general-purpose programming language designers with some evidence about expert problem solving and programming behaviour. The contrast is summarized in a rough 'wish list' of what experts want from general-purpose programming languages.

2 Design aspirations

Hoare wrote in 1973: '...good language design may be summarized in five catch phrases: simplicity, security, fast translation, efficient object code, and readability'. These provide a good centre for more recent aspirations, although, as Hoare anticipated: '...many language designers have adopted alternative principles which belittle the importance of some or all of these criteria, perhaps those which their own languages have failed to achieve'. Much language theory (e.g. denotational semantics) is concerned with separating the basic semantic description of a language from the notion, however abstract, of an evaluating mechanism or computer. This theoretical goal has infiltrated design attitudes: fast translation and efficient object code are 'implementation issues' considered separable from (and often subordinate to) qualities of the language model. Hence, these will not be discussed in this section. Hoare's three remaining 'catch phrases' will lead this (by no means exhaustive) discussion onto other design aspirations popular in the literature.

2.1 Simplicity

By simplicity, Hoare means a small range of instructions with uniform format, each having a simple effect that can be described and understood independently of other instructions. Terseness or simplicity of syntax is a common goal, one supported by studies of notations (e.g. Green, 1977) and reflected to some extent in recent designs (e.g. Miranda).

2.2 Orthogonality

Related to simplicity is orthogonality: the notion that there should not be more than one way of expressing any action in the language, and that all language components are mutually independent (cf. Denvir's more comprehensive examination of the notion, 1979). In a truly orthogonal language, a small set of basic facilities may be combined without arbitrary restrictions according to systematic rules. Orthogonality may result in arbitrary complexity via this unrestricted combination. It is endorsed in the literature in a limited role: as a principle for eliminating redundant expressions and hence for contributing to overall simplicity. This limited orthogonality is implied above in Hoare's meaning of simplicity.

2.3 Security

Hoare's principle is that only syntactically correct programs should be accepted by the compiler and that results (or error messages) should be predictable and comprehensible in terms of the source language program. No program should cause the computer to 'run wild' The classic examples for this aspiration are Pascal, a 'secure', strongly typed language, and C, a 'dangerous' one that produces occasional 'nasty surprises'. Building on the principle of security is correctness.

2.4 Correctness

This is the notion that a program can be proven to conform to a specification or to exhibit specified properties. This goal drives the aspiration to develop languages that conform to formal models and so are amenable to formal manipulation. Indeed, the pursuit of correctness is one of the *principal* aspirations of recent designs.

2.5 Readability

Hoare argues, simply, that programs are read by people, and that the programming language should encourage clarity. He asserts: 'The readability of programs is immeasurably more important than their writeability'. Unfortunately, a certain amount of myth adheres to readability: 'Authors choose stylistic factors commonly thought to influence style with little or no supporting evidence demonstrating the importance or effect of that characteristic on program comprehension and maintainability' (Oman and Cook, 1988). This notion leads to another: clarity of structure.

2.6 Clarity of Structure

Language designers aspire to express the intended solution structure visibly, without artificial imposition of extraneous structure, including artificial sequentiality. A related goal is to express any inherent parallelism (and to exploit it via the language implementation).

2.7 Modularity

One of the techniques for structuring larger programs is to segment or divide them into manageable pieces. Modularity allows programs to be segmented into independent chunks or *modules* which have *locality*, that is, which have internal scope and which interact with other modules via a defined interface. (cf., for example, Hughes, 1984.) The independence of the chunks is important, permitting separate compilation, separate debugging, independent testing, and reasonably easy replacement or modification. The independence and locality of modules is sometimes interpreted as 'hiding' an enforced rather than notional restriction of access to internals.

2.8 Abstraction

The motivation behind many language design decisions is to spare the programmer the 'messy' bits, the low-level, machine-oriented details of programming. The emphasis is on high-level operation, with the idea that high-level expression matches more closely the problem domain. As will be discussed later, this is a restrictive interpretation of abstraction.

3 The influence of the programming language on programming (that is, on devising solutions)

'Computer scientists have recognized that the features of a programming language can have a significant effect upon the ease with which reliable programs can be developed. It has also been observed that certain languages and language features are particularly well suited for the use of systematic programming techniques, while others hinder or discourage such discipline' (Wasserman, 1975, with reference to the then-recent structured programming techniques). The usual way to interpret this sort of statement is that the programming language guides the solution. Indeed, many of the arguments of modern designers are based on a conviction that programming languages influence the sorts of solutions that programmers will devise and that good languages guide the programmer's thinking (into whatever avenue is endorsed by the particular language model). Associated with this conviction is a certain amount of evangelism about how programmers *should* think.

Yet it is a vain hope that the programming language itself will provoke good code. (Just about everyone knows someone who can write Fortran in any language.) Flon's axiom (1975) states the case strongly: 'There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs'. Experience with expert programmers (described in Section 4.2) offers an alternative interpretation of Wasserman's observations: that the programming language may facilitate clear *expression* of a solution, or it may obstruct it, but that the language does not influence strongly the nature of the solution strategy.

In the general case, language is a tool. Tools make easy the tasks for which they are designed, but the outcome depends on the intention and expertise of the wielder. Whereas a language can provide convenient categories, the support and heuristics for choosing the right abstractions in solving the problem are at best only implied. The *user* mediates the relationship between language and programming (i.e. devising solutions): individual expertise can compensate for deficiencies in language, whereas language cannot wholly compensate for deficiencies in use.

As summarized by Hoare (1981): 'I have regarded it as the highest goal of programming language design to *enable* good ideas to be elegantly *expressed*'. (Emphasis added.) Similarly, Dijkstra (1976) wrote: 'I view a programming language primarily as a *vehicle* for the *description* of (potentially highly sophisticated) abstract mechanisms'. (Again, emphasis added.) The goal of designers should not be to influence thinking but to reflect it clearly, to remove obstructions and restrictions. As Zemanek (1985) argues (as part of his case for formal definition): 'One cannot standardize thinking and one should not even attempt to do so'.

The following sections justify this more moderate view of the role of the programming language by considering some evidence about how experts solve problems and how expert programmers program.

4 How experts behave

4.1 Expert problem solving

Expertise in problem solving has been studied in a variety of domains, and results tend to be consistent across domains (for reviews, see Kaplan *et al.*, 1986; Allwood, 1986). Studies typically contrast novice with expert performance in solving problems. Experts differ from novices in both their breadth and their organization of knowledge: experts store information in larger chunks organized in terms of underlying abstractions. This organization apparently facilitates quick recognition of problem types and recall of associated solution strategies.

Experts sort problems in terms of underlying principles or abstract features (e.g. Chi *et al.*, 1981; Weiser and Shertz, 1983), whereas novices rely on surface features. Recall exercises (e.g. Chase and Simon, 1973; Shneiderman, 1976; Adelson, 1981, 1984; McKeithen *et al.*, 1981) have shown that experts are better able to reconstruct realistic domain configurations, such as chess positions or computer programs, although they perform no better than novices with random or scrambled stimuli. The interpretation for programming is that experts represent programs in terms of semantic structure, whereas novices encode them syntactically.

Experts tend to spend more time than novices planning and evaluating. Experts run programs more frequently while debugging than novices do (Gugerty and Olson, 1986). Experts are better able to form overviews, but thereafter they take longer to develop their understanding and representations, and they consider more fully interactions among functions or components of a system (Adelson *et al.*, 1984). Experts form a detailed conceptual model of a system and tend to incorporate abstract entities rather than the concrete objects specific to the problem statement (Larkin, 1983). Their models accommodate multiple levels and are rich enough to support mental simulations (Jeffries *et al.*, 1981; Adelson and Soloway, 1985; see also Chapter 3.1 concerning a schema-based view of expert programming knowledge and mental simulation).

The pattern overall is that experts are able to handle information at different levels. They differ from novices in two important respects: their ability to develop overviews or abstract models of solutions, and their ability to understand the consequences of implementation detail.

4.2 What experts do (and don't do) with programming languages

Given the emphasis, in the expertise literature, on abstract, semantic models rather than syntactic representations, where does the programming language impinge on programming?

One study of expert programmers solving problems in several general-purpose programming languages (Petre and Winder, 1988) has suggested that a programming language has only a weak influence on the solution. In this investigation, experts were presented with five non-trivial problems for solution in three languages each. The languages were chosen by the experts themselves, most choosing languages in more than one language style (e.g. Basic, Pascal, and Scheme; or Forth, KRC, and Prolog). In follow-up questionnaires and interviews, experts were asked to discuss, among other things, the suitability of each language for solving the problems or expressing the solutions.

Particular languages or language types did *not* correspond to particular solution strategies, although there *was* some correspondence between languages and oversights. It seems that experts conceive an abstract model of the solution separate from its expression in a particular programming language. There was evidence that they solve problems, not in the target programming language, but in a private, pseudo-language that is a collage of convenient notations from various disciplines, both formal and informal. The pseudo-language might be taken as the surface reflection of the expert's computational model, itself a composite of models borrowed from many sources.

Examples and remarks volunteered by the experts were sufficiently consistent to suggest the natures of these pseudo-languages. Variety of representation was important. Some of the notations derived from programming languages, including languages of different styles, others from mathematics, logic, natural language description, and sketches or diagrams – or combinations of these. Some code fragments were incorporated verbatim. The collections were apparently not made coherent but were assemblages of overlapping and possibly inconsistent fragments. The implication was that, although the pseudo-languages may not be self-consistent, coherence was imposed in use.

Different notations were used for different sorts of tasks. The associations between notations within the pseudo-language and general categories of problems and solutions were made on the basis of suitability; that is, the pseudo-language fragment made accessible some problem information or solution feature, or facilitated some set of operations, that was then of particular interest to the expert. A 'mix and match' description was typical, e.g. 'I find Algol-like pseudo-code useful for control-structure problems, together with arrays, sets, etc. from mathematics. This doesn't, however, cope well with control abstraction...for which I tend to use Scheme and Miranda-like notations...'. This sort of association is consistent with findings about expert problem classification (mentioned in Section 4.1) and makes good sense in terms of findings that problem representation is critical to ease of solution (See Section 2 on problem representation in Chapter 1.3). The borrowing in the pseudo-languages from many disciplines is suggestive; it may be that the pseudo-language provides a means of mapping or translation between problem and programming domains.

Given that the abstract algorithm is captured first in the programmer's own pseudo-language, expression in the target programming language is a matter of subsequent translation. This notion is by no means alien to the programming culture; Kernighan and Plauger (1974) endorsed it years ago as a principle for good design: 'Write first in an easy-to-understand pseudo-language; then translate into whatever language you have to use'. Hence, the algorithm is largely programming language independent, but it is dependent instead on some computational model private to the programmer.

This decoupling of programming – devising a solution – from programming language accounts for phenomena observed in the study. Experts were tenaciously resistant to a change of algorithm; in no case did a change of language *itself* provoke a change of algorithm. Unless provoked strongly, as by failure or inefficiency, experts did not look beyond the first algorithm 'good enough' to satisfy the problem. Rather, they were willing to tolerate heavy translation overheads in recoding a solution into an alternative language. Rather than guiding algorithm choice, programming languages may be distorted into conformance with the programmer's abstract solution.

In contrast, languages were changed much more readily than algorithms. Experts were willing to change languages just for fun as well as under the provocation of unacceptable obstruction (see next subsection).

4.3 Obstructions to coding; how programming languages get in the way

Apparently, although a programming language is unlikely to contribute directly to a solution, it may obstruct solution, even contributing to errors or oversights. Du Boulay and O'Shea (1981) also suggest that different error types may correlate to particular programming languages. In the study cited above (Petre and Winder, 1988), experts observed a number of coding obstructions.

Strongest among the experts' objections were obstructions to performance: inefficiency and an inability to access the facilities in the hardware. In other words, and in contrast to the aspirations of language designers, languages which did not support low-level manipulations were considered obstructive (e.g. 'I'd love to see someone try and do that in Scheme; you can't. You can't get at the bits.' and 'There's a lot of...design decisions which you can't easily express in a language which doesn't allow you to control storage'). All of the experts had an awareness of the machine that they instruct and, more broadly, of the world in which they operate. All of them claimed to have a reasoning model of the underlying machine; their aim was to produce a program that would run effectively on a computer. Many of the experts were not satisfied with a solution until the code had been compiled and run. Their concern about efficiency underlines the persistence with which reality constrains computing problems; there is a tension between the importance of the abstract algorithm and practical demands.

In order to satisfy their concerns about performance, experts demonstrated a wily evasion of restriction and a canny persistence in uncovering necessary details.

Other obstructions inhibited clarity of expression, a well-recognized goal: 'The problems that must be solved with today's languages are not simple. It is important that the programmer's task not be compounded by an additional layer of complexity from the very tool that is being used to solve the problem' (Marcotty and Ledgard, 1987). Experts bemoaned a lack of data structuring tools, the inadequacy of abstraction facilities, an inability to reflect the solution structure, and verbosity or notational clutter (e.g. 'So, how do you arrange, in a purely array world, to imitate a list?', '...you can actually manage to package up control structures in Scheme, whereas you can't really in C++ or Pascal. Now, the lack of that is something I've felt for a long time...' and '...the sheer textual complexity was enough to stop me even trying'). The conclusion was that 'ugliness' matters; programmers will refuse to use a programming language that is too obstructive or unpleasant. (Typical examples were Cobol, rejected for its verbosity, and Scheme, rejected for its messy punctuation.)

5 What expert language users want: a programming language 'wish list'

This section draws on the reported investigations of expert programmers to offer a wish list of general language facilities and qualities. Although this list is by no means comprehensive, it does attempt to capture the gist of what expert programmers expect from, and how they expect to use, general-purpose programming languages.

5.1 Granularity and control

The characteristic mastery by experts of both abstraction and detail, and the tension demonstrated by expert programmers between the abstract algorithm and practical demands, reflect the key tradeoff in modern programming languages: that between power, in the sense of abstraction and combination, and specific control. The experts want, on the one hand, the ability to abstract, to think in terms of high-level constructs, and, on the other hand, the ability to manipulate hardware or 'to get at the bits'. Experts need to look at different things at different times; in particular they need to choose their *grain of focus* at any one time (cf. Chapter 3.3 concerning opportunistic design strategies).

Expert programmers resent restrictions, particularly restrictions masquerading as 'protection'. The trouble in many languages is the assumption that programmers *can* forget details, that concrete detail, whether explicit or embedded in the language implementation, is 'accompaniment' to a program rather than part of it. These languages confuse 'exclusion of detail' with 'granularity of detail'. The notion of protecting the programmer from low-level details is, in the case of the expert, misguided. Lesk (1984) characterizes the problem: 'The good news about this is that you don't have to decide on the recursion or iteration yourself. The bad news is that you may not like what you get'.

A more appropriate aspiration than *protection* is *selection*: the ability to select the granularity of information for the context, when the context includes the intentions and goals of the expert as well as the nature of the problem, and to dip among the levels during a single task. The aspiration should be to provide information only when desired and to provide sensible defaults, so that, if the programmer is not interested, the implementation will handle the details.

Moreover, the language should provide comprehensive *control*. Experts want the language to be able to make full use of facilities in hardware, including interface facilities.

5.2 Basic, flexible tools

Expert programmers are pragmatists. They tend to rely on the language features they can count on getting. Like smiths, they expect to create most of their own tools. Discovering and remodelling hidden, high-level mechanisms is perceived to entail more work and esoteric knowledge than building utilities up from a low level. Hence; experts tend to prefer generality and flexibility to specific in-built power, although they will make use of appropriate tools when convenient.

Much of the enthusiasm for the newer programming styles (e.g. functional, object oriented) is accounted for as recognition rather than conversion. Programmers receive gladly tools for doing what they intend and have been achieving by less direct routes, i.e. by writing programs in a style other than that of the programming language (e.g. 'structured' assembler, or 'functional' C), or by implementing the desired evaluation mechanisms in whatever language is available. One common pattern is to use some crude but effective language (e.g. C) to build an intermediate language customized for the problem domain (e.g. writing an object-oriented superstructure for C, or a schematics-encoding language in Lisp).

But experts don't view handy facilities as substitutes for basic utility; these same celebrants argue against restriction. Often they achieve the desired combination of expressive power and control by mixing languages (e.g. writing the bulk of a prototype in Prolog, but handling the input/output and graphics in C).

Implicit in arguments for granularity of focus and for flexibility is recognition both that any general purpose language can express any solution expressible in another, and that each language expresses some things well at the expense of others.

5.3 Structural visibility, modularity, and abstraction

The desire for abstraction facilities extends in part from the desire for structural visibility. Experts want to be able to express the solution structure as they perceive it, with support for the conception and amendment of that structure. They want structuring tools both for expressing strategy and for representing data. They want tools for packaging mechanisms into structurally and notionally sensible parcels (or 'modules' with locality (that is, with internal scope and a defined interface to other parcels). Further, they want tools for abstraction, so that they can generalize parcels into program concepts to be used and reasoned about in a way that de-emphasizes their internals.

Reiterated within this sense of abstraction is the argument about granularity of focus. A choice of granularity leaves the constructs within a program abstraction beneath the level of concern. It is critical to distinguish the choosing of focus from invisibility. The distinction is between the option to ignore the detail encompassed within an abstraction, and inaccessibility; i.e. between 'needn't look' and 'cannot see'.

Such powerful structuring tools are not without their dangers. Abstraction building is seductive; forming generic abstract types can lead into confusing excess and even eclipse the problem itself. It is not clear that the languages which offer abstraction tools actually help in choosing – and choosing the extent of – abstractions, or facilitate rebuilding if the abstraction structure fails.

5.4 Accessibility

Experts want access to a complete, workable model of the language, preferably without digging into the implementation. In general, computational knowledge, not machine knowledge, is critical to solution (e.g. an effective knowledge of what a stack is and how it operates need not include the details of the machine instructions and memory locations used to implement it). Unfortunately, computational knowledge is still often unavailable except via hardware or language implementation knowledge. The notion of 'freeing the programmer' from implementation details is misguided when, as in the case of hidden evaluation mechanisms, the language model is not deducible from the language surface, or when, by reading a program, the programmer cannot anticipate what the machine will do.

In the case of the more 'powerful' declarative or specialist languages (e.g. Prolog, SQL), some algorithmic decisions are pre-empted by the language implementation. That is, the language implementation injects algorithmic information not in the program, so that part of the solution is embodied in the language. To that extent, the language contributes directly to the solution, and the programmer requires at least an abstract knowledge of those in-built mechanisms in order to comprehend a solution strategy and its behaviour completely.

In order to code effectively in a language, the programmer needs a fairly accurate model of the language for mapping with the solution model. The idea of separating the language definition and design from the language implementation is a good one – but if a complete, workable model of the language is not accessible via an alternative source, the programmer will delve through the implementation and will exploit whatever 'hacks' were needed to make the language work.

5.5 Predictability and efficiency

Experts need to be able to predict, at least roughly, the behaviour of their programs, including efficiency, side-effects, by-products, and data end-state. Wirth (1974) called this *transparence*, and considered it an aspect of simplicity: 'Do not equate simplicity with lack of structure or limitless generality, but rather with transparence, clarity of purpose, and integrity of concepts'. By transparence, Wirth means that a language feature 'does not imply any unexpected, hidden inefficiencies of implementation' and that 'the basic method of implementation of each feature can be explained independently from all other features in a manner sufficiently precise to give the programmer a good estimate of the computational effort involved'.

Efficiency, far from being an 'implementation detail' separable from program – and language – design, is of primary importance to expert programmers. 'What seems clear is that people settle on fairly efficient languages, regardless of the claims of the high level language designers' (Lesk, 1984).

5.6 Conciseness or simplicity

Experts appreciate an uncluttered notation. As reported earlier, 'ugliness' matters.

6 Summary and conclusion

The list of what expert programmers want in a general-purpose programming language differs from the aspirations of language designers less in detail than in emphasis. The well-foundedness versus utility schism is reflected in the designers' emphasis on correctness and the expert programmers' emphasis on predictability and efficiency. To exaggerate the point: whereas designers often aspire to provide a panacea, users want tools. Although programmers do not object to the advantages of formal manipulation, they will not, in general, accept them at the expense of control and performance. A second important discrepancy is in the interpretations of abstraction: what designers discuss in terms of hiding and restriction, programmers desire as a selection of focus paired with sensible defaults.

What expert programmers seem to want is what Hoare prescribed back in 1973: 'A good programming language should give assistance in expressing not only how the program is to run, but what it is intended to accomplish; and it should enable this to be expressed at various levels, from the overall strategy to the details of coding and data representation'. Or, as Tennant (1981) phrased it: 'In short, an ideal programming language would combine the advantages of machine languages and mathematical notations...'.

Programmers often address inherently complex tasks. If the programming language is itself complex, it may be an obstruction rather than a tool; it may contribute to the problem instead of to the solution. The point remains to provide good tools for people who instruct computers, that is, tools for tool-users.

References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition,* **9**, 422-433.

Adelson, B. (1984). When novices surpass experts: the difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition,* **10**, 483-495.

Adelson, B. and Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering,* **11(11)**, 1351-1360.

Adelson, B., Littman, D., Ehrlich, K., Black, J. and Soloway, E. (1984). Novice-expert differences in software design. *Interact '84, First IFIP Conference on Human-Computer Interaction.* Amsterdam: Elsevier.

Allwood, C.M. (1986). Novices on the computer; a review of the literature. *International Journal of Man-Machine Studies,* **25**, 633-658.

Chase, W.G. and Simon, H.A. (1973). Perception in chess. *Cognitive Psychology,* **4,** 55-81.

Chi, M.T.H., Feltovich, P.J. and Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science,* **5,** 121-152.

Denvir, B.T. (1979). On orthogonality in programming languages. *SIGPLAN Notices,* **14(7),** 18-30.

Dijkstra, E.W. (1976). *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice Hall.

du Boulay, B. and O'Shea, T. (1981). Teaching novices programming. *In* M.J. Coombs and J.L. Alty (Eds), *Computing Skills and the User Interface.* New York: Academic Press.

Flon, L. (1975). On research in structured programming. *SIGPLAN Notices,* **October 1975,** pp. 16-17.

Fung, P. (1987). Novice PROLOG programmers: a consideration of their problems. CITE Technical Report No. 26. Milton Keynes: Open University.

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

Gugerty, L. and Olson, G.M. (1986). Debugging by skilled and novice programmers. *Proceedings CHI'86: Human Factors in Computing Systems.* New York: ACM.

Hoare, C.A.R. (1973). *Hints on programming language design. SIGACT/SIGPLAN Symposium on Principles of Programming Languages,* October 1973.

Hoare, C.A.R. (1981). The emperor's old clothes. *Communications of the ACM,* **24(2),** 75-83.

Jeffries, R., Turner, A.A., Polson, P.G. and Atwood, M.E. (1981). The processes involved in designing software. *In* J.R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum, pp. 255-283.

Kaplan, S., Gruppen, L., Leventhal, L.M. and Board, F. (1986). *The Components of Expertise: A Cross-Disciplinary Review.* Ann Arbor, MI: The University of Michigan.

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

Kowalski, R. (1982). Logic as a computer language. *In* K.L. Clark and S.-A. Tarnlund (Eds), *Logic Programming.* London: Academic Press.

Larkin, J.H. (1983). The role of problem representation in physics. *In* D. Gentner and A.L. Stevens (Eds), *Mental Models.* Hillsdale, NJ: Erlbaum.

Lesk, M. (1984). Programming languages for text and knowledge processing. *In* M. Williams (Ed.), *Annual Review of Information Science and Technology,* vol. 19. Knowledge Industry Publications, pp. 97-128.

McKeithen, K.B., Reitman, J.S., Reuter, H.H. and Hurtle, S.C. (1981). Knowledge organisation and skill differences in computer programmers. *Cognitive Psychology,* **13,** 307-325.

Oman, P.W. and Cook, C.R. (1988). A paradigm for programming style research. *SIG-PLAN Notices,* **23(12),** 69-78.

Petre, M. and Winder, R.L. (1988). Issues governing the suitability of programming languages for programming tasks. *People and Computers IV: Proceedings of HCI'88.* Cambridge: Cambridge University Press.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer and Information Sciences,* **5,** 123-143.

Taylor, J. and du Boulay, B. (1986). Why novices may find learning PROLOG hard. *In* J. Rutkowska and C. Crook (Eds), *The Child and the Computer: Issues for Developmental Psychology.* Chichester: Wiley.

Tennant, R.D. (1981). *Principles of Programming Languages.* Englewood Cliffs, NJ: Prentice-Hall.

Wasserman, A.I. (1975). Issues in programming language design–an overview. *SIGPLAN Notices,* July 1975, pp. 10-12.

Weiser, M. and Shertz, J. (1983). Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies,* **19,** 391-398.

Wirth, N. (1974). On the design of programming languages. *Proceedings of IFIP Congress 74.* Amsterdam: North-Holland, pp. 386-393.

Zemanek, H. (1985). Formal definition the hard way. *In* E.J. Neuhold and G. Chroust (Eds), *Formal Models in Programming.* Amsterdam: Elsevier, pp. 411-417.