This chapter appeared in P. Brna, B. du Boulay and H. Pain (Eds.), *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*. Stamford, CT: Ablex, 1999. (Chapter 1, pp. 7 –27)
This document can be downloaded via links at
http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/

BUILDING AND MANIPULATING COMPLEX INFORMATION STRUCTURES: ISSUES IN PROLOG PROGRAMMING

T. R. G. Green, Computer Based Learning Unit, University of Leeds

An overview

Prolog is notorious: learners find it hard — so hard, compared to Basic or spreadsheets, that many papers have been written on how to make it easier to learn. Here is a remarkable fact: very few of those papers have reported comparisons across languages to look for reasons for Prolog's exceptional difficulty.

In this chapter I present a very brief account of a unified approach to usability of programming languages and the like, and I show how this 'cognitive dimensions of notations' framework might be applied to Prolog. Regrettably, the empirical evidence is quite scanty, but at least my approach generates some ideas to try.

Prolog programs are examples of information structures. Like all information structures, whether they are programs, timetables, spreadsheets, or recipes for cooking, Prolog notation has certain structural properties, and these properties affect 'usability' - that is, what can readily be done and what can only be done with difficulty. Because Prolog has a different notational structure from, say, Pascal, it has different usability properties. Looking ahead, the major usability difficulty I shall pick out is poor 'role-expressiveness', which means that it is difficult for learners to perceive the code in terms of the higher-order cognitive structures that they need to learn. Empirical evidence, such as it is, appears to support this claim.

If my argument is correct, learners would benefit from a learning environment which improved the role-expressiveness by highlighting structures in the code that conform to the desired cognitive structures. According to preliminary evidence, these cognitive structures might be 'schemas', rather than say 'techniques' (these terms are defined below).

The most convincing test of this framework would be to design an environment with its aid and then to observe success. Of course, the framework would have to be supplemented by pedagogic considerations, system considerations and other HCI considerations, so the success would not depend on it alone. Such a test has not been performed, but at least an illustrative example of retrospective analysis can be supplied, albeit in very sketchy form, by applying it to the one attempt I know of to create a specialised learning environment for Prolog by notational means, the 'Ted' editor. This environment met with limited success (Ormerod and Ball, this volume). The cognitive dimensions analysis suggests possible reasons, and thereby suggests ideas for a second generation version of Ted, which might be a basis for empirically testing the framework.

A caveat

The cognitive dimensions approach has a different focus to most of the chapters in this book, which are concerned with the learning processes and the cognitive happenings inside the learner. For example, Josie Taylor uses a discourse model to categorise the 'stories' that learners tell themselves in trying to comprehend Prolog, and Pat Fung describes learners' misconceptions and the importance of a correct model of the underlying machine. The cognitive dimensions framework focuses simply on notational design and has nothing to say about these internal happenings, but there is no conflict between the different approaches, despite their different focus.

My suggestion is simply that the notational difficulties exacerbate the deeper difficulties experienced by learners. A learner who is faced with a program whose behaviour is hard to explain, couched in a notation

that is hard to unscramble, is very likely to invent 'stories' bearing little relation to fact; whereas the same learner, still faced with inexplicable program behaviour but now looking at a notation whose parts are easily discerned, should be more likely to find correct explanations.

Be it noted: I do not claim that the difficulties of learners are only notational, nor that notational manipulations can solve all the difficulties.

STRUCTURE OF THE COGNITIVE DIMENSIONS FRAMEWORK

The framework I shall present is intended to apply to all information artefacts, and so it necessarily has a very abstract nature to allow it to cope with everything from programming languages to timetables and from mobile telephones to drawing applications. Obviously, not all of the framework is likely to be relevant to the design of Prolog notation. I shall therefore describe the framework only in the most general terms, expanding only on those parts that seem to be most relevant to the peculiarities of Prolog. Interested readers can find a more complete account of the framework in other publications (see 'Further Reading').

How the framework works

The cognitive dimensions framework contains three important components: dimensions, activity types, and environment.

Dimensions

The core of the cognitive dimensions framework is a list of about a dozen descriptors ('dimensions') which capture those aspects of an information artefact that affect how people use it. Each descriptor is a high-level term, and ideally these terms would be reasonably comprehensible at first exposure[1]. The concepts that are picked are not necessarily new, but most people have no names for them and never thought about them as a coherent group. By giving names to concepts that are frequently familiar but have not previously been 'lexicalised', the dimensions help to focus attention on important general issues and provide a means to compare design alternatives at a structural level rather than merely by counting features. In this light, the dimensions can be seen as *discussion tools*, helping to raise the level of discourse.

Activities

The dimensions are not, however, evaluative in themselves. Different types of activity demand different 'profiles' to support them. Four main types of activity can be distinguished, for our purposes:

incrementation:	adding a new card to a cardfile; adding a formula to a spreadsheet
transcription:	copying book details to an index card; converting a formula into spreadsheet terms
modification:	changing the index terms in a library catalogue; changing the layout of a spreadsheet; modifying the spreadsheet to compute a different problem
exploratory design:	typographic design; sketching; other cases where the final product cannot be envisaged and has to be 'discovered'.

Environment and medium

It is easy to think of the cognitive dimensions as properties of the notation, and indeed I shall often use that wording, but it ought to be remembered at all times that they are really properties of the notation *plus the working environment*. A notation may be 'viscous' (hard to change) when it is used with pen and paper, yet that same notation may be relatively fluid when it is used in an environment that contains suitable editing tools.

---

[1] It must be admitted that the ideal has not been achieved in the present version. Rather than rename the terms at this stage I am waiting until the framework as a whole can be improved.

For the purposes of this chapter I shall assume that Prolog is to be used with a simple text editor, rather than with a system that is specialised for Prolog. In the same way I shall ignore another important consideration, the *medium* in which the notation is used; once again, a notation that is suitable for one medium, e.g. writing, may be unsuitable for another, such as speech, but we can restrict attention to the written medium for these purposes.

THE DIMENSIONS

Thumbnail definitions will have to suffice for most dimensions (Table 1).

Abstraction	types and availability of abstraction mechanisms
Closeness of mapping	closeness of representation to domain
Consistency	similar semantics are expressed in similar syntactic forms
Diffuseness	verbosity of language
Error-proneness	notation invites mistakes
Hard mental operations	high demand on cognitive resources
Hidden dependencies	important links between entities are not visible
Premature commitment	constraints on the order of doing things
Progressive evaluation	work-to-date can be checked at any time
Provisionality	degree of commitment to actions or marks
Role-expressiveness	the purpose of a component is readily inferred
Secondary notation	extra information in means other than formal syntax
Viscosity	resistance to change
Visibility	ability to view components easily

Table 1: the cognitive dimensions in their present state.

For the present discussion, four of the dimensions seem to be particularly relevant: viscosity, closeness of mapping, role-expressiveness and hidden dependencies. The other dimensions will be ignored, but at some cost, because a proper evaluation of a notation requires consideration of them all.

Viscosity

Resistance to change: the cost of making small changes. Viscosity in hydrodynamics means resistance to local change – deforming a viscous substance, e.g. syrup, is harder than deforming a fluid one, e.g. water. It means the same in this context. As a working definition, a viscous system is one where a single goal-related operation on the information structure requires an undue number of individual actions, possibly not goal-related.

Repetition viscosity: a single goal-related operation on the information structure (one change 'in the head') requires an undue number of individual actions. Example: manually changing US spelling to UK spelling throughout a long document.

Knock-on viscosity: one change 'in the head' entails further actions to restore consistency. Example: inserting a new figure into a document creates a need to update all later figure numbers, plus their cross-references within the text; also the list of figures and the index.

The ‘goal-related operation’ is, of course, at the level that is appropriate for the notation we are considering. It can also be thought of as a function of the misfit between a ‘cognitive semantics’ (see below) and the notation.

Viscosity is most definitely a property of the system as a whole, and it can be reduced by specialised editors (at the cost of certain trade-offs, usually with abstraction level).

Viscosity may be very different for different operations, sometimes making it hard to give a meaningful average figure: however, the crucial measure is probably the most viscous operation that has to be performed fairly often.

Closeness of Mapping

To represent a domain in some external form, the concepts of that domain must be somehow transformed into the external representation. For example, the notes of the Western musical scale must be transformed into marks on the musical staff to notate them, or transformed into the keys of the piano in order to play them. Here the transformation is from the pitch of the note into a scale of height on the page or left-right position on the piano. (The transformation is a good deal more subtle than that, in fact, but the subtleties do not affect the main point.) If the pitch were instead transformed into a representation where alternate notes were represented or sounded in physically different ways (e.g. C with the left hand, D with the right, etc) the transformation appears to be more complex and the mapping less ‘close’. Such a representation would be harder to get familiar with. The curious reader may like to know that precisely such a system (C with one hand, D with the other) is a characteristic of the fingering of the ‘English-system’ concertina, devised by a very inventive physicist (Wheatstone) who probably gave little thought to closeness of mapping.

This ‘distance’ between a domain ‘in-the-head’ and its external representation is a concept that is easier to illustrate than to define, partly because there is rather little agreement about what is in the head, or even how to talk about it. Considerable analysis would be required to take it from the hand-waving level. Nevertheless, useful examples have been given by, among others, Norman (1988).

Accepting the hand-waving quality of this dimension, for the purposes of discussion, we can see that to estimate closeness of mapping we need to know the cognitive representation of the domain. In the foregoing, I tacitly assumed that the cognitive representation of pitch is a simple monotonic dimension. But suppose it isn’t? For example, experienced Western musicians place notes in a chord structure as well as on a simple pitch dimension. An instrument or a notation that reflected the chord structure would potentially have a closer mapping than one that only reflected the pitch dimension.

Role-expressiveness

This dimension describes the ease with which the notation can be broken into its component parts, when being read, and the ease of picking out the relationships between those parts.

There is a relationship between closeness of mapping and role-expressiveness. Closeness of mapping is an indication of whether the user’s concepts are readily transformed into the concepts of the notation. Role expressiveness is an indication of how easy it is to perceive (or create) *structures* built from those components.

For a considerable time electronics dealt in individual components such as transistors, rather than integrated circuits. During that stage, the notation of electronics had good closeness-of-mapping, since distinctive individual components were mapped onto distinctive individual symbols, wiring was mapped onto connecting lines, and so on; but the role-expressiveness was quite poor, and only an experienced user could look at a radio circuit and quickly pick out the parts that deal with different stages of the process (detecting the signal, the first amplification stage, and so on).

Role expressiveness can be supplemented by ‘secondary notation’, in which users can adopt some informal means to indicate that particular components are related, using some indication which is not part of the formal notation. In the circuit diagram, related components could be placed near each other to help to indicate the structures that they formed; this was quite independent of the real-world physical position of the components on the built circuit.

In other examples, related components might be printed in the same colour, or related by some other means.

Hidden Dependencies

A hidden dependency is a relationship between two components such that one of them is dependent on the other, but that the dependency is not fully visible. In particular, the one-way pointer, where A points to B but B does not contain a back-pointer to A. Example: HTML links – if your page is linked to someone else's, how will you know if and when that page is moved, changed, or deleted?

A dependency can be hidden in two different ways, by being one-way or by being local. The HTML link is a one-way pointer, so it is invisible from the other end – I cannot find out what links to my page have been created, unless I use a search engine. As it happens, HTML links are also local: they point to the next page on a chain, but say nothing about what lies beyond.

Spreadsheet links are just the same in structure. Here their local nature is more important, because I may well want to know which cells use the value in a given cell, whether directly or indirectly.

SUPPORTING EXPLORATORY DESIGN AND OTHER ACTIVITIES

Each of the four activity types mentioned above has its own desirable profile of dimensions. Again, lack of space forces a very terse list. See Green and Petre (1996) or Green and Blackwell (1998) for a more considered account.

	transcription	*incrementation*	*modification*	*exploration*
viscosity	acceptable	acceptable	harmful	harmful
role-expressiveness	desirable	desirable	essential	essential
hidden dependencies	acceptable	acceptable	harmful	acceptable for small tasks
premature commitment	harmful	harmful	harmful	harmful
abstraction barrier	harmful	harmful	harmful	harmful
abstraction hunger	useful	useful (?)	useful	harmful
secondary notation	useful (?)	–	v. useful	?
visibility / juxtaposability	not vital	not vital	important	important

Table 2 Suggested relationship between activities types and cognitive dimensions (from Green and Blackwell, 1998)

THE NOTATION OF PROLOG

It is now time to apply the cognitive dimensions framework to the notational structure of Prolog. This process must necessarily be a cursory and mostly off-the-cuff sketch: we have neither the evidence, nor the space, for a detailed analysis.

Activity

To start with, what kind of activity is taking place? It might be thought that programming is an incremental activity, since at first sight it appears that all the programmer needs to do is to keep adding some more program code to what has already been written. Perhaps not surprisingly, that is not a good picture of what really takes place. Even highly experienced Prolog programmers develop their code gradually, with modifications such as adding another argument to a predicate (Green, Bellamy & Parker, 1987).

Green et al. recorded the activity of highly experienced programmers solving very simple problems in their usual working language (Basic, Pascal, or Prolog – later data, unpublished, also covered C and Lisp). The development of these programs showed shifts of focus in all languages, such that code that had already been written was later revised. Subsequent studies using the same basic methodology, by Davies (1991) and Ormerod and Ball (1993) confirmed this finding.

There has been some discussion about whether and how to characterise the control of programming behaviour. The balance of evidence has shifted from an early picture of 'opportunistic' programming towards other views such as the 'children-first' view proposed by Ormerod and Ball (1993). For present purposes, I shall simply characterise the behaviour as exploratory activity, not intending to make any hypothesis about the nature of the underlying cognitive processes but certainly intending to convey the fact that in this type of activity, programmers sometimes make localised changes of detail and sometimes make big changes that affect much of the program structure.

As Table 2 shows, the successful support of exploratory activity places certain stringent demands on the notation and its working environment: low viscosity and high role-expressiveness are essential. These are problematic in Prolog, I believe.

Dimensions

Viscosity of Prolog

Viscosity has to be considered for individual tasks, so a basket of representative tasks is necessary to create a representative overall measure. Consider five tasks: adding a new predicate to an existing program; adding a new 'chunk' to a program; changing the arity of an existing predicate already used in the program; changing a data structure; and changing the method of attempting a solution. The first two tasks characterise simple incrementation, the others characterise modification.

Adding a new predicate is easy. Because Prolog has no declarations, nor any other similar dictionary-style structure, the predicate can simply be inserted into the program, wherever desired between other predicates, and it can invoke any other predicates in the program.

Pascal makes much more fuss. Adding a new procedure will require declarations, and often will require careful placement in the identifier hierarchy to allow the new procedure to call whichever existing procedures it needs; sometimes, the program structure may need to be altered to make the requisite identifiers visible, an example of heavy knock-on viscosity.

Prolog is therefore exceptionally effortless for simple incrementation.

Adding a new 'chunk' to a Prolog program is not difficult but has some unusual properties. A simple illustration, from Green (1990), showed that for a plausible example, structured Basic has less viscosity than Prolog. (Siddiqi and Roast (1997) offered a Prolog version that was less viscous than Green's, but even so, it was more viscous than the Basic program.) The example in question starts with a program to compute the average of a series of integers and modifies it to a program that also computes the 'filtered average' of the series, where the filtered average is the average of all non-zero elements.

```
(a) BASIC: initial version of 'average'

 1: Count = 0
 2: Sum = 0
 3: READ Item
 4: WHILE Item <> 9999
 5:     Sum = Sum + Item
 6:     Count = Count + 1
 7:     READ Item
 8: WEND
 9: PRINT Sum / Count
```

```
(b) BASIC: as (a) but with a filter

 1: Count = 0
 2: Sum = 0
 3: FCount = 0
 3: READ Item
 4: WHILE Item <> 9999
 5:     Sum = Sum + Item
 6:     Count = Count + 1
 7:     IF Item <> 0 THEN FCount = FCount + 1
 8:     READ Item
 9: WEND
10: PRINT Sum / Count
11: PRINT Sum / FCount
```

```
(c) PROLOG: initial version of 'average'

average1(L, Result) :-
   sumAndCount(L, Sum LL),
   Result is Sum div LL.

sumAndCount( [], 0, 0).
sumAndCount( [H|T], Sum, L) :-
   sumAndCount(T, SubSum, SubL),
   Sum is SubSum + H,
   L is subL + 1.
```

```
(d) PROLOG: as (c) but with a filter

average2(L, Result, FResult) :-
   sumAndCountF(L, Sum LL, FLL),
   Result is Sum div LL.

sumAndCountF( [], 0, 0, 0).
sumAndCountF( [0|T], Sum, L, FL) :-
   sumAndCountF(T, SubSum, SubL, FL),
   Sum is SubSum + H,
   L is subL + 1.
sumAndCountF( [H|T], Sum, L, FL) :-
   H > 0,
   sumAndCountF(T, SubSum, SubL, SFL),
   Sum is SubSum + H,
   L is subL + 1,
   FL is SubFL + 1.
```

Figure 1 Introducing a filtered average is more viscous in Prolog than in Basic. It can also be seen that the elements introduced (shown underlined) are not only more numerous in Prolog, but are widely scattered, increasing the problems of comprehending the structure of the program and lowering the role-expressiveness.

Changing the arity of an existing predicate creates repetition viscosity. Each case where that predicate is invoked as a goal must be individually updated. This operation is no more arduous than during Pascal or Lisp

development, but it may be more frequent in Prolog. Examples of coding development from Green et al and from Ormerod and Ball show that this task certainly occurs during Prolog development. In the case of learners, it seems very likely that they will overlook the need for an accumulator or for an output argument and then have to revise their program; empirical data sorely needed ....

Changing a data structure creates very substantial viscosity effects, but probably no different from those in Pascal or Lisp. The sort of case that I believe typifies the development of programs by learners is where the program processes a list of records each containing a fixed number of atomic items of information, and then it appears that one of those items should itself be a list. (For instance, the record originally contains name, email address, and telephone number, but it is then discovered that some people have several telephone numbers.) Every access to that data structure must then be recoded. This modification, like the previous one of changing arity, is quite often an indication that the programmer was unable to analyse the problem sufficiently in advance of starting to code, and therefore had to make an *early commitment* that turned out to be a wrong guess.

Changing the chosen method may have extensive knock-on consequences. Here, what I imagine is that the programmer decides to use an 'arithmetic-first' method rather than an 'arithmetic-last', or some similar change. This is clearly relevant for exploratory program development. Unfortunately we have no comparative data at present, although Gray and Anderson (1987) report an analysis of change episodes in the development of Lisp programs by novice programmers; it would be of considerable interest to create a similar analysis for Prolog (and indeed for other languages).

Closeness of Mapping

It is hard to gauge the closeness of mapping of a programming language because programmers' mental representations of programs are not well understood. Historically, it has been proposed that Pascal novices think in terms of 'plans', groups of statements that typically consume some data, operate upon on it in some characteristic way, and produce results for use by another plan (Soloway and Ehrlich, 1984; Rist, 1986); there is some empirical evidence to support this proposal, although the 'plan' is at best only one of the possible cognitive structures that programmers may employ. (E.g. Pennington, 1987, demonstrated that text structures were also relevant; Gilmore and Green, 1988, demonstrated that control-flow structures were also important for Pascal novices; Bellamy and Gilmore, 1990, suggest that the apparent structure depends on the task; Burkhardt et al., 1997, distinguish between different types of text-derived structures, the textbase (from reading-to-recall) and the situation model (from reading-to-do); Green and Navarro, 1995, showed that visual/spatial components are sometimes relevant; and so on.)

Very little comparative research has been reported on the cognitive components developed by novice Prolog programmers. There appear to be three lines of research. One line has investigated the temporal structure of comprehension – the order in which Prolog programmers seek for data flow, control flow, operations, etc. (Bergantz and Hassell, 1991; Good et al., 1997) A second line has looked for descriptions of a 'cognitive semantics' that could be related to one of the various models of formal semantics (Andrews, 1997).

The third line of research, which is the most relevant in this context, has sought to determine empirically what components in Prolog might have the same function as 'plans' in Pascal. In a very relevant study, Romero (1998) compared recall rates for different hypothetical structures. The 'recall rate' is simply a count of how many instances of a given type of structure were *completely* recalled. The argument was that if structure A corresponds well to a cognitive structure while structure B corresponds badly, then the recall of programmers will exhibit many complete examples of type A; parts of type B structures might also be recalled, if they overlap with type A, but these will just be fragments that A and B share – there will be very few examples of complete type B structures being recalled.

Romero gave Prolog programmers 3 minutes to study a program and then 5 minutes to recall it, then analysed the material they recalled to see whether it was best described in terms of 4 different hypothetical cognitive structures. The structures compared were taken from suggestions in the literature, named by their various originators 'techniques', 'schemas', 'focal lines', and 'critical control flow points'.

Techniques (Bowles and Brna, this volume) are generalised patterns of arguments and uses that can be combined together to form complete predicates. Romero's example is the *after* technique, in which a process `g()` takes place after a recursive call `p()`:

```
p(X) :-
   p(Y),
   g(Y, X).
```

The *after* technique

Because techniques are components of predicates, rather complete predicates in themselves, they are not always easy to spot in Prolog code. Here is a simple predicate to find the length of a list, this time using the *before* technique:

```
length([], L, L).
length([H|T], L0, L):-
   L1 is L0 + 1,
   length(T, L1, L).
```

Schemas (Gegg-Harrison, 1991) are slightly higher-level than techniques; they appear to be complete Prolog predicates in skeletal form.

```
schema_C([E|T], E,  ...).        succeed when E is found
schema_C([H|T, E, ...):-
   <not(EH)>,                    optionally check E has not been found
   <pre_pred(..., H, ...)>,      optional pre-process
   schema_C(T, E, ...),          mandatory recursive call looking for E
   <post_pred(..., H, ...)>.     optional post-process
```

A schema for finding an element in a list

Focal lines and *critical control-flow points* , the remaining candidates for cognitive components, are easier to describe. Focal lines (a term taken from earlier research on Pascal 'plans') are the clauses in which a key operation is performed; in a counting program, it would be the clause in which the counter is incremented. Control-flow points are those points where recursion is invoked or stopped.

When each attempted recall was analysed in each of these four ways, Romero found that the schema structure was much the best match to the material recalled by experts, and was also (but less strongly) the best match to the material recalled by novices: for non-programmers, there were no differences between the four structures.

No one experiment should be taken as final, but the first indications are that schemas are a good model of at least some aspects of the cognitive structures of experienced Prolog programmers.

Whatever model emerges after further research, the closeness of mapping will depend on the match between that model and the problem domain. The focal line of the plan structure seems to have some degree of closeness of mapping, since one can see how a programmer could reason from the problem statement to the need for a particular focal line. The schema structure may also have some degree of closeness of mapping, for the same reason. The technique structure, being far more general, appears to have a more distant mapping: that is, from a problem statement, it is not always easy to see which techniques to use.

Role-expressiveness

Assuming that programs are written by translating cognitive structures such as techniques or schemas into code, it follows that part of the process of comprehending a program is parsing it back into the original cognitive components.

Anecdotal evidence suggests that Prolog is much harder to decompose than Pascal: according to at least one theory of parsing, that is exactly what one would expect. Prolog, like assembly code, allows many different program structures to be built by combining a very small number of notational elements in different ways. There are no lexical distinctions between these elements, merely different rearrangements of the same symbols. Pascal, on the other hand, has plentiful lexical cues which help to distinguish different program structures from each other.

The importance of lexical cues in human natural language parsing was postulated many years ago, and Bratley et al. (1967) even demonstrated a parser that could achieve surprising degrees of success in parsing English using no information except the 'closed class morphemes' such as *-ly*, *-s*, *the* etc. Green (1979) demonstrated that learning miniature artificial languages was very much easier when those languages contained effective lexical indicators cueing the phrase structure of the 'sentences.

Because of the lack of lexical cues, different Prolog programs can appear very similar on the page: differences are subtle, though important. Consider, for example, the two fragments in Figure 2 (from Brna, 1995): one uses the technique of building structure in the clause head, while the other uses an accumulator. To distinguish them it is necessary to see how the arguments are used. (Don't think you can just count the arguments, because in real examples there will be other arguments dealing with other aspects of the processing.) To see how the arguments are used, we have to see where they occur in the body of the rule, build up a structure, and recognise that structure as an example of a particular schema. Pascal, in contrast, contains important lexical indicators, such as *while*, *for*, etc., which immediately distinguish between different syntactic constructs and which give big clues to distinguishing between different schemas.

```
tripleA([], []).
tripleA([H1|T1], [H2|T2]):-
      H2 is 3*H1,
      tripleA(T1, T2).


tripleB([], Y, Y).
tripleB([H1|T1], X, Y):-
      H2 is 3*H1,
      tripleB(T1,[H2|X],Y).
```

Figure 2 Two Prolog fragments using different methods for very similar purposes.
The difference is not strongly apparent.

Compare the Pascal and Prolog versions of a count fragment(Figure 3). In each case we have some distinctive clues (the zero and the expression `N=N+1` or its Prolog equivalent) but the process that is being applied is more clearly indicated in the Pascal in two ways: the `while`, obviously enough, and the indentation, which is a strong help. Prolog, of course, has indentation, but it is less useful in picking out control structures.

(This sort of comparison is never clearcut. The most convenient data structure for Prolog is the list, which lends itself readily to algorithms that consume each member; lists are clumsy in Pascal, so I have used an abstract form of expression.)

```
N := 0;
while expression true do
   begin
   N := N+1;
   compute new expression
   end
```

```
count([],0).
count([A|B], N) :-
      count(B, N1),
      N is N1 + 1.
```

Figure 3 Pascal (above) versus Prolog (below).

Green and Borning (1990) applied a plausible model of human natural-language parsing to the parsing of Prolog and Pascal, using the counting fragment as their simplest example. Obviously, they were not concerned with whether Prolog sentences were syntactically correct, but with extracting chunks corresponding to postulated cognitive structures; in the absence of empirical evidence at that time, they adopted 'program

plans' as a reasonable model of cognitive structure. The parsing model they adopted was unification parsing with simulated annealing (Kempen and Vosse, 1989). In this model, the grammar describes fragments with features, and parsing proceeds by unifying fragments with the text and then unifying fragments with each other. Successful parses occur when all the fragments can be unified into a whole.

In this model, there can be many fragments that are candidates for unification with each other. If the wrong ones join up, the parsing will fail. The more lexical cues are present, and the more powerful they are, the better the success rate. When applied to Prolog, with its relative dearth of lexical cues, the success rate was much less than when applied to equivalent programs in Pascal. Figure 4 shows how complex the Prolog parse tree is even for the simple counter example illustrated above.

Unpublished results showed that a simple Pascal program was very much easier to parse than the equivalent Prolog program. Worse, when the complexity of the program was increased by adding extra components, the difficulty of parsing the Prolog version increased much faster than the difficulty of parsing the Pascal version.

Hidden dependencies

The structure of Prolog contains many hidden dependencies. For example, the interpretation of the main clause of a predicate will depend on how the base case is handled (see Figure 3). On the whole, however, the dependencies are short-range and tend to be stereotyped and predictable; moreoever, it is easy to know where to search for them. The interpretation of Prolog code seems to suffer less from long-distance surprises than, say, C.

The profile of Prolog

A proper analysis of Prolog as a notational structure would need to consider all the remaining cognitive dimensions; moreover, it would be based on serious evidence rather than the shreds that are all that can at present be offered. So the conclusions are at best tentative.

Within those limitations, what we have is a notation with very low role-expressiveness, especially for the way that data structures are manipulated. Obviously, learners need to discover what are the relevant chunks, need to discover what their domain relevance is, need to be able to recognise them in a program, need to be able to correct inappropriate or malformed chunks, and need to able to revise the structure of the program. When role-expressiveness is low, learners will find it difficult to analyse their own or anyone else's programs.

Moreover, I would contend that although Prolog has low viscosity on some tasks, it has quite high viscosity for the important task of rebuilding a program using different schemas. Thus, simple experimentation using trial and error becomes laborious and error-prone.

On the plus side, which has not been mentioned, Prolog avoids damaging hidden dependencies, has good visibility, and avoids enforced lookahead and premature commitment reasonably well.

NOTATIONAL POSSIBILITIES?

If this analysis is correct, the problems of learners might be improved by changing the notation or its support environment, to reduce the viscosity and increase the role-expressiveness.

If the chief weakness of the Prolog notation is its poor role-expressiveness, one possibility is to use secondary notation to unify components that form part of one larger component. For example, Gilmore and Green (1988) showed that colour could help their subjects identify plan structures in Pascal. Support for using perceptual features to assist parsing comes from work by also from work by Payne, Sime and Green (1984) who showed that reasoning about complicated string-matching patterns was assisted by quite simple typographical cues.

The Prolog-parsing experiments of Green and Borning, described above, were extended by adding virtual colour as an additional feature. A unification parser tries to join components with features that will unify, so by marking components of the same plan structure as having the same colour the parser was encouraged to join them together. This brought the Prolog difficulty level down almost to the level of Pascal.

The unification parser has proved to be a good but not perfect model of human natural language parsing. Little is known, however, about its accuracy when artificial notations are the target. If we accept the results at face value, then adding an automatic colouring method to a Prolog environment would be a very simple and possibly effective approach to making the learner's problems fewer. One could imagine that each of a set of schemas or templates was assigned its unique colour, with unrecognised components in some other colour; the learner, reading a program, would then be able to readily see what approach the program was using, and when writing a program would be able to see whether the intended schema had been successfully implemented.

REDESIGNING THE ENVIRONMENT: THE NOTATION OF 'Ted'

Since the cognitive dimensions analysis offers assertions about the problematic aspects of the Prolog notation as used in its conventional environment, it is possible to test the framework by gauging the success of a redesigned environment. The Ted editor described by Ormerod and Ball (this volume) is one such redesign. While a better test of the framework would be to predict the result in advance, at least some test is possible by considering the results they report, working with the benefit of hindsight.

Instead of changing the displayed notation, the Ted designers chose to change the *input* notation, using a set of seven 'techniques'. Their example is generating the code for the finding the length of a list (see Figure 1 of their chapter).

To generate that code, the learner combined two 'techniques' (see above). One technique was the 'List_head' technique, specifying that one argument of the predicate was a list whose tail was recursively used in the body; the other technique was 'arithmetic_after', specifying a head variable, `L`, a recursion variable, `L1`, and a subgoal, `L is L1 + 1`.

Pedagogically, the argument was no doubt that building programs from a small repertoire of techniques, combinable in only a few ways, was an easier learning problem than building them from a larger repertoire of lexemes combinable in a great many ways.

The Ted environment showed two outputs. One was the assembled code; the other was a history of the construction, shown as a tree.

It turned out that the use of techniques did offer some improvement in learning but the improvement offered by using techniques instead of 'raw' Prolog seemed to fade away on the harder problems. The Ted editor gave no overall benefit over the techniques alone; results were mixed, suggesting perhaps that there was some problem that was interfering with its potential effectiveness.

Can we, working with the benefit of hindsight, understand why this promising-looking idea was not as successful as its designers hoped? Let's very briefly consider how the cognitive dimensions framework applies to this notation (and if the analysis of standard Prolog was tentative, the following must be even more so, since the paper gives very little more detail than has already been mentioned).

Viscosity

Programmers using Ted had only one way to revise their code: they had to edit the history tree. This was presumably to ensure that the code generated was always built from the technique library and could therefore be processed by the Ted environment (whereas if arbitrary changes to the code were allowed, the results might have been impossible to interpret in terms of techniques).

Having to decided that program revision would always require the history tree to be edited, the developers were no doubt faced with a hard choice; arbitrary editing of the history tree would be difficult to implement and might also be difficult for the learner to comprehend. Be that as it may, they took the easier road of insisting on 'Destructive Undo', a restricted form of editing: to change the program, learners had to step back through the history tree to where the offending component had been added, and then delete that component, *thereby at the same deleting every subsequent editing operation*.

Evidently the knock-on viscosity could be extremely high if the learner needed to change a choice that had been made far back in the tree. To avoid this fate (premature commitment), learners no doubt attempted to look far ahead, which may or may not have been pedagogically useful.

It would seem that the Ted designers thought of the coding activity as incrementation, rather than as exploration; an easy mistake, but with serious consequences.

Role-expressiveness
Since the Ted editor displayed standard Prolog as its output code, the role-expressiveness was not improved. The role-expressiveness of the techniques was also very poor, because the output code does not indicate which part of the code is associated with which technique. On the other hand, the difficulty of parsing the output code into the seven techniques should have been less than the difficulty of parsing the same code

Thus, overall the role-expressiveness might been made *worse*, not better.

(It would have been possible to make an empirical test of my conjecture by asking learners to perform tasks of translating between code and techniques, preferably in both directions, to discover whether they understood how the two were related; or by presenting code and asking how to generate it using Ted, but no such tests were reported.)

Secondary notation
In conventional Prolog, the opportunities for secondary notation are severely limited – comments and indenting, nothing more – but it seems that in Ted, even those possibilities are removed: learners were therefore unable to document the reasons for their editing choices.

Closeness of mapping
The techniques do not seem to offer any improvement in terms of closeness of mapping. It is difficult to see how to make it easy for learners to proceed from the problem statement to the use of the 'arithmetic_after' technique, for example. If the choice had been to use schemas, it might have been easier to associate the schema as a whole with the type of problem to be solved.

Error-proneness
The Ted editor removed many of the trivial problems of conventional Prolog – mismatched parentheses, inconsistent spelling, unfinished comments, and so on. Trivial though they may be, these add to learners' problems and in the case of inconsistent spelling they can be very difficult to catch.

The profile of Ted
If my cursory analysis is correct (and it must be emphasized again that it *is* cursory, and is not based on experience with the actual tool), then Ted seems to have taken a wrong path. The problems that were identified in conventional Prolog, using the cognitive dimensions framework, were poor role-expressiveness and high viscosity. Ted seems to have made no contribution to solving those problems.

With hindsight, therefore, it seems not too surprising that the results were disappointing.

HOW TO DESIGN PROGRAMMING SUPPORT FOR LEARNERS

The cognitive dimensions framework needs to be further developed and it needs to be tested more rigorously than has yet been possible. It would be extremely foolish to put too much trust in it at present. Nevertheless it can offer analyses that are provocative and possibly useful.

The way forward seems to be to compare notational properties of different languages and relate them to the performance of programmers, of every skill level, performing tasks that fall into each of the activity types that were listed above. If the framework survives that stage without serious modification, it will acquire more credibility.

Assuming for the nonce that the analyses above are adequate, then it seems that programmers at all levels of experience could benefit from an environment that reduced the viscosity of Prolog. Unfortunately, modifications to notations are invariably victims of the 'no free lunch' principle: that is, most of the ways to reduce viscosity incur costs on other dimensions. The usual 'design manoeuvre' (Green and Blackwell, 1998) is to create abstractions in order to reduce viscosity; but defining and editing abstractions is not easy, and it requires an 'abstraction manager' to look after them, thus increasing the learning load. This would be a route that required careful planning.

In the meantime, it would be interesting to make the very cheap experiment of simply increasing the visibility of 'schema' constructs in Prolog programs, as suggested above, to see what happens when role-expressiveness is improved.

FURTHER READING

The cognitive dimensions approach was first proposed by Green (1989). It was applied to visual programming languages by Green and Petre (1996). Modugno et al (1994) and Yang et al. (1998) describe how various programming environments were redesigned following analysis. Currently the most complete account of its applicability outside programming is a tutorial by Green and Blackwell (1998), available as a web-downloadable document.

REFERENCES

Andrews, J. H. (1997) Cognitive and formal semantics of Prolog. Unpublished MS ; Dept of Computing Science, Simon Fraser University, British Columbia, Canada.

Bellamy, R. K. E. and Gilmore, D. J. (1990) Programming plans: internal or external structures. In K. J. Gilhooly, M. T. G. Keane, R. H. Logie and G. Erdos (Eds.) *Lines of Thinking: Reflections on the Psychology of Thought*. (Vol 1.) Wiley. 59-71

Bergantz, D. and Hassell, J. (1991) Information relationships in Prolog programs: how do programmers comprehend functionality? *Int. J. Man-Machine Studies*, 35,313-328.

Bowles, A. and Brna, P. (This volume). Introductory Prolog: A suitable selection of programming techniques. In P. Brna, B. du Boulay and H. Pain (Eds.), *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*. Stamford, CT: Ablex

Bratley, P., Dewar, H. and Thorne, J. P. (1967) Recognition of syntactic structure by computer. *Nature*, Vol. 216, No. 5119, 969-973.

Brna, P. (1995) *Prolog programming, a first course*. Unpublished MS. Computer-Based Learning Unit, University of Leeds, UK.

Burkhardt, J.-M., Détienne, F. and Wiedenbeck, S. (1997) Mental representations cnstructed by experts and novices in object-oriented program comprehension. In Howard S., Hammond J. and Lindgaard J. (Eds)*Human-Computer Interaction - INTERACT '97*. Sydney: Chapman & Hall. 339-346

Davies, S. P. (1991) Characterising the program design activity: neither strictly top-down nor globally opportunistic. *Behaviour and Information Technology*, 10 (3), 173-190.

Gegg-Harrison, T. S. (1991) Learning Prolog in a schema-based environment. *Instructional Science*, 20, 173-192.

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

Good, J., Brna, P. and Cox, R. (1997). Program comprehension and novices: does programming language make a difference? Technical Report 97 / 10, Computer Based Learning Unit, University of Leeds.

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. 187-197

Green, T. R. G. (1979). The necessity of syntax markers: Two experiments with artificial languages. *Journal of Verbal Learning and Verbal Behavior*, 18, 481-496.

Green, T. R. G. (1989) Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay (Eds.) *People and Computers V*. Cambridge University Press. 443-460

Green, T. R. G. (1990) The cognitive dimension of viscosity: a sticky problem for HCI. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.) *Human-Computer Interaction – INTERACT '90*. Amsterdam: Elsevier.79-86

Green, T. R. G. , Bellamy, R. K. E. & Parker, J. M. (1987). Parsing and Gnisrap: a model of device use. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex. 132-146

Green, T. R. G. and Blackwell, A. F. (1998) Cognitive dimensions of information artefacts: a tutorial. Web-based document. URL: `http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/`

Green, T. R. G. and Borning, A. (1990) The Generalized Unification Parser: modelling the parsing of notations. In D. Diaper, D. Gilmore, G. Cockton and B. Shackel (Eds.) *Human-Computer Interaction – INTERACT '90*. Amsterdam : Elsevier. 951-957.

Green, T. R. G. and Navarro, R. (1995) Programming plans, imagery, and visual programming. In Nordby, K., Helmersen, P. H., Gilmore, D. J, and Arnesen, S. (1995) *Human-Computer Interaction - INTERACT-95.* London: Chapman and Hall (pp. 139-144).

Green, T. R. G. and Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing*, 7, 131-174.

Kempen, G. and Vosse, T. (1989) Incremental syntactic tree formation in human sentence processing: an interactive architecture based on activation decay and simulated annealing. *Connection Science*, 1, 273-290.

Modugno, F. M., Green, T. R. G. and Myers, B. (1994) Visual programming in a visual domain: a case study of cognitive dimensions. In G. Cockton, S. W. Draper and G. R. S. Weir (Eds.) *People and Computers IX*. Cambridge University Press. 91-108.

Norman, D. A. (1988) *The Psychology of Everyday Things.* New York: Basic Books.

Ormerod, T. C. and Ball, L. J. (this volume)) An empirical evaluation of Ted, a techniques editor. for Prolog programming. Also in W. D. Gray and D. A. Boehm-Davis (Eds.), *Empirical Studies of Programmers: Sixth Workshop*. Newark, N. J. : Ablex, 1996.

Ormerod, T. C. and Ball, L. J. (1993) Does programming knowledge or design strategy determine shifts of focus in Prolog programming? In Cook, C. R., Scholtz, J. C. and Spohrer, J. C. (Eds.), *Empirical Studies of Programmers: Fifth Workshop*. Ablex.

Payne, S. J., Sime, M. E. and Green, T. R. G., (1984) Perceptual structure cueing in a simple command language. *Int. J. 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.

Rist, R. S. (1986) Plans in programming: definition, demonstration, and development. In E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers*. Norwood, NJ: Ablex. 28-47.

Romero, P. (1998) Focal structures in Prolog. Unpublished MS, School of Cognitive Science, University of Sussex, UK.

Siddiqi, J. I. and Roast, C. R. (1997) Viscosity as a metaphor for measuring modifiability. *IEE Proc. Software Engineering*, 144(4: August), 215-223.

Soloway, E. and Ehrlich, K. (1984) Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10, 595-609.

Yang, S., Burnett, M. M., DeKoven, E. and Zloof, M. (1998) Representation design benchmarks: a design-time aid for VPL navigable static representations. *Journal of Visual Languages and Computing*, 8 (5/6), 563-599.

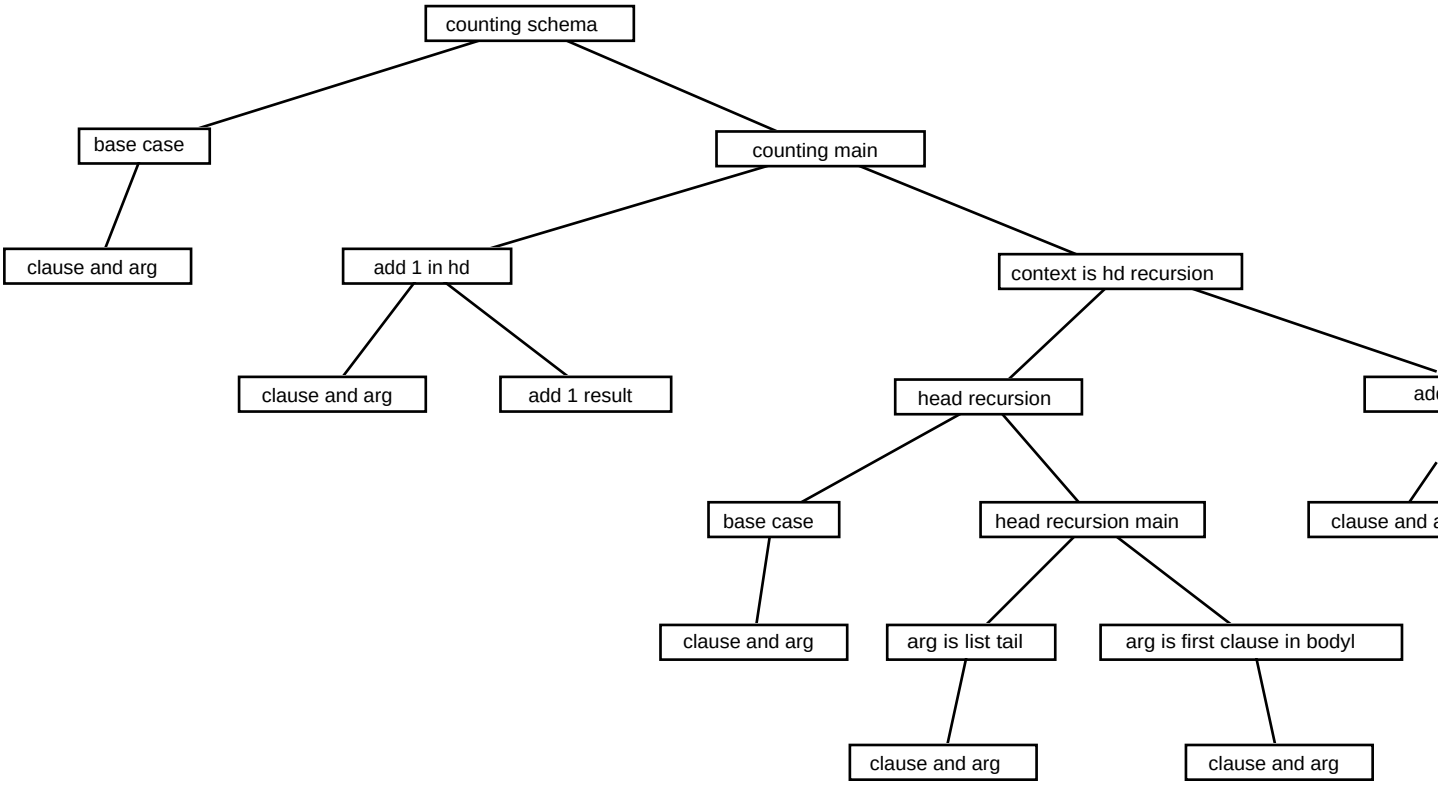

Figure 4 PROLOG TREE Parse tree for a very simple Prolog predicate, as determined by Green and Borning's Generalised Unification Parser. Among the unification features omitted for simplicity are those that ensure that each instance of 'clause and arg' has unified with the appropriate argument.