Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension

Judith Good

Ph.D.
University of Edinburgh
1999

Abstract

This thesis describes research into the role of various factors in novice program comprehension, including the underlying programming paradigm, the representational features of the programming language, and the various types of information which can be derived from the program.

The main postulate of the thesis is that there is no unique method for understanding programs, and that program comprehension will be influenced by, among other things, the way in which programs are represented, both semantically and syntactically. This idea has implications for the learning of programming, particularly in terms of how these concepts should be embodied.

The thesis is focused around three empirical studies. The first study, based on the so-called 'information types' studies, challenged the idea that program comprehension is an invariant process over languages, and suggested that programming language will have a differential effect on comprehension, as evidenced by the types of information which novices are able to extract from a program. Despite the use of a markedly different language from earlier studies, the results were broadly similar. However, it was suggested that there are other factors additional to programming notation which intervene in the comprehension process, and which cannot be discounted. Furthermore, the study highlighted the need to tie the hypotheses about information extraction more closely to the programming paradigm.

The second study introduced a graphical component into the investigation, and looked at the way in which visual representations of programs combine with programming paradigm to influence comprehension. The match-mismatch conjecture, which suggests that tasks requiring information which is highlighted by a notation will be facilitated relative to tasks where the information must be inferred, was applied to programming paradigm. The study showed that the match-mismatch effect can be overridden by other factors, most notably, subjects' prior experience and the programming culture in which they are taught.

The third study combined the methodologies of the first two studies to look at the match-mismatch conjecture within the wider context of information types. Using graphical representations of the control flow and data flow paradigms, it showed that, despite a bias toward one paradigm based on prior experience and culture, programming paradigm does influence the way in which the program is understood, resulting in improved performance on tasks requiring information which the paradigm is hypothesised to highlight. Furthermore, this effect extends to groups of information which could be said to be theoretically related to the information being highlighted.

The thesis also proposes a new and more precise methodology for the analysis of students' accounts of their comprehension of a program, a form of data which is typically derived from the information types studies. It then shows how an analysis of this qualitative data can be used to provide further support for the quantitative results.

Finally, the thesis suggests how the core results could be used to develop computer based support environments for novice visual programming, and provides other suggestions for further work.

Acknowledgements

I am grateful to Paul Brna, my first supervisor, for his patience, creativity, 'speeding bullet' email replies and a number of shared laughs, margaritas and glasses of muscadet. Thanks are also due to Richard Cox, my second supervisor, for making my research more statistically rigorous, and my thesis more clear. Helen Pain, initially my first supervisor, has remained a very dear friend and a source of excellent advice, optimism and encouragement.

The AI & Education group (past and present) provided a wonderful environment in which to conduct research: support and feedback were always forthcoming, and the fabulous socials were an added bonus.

Expert advice was provided by Andy Bowles and Simon Perkins on computer science issues, and Jean Carletta on coding schemes and Word macros. Numerous persons took part in the experiments and I am grateful to all of them for their help.

Two institutions eased some of the pressure of funding myself through my PhD: first and foremost, the St. Andrew's Society of Washington, D.C., and the Crowther Fund. Work on final stages of the thesis was carried out within the context of the GRiP (Graphical Representations in Programming) Project, funded by EPSRC grant GR/L36987.

I am thankful to the following people for a wide range of useful and insightful comments on earlier drafts: Eshaa AlKhalifa, Márcio Brandão, Marco Carvalho, Rafael Morales, Jean McKendree, Mike Ramscar, Jon Oberlander, Helen Pain, Pablo Romero, Chi-Chiang Shei and Shari Trewin. A special thank you to Judy Robertson, who made it through an entire draft and managed to rekindle my initial enthusiasm at a time when I really needed it.

'The boys' (Simon Perkins, Peter Funk, William Chesters and Enrique Filloy) were amusing fellow students, while Mark Burton gave me a home away from home in Leeds, fabulous meals and endless entertainment. Tim Taylor provided commiseration over "teatime breakfasts" at Teviot, and friendly competition from Mike "spit on palm" Ramscar helped me make my deadline. Thanks to Jonathan Kilgour, Jean McKendree and Carol Small for encouraging me to submit once my questions had reached the appropriate degree of triviality. Thanks to Jean for helping me to get to that stage by sharing information, and more importantly, knowledge, about many fields.

I feel like Stevie Wonder accepting a Grammy here, but just a few more:

Thomas Green has been an inspiration to me, as he has to many in the field, and I am honoured that he agreed to examine this thesis.

Mony Elkaïm, a friend so dear that he's really family, provided help and support in ways too numerous to mention. The fact that I even considered doing a PhD is a direct result of his encouragement.

Finally, to my husband, Bernard Goffin, for love, humour and shared dreams: *crescit eundo.*

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Judith Good
Edinburgh
February 25, 1999

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Novice Program Comprehension Reconsidered

Program comprehension is important, and yet difficult. It is an integral part of the programming process, playing a role in activities such as coding, debugging, and maintenance. Unfortunately, novices often find it extremely problematic to understand a program, and the types of difficulties they encounter have been well documented (see (Mayer, 1988) for a summary of some of these).

Novice problems may be compounded by the fact that comprehension per se is often not an explicit part of the curriculum. This may be because attempting to isolate the skill of comprehension and teach it directly can prove to be difficult, as there seems to be no universally agreed upon definition of what it is, and how it proceeds. This is unfortunate, as comprehension is an implicit first step in coding: learning a new language almost inevitably starts with the teacher showing the students a short program (*e.g.* the ubiquitous 'hello world') and then asking them to write similar programs. Thus, before even writing programs, students must be able to understand them.

If teaching comprehension is difficult, a number of other techniques might be used to approach program comprehension in a more indirect way, for example, by choosing languages which claim to make comprehension less painful, or by building learning environments to tackle the program understanding difficulties in novel ways. These are not without their own problems however, and will be discussed briefly in Section 1.4.

This thesis takes the view that novice program comprehension should be supported as a recognised activity rather than as a by-product of learning to program. It envisages an approach based on the combination of a number of external factors, many of which are present in some form in previous solutions to novice programmer difficulties. It is felt that a more detailed examination of the effect of the semantic and syntactic properties of a programming language on comprehension, combined with a change in the conceptualisation of program comprehension itself, have implications for the ways in which novice problems can be addressed: by moving from a traditional "process" view to one based on information entities, novel types of program comprehension support can be envisaged. By combining this conceptualisation with an approach which takes into account potential novice difficulties with particular types of information display, the thesis lays the groundwork for a flexible support system for program comprehension.

This chapter provides an introduction to the thesis by first describing novice comprehension difficulties and explaining why comprehension is difficult to teach. It then explores a number of potential solutions and examines their shortcomings. Following this, it suggests a new approach, based on some of the attributes of previous solutions. Before this approach can be implemented however, empirical work is necessary, and this work forms the core of the thesis questions. Finally, the chapter concludes with an outline of the thesis.

1.2 Novice Difficulties with Program Comprehension

Program comprehension is a daunting task for novices: where should one start and, once a starting point has been chosen, what is the most appropriate place to go to from there? How does one remember what one has looked at, and understood, and what one hasn't? Once some fragments have been understood, how can they be fit together into a coherent overview of the program? In short, *what does it all mean*?

The amount and variety of information in a program can seem overwhelming: there is information about the data the program works on, the data the program produces, the way in which it produces the data, and the reasons why it does so in the first place. Add to this all the low-level details of having to remember what particular keywords might mean in a program, the rules by which these keywords can be combined, ways of referring to different types of objects and changes in those objects, and it is not surprising that novices frequently feel lost.

These problems are likely exacerbated because novices have not yet developed the "coping strategies" which experts seem to possess. Experts are able to control the amount of information they are confronted with by chunking it into a smaller number of meaningful, higher level groupings (McKeithen *et al.*, 1981). Furthermore, expert search through a program is more focused, and is thought by some to function as a sort of hypothesis verification process (Brooks, 1983): based on their prior programming knowledge, experts form hypotheses about what they are looking for and use the program text in order to verify these hypotheses. While performing the search, they are able to take advantage of cues in the program, *e.g.* beacons (Brooks, 1983), in order to direct their search most fruitfully.

Without this knowledge and skill, novice program inspection may seem to lack focus: faced with so much information, an inability to consolidate it into chunks, and bereft of a search strategy, novices are not able to direct the decision making process which determines what would be the most appropriate thing to focus on next, or how to change focus when appropriate. Anderson *et al.* (1988) give an example of the latter case in the context of novices learning recursion. They feel that difficulty with the concept stems in part from the fact that recursion can either be seen as *data*, or *complex operations*, depending on one's viewpoint. The problem for novices is that they often persevere with a particular view of the problem and are blinded to a solution which could easily be reached via the other view, for example, they persist in tracing the control flow of the program rather than considering the output of a particular recursive call, a remark that seems as applicable to program comprehension as to program construction.

1.3 Why is Program Comprehension Difficult to Teach?

In looking for solutions to novice problems, a reasonable suggestion might be: why not look at what expert programmers do, and then try and teach novices to emulate them? Indeed, there is a wealth of literature on program comprehension focusing essentially on the comprehension *process*, in other words, on the steps by which programmers use their knowledge about programming in order to make sense of the program and extract the information they need.

This approach is problematic for a number of reasons. Firstly, there is no one model of program comprehension, and wide variation exists between the models which have been proposed, particularly with regard to the direction in which processing occurs. A number of theories postulate top-down processing, *e.g.* (Soloway *et al.*, 1988), while researchers such as Pennington argue for a bottom-up process of comprehension which occurs in two stages (Pennington, 1987a). Models of iterative processing have also been put forward (Brooks, 1983), while von Mayrhauser and Vans suggest an "integrated meta-model" which seeks to include both top-down and bottom-up elements (von Mayrhauser and Vans, 1994). Without a single normative model of program comprehension, it is difficult to see how to choose a model for teaching, or even if it would be appropriate to do so. Indeed, Gilmore (1990) maintains that one of the distinguishing features of expert programmers is the fact that they possess a repertoire of strategies, and are capable of choosing an appropriate strategy based on the programming situation, task characteristics and language requirements.

Furthermore, program comprehension is not a single, invariant activity comprising the same cognitive processes in all situations. Comprehension can be an activity in which one takes an entire program and "comprehends" it, but in many situations, comprehension of a program is required at different levels of granularity, for different purposes, and involving greater or lesser sections of the program. For example, one may need to understand the inputs and outputs to a program in order to link it up with another program, or understand a particular low-level detail so as to modify it. In this case, comprehension is grounded in a context and usually associated with a task. The context may also be communicative: obtaining the information needed to explain a program to a client or to a fellow programmer who is to modify the code will require focusing on different parts of the program and at different levels of abstraction.

Finally, although much has been made of the differences between novices and experts in terms of comprehension, little is known about the progression from novice understanding to expertise, particularly in terms of the nature of intermediate stages.

To summarise:

- there are a number of different theories of the comprehension process: how to choose one to teach?
- experts seem to vary widely in the strategies they use: again, how to decide which

would be the “best” one to teach?

- comprehension is not a single activity, it varies according to task, context, purpose, etc.

- little is known about the stages through which novices pass on their way to becoming experts.

1.4 Previous Approaches

There are a number of ways in which novice program comprehension problems might be tackled: this section describes some possible solutions, and points out potential problems. Note that most of these approaches relate to learning to program in general, of which program comprehension is seen as a subskill.

Ignore: Although it may seem like a radical alternative, one approach to comprehension problems might be to ignore them. After all, experts don’t exhibit these same difficulties, therefore it could simply be a question of time and patience. This is obviously risky: Soloway *et al.* (1992) describe the steep learning curve associated with learning to program, with serious effort invested in the initial stages for little return, and significant payoff realised only much later. Out of sheer frustration, students may never progress beyond the first stages to become more skilled at programming.

Additionally, an increase in end-user programming means that many individuals have no intention of becoming professional programmers, but would simply like to write small programs which serve their aims. As Soloway *et al.* point out, “...there will be a great number of people who will program computers in carrying out their daily activities. For such casual programmers, initial difficulties in learning a programming language may become a permanent barrier to their continuing interaction with computers” (Soloway *et al.*, 1982, p. 28) . Although specific end user languages and applications are being developed, it seems misguided to think that any new programming language will allow users to bypass the novice stage completely.

Choose the 'best' language: Much thought goes into the decision about which language to teach first. Traditionally, the preferred strategy has been to teach students a 'clean' or educational language, such as Pascal or Smalltalk, in order to convey the basic concepts of programming. Once this language was mastered, students were introduced to 'dirty', but commercially useful, languages such as C or C^{++}. Recent trends show that there appear to be two camps in this debate, with some maintaining that it makes no difference which language students learn first, and therefore starting students out with the most potentially useful languages for their career, and others continuing the search for the 'best' language. All would probably agree though that no one language will be a panacea for all novice programming difficulties, both generally and with respect to comprehension.

Invent a new language: New languages are constantly being invented, as are new ways of representing existing paradigms (*e.g.* visual programming languages (VPLs) based on control flow or functional paradigms). These languages are invariably accompanied by a series of claims, often the same for all languages (both object-oriented and visual programming languages spring to mind), such as, "the language is more natural", "it makes data structures/control flow/execution more apparent", "it will allow novices/end users to become experts in no time". Indeed, new languages often fall prey to what Green *et al.* (1991) term the 'superlativist' claim. However, this initial enthusiasm tends to wear off once the languages have been subjected to empirical testing, as there is ample evidence to show that no one language is best for all situations, expertise levels, or tasks (comprehension included).

Use multiple representations: Given the wealth of information contained in a program, another possible solution would be to present the student with multiple representations of the program, each of which highlights a particular aspect or type of information present in the program. Certainly, there is evidence to suggest that one characteristic which distinguishes experts from novices is their ability to use multiple representations and move between them with relative ease so as to exploit the inherent advantages of each one (Tabachneck *et al.*, 1994).

Petre *et al.* (1998) describe the possible benefits of having to translate between representations as a sort of 'useful awkwardness', postulating that it might encourage deeper levels of cognitive processing. However, the authors do acknowledge that multiple representations may be "provocative" in some instances, and "obstructive" in others. Indeed, although useful awkwardness might be exploitable in particular learning situations, one wonders how useful it might be when the novice does not have a clear grasp of the base representation to start with. Certainly, the advice from other fields, *e.g.* mathematics education, is to start by teaching one representation in depth rather than superficially teaching several forms simultaneously (Moss and Case, pear). Furthermore, results from the physics education field suggest that unless the student has a mastery of all of the representations used, they are likely to be a hindrance and lead more frequently to incorrect solutions (Scanlon and O'Shea, 1988).

Build a tutoring system: Helping novices learn to program has in some cases involved the provision of specially designed environments in which to learn programming before switching to the language in question. This solution often creates additional problems as novices must first learn to use one environment, then negotiate the transfer to the full-scale target language, which may involve unlearning some constructs and relearning others. For example, GIL (Merrill and Reiser, 1994) is a graphical environment which offers support of various types to novices learning LISP. At some point however, the novice must transfer to LISP, a textual language offering very few of GIL's features.

Additionally, the intensive nature of learning environment development means that often only a small portion of the programming curriculum can be included. For example, the Bridge system provided detailed support for learners to formulate their programming plans, but restricted novice support to a fragment of Pascal (Bonar and Cunningham, 1988). Similarly, GIL, mentioned above, was a very well thought out environment but did not offer the LISP novice any help with recursion (see (Good and Brna, 1996b) for a discussion).

1.5 A New Approach to Novice Program Comprehension Difficulties

This thesis suggests a new approach to supporting novice program comprehension, which is based on lessons learned from previous approaches. The proposed solution represents in many ways an ideal one. It relies on a number of as yet unsubstantiated suppositions about ways in which program understanding by novices might usefully be fostered. This thesis undertakes a detailed exploration of some of these hypotheses, looking at the factors involved in supporting novice program comprehension, and the ways in which they interact. The results of the thesis should inform the design of a novice support system, the tenets of which are described below:

Choose a full-scale language: this avoids the problems associated with inventing a new language, and those of moving up to a full-scale language or transferring from a novice programming environment.[1]

Replace process with information types: this thesis puts forward the idea that many differences in program comprehension models are not so much related to the information or knowledge necessary for program comprehension, but to the processes used in searching for/deploying this information and knowledge.

Therefore, rather than focusing on the temporal aspects of program comprehension, one can focus on the *entities* thought to be involved in comprehension. It is postulated that the comprehension process can be conceived of as combinations of steps, where steps involve the search for particular types of information. Different processes (*e.g.* top-down, bottom-up) would therefore involve different combinations of steps. Rather than trying to determine a fixed ordering on the sets of steps carried out during comprehension, it might be more useful to focus on the steps themselves, in other words, focus on the *product* of each particular step, rather than the *process* which combines them. This implies a less prescriptive approach, which leads to the following research question: can we teach novices about the different types of information present in a program, and how to locate

[1] Note that this point refers to the final solution: the empirical work necessary prior to implementing this approach requires the use of scaled down micro-languages in order to eliminate potential sources of confound in experimental settings.

this information, providing support for them as they do so, rather than limiting teaching to a single, invariant process?

Information types are a way of describing different types of information which are present in the program text, whose detection is necessary for program comprehension (Pennington, 1987b). They include such entities as functional information, data flow, control flow, etc. Novice support could be based on these types of information, firstly, by teaching novices what they are, and secondly, by helping them to learn how to recognise information types in programs. Despite their potential usefulness, unanswered questions remain, both on a theoretical and empirical level. In theoretical terms, Pennington sought to embed information types within a theory of program comprehension, based on Kintsch and van Dijk's theory of text comprehension (Kintsch and van Dijk, 1978; van Dijk and Kintsch, 1983). This thesis examines whether information types can have a useful role outwith this theory of comprehension.

From an empirical perspective, work on information types has focused on finding evidence for the comprehension theory described above, and on uncovering the nature of programmers' mental representations (Pennington, 1987b; Corritore and Wiedenbeck, 1991). Issues such as the effect of the particular language used in the study, the role of the task, the embodiment of information types in the particular language, and the interaction between information type and task have not been investigated in detail. However, all of these issues have important implications for support.

Fit this support onto the language itself: the aim is to provide support for program comprehension which is essentially layered onto the representation of the program itself. This support would be optional and act as a sort of scaffolding which could be turned on and off by the user at will. The idea is to end up with a programming language with support features, rather than a novice support environment which incorporates a programming language. This would hopefully avoid problems of transfer and scaling up. Furthermore, this approach sidesteps novice problems with multiple representations by operating on a single representation.

1.6 Main Thesis Questions

The proposed support system centres on activities such as searching for and identifying particular types of information in a program. As such, the features of the base language may play a role in determining the difficulty (or ease) with which the information can be accessed. This suggests that preliminary work should focus on issues such as the relationship between language features and the understanding of information types. In light of this aim, the thesis investigates the following issues:

- How do particular languages interact with the extraction of information types, particularly with respect to novices? Previous studies, which showed a predominance of low-level control flow information in the initial stages of comprehension, used control flow based languages. It is an open question whether this effect holds for different types of language, *e.g.* declarative, event driven.

- Some language paradigms could be said to mirror information types, *e.g.* control-flow languages and control-flow information. What is the relationship between information types and languages whose underlying paradigm mirrors a particular information type? Will there be an influence on the types of information extracted from the program?

- How does the task interact with the information highlighted by the representation?

- Can errors in comprehension be cast uniquely in terms of information types? Specifically in the case of VPLs, does the notation introduce difficulties on a syntactic level which cannot be accounted for by a semantic description of the language in terms of information types?

- From a methodological point of view, how can information extraction be measured most effectively, and in a way which is ecologically valid: what techniques should be used to gather and analyse the data?

- What might support for comprehension based on information types look like, and on what type of language could it be built?

1.7 Outline of the Thesis

This thesis examines the role which different *types* of information, hypothesised to be present in the program, play in program comprehension for novices. As such it does not make the claim that any of the hypotheses or findings described are in any way applicable to expert programmers. Furthermore, it considers primarily the notion of program *comprehension* rather than program coding.

Chapter 2 proposes a generic model of program comprehension, and then looks at how the most well-known models of comprehension fit into that model, based on their primary focus. It considers how these models might be relevant to novice program comprehension, and comprehension support. Finally, it looks at the notion of program comprehension as information extraction, before going on to describe experiments which have looked specifically at information types.

Chapter 3 describes a study carried out on Prolog novices using a similar methodology to the information types experiments discussed in the previous chapter. Prolog, a declarative language, was used instead of the procedural languages traditionally used in these experiments in an attempt to determine whether language paradigm has an effect on the pattern of extraction of information types by novices.

Chapter 4 discusses the rationale behind a proposed comparative study of control flow and data flow VPLs. It looks at the match-mismatch conjecture (Gilmore and Green, 1984) in detail, and at various studies which have tested it. It then considers the decision to use VPLs as a vehicle for studying the retrieval of information types.

Chapter 5 defines the notion of control flow and data flow, and looks at the historical development of control flow and data flow VPLs, with examples from the literature. It then describes the development of two micro VPLs, used in the studies reported in Chapters 6 and 7.

Chapter 6 reports on a pilot study which looks at the interaction between information type and language paradigm in more detail, using the methodology of the match-mismatch conjecture. Using the two micro VPLs described in Chapter 5, it investigates the way in which representations highlight or obscure the information required by the task, and also at the types of strategies novices use to make sense of a VPL and the misunderstandings they harbour.

Chapter 7 reports on an experiment which uses a new methodology, combining the information types approach used in the experiment described in Chapter 3 with the match-mismatch approach used in the experiment described in Chapter 6.

Chapter 8 looks at the question of program summary analysis. It first discusses problems inherent in attempting to apply the program summary analysis described by Pennington (1987b), and proposes a new approach based on finer-grained, orthogonal analyses.

Chapter 9 presents the results from applying the program summary analysis scheme to the data obtained in the experiments described in Chapters 3 and 7. It includes a comparison of the data across experiments and a general discussion of the results.

Chapter 10 summarises the main findings of the thesis, by relating them back to the original questions, and provides suggestions for further work.

Chapter 2

Program Comprehension

2.1 Introduction

What is program comprehension and why is it so important? The introduction to this chapter will consider these two questions in turn.

2.1.1 What is Comprehension?

Strangely enough, it turns out to be difficult to find a definition of program comprehension. Papers on comprehension tend not to define comprehension explicitly, perhaps because it seems so intuitively obvious, in the same way that, say, research papers on reading don't start out by asking what reading is. On the other hand, it may be because, like reading, comprehension covers a wide range of activities, with subtle differences between them.

Pennington and Grabowski (1990), in describing the tasks of programming, offer a notable exception:

> Understanding a program involves assigning meaning to a program text, more meaning than is literally 'there'. A programmer must understand not only what each program statement does, but also the execution sequence (control flow), the transformational effects on data objects (data flow), and the purposes of groups of statements (function) (Pennington, 1987b,a). In order to do this, the programmer will employ a comprehension strategy that co-ordinates information 'in the program text' with the programmer's

> knowledge about programs and the application area. This results in a mental representation of the program meaning." (Pennington and Grabowski, 1990, p. 54).

One of the reasons a unified definition of comprehension is not forthcoming may result from the scope of the activity: it can involve trying to understand an entire program in detail, scanning a program to look for a particular piece of information, or gaining a general overview of the program. Program comprehension is carried out by persons with differing levels of prior knowledge and experience, and can involve languages with very different characteristics. It comes into play whether one is reviewing one's own work, attempting to comprehend a program written by someone else, or trying to understand a program for a particular task (debugging, maintenance, communicating some aspect of the program to another person, etc.). As the nature of tasks themselves are wide-ranging, they will necessarily have very different information requirements, in terms of the amount of information, the type of information, and the way it is combined.

Program comprehension has been studied extensively, and from various angles. Many attempts have been made to derive a theory of program comprehension, focusing on the processes which occur when people try to make sense of programs, the sorts of knowledge they have about programming, and the models they form of programs. The role of information which can be derived from the program has also been studied, although to a lesser extent.

The following section provides a "generic" model of program comprehension. The reason for doing so is not to argue that this definition is superior to any others, but to try and encompass as many facets of comprehension as possible. The generic definition can then act as a framework in which to position theories and research on program comprehension, allowing a better understanding of the relationships between the various approaches.

2.1.2 A Generic Model of Program Comprehension

At the most basic level, program comprehension can be defined as follows: given a program in a particular language, program comprehension is a process in which the programmer uses prior knowledge about programming and information present in the program to form a dynamic, evolving model of the program which can then be applied to a task.

This definition makes use of a number of entities and processes, which are listed below:

- *knowledge* about programming, in other words, information the programmer has about programming in general and about programs he/she has previously encountered. The programmer's level of expertise is most likely the greatest influence on his/her knowledge, both in terms of the amount and type of knowledge, but also the ease with which new knowledge can be assimilated. The *form* in which the knowledge is stored is widely debated.

- a *mental model or representation* of the program. This represents the programmer's current state of knowledge about the program being studied, and so may be incorrect and/or incomplete. The model is dynamic and evolving.

- *information* contained in a program, either in read-off or derivable form. An example of information present in the program is a data object. Derivable information includes the program's function, in the sense that it is difficult to point to it in the same way that one points to a data object, but it is nevertheless determinable on the basis of the program code. Program information is determined in part by the programming notation.
  There is an interaction between programming knowledge and information in the program in the sense that different levels of knowledge allow one to "see" different things in the program.

- the *comprehension process*: the process by which the programmer extracts the information he/she needs from the program. This can vary in terms of the "direction of approach", in other words, whether programmers start from a hypothesis about the program and use the program code to verify it (top-down), whether they start from the concrete representation of the code and build up an understanding of the program based on the elements contained in the program (bottom-up), or (more likely), some combination of the two.

- *comprehension strategies*: an issue which has not received as much coverage, as many theories are built around a single model of comprehension. The work on strategies assumes that there are different ways of understanding a program, and looks at the circumstances in which these strategies are deployed.

- a comprehension *artefact*, *i.e.* something produced after viewing a program and which provides evidence of comprehension. It could take the form of a verbal or written explanation of some aspect of the program, a new piece of code, a modification to the code, an answer to a question about the program, comments in the code, etc. Note that production of an artefact is optional, but it is, however, interesting from the point of view of research into the psychology of program comprehension.
- a comprehension *task* or *purpose*, in other words, why the programmer is looking at the program in the first place, and what he/she wants to get out of it. This will determine in turn the *scope* of the comprehension process.

Various issues which are central to theories of program comprehension can be grouped around these main entities: hypothesis formation and level of expertise are related to programming knowledge and to process, while issues of programming notation relate to the information in the language, and the ways in which it can be characterised.

Although these entities have been separated out for the sake of discussion, they are highly interrelated: general programming knowledge will influence the model one builds of a program, while programming strategy will also be influenced by programming knowledge, and likely by task as well.

2.1.3 Why is Comprehension Important?

In the first chapter and in the above paragraphs, it was mentioned that program comprehension plays a role in many of the tasks of programming. However, it is in some ways an even more important part of learning how to program. Lecturers often start a programming course by describing a completed program, before going on to describe how to write one. Programming textbooks frequently use an expository method of introducing new programming concepts based on an example, and then require students to produce similar programs as an exercise. Students often continue to adopt this method, searching through the textbooks to find an example program which might serve as a good starting point for a new program. However, in order to successfully adapt/change a program, it is necessary to first understand it.[1]

Shneiderman and Mayer (1979) identify the tasks of programming as composition, comprehension, debugging, maintenance and learning. In this categorisation, the last task, learning, would logically cover the first four tasks as each comprises a skill to be learnt. However, the extent to which program comprehension is taught explicitly is an open question. Certainly, it is not highlighted in the literature: in a very interesting article on teaching programming, du Boulay and O'Shea (1981) consider novice difficulties in three main skill areas: planning, coding and debugging. Likewise, *Studying the Novice Programmer*, by Soloway and Spohrer (1989), covers such issues as transfer, learning programming, misconceptions and the design of programming environments, but again, does not consider the teaching of comprehension. Debugging is frequently addressed as an issue in its own right, often in the form of tips for troubleshooting, but it is unclear whether students are taught how to look at a correct program, make sense of it, and pick out and synthesise the information they need.

The lack of instruction in program comprehension may be due to several factors, many of which may be theoretical. Firstly, as mentioned above, program comprehension is multi-faceted, yet many theories seem to take a "single activity" stance, and attempt to propose an invariant model for comprehending an entire program. This does not take into account the fact that, as von Mayrhauser and Vans (1994) point out, the processes delineated by such an approach may not be appropriate to situations in which partial understanding of the code is required, *e.g.* trying to pinpoint an error in a very large piece of code. Furthermore, it ignores the issue of novice/expert differences: von Mayrhauser and Vans (1994) make the point that some comprehension processes, top-down in particular, require previous knowledge, and so would not even be available to novices, at least in the first stages of comprehension. Finally, even given that most theories are based on comprehension of the whole program, they vary widely in terms of how they postulate that programmers go about understanding the program. This makes it difficult to choose the most appropriate model to teach.

The above points notwithstanding, program comprehension is a process: there is a definite temporal aspect to it, and it seems natural to describe it in terms of a series

[1] Even if it sometimes looks as if students skip that first step ...

of steps. However, given the issues of variability in process, and the fact that the process for general program understanding will not be appropriate in specific instances, it may make more sense to look at comprehension at a lower level of granularity. Rather than trying to describe the entire process of comprehension from start to end, perhaps one can identify which types of information and which actions are important to comprehension. This makes for a less predictive theory, but it also allows the "theory" to be more widely applicable, perhaps resulting in a more flexible approach to program comprehension teaching.

2.1.4 The Rest of the Chapter

The following sections describe ways in which program comprehension can be characterised, organising theories, research and empirical work around the various entities making up the generic model described above. The aspects considered are:

- comprehension as a process (Section 2.2);
- comprehension strategies (Section 2.3);
- the role of the programmer's knowledge in comprehension (Section 2.4);
- the programmer's mental representation or model of the program (Section 2.5);
- comprehension as the search for information in the program (Section 2.6).

Is it impossible to cover all theories and research in the space of a chapter, therefore selected theories and research will be used to highlight the issues involved in each aspect of the generic model. Again, as mentioned above, these distinctions are not mutually exclusive: theories about the process of programming obviously relate to programming knowledge. However, the theories have been organised according to their perceived main focus. In some cases, theories and research will be spread over more than one category: this is the case for Pennington's work, which has considered several of the aspects in question. Despite surface differences and differences in focus, many theories are reassuringly similar: most, if not all, make some provision for the entities described above. In many cases, it is the terminology chosen to describe these entities which varies.

Following the sections on program comprehension theories and research, the chapter considers their implications for providing comprehension support to novices. It discusses the usefulness of focusing on the external products of program comprehension, rather than on the internal processes, and discusses the idea of *information types* in this context. The final section will consider empirical work on information types, and highlight questions to be answered in this thesis.

2.2 Comprehension as a Temporal Process

The theories described in this section will be considered as "temporal theories", as their description of program comprehension relies heavily on a notion of sequence, describing the steps which the programmer takes to understand the program, and the order in which they are taken.

The temporal sequencing in these theories is linked to the level of abstraction: *top-down* theories postulate a progression from a high level of abstraction to a lower level verification process, whereas *bottom-up* theories hypothesise the building up of higher level abstractions from low level entities present in the program code. *Mixed* theories postulate greater movement between levels of abstraction. These theories are described below.

2.2.1 Top-Down Models of Program Comprehension

A number of theories conceive of program comprehension in terms of top-down processing, with Brooks (1983) proposing one of the first models. He describes program comprehension as the inverse of coding: whereas coding involves a mapping from the problem domain into the programming domain, often through a series of intermediate domains, program comprehension requires the reconstruction of these mappings. In contrast with the usual program/problem domain distinction used by many researchers, Brooks uses the term "domain" to refer to many aspects of the comprehension process: a real-world object/program object mapping domain, the algorithmic domain, the programming language domain (embodying the particularities of that language in terms of its data structures and primitive operations), and an execution domain, couched in terms of memory locations and hardware.

Comprehension is seen as a hypothesis verification process. Sometimes simply hearing the name of a program will trigger the construction of a hypothesis about the program. This high level, functional hypothesis leads to the development of a tree of subsidiary hypotheses, generated in a top-down, depth-first manner, based on domain knowledge and familiarity with similar programs. When the level of detail matches that of the program text, hypothesis verification occurs by searching for *beacons* (described in more detail in Section 2.6.1), defined by Brooks as "typical indicators of the use of a particular operation or structure" (Brooks, 1983, p. 548). The beacon which acts to confirm the current hypothesis will be triggered, as will any beacon which strongly indicates the presence of other structures (this reduces the need for several searches). New subsidiary hypotheses may also be formed at this stage. Lines of code which confirm the subsidiary hypothesis will become bound to it, thus creating a tree-like structure whose leaf nodes are actual code segments. At this stage, comprehension is complete.

2.2.2 Bottom-up Models of Program Comprehension

In bottom-up theories of program comprehension, programmers are hypothesised to build up an abstract representation of the program based on the program text.

Although an article by Shneiderman and Mayer (1979) is often cited as one of the first bottom-up theories, it is the author's feeling that this is not necessarily the most accurate categorisation. Shneiderman and Mayer focus much more on the structure of the programmer's knowledge than on the order in which the comprehension process occurs, therefore, this theory will be described in Section 2.4. This section will instead use Pennington's theory of program comprehension as a means for describing the general characteristics of the bottom-up model. As Pennington's theory appears throughout this chapter under various headings, it will be described only briefly here.

Pennington (1987b) based her work on the theory of text comprehension put forward by Kintsch and van Dijk (1978) and van Dijk and Kintsch (1983). This theory postulates that text comprehension results in the production of two distinct but interrelated representations of the text, the *textbase* and the *situation model* (described in more detail in Section 2.5.2). According to Pennington's adaptation of this theory to program texts, the textbase is first built, based on a procedural reading of the program, and couched in terms of the programming language. The situation model is built from this textbase (or "program model" as Pennington calls it), and highlights functional relationships between domain objects.

Pennington therefore hypothesises that program comprehension occurs from the bottom up, starting with the program. Programmers first divide the program into small control segments, making inferences about each segment's procedural role. Function and data flow run through segments, therefore they are thought to be uncovered through a process of integration later in the comprehension process.

2.2.3 Mixed Theories of Program Comprehension

Letovsky's model (1986) can be considered to be a mixed theory in that program comprehension is hypothesised to take place using both top-down and bottom-up processes. This model complements the model described by Littman *et al.* (1986) in that the latter describes macro-scale events while Letovsky's model focuses on meso-scale events, *i.e.* events which occur over the space of seconds and minutes. In many ways, this is one of the most complete and flexible models of comprehension, hypothesising a range of different types of knowledge and reasoning processes.

Letovsky's cognitive model centres around the idea of the programmer as *knowledge based understander*. Knowledge based understanders (human or otherwise) consist of three entities: a *knowledge base*, containing prior programming knowledge and expertise; a *mental model*, which is the current understanding of the program in question and therefore dynamic; and an *assimilation process*, which constructs the mental model through interaction with the program code and documentation, and the knowledge base.

A number of different types of knowledge are hypothesised to reside in the knowledge base: programming language semantics, goals, plans, knowledge about efficiency, domain knowledge, and the rules of programming discourse.

The mental model is a layered network consisting of a specification at the top, an implementation at the bottom, and *annotation* layers in the middle. The specification describes the goals of the program, while the implementation describes the actions and data structures. The intervening layers are explanations linking goals to actions and vice versa. The model will be incomplete during the comprehension process, and nodes and links can be added in either direction. Starting from the specification layer would indicate a top-down process, while starting from the implementation layer would imply a bottom-up process. In Letovsky's view, the comprehension process is opportunistic in the sense that humans can exploit any available cues.

One of the interesting features of this model is that Letovsky analysed verbal protocols of programmers working on a comprehension task, and attempted to isolate various types of events which occur. He defined *inquiries* as a high-level structure comprising questions, conjectures and searches centring on the same topic. For example, a programmer might wonder about the importance of a particular data structure in the program, hazard a guess as to what its role might be, and then search the program for confirmation. Letovsky discusses how to identify questions and conjectures, and provides a taxonomy of each based on content. Furthermore, for conjectures, he elaborates the idea of *conjecture certainty*, ranging from *facts* to *guesses*.

Finally, Letovsky "explains" conjectures and questions, by which he means finding computational mechanisms which would produce the behaviours in question. Different question types can be associated with the directionality of processing (*e.g.* "how" questions indicate top-down reasoning). The notion of "urgency" is also introduced to explain why some types of questions must be answered immediately while others can be delayed, and this notion is linked to hypothesised memory limitations. In terms of conjectures, the model postulates that different reasoning processes underlie the formulation of different types of conjectures (*e.g.* slot filling, abduction) and the knowledge or artefacts on which these processes operate. Inquiries can then be explained in terms of the processes postulated for questions and conjectures.

This model has a lot of scope for explaining novice programmer difficulties. It is widely accepted that novices cannot use top-down comprehension as they are lacking the knowledge to formulate the hypotheses in the first place, and that they therefore resort to a bottom-up process based on the program code. Letovsky's model allows for a finer-grained view of the process: analysing an individual novice's protocols might allow one to identify those moments in the comprehension process where an impasse is reached. It may then be possible to discern patterns suggesting difficulties with particular types of reasoning, or that particular types of knowledge which are either missing, incomplete or incorrect.

Boehm-Davis (1988) proposes a model which is, in many ways, a variation on Letovsky's model. She postulates an iterative segmentation, hypothesis generation and verification process in which programmers use their knowledge along with the information present in the program (*e.g.* plans, beacons) to segment the code into manageable pieces, formulate hypotheses about program function, and verify the current hypothesis with the actual code. The iterative process is opportunistic. An integration stage coordinates hypothesis generation based on the programmer's knowledge base and current understanding of the program.

2.3 Comprehension as Strategy Deployment

So far the theories described have dealt with the comprehension of an entire program. von Mayrhauser and Vans (1994) point out that these models may not be appropriate to situations in which partial understanding of a program is required, *e.g.* pinpointing a single bug in a very large piece of code. However, models do not seem to exist which describe the processes involved in looking for particular types of information, or at isolated code segments for specific reasons. The following study describes differences in comprehension strategy as they relate to debugging, but the message seems to be that best performance is nonetheless associated with one of the two strategies.

2.3.1 Systematic *vs.* As-Needed Strategies

Littman *et al.* (1986) describe two types of strategy used by experts in the context of a maintenance task: a *systematic* strategy, and an *as-needed* strategy. In the systematic strategy, the programmer attempts to make sense of the entire program before modifying it. In the as-needed strategy, the programmer concentrates on the part in the program where the modification should be made. The danger with this approach is obviously that the modification will have effects on other parts of the program which the programmer did not anticipate.

The authors suggest that two types of knowledge must be gleaned from the program: *static* knowledge, which concerns the objects, actions and functional components of the program, and *causal* knowledge, which describes the connections and interactions between the functional components, acquired by mentally executing the flow of data and control between the components.

Littman *et al.* maintain that the strategy used by the programmer has a direct influence on the knowledge which is acquired. This knowledge is stored in the programmer's *mental model*, which may be of two types. *Weak* mental models are derived from the as-needed strategy and contain static knowledge only. *Strong* mental models are built from the systematic strategy and contain both static and causal knowledge.

A few points are worth noting with respect to this study. Firstly, the comprehension task occurs in a realistic setting (debugging a program), but the findings may not apply to contexts other than debugging. Secondly, they state that subjects were not allowed to carry out a "test and debug" sequence, therefore, a systematic strategy may well have been more appropriate *in that context*; whether the "as needed" subjects would have been as disadvantaged in a situation more akin to most programmers' actual working conditions is open to question.

Finally, and this is a point that Littman *et al.* readily acknowledge, this type of approach is feasible for a program of moderate length such as the one used in the study: for very large pieces of code, a systematic strategy will not be possible.

2.3.2 Positive Strategy Differences

While the above study postulated that one of the two strategies observed was the more successful one, Gilmore (1990) suggests that having a wide repertoire of programming strategies available is one of the hallmarks of an expert programmer. Looking in more detail at comprehension and debugging, he notes that many theories of comprehension focus on the static knowledge possessed by experts, while ignoring the fact that experts also possess strategies for making use of that knowledge. He maintains that differences in strategy, in addition to differences in knowledge, contribute to the novice-expert distinction in programming, and cites a study by Widowski (1987) showing that experts, unlike novices, were able to change comprehension strategy depending on the "typicality" (*i.e.* meaningfulness) of the program, and on its complexity.

2.4 Comprehension as Programming Knowledge

This section describes theories which focus on the knowledge possessed by the programmer, and the ways in which this knowledge influences comprehension. This is not to say that there is no implicit or explicit temporal ordering in these theories, simply that the ordering is perhaps not the most important aspect of the theory.

2.4.1 Semantic/Syntactic Knowledge

Shneiderman and Mayer (1979) proposed one of the first cognitive theories to cover the whole of the programming process, including coding, comprehension, debugging, modification and learning. The theory is grounded in the information processing approach, and looks in detail at the role of various memory stores in programming (short-term, long-term and working). These stores motivate the main questions which the theory aims to answer, namely, what kind of knowledge does the programmer have in long-term memory, and what processes does he/she use to construct a solution in working memory? The former question will be considered in detail here, as it is most applicable to program comprehension.

Two types of programming knowledge are hypothesised to reside in long-term memory: *semantic knowledge* and *syntactic knowledge*. Semantic knowledge consists of general programming concepts which are language independent. This knowledge ranges from low-level knowledge of what specific program actions are, up to problem solving knowledge for particular application areas. Syntactic knowledge is language dependent, making it at once more precise and more arbitrary (*e.g.* the use of particular keywords to denote loops).

Shneiderman and Mayer hypothesise that syntactic and semantic knowledge interact in program comprehension as follows: programmers use their syntactic knowledge of the programming language to construct a multileveled, semantic representation of the program. The higher levels of the structure will include information about the program's function, while lower levels may include familiar algorithms or operations. Although they hypothesise that program comprehension proceeds by successively chunking groups of statements into larger and larger chunks until the program is comprehended, they acknowledge that high-level comprehension may occur in the absence of full knowledge of the low-level details, and vice versa, suggesting that directionality of processing was not the main focus of the theory.

2.4.2 Plans

Plans have played a very important part in the development of theories of program coding and comprehension, and have been studied extensively, most notably by Soloway and his colleagues. Research has been carried out to establish the psychological reality of plans (Soloway *et al.*, 1982; Soloway and Ehrlich, 1984; Soloway *et al.*, 1988), and they have been used as the basis for novice programming environments and tutors (Johnson and Soloway, 1985; Bonar and Cunningham, 1988).

Definitions of Plans

In some ways, the position of plans in the generic model described in Section 2.1.2 is unclear: should they be seen as information which can be identified in a program, or as a programmer's internal representation of his/her programming knowledge? On the one hand, plans are defined as "program fragments that represent stereotypic action sequences in programming" (Soloway and Ehrlich, 1984, p. 595), while on the other, they are defined as "a procedure or strategy in which the key elements of the process have been abstracted and represented explicitly." (Soloway *et al.*, 1982, p. 30). The latter definition considers a plan as a knowledge structure, much like a schema, which is possessed by expert programmers. This thesis will use the latter definition, but it should be remembered that, using the former definition, plans could also be considered to be "information present in the program".

As the last paragraph shows, there is considerable variation in the way plans were initially defined. In addition to the definitions given above, Soloway *et al.* (1982) define three types of plans: *strategic* plans, *tactical* plans, and *implementation* plans. Strategic and tactical plans are language independent plans for solving problems, with strategic plans operating at an algorithmic level, and tactical plans operating on smaller problem segments. Implementation plans are language dependent techniques which allow strategic and tactical plans to be realised.

Rist (1986) supplements this with *global plans*, a type of high-level plan which can describe a simple program in terms of its input, process/calculation, and output. He also describes the idea of *focal lines* which are "key features" in a program which convey the gist of the program, and are the "driving force of the plan". According to Rist, a focal line often appears deep inside the program, and will be identified more easily by experts than novices, who process the code in a sequential manner.

Robertson and Yu describe two types of plans: task related plans and global plans. Task related plans are subgoals expressed in what they term the "task language" (*i.e.* the domain) and are described at a functional level, in that they specify what is to be done, but not how. "Global plans" refer to "subgoals that are abstractions from a specific task domain" (Robertson and Yu, 1990, p. 344) and may apply to several different tasks. These are domain independent, but not program independent. In other words, these plans are couched in the language of programming constructs, *e.g.* "iterate using a loop". These global plans can be implemented via a single line of code or several lines.

It is difficult to reconcile the different definitions of plans, even by the same authors, and to determine whether alternative definitions complement the original definitions, or in fact slice up the domain in a different way. Given the scope of many of these definitions, it is not surprising that many studies found support for plans. One of the first studies to be carried out which relates specifically to program comprehension (rather than coding) is described below.

Plans: Empirical Support

Soloway *et al.* (1988) state that plans are only one type of knowledge available to the expert programmer: the other type is *rules of programming discourse*, which are conventions in programming, and are analogous to discourse rules in conversation. These rules are in some ways common sense coding conventions, *e.g.* "use meaningful variable names", "make sure the code is actually used, but that it isn't doing non-obvious double duty". According to the authors, programs which conform to the rules of programming discourse are "plan-like", while those programs which violate the rules are "unplan-like".

This theory was tested by presenting novice and expert programmers with two versions of a program: one which followed the rules of programming discourse, and one which violated them. Subjects were asked to fill in a blank in the program.[2]

They found that subjects take longer to complete unplan-like programs, and make more errors when doing so (some of which correspond to providing the plan-like answer). In addition, experts' behaviour is significantly impaired on these programs.

Based on this experimental evidence and on the development of an AI simulation of program understanding (Johnson and Soloway, 1985), program comprehension is hypothesised to occur as follows: given that the program goals were not known, subjects used a bottom-up process, reading the program from the beginning. The recognition of programming plans in the program triggered a top-down process, with subjects using *shallow reasoning* in order to match the plans to the actual code. The unplan-like programs forced the use of bottom-up processes such as program execution, and *deep reasoning*, or reasoning causally about the relationship between the program code and the program's goals.

Plans: Unanswered Questions

Although plans have played an important role in studying programming, their status is unclear. Firstly, they suffer from the lack of a formal definition (Bellamy and Gilmore, 1990), which may have led to the proliferation of definitions available. Secondly, their degree of universality is unclear: although Robertson and Yu (1990) suggest that plans are language independent, Gilmore and Green (1988) provided evidence to the contrary. Davies (1990) suggested that the observed language dependence may in fact be due to differences in prior experience and culture which determine whether plans are recognised and/or used. Finally, the relationship between programming plans and rules of discourse has never been fully explored. Although Soloway *et al.* (1988) showed that violating rules of discourse to produce unplan-like programs affected experts' ability to understand programs, it also had a marked effect on novice performance. Comparing plan-like programs to unplan-like versions, experts showed a 53% decrease in scores, while novice performance decreased by 41% (Soloway *et al.* do not say whether the latter difference was tested for significance). Whether this implies that novices also have a reasonable amount of knowledge about plans is a point which does not seem to

[2] A technique very similar to the one used by Gellenbeck and Cook (1991), except that they were looking for the existence of beacons, rather than plans ...

have been addressed.

2.5 Comprehension as Mental Representations

This section describes work which has focused on the way in which a particular program is represented by the programmer, as opposed to the programmer's representation of general programming knowledge. Much of the work on mental representations has been empirically based: a number of studies have aimed to elicit a programmer's mental representation of a program, sometimes at distinct intervals (*e.g.* before and after modification/debugging), and to characterise its structure and content. A selection of this work is described below.

2.5.1 Novice/Expert Differences in Mental Representations

Adelson

Adelson (1984) was interested in the types of representations which novices and experts form of a program. She hypothesised that novices form *concrete* representations based on how the program works, while experts form *abstract* representations based on what the program does.

Adelson tested this hypothesis by having novice and expert programmers answer concrete and abstract questions after examining short programs accompanied by either an abstract flowchart, a concrete flowchart, or no flowchart. Her results tended to bear out her hypothesis: experts scored higher on abstract questions in all conditions, while for concrete questions, novices scored higher than experts, except, strangely enough, in the case where they received a concrete flowchart. In a further experiment in which the flowcharts were replaced with tasks designed to "channel" the subject's understanding of the program in either an abstract or concrete way, Adelson found much the same effect.

A few points do seem worth noting:

- the *abstract* task required subjects to insert a missing line of code into the program, which surely requires quite low-level knowledge of how the program works.

- in the second experiment, the percentage of errors for experts on the abstract questions was 13% when the task did not match the questions (*i.e.* when a concrete task was used) as compared to 38% when it did. This is a strange result, which Adelson does not discuss.

- Adelson states that the novices were taking a course in PPL (Polymorphic Programming Language), while the experts were the teaching fellows for the course. It may be that the problems were not familiar to the novices, who therefore adopted a bottom-up process, but were familiar to the experts, which allowed them to use a top-down process, perhaps (erroneously) filling in the low level execution details based on prior knowledge rather than on the code.

Holt *et al.*

Holt *et al.* (1987) carried out an interesting study to investigate differences in mental representations between student and professional programmers. The experiment required subjects to perform easy or difficult modification tasks on programs based on three different design approaches: serial (in-line code), functional decomposition or object-oriented.

Subjects' mental representations of the program were elicited by asking them to recall program components and to specify the relationships between components. The representations were then analysed in terms of the number of components and relationships, the depth (longest chain of linked segments) and width (largest number of branches) of the representation, and degree of connectedness.

Interestingly, they found no significant differences overall between students and professionals in terms of the structure of their mental representations. However, they found that the two groups were differentially affected by certain variables: complex modifications were associated with more complex mental representations for the expert programmers, while the student programmers' representations were affected by the structure of the program (serial, functional, object-oriented) and by the particular program.

2.5.2 Mental Representations and Information Types

Pennington (1987b) investigated mental representations using the idea of information types, described in Section 2.6.2. Her model of comprehension, described in Section 2.2.2, postulates the existence of two types of mental representation of a program: the *textbase* and the *situation model.* The *textbase*, according to Pennington, "includes a hierarchy of representations consisting of a surface memory of the text, a microstructure of interrelations among text propositions, and a macrostructure that organizes the text representation", while the *situation model* is "a mental model of what the text is about referentially" (Pennington, 1987a, p. 101). Pennington hypothesised that the textbase for a program would largely include procedural relationships structured in terms of the program text, while the domain model would contain functional relationships between real world objects.

According to this theory, the textbase is first built, based on a procedural reading of the program, and couched in terms of the programming language. The domain model is based on this textbase (or "program model" as Pennington calls it), and highlights functional relationships between domain objects. Pennington suggests a link between the type of description and the level of granularity: namely that functional relationships are more "comprehensible" when couched in terms of real world objects, while procedural relations will be expressed in programming language terms. There is also an implicit temporal ordering as the domain model is thought to be dependent on the construction of the textbase. The representations are, however, cross-referenced, and Pennington hypothesises that cross-referencing is one of the requirements of effective comprehension.

This work was followed up by Corritore and Wiedenbeck (1991), who investigated the mental representations of novices, and Ramalingam and Wiedenbeck (1997), who looked at the effect of paradigm on mental representation. As the work described in this thesis borrows extensively from the methodology used by all of these researchers, their work will be described in detail in Section 2.8.

2.6 Comprehension as the Identification of Program Entities

This section describes research which focuses on the information said to be present in, or derivable from, a program. This is information which the programmer attempts to uncover during program comprehension, as opposed to knowledge which he/she already has about programming.

2.6.1 Beacons

Brooks (1983) used the term *beacons* to describe "sets of features" which signal the occurrence of particular types of structures or operations. The mapping between structures/operations and beacons is many-to-many: one structure or operation can be signaled by multiple beacons, while "the same feature may participate in beacons for a variety of structures or operations" (Brooks, 1983, p. 348). Wiedenbeck (1986) used the example of a beacon from Brooks (1983) in a program memorisation and recall task, and was able to show that experts recall these key lines better than other parts of the program, unlike novices, who correctly remembered more lines from the beginning of the program.[3]

Gellenbeck and Cook (1991) also studied the role of beacons in comprehension. They acknowledge that the concept of beacon is ill-defined, and attempted to look at how entities such as procedure names, variable names, and key lines in the program (*e.g.* the swap operation) might act as beacons. While their research is interesting in determining where the effect lies for each of these changes to a program, the notion of beacon remains indeterminate.

2.6.2 Information Types

Information types have been defined as "different kinds of information explicit 'in the text' that must be detected in order to fully understand the program" (Green *et al.*, 1980; Green, 1980) in (Pennington, 1987b, p. 299). Pennington described information

[3] Wiedenbeck states that it is an open question why two-thirds of novices were able to correctly state the program's function, even though they seemed to recall the program in a linear order. However, it is plausible that comprehension and memorisation of the program occurred as two separate processes: students first inspected the program for meaning (in whatever order), then set about trying to memorise the program line by line.

types in terms of internal, rather than external, abstractions of a computer program. In other words, they are not meant to be mental representations of the program, but are "based on formal analyses of programs developed by computer scientists" (Pennington, 1987b, p. 298), and can be compared with the abstractions which are made with respect to natural language, such as referential or causal abstractions.

Pennington identified five types of information: *function*, *control flow*, *data flow*, *state* and *operations*, defined as follows:

Function: information about the overall goal of the program, essentially, "What is the purpose of the program? What does the program *do*?". Function also includes program subgoals, therefore goals and subgoals can be represented in a goal hierarchy. Some information about which events precede others can be inferred, but not details of how the events are implemented.

Control flow: information about the temporal sequence of events occurring in the program, *e.g.* "What happens after X occurs? What has occurred just before X?" If the information is represented graphically, then the links will correspond to the direction of control, rather than to the movement of data. Data flow information can be inferred in a program by searching for repeated occurrences of data objects, but goal/subgoal information is harder to infer.

Data flow: essentially concerned with the transformations which data objects undergo during execution, including data dependencies and data structure information, *e.g.* "Does variable X contribute to the final value of Y?" The data flow abstraction is linked to the function abstraction in the sense that function information can be partially reconstructed from a data flow abstraction. Control flow information is also more readily inferable than from the function abstraction.

Operations: information about specific actions which take place in the code, generally corresponding to a single line of code or less, such as "does a variable become instantiated with a particular value?" Although Pennington doesn't describe these in great detail, they seem most linked to control flow information, in the sense that describing the control flow of a program would lead to "stringing together" a series of operations.

State: time-slice descriptions of the state of objects and events in the program, *e.g.* "When the program is in state X, is event Y taking place, or has object Z been created/modified?" This abstraction is quite "distinct" in the sense that other types of information are hard to infer from it, and likewise, state information is difficult to infer from the other types of representation.

The categories are orthogonal in terms of information coverage. In terms of level of granularity, although Pennington doesn't address this point explicitly, the categories vary: *function* can often cover the entire program, while *operations* will concern only a single line (or node, in the case of a visual programming language).

Pennington was interested not so much in information types per se, but in the relationship between information types and what she called *programming knowledge structures.* She looked at two competing knowledge structures: *text structure* knowledge, which is organised around control structure primes, and *plan* knowledge, which, according to her, is primarily functionally oriented. Pennington carried out two experiments to investigate these ideas, which are described in Section 2.8.

2.7 Comprehension Theories and Research: Implications for Novice Teaching

From the theories and research examined in the preceding sections, it is obvious that there are differences between novice and expert programmers. Results of program comprehension experiments comparing novice and expert behaviour (Adelson, 1981; McKeithen *et al.*, 1981) show many of the same patterns found in other fields (Chase and Simon, 1973). The main differences between the two groups seem to be in terms of syntactic *vs.* semantic knowledge, and shallow *vs.* deep reasoning. In a very interesting paper on novice/expert differences, Mayer (1988) describes four types of programming knowledge: syntactic (language syntax), semantic (language meaning), schematic (common subroutines or functional code units), and strategic (how to use the available information to achieve a goal).

Summarising novice/expert differences, Mayer draws the following conclusions for each type of knowledge respectively:

- experts, unlike novices, have automated their processing of syntax, so are able to focus on higher levels of processing;

- in terms of semantic knowledge, experts have a more coherent conceptual model, but (or possibly "therefore") novices benefit, more than experts, from instruction focusing on conceptual models;

- experts possess more schematic knowledge, as evidenced by the fact that there is a difference in recalling ordered *vs.* scrambled programs for experts, but not for novices (Shneiderman *et al.*, 1977).

- in terms of strategic knowledge, experts take a more top-down, breadth first approach to comprehension than novices, attempting to understand the abstract goals of the program.

What are the implications of novice/expert differences for teaching? Some of the research described in this chapter is directly relevant. Mayer (1988) points out that most programming instruction focuses on syntax, and that far more attention should be given to the other types of knowledge than is currently the case. Looking at the issue from the point of view of information types, Corritore and Wiedenbeck (1991) state that although many textbooks consider the issue of control flow in a program, they seldom consider data flow or state. Furthermore, they do not explain how to derive function information from a program. These authors call for a curriculum in which novices are not only made aware of the specific types of information, but are also taught how to extract this information from a program when needed.

However, many theories have not addressed the issue of teaching novices how to become experts, which is not surprising given that it wasn't the researchers' intention to do so in the first place. It might be more sensible to ask whether program comprehension theories and research can be interpreted as having implications for teaching comprehension: this question will be the focus of the next few sections. Rather than consider each aspect of the generic model in a separate section, they have been grouped together under three main headings: comprehension as a process, internal representations (*i.e.* programming knowledge and mental representations), and external manifestations of information search and/or strategy.

2.7.1 Comprehension Processes

Many of the theories which describe comprehension in terms of process aim for a unique description of events which occur when understanding a program. In this sense, they could be accused of being insensitive to differences between task environments which might affect the comprehension process.

Furthermore, there is no agreement as to which process theory is the "correct" one, which leaves educators with the dilemma of which theory to pick. Top-down and bottom-up theories are diametrically opposed: either programmers start with high level hypotheses and use the code to verify them, or they start with the code and build up chunks until a complete understanding of the code is reached. They can't do both, unless of course, one opts for a mixed model. Lack of agreement on a single model is probably healthy, as it is in some ways an implicit acknowledgement of the difficulty of characterising comprehension as a unique, well-defined activity. However, it does not help the teacher.

Many models of comprehension are normative models: they describe what happens when competent programmers correctly understand a program. The case for novices is less clear: one of the biggest problems facing novices is their lack of prior programming knowledge and experience. Therefore, a top-down model of program comprehension would in principle be out of their grasp, as the initial stages depend on information they don't possess. They may fare better with bottom-up comprehension (and many studies seem to suggest this is what novices do), but when they come to the actual chunking stage, they seem to chunk according to the wrong criteria (*e.g.* when asked to group problems based on how they would be solved, novices group them together according to the domain in which they operate rather than the way in which they work (Weiser and Shertz, 1983)).

The scope of many models is also potentially problematic, as they assume that program comprehension implies a complete understanding of the entire piece of code. Firstly, it is difficult to know what a "complete" understanding might mean: is describing the program's function sufficient, or must one know exactly how the program accomplishes its task? In terms of code coverage, some situations may require comprehension of a small part of the program only, or a sketchy overview of the program, or the retrieval of a specific piece of information. Furthermore, the program may be so large that a complete understanding of it cannot be reached.

Indeed, many theories of program comprehension have attempted to cover "comprehension", as opposed to "comprehension of short Prolog programs" or "comprehension of full-scale C programs", or even "comprehension of a full-scale C program with a view to performing a highly specific change in the code". Enough attention has perhaps not been given to the parameters which might change the results observed, such as how markedly different languages interact with comprehension, or whether comprehension is embedded in a task and, if so, how task factors might influence comprehension.

To summarise, an approach to teaching novice comprehension based on process theories would require a unique, agreed upon theory which can chart the progression from novice to expert (including "buggy" sideshoots), and is detailed enough to account for variations in comprehension "setting", including language and task factors. This seems unrealistic, at least in the near future, which suggests that provisions for novice support might be more usefully examined from another angle.

2.7.2 Internal Representations

This section considers the relevance of internal representations of programming knowledge and mental representations of programs to supporting comprehension.

An obvious difference between novices and experts is the amount of knowledge they possess, in whatever form it is postulated to reside in memory. The implications of this on the comprehension process were discussed in the previous section.

According to Gilmore (1990), proponents of knowledge-based theories take the view that novice difficulties arise from a lack of knowledge, and that teaching programming involves a passing on of expert knowledge. Unfortunately, studies show that this is not the whole story: novices often have the necessary knowledge, but fail to use it when appropriate (Perkins and Martin, 1986). Gilmore calls for a more explicit focus on programming strategies, rather than static knowledge, a proposal which will be considered in the next section.

In contrast to programming knowledge, mental representations are more ephemeral in nature, and represent in some sense the end product of program comprehension: they are the internalisation, in some form, of an external program. Describing the exact nature and structure of internal representations is a difficult and thorny issue, namely because there is no direct access to them. It is often suggested that mental representations are neatly organised in memory, taking the form of a network of linked propositions or schemas. These structures sometimes seem more appropriate to machines than to humans: it can be hard to imagine, based on performance measures, that information is stored in such an organised fashion. Many theories of program comprehension make an implicit supposition that the mental representation is available in "read-off" form, with performance being simply an outward display of an internalised model. Very few, if any, consider that task performance might require the reconstruction of a model of the program based on fragments of information. Perhaps mental representations are more like external representations in that they comprise some information which is readily available, and some which can be inferred, for example, by applying general purpose reasoning mechanisms and prior knowledge.

In any case, there seem to be a number of reasons why mental representations might not be the ideal basis for novice comprehension support. Firstly, as mentioned above, there is a great deal of incertitude surrounding the form that they might take. Secondly, mental representations would need to be equated with performance measures in order to determine if one particular type of mental representation is "best" in particular comprehension situations (some researchers have attempted to do this, *e.g.* Pennington's description of cross-referenced mental representations (Pennington, 1987b), described in Section 2.8.2). Finally, it is unclear how one would go about teaching a novice to have a particular mental representation of a program.

2.7.3 External Strategies and Information Extraction

As described above, Gilmore (1990) maintains that differences between novices and experts have as much to do with programming strategy as with knowledge. He claims that experts are able to choose from a range of comprehension strategies based on the characteristics of the particular situation, including task requirements and the programming language in question. One outcome of their use of different strategies, according to Bellamy and Gilmore (1990), is their ability to make multiple representations of a program (Pennington, 1987b). In other words, experts have the ability to extract different types of information from a program.

Empirical evidence of this is provided by Holt *et al.* (1987), who found that the ability to extract information from the code was a crucial determinant in performing a modification task. Additionally, they found that student programmers are particularly affected by external factors such as the structure and content of the program when attempting to do so.

Focusing on the identification of information in the code as a basis for providing comprehension support for novices has a potential advantage in that it is easier to emulate expert behaviour in searching for information in a program, than to attempt to teach static expert knowledge representations to novices.

The concept of information types is interesting from this perspective in that, of the categorisations of the external features of a programming language which were examined, information types seem to offer the most comprehensive account of information present in, or derivable from, a program. The categorisation offers concepts at varying levels of granularity, and the function and control flow categories cover information which is central to many other theories, namely, *what a program does* and *how it does it*.

Information types would also appear to be language independent, in that concepts of function, data flow, etc. occur in most, if not all, programs, regardless of the language in which they are written.[4] In other words, they relate more to the semantics of a program than to its syntax. From a pedagogical point of view, this is important: Shneiderman and Mayer make the point that "it is apparently easier for humans to learn a new syntactic representation for an existing semantic construct than to acquire a completely new semantic structure" (Shneiderman and Mayer, 1979, p. 222). This would explain then, why the teaching of the first programming language is so crucial (or why the teaching of the second and following languages are less crucial, depending on how one looks at it). Providing support for information types, rather than for particular programming languages, focuses attention on semantic issues rather than on syntactic ones. In principle, this should allow students to build up a semantic structure which is transferable between languages.

Of course, information extraction occurs within the context of a strategy: it is important

[4] Although this would need to be corroborated with data from studies using a number of paradigms.

to know not only *how* to access the information one needs, but *when* to access it, and for what purpose. Although strategy will not be the specific focus of this thesis, it is acknowledged that strategic aspects must necessarily form a part of any well-rounded theory of comprehension, and would also need to be addressed explicitly in teaching situations.

2.7.4 Program Comprehension and Information Types: Summary and Comments

In summary, it is felt that information types offer a good basis for providing novice support: they are external, rather than internal, entities, which makes them easier to identify and manipulate. Furthermore, they cover a range of different types of information, and they focus on the semantic aspects of programs, making them language independent.

To date, information types have primarily, if not exclusively, been used to investigate programmer's mental representations, and so considering applying them to teaching raises a number of questions. One issue to be considered is that of paradigm: if support for comprehension is to be based on the idea of picking different types of information out of a program representation, then it is worthwhile knowing if differences in representation alter the difficulty of the information extraction process. Different paradigms stress different programming concepts, at least in theory. Even if information types are assumed to be language independent, there is likely to be an interaction between the information type and the programming representation. The effect of notation on information extraction was considered in detail by Gilmore and Green , who state that "the structure of a notation affects the ease with which information can be extracted both from the printed page and from recall" (Gilmore and Green, 1984, p. 47). Therefore, different programming paradigms, to the extent that they highlight different types of information at the expense of others, should play a role in the extraction of information types. This idea has been partly investigated by Ramalingam and Wiedenbeck (1997) in work that postdates the work described in the following chapter, however, more varied forms of data would serve to confirm their findings.

Note that the reason behind an investigation of paradigm and information types is not one of "naturalness" or attempting to find the "right" programming language. Petre and Winder (1988) make the point that when designing programs, experts devise solutions in a personal pseudo language. Once an algorithm has been developed, it is "translated" into the programming language to be used. Although this is a valid critique of studies which attempt to find "the" language which will solve all of a programmer's problems, it does not conflict with research on program comprehension: in design situations, the designer is at liberty to choose any design language which he/she deems appropriate or make up a personalised language. In program comprehension, a representation is imposed on the programmer, whose reasoning must start from that representation. If the representation had no effect on reasoning processes, then the match-mismatch effect (described in Chapter 4) would not be observed, and much less energy could be expended on finding suitable/efficient/useful representations.

2.8 Information Types in Empirical Work

This section describes a series of experiments based on the idea of Pennington's classification of information types described in Section 2.6.2. Although the aim of all of these studies was an investigation of mental representations, their use of information types as a basis for investigating comprehension is of interest. The work reported in this thesis looks strictly at the influence of programming paradigm on comprehension, and makes no claims about the nature of the programmer's internal representations. However, it uses the concept of information types in order to do so, and borrows from the methodology used in previous studies, therefore, they are described in detail below.

2.8.1 Pennington - Experiment 1

In a first experiment, Pennington gave 80 programmers (40 COBOL and 40 Fortran) short program segments to study. Between brief periods of study, they were required to perform one of the following tasks:

- answer yes/no questions corresponding to the various information types;
- write down as much of the program as they could remember (free recall);
- confirm whether they had seen a particular program snippet or not. These were primed, either with snippets originating from the same cognitive unit according

> to the text structure (TS) analysis of the program, or the plan knowledge (PK) analysis, where text structure knowledge is organised around control structure primes and plan knowledge is primarily functionally oriented.

Her results showed that TS primed items were answered more quickly, thus supporting a control flow view of the program. Furthermore, in terms of the information type questions, there was a significant difference between categories, with the lowest error rates occurring on operations and control flow questions. However, there are interesting differences across languages. The error rates (from lowest to highest) for the FORTRAN subjects were:

> operations – control flow – functional – data flow – state

while for COBOL programmers they were:

> operations – data flow – control flow – state – functional

In other words, it appears that COBOL programmers make fewer errors on data flow questions than on all other types of questions apart from operations. Additionally, their data flow scores are higher than those of the FORTRAN programmers. Thus, there seems to be an interesting effect of language, which Pennington addresses to some extent.

2.8.2 Pennington - Experiment 2

In Pennington's second experiment, she used the top and bottom quartile comprehenders from her first study (with equal numbers of COBOL and FORTRAN programmers).

Subjects studied a moderate length program (with half of the subjects asked to "think aloud" while they worked), then summarised the program and answered a series of comprehension questions. After performing a modification task (again, with half of the subjects verbalising), they summarised the program again, and responded to a second list of comprehension questions.

Pennington found that error rates on the comprehension questions preceding the modification task were lowest for control flow and data flow questions, and lowest for function and data flow questions after the task, particularly for the "think aloud" group.

In terms of the program summaries, Pennington was only able to analyse those obtained before the modification task, as once they had completed the task, "programmers tended to refer to their earlier summaries and then to concentrate on describing their modifications rather than giving complete program summaries as instructed." (Pennington, 1987b, p. 331).

The summary statements were categorised according to *type* (data flow, control flow, function), and *level of detail* (detailed program, domain, vague[5]). Pennington found a predominance of control flow statements (57%) compared to data flow (30%) and function (13%). The percentages in terms of level of detail were: program level, 38%; detailed, 18%; domain, 23%; vague, 21%. Comparing upper and lower quartiles, Pennington found that lower quartile subjects made more statements which were either more detailed, or more vague. In addition, the majority of functional statements were expressed in terms of the domain, while the majority of procedural statements were expressed in terms of program objects. As mentioned above, it was not possible to compare the summaries written before program modification with those written afterward to see if there was a change in statement type and level across summaries.

In a further analysis of the program summary data, Pennington (1987a) divided subjects into groups according to the proportion of statements at different levels of detail contained in their summaries, as follows:

- Program level summaries: mainly operational and program level statements;
- Cross-referenced summaries: containing a more even distribution over operations, program and domain levels;
- Domain summaries: containing a majority of domain and vague statements.

She found that the majority of the upper quartile comprehenders produced summaries which were cross-referenced, and that error rates were lower for the cross-referenced

[5] Vague statements are defined by Pennington as statements without specific referents, *e.g.* "this statement reads and writes a lot of files" (Pennington, 1987a, p. 105).

group. She concludes that best performance is associated with representations which are richly cross-referenced, utilising the relationships that hold between the real world and the program world.

One issue relating to this categorisation is however open to question. The domain summary group categorisation could potentially be skewing the results in favour of cross-referenced summaries. Domain summaries are those containing either domain or vague statements. Although Pennington does not give exact figures, the percentage of vague statements in the domain group is over 50, while they appear to be around 10% for the program and cross-referenced groups. Domain statements account for just over 20% of statements in the domain group, compared to over 30% in the cross-referenced group. Vague statements would seem, given the information provided, to be statements which are characteristic of a lack of understanding of the program. The existence of a *domain* group in which the predominant statement is of type *vague*, and the fact that the proportion of domain statements is lower than in one of the other (non-domain) groups, must surely have an unwanted effect on the results. If program level statements are characteristic of the first stages of comprehension, and functional understanding follows (and if one agrees that function and domain go hand in hand), then presumably a domain group should show good program understanding, as this would signify that they have reached the second stage of the situation model. It would be interesting to see the comparison if only a majority of domain statements characterised the domain summaries, rather than a majority of vague ones: it may well be that the performance of the domain group would have been found to be much higher.

Finally, Pennington analysed the verbal protocols obtained from half of her subjects, both at a propositional level, and in terms of episodes, which are formed by series of propositions. Of the episodes she identified, she reports on *connection* episodes in which a “trigger event” causes a hypothesis about the domain world to be formed. In looking at three subjects, one from each group, she found that the program level summary subject made almost no connections, staying at the level of program simulation, the cross-referencing subject made regular connection episodes, with attempts to relate the program text to domain function, while the domain summary subject shows numerous connection episodes, but these are 1) based on minimal triggers and 2) not verified.

Pennington concludes that the results provide evidence for a two-stage comprehension process, whereby a textbase and then a domain model are constructed.

2.8.3 Corritore and Wiedenbeck

Corritore and Wiedenbeck (1991) carried out two studies similar to Pennington's second study but using novice Pascal programmers. They were interested in finding out how novices differed from experts, and also whether the upper quartile performers were in any way like experienced programmers.

In their first experiment, they asked 80 subjects to study small programs, and then to write a program summary and answer five comprehension questions corresponding to the five information types. The program was not visible while subjects completed the two tasks.

Corritore and Wiedenbeck found significant differences between question types in terms of the number of errors made. The order of error rates by question, from lowest to highest, was: operations, control flow, data flow, function and state. They also compared the performance of upper and lower quartiles, and found that the overall pattern was similar, but that the greatest disparity between quartiles occurred for state and function questions.

Program summaries were classified in accordance with Pennington's categories, *i.e.* in terms of type (procedural, data flow, function), and level of detail (detailed, program, domain, vague). They found 50% procedural, 27% data flow, and 23% functional statements. No significant differences were found for quartile, nor was there an interaction between quartile and statement type, however, there was a trend for more functional statements in the upper quartile. In terms of level of detail, they found significantly more detailed and program statements (accounting together for over 80% of the statements) than domain or vague statements.

They concluded that the novice comprehension process works from the bottom up, starting with a detailed concentration on operations and procedural information.

In their second study, Corritore and Wiedenbeck looked at the performance of the upper and lower quartile comprehenders from the first study on a longer program (85 lines).

In terms of comprehension questions, they found increased errors overall, and greater differences between groups (for the lower quartile, the number of errors, from lowest to highest was: control flow, data flow, operations, state and function; for upper quartile subjects, this was: operations, state, data flow, control flow, function), with lower quartile subjects performing surprisingly badly on operations questions.

Examining the program summaries, they found that data flow statements made up the highest proportion of statements, followed by procedural and then functional. With respect to the lower percentage of procedural statements observed, compared to the first study, they hypothesised that novices "lost control" of the low level, procedural information as the program increased in size. In terms of level of detail, most statements were detailed. Corritore and Wiedenbeck hypothesise that the increased length of the program made it more difficult for novices to progress beyond a detailed, concrete account of the program, and that they therefore miss out on the function of the program. At the same time, they feel that the problems with operational questions might also derive from the increased program length.

Finally, Corritore and Wiedenbeck divided subjects into groups according to their summary strategies, as did Pennington. Rather than a predominance of cross-referenced strategies among the upper quartile comprehenders, they found that the majority of subjects in both quartiles tended to use a program strategy. However, more upper quartile than lower quartile subjects used a domain strategy. It should be noted however that a cross-referenced strategy, as defined in this research, is not necessarily one in which the different types of information are well integrated and coordinated: it may simply denote a procedural statement followed by a functional statement, without explicit links between the information.

2.8.4 Ramalingam and Wiedenbeck

Ramalingam and Wiedenbeck (1997) carried out a very similar study to that of Corritore and Wiedenbeck, but focused on how different languages might affect novices' mental representations. They gave C^{++} novices programs written in either an imperative or an object-oriented style. Again, subjects answered questions about the program corresponding to the five information types.

Overall, the error rate was higher on object-oriented programs than on imperative programs. However, they found quite striking differences between the two types of program. For imperative programs, the error rates per question, from lowest to highest, were: control flow, operations, state, data flow, function. For the object-oriented programs, the order was: function, data flow, operations, state and control flow.

On the basis of this data, the authors claim that the object-oriented group formed a domain model of the program, while the imperative group formed a program model. Interestingly enough, this seems counter to Pennington's two-stage theory, in that the object-oriented group appear to be developing a domain model in the absence of the program model on which she postulates the domain model is based.

Ramalingam and Wiedenbeck go on to claim that the object-oriented style is more "natural". While the evidence is certainly intriguing, the claim seems over-confident. The data comes from one source only: binary choice questions, as no program summaries were collected during the experiment. Additionally, given the higher overall error rate of the object-oriented group, it seems unlikely that this paradigm is more "natural" (assuming that such a question is a useful one to ask). Nevertheless, this study does point to the role of representation in program comprehension.

2.8.5 Bergantz and Hassell

Bergantz and Hassell (1991) used information types within a very different methodology in order to examine non-procedural languages, in their case, Prolog. Their research centred on a detailed analysis of a very small number of verbal protocols made by industrial programmers as they examined a medium length program with a view to modifying it. The protocols were analysed in terms of the frequency of occurrence of information types (their information types differed slightly in that they considered function, control flow, data flow and data structure[6]), as well as their temporal ordering. Bergantz and Hassell found a high frequency of function statements, and a shift from data structure to functional statements as comprehension proceeded. They concluded that this provided support for a two-stage model as postulated by Pennington. This could be debated, depending on whether one feels that data structure relationships do indeed reflect a procedural understanding of the program: an alternative interpretation

[6] The latter being defined by Bergantz and Hassell as "the type and number of program objects that are transformed during the course of program execution" (Bergantz and Hassell, 1991, p. 318).

is that understanding is differentially influenced by language. Replication of this study would be insightful, as only three subjects were studied, and differences in experience level between the three appear to have led to differences in performance. Nevertheless, this type of methodology, which takes an in-depth look at the process of comprehension, offers increased ecological validity compared to other studies.

2.8.6 A Summary of Information Types Studies

Table 2.1 provides an overview of the program comprehension research carried out using the information types approach. The research is characterised and compared along a number of factors. Note that because the methodology used by Bergantz and Hassell (1991) was so different from that used by the studies reported above, it could not usefully be included in the comparative table.

Characteristics	**Studies**				
	Penn1	Penn2	C&W1	C&W2	R&W
Subjects	experts	experts	novices	novices	novices
Language(s)	Cobol Fortran	Cobol Fortran[7]	Pascal	Pascal	C^{++} (imperative or OO style)
Program Length (in lines)	15	200	12-15	85	"brief"
Comp. Questions	yes	yes	yes	yes	yes
Free Recall of Program	yes	no	no	no	no
Recognition test	yes	no	no	no	no
Prog. Summaries	no	yes	yes	yes	no
Verbal Protocols	no	yes (half)	no	no	no
Modif. Task	no	yes	no	no	no

Table 2.1: A Comparison of Experimental Work on Information Types

2.9 Chapter Summary

This chapter proposed a generic model of program comprehension, and considered a number of theories of program comprehension and empirical work in terms of the components of the generic model. It then considered the implications of each approach for providing support for novice comprehension, and called for a focus on the external factors affecting program comprehension. It described reasons why the idea of information types might be useful in this context, and highlighted unanswered questions, namely the issue of how information types interact with paradigm in comprehension. Finally, previous empirical work on information types was reviewed.

The next chapter reports on an experiment which looks at the retrieval of information types from a very different programming language from the ones used in previous work: Prolog, which is based on the declarative paradigm.

[7] Although differences between languages were not compared.

Chapter 3

Interactions between Comprehension and Programming Language

3.1 Introduction

In the last chapter, a review of the results from experiments by Pennington (1987b) and Corritore and Wiedenbeck (1991) suggested that program comprehension could be characterised essentially in terms of control flow and low-level operations (at least in the initial stages of comprehension). This was evidenced both by responses to program comprehension questions and written summaries of the program.

However, the point was also made that this largely control flow-oriented view of comprehension stems from studies which used control flow languages such as FORTRAN, COBOL and Pascal to test theories of comprehension. In that sense, the notation used may have had a confounding effect on comprehension, in other words, comprehension of control flow relative to other types of information was best because the programming language being used was, by definition, highlighting the control flow aspects of the program.

One question which stems from this research is the extent to which the style of language interacts with the way in which programs are comprehended. It is quite likely that there is no generic method for comprehending a program, and that it will be dependent, at least in part, on the language paradigm in which the program is represented.

The experiment described in this chapter investigates this issue using a language which is radically different from traditional procedural languages. A non-procedural language was chosen (Prolog), and an experiment broadly similar to that of Corritore and Wiedenbeck's was carried out.

3.1.1 Claims for and against Prolog

Prolog is a declarative language based on predicate logic. As such, it is very different from languages such as Pascal or COBOL. Clocksin and Mellish (1981) state that it is *descriptive* as well as *prescriptive*, in other words, programming in Prolog is as much about stating what is known to be true about a given problem, than it is trying to find a step-by-step solution to the problem, a point echoed by Kreutzer and McKenzie (1991). According to Clocksin and Mellish, the Prolog interpreter is dependent only in part on the control flow information which the programmer specifies.

Early proponents of Prolog stressed the beneficial nature of Prolog's declarativeness, particularly for novices. In an introduction to Prolog for teachers, Ennals (1981) considers some of the many advantages of Prolog as being easily understandable by "ordinary people", and a good representational medium for real world problems and information. He stresses that users are able to write and understand simple programs immediately, and, in his words, quickly become "promoted" to programmers. Kowalski (1979) shares this view: he states that logic based computer languages are both machine independent and human oriented, in contrast with conventional languages, which express the behaviour which the machine is expected to manifest. For this reason, he feels that logic based languages are, "easier to construct, easier to understand, easier to improve, and easier to adopt to other purposes' (Kowalski, 1979, preface).

Similar claims were made by Baron *et al.* (1985) for non-procedural languages in general. They feel that non-procedural languages are more appropriate, and even "natural" for problem solving, as "any problem can be stated in terms of the constraints to be satisfied" (Baron *et al.*, 1985, p. 127). There is an implicit assumption that if one focuses on the structural aspects of the problem, the computer will somehow take care of the procedural aspects.

The strength of many of these claims was later tempered, both by empirical work and by the experience of teaching Prolog: novices were found to have immense problems learning Prolog. Apart from the complexities of trying to translate problem specifications into a formal, logical language, many novices found Prolog's execution baffling, particularly the use of backtracking and cuts. Taylor (1988) claimed that many novice problems with Prolog could be attributed to two sources: the logical structure of the language, which means that many conceptual difficulties and problems of expression in logic are also present in Prolog, and execution, as novices find Prolog's automatic execution mechanisms such as backtracking difficult to control. Indeed, many Prolog "stories" (Pain and Bundy, 1987), such as AND/OR trees or Byrd box models, and programming environments focused on execution, as Dichev and du Boulay (1988) point out. However, van Someren also describes difficulties in matching data structures, or *unification* (van Someren, 1990a,b), and Dichev and du Boulay's data tracing system for Prolog also suggests that novices have difficulties in areas other than execution.

3.1.2 Implications for the Study

The use of Prolog as the vehicle for a program comprehension study has various implications: firstly, the fact that it differs substantially from procedural languages should mean that different types of information are made more salient (or conversely, are obscured). Secondly, the claims for and against Prolog will lead to different predictions about the pattern of responses one might expect to observe: these are considered in turn below. Before doing so, however, it should be noted that the most specific claims tend to refer to program construction rather than comprehension, even though most authors also contend that Prolog programs are easy to understand. Additionally, many of the claims are quite general, and slightly vague. They do not map directly onto information types, therefore some speculation is called for.

Based on the claims in favour of Prolog, one might expect to find higher error rates for control flow, given that novice programmers will have been taught to focus on the declarative aspects of the program, and leave the procedural aspects to the interpreter. Errors for operations questions should remain low, if only because by definition they focus on small segments of code (one line or less according to Pennington (1987b)).

Likewise, it could also be hypothesised that performance on data flow questions will be quite good, as the presence of explicit arguments to "hold" data, and the lack of multiple assignment to the same variable, could mean that data flow is more clear and hence better understood by novices.

Finally, the notion of state is relatively straightforward in Prolog in the sense that predicates can, in principle, be understood in isolation from other predicates. However, given the small size of the programs used in the experiment (usually only one predicate), this definition of state is not really applicable. State seems to have been defined on a more narrow scale by Pennington to correspond to a time-slice description of the program, *e.g.* "when procedure X is executed, does variable Y have a value?". Using this definition of state, it is unclear how performance on state questions will differ between Prolog and procedural languages.

If, on the other hand, one bases predictions for performance on the claims against Prolog, one might also expect an overly high error rate for control flow questions, given difficulties with backtracking and other control structures. One might expect similar problems with data flow questions: matching data structures, particularly in recursive programs, is known to be problematic, therefore, this may offset any benefits for data flow questions as described above. Assuming that novices come to an understanding of the program's function via comprehension of lower level structures, then problems with control flow and data flow may well have negative repercussions for performance on function questions. Following this line of reasoning then, one might expect a high overall error rate, with even higher error rates on control flow, data flow and possibly function questions relative to the overall mean error rate. In terms of the program summaries, one should find a different pattern of distribution of statements of each type, with control flow statements less frequent than in previous studies.

To explore these issues, Corritore and Wiedenbeck's methodology was taken and modified for use with Prolog. Additionally, because the focus of the experiment was on the effect of different programming languages on performance, rather than an attempt to capture novice mental representations of programs, there was a corresponding shift in emphasis away from memory based tasks to tasks which required searching through the representation for the necessary information.

3.2 Method

3.2.1 Design

The experiment was a within-group, repeated measures design. Participants were presented with five different types of comprehension questions: function, data flow, control flow, operations and state questions. The questions (and the programs to which they were referred) were presented in random order across subjects.

3.2.2 Subjects

Seventy four subjects took part in the experiment. All subjects were first year undergraduate students at the University of Edinburgh, and were nearing the end of a half year course in Artificial Intelligence (AI1Bh) in which Prolog had been taught. They had received approximately ten lectures on Prolog, and had completed various programming exercises, most of which made use of recursive constructs.

3.2.3 Materials

The experiment took the form of a paper and pencil exercise. Six programs were used (the first being a practice program). All programs were between 6–12 lines long and all were recursive. The programs used were chosen because they implemented different types of recursion and included reasonably complex passing of values between arguments. In everyday use (with Prolog and with other languages), meaningful predicate and variable names provide information about the program's function *e.g.* `reverse(List, ReversedList)`, making it relatively easy to construct a "pseudo-functional" account of the program (*e.g.* "This program takes a list, reverses the order of the elements and returns a reversed list). In order to avoid a potential functional bias, predicates and variables were purposely given meaningless or ambiguous names.

Each program was accompanied by five comprehension questions, corresponding to the five information categories identified in (Pennington, 1987b). These questions were presented in random order, and followed by a request for a summary of the program. In order to be consistent with Corritore and Wiedenbeck's study, this question was functionally oriented[1], asking subjects to "write a summary of what this program does".

A sample problem used (along with a line showing how it is called) is shown below, while Table 3.1 shows the accompanying comprehension questions, answers and relevant information types:

```
?- adjust([1, 3, 2, 7], Res).


adjust(X, R):-
       adjust_sub(X, 0, R).


adjust_sub([], _, []).
adjust_sub([X|Xs], Y, [Z|Zs]):-
        Z is X + Y,
        adjust_sub(Xs, X, Zs).
```

Type	**Question**	**Response**
data	Is the variable Y always set to 0?	No
ops	Is Z initially instantiated to 0?	No
control	Does the program recurse over all of the elements of the list?	Yes
state	When X is instantiated to 3, is the value of Y equal to 2?	No
function	Does this program total the numbers in the list?	No

Table 3.1: Comprehension Questions and Correct Responses for Sample Problem

The programs were shown one to a page, along with the accompanying comprehension questions and request to summarise the program.

A self-report questionnaire was also devised to determine subjects' prior programming experience and familiarity with other programming languages.

A full listing of the instructions to subjects, all problems, accompanying comprehension questions, and the self-report questionnaire can be found in Appendix A.

3.2.4 Procedure

The experiment was carried out during a one hour practical session. Subjects were given a packet containing a short description of the experiment, instructions, a practice

[1] Corritore, personal communication.

problem, the remaining five programs in random order and the self-report questionnaire.

After reading the description and instructions, subjects were given five minutes to complete the practice problem. Subjects then had the opportunity to ask questions of clarification before the experiment began.

Subjects had five minutes to study each program, answer the accompanying questions and write a program summary before being asked to turn to the next program. As mentioned above, because the experiment was designed to look at the effect of different programming languages on performance rather than on their mental representations, it was decided that programs should be visible to the participants while they were answering the questions and writing the program summaries. If subjects finished before the five minutes had elapsed, they were instructed to wait before turning the page until told to do so. Likewise, subjects were requested not to return to previous questions.

Finally, subjects were asked to fill out the questionnaire on programming knowledge.

The experiment lasted approximately 40 minutes.

3.3 Results

3.3.1 Programming Experience

On the self-report questionnaire, 80% of subjects reported knowing three or more languages[2]. 5% of the subjects knew only one language (Prolog).

Of the subjects, 58% had studied a programming language in school (prior to starting university), with Basic, Comal (a language with a Pascal-like structure added to Basic) and Pascal being the most popular languages. 73% of the subjects were studying another language in addition to Prolog at university level: C was by far the most studied language. All subjects apart from the four who knew only Prolog had some knowledge of a procedural language.

[2] Note that the questionnaire asked them to rate their level of understanding at either a novice/intermediate/expert level, so their "knowledge" of a language may not have been more than a passing familiarity in some cases

3.3.2 Comprehension Questions

Each subject answered five questions about five programs (25 questions in total). Each question represented one information type (FUNC, CF, DF, OPS, STA), therefore, there were five separate measures for each information type.

Figure 3.1 shows the percentage of errors per information type. The fewest errors occurred for control flow questions (CF), followed by operations (OPS), state (STATE) and function (FUNC), with data flow questions (DF) having the highest rate of errors.

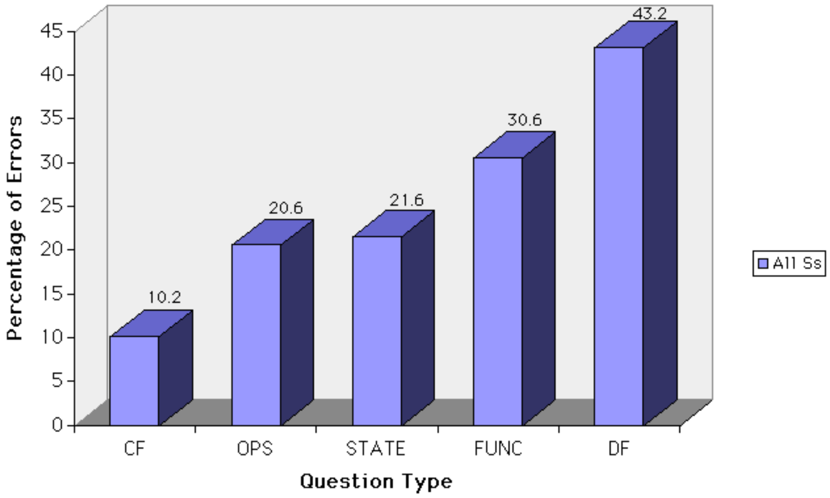


Figure 3.1: Percentage of Errors by Information Category

Table 3.2 shows the mean score (out of 5) for all subjects per question type, along with standard deviations.

Ss (n=74)	Question Types				
	FUNC	CF	DF	OPS	ST
Mean	3.47	4.49	2.84	3.97	3.92
Std Dev	1.08	0.67	1.18	1.10	1.06

Table 3.2: Mean Scores and Standard Deviations for the 5 Comprehension Question Types

A one-way ANOVA for repeated measures revealed that there was a significant difference in the number of errors between information categories (function, control flow, data flow, operations, state), F(4,292)=35.10, $p < .001$. Post-hoc pairwise comparisons between mean errors were made using the Bonferonni adjustment as recommended for within-subjects repeated measure designs in (Tabachnick and Fidell, 1983). Therefore, only probabilities of less than .005 were considered to be significant.

Table 3.3 summarises the results of the paired t-tests.

Info Type	DF	FUN	ST	OPS	CF
DF	-	t=-4.31 p<.001	t=-7.00 p<.001	t=-7.53 p<.001	t=11.51 p<.001
FUN	- -	- -	t=-2.75 ns	t=-2.83 ns	t=7.14 p<.001
ST	-	-	-	t=.36 ns	t=4.49 p<.001
OPS	-	-	-	-	t=4.43 p<.001
CF	-	-	-	-	-

Table 3.3: Results of Pairwise Comparisons

There was a significant difference between data flow questions and questions relating to all other information types, and between control flow and all other types. The remaining comparisons (function-state, function-ops, and state-ops) were not significant.

In line with Corritore and Wiedenbeck (1991), the scores of the lower and upper quartile subjects were examined separately to investigate whether there were differences in the pattern of response. Table 3.4 shows the mean number of errors per question type for subjects in the top and bottom performing quartiles, with standard deviations.

Quartile	**Question Types**				
	FUNC	CF	DF	OPS	ST
Upper Quartile (n=18)					
Mean	4.50	4.78	4.17	4.56	4.78
Std Dev	0.62	0.43	0.79	0.62	0.55
Lower Quartile (n=18)					
Mean	3.00	3.94	1.83	2.72	3.00
Std Dev	1.24	0.87	0.99	1.13	1.19

Table 3.4: Mean Scores and Standard Deviations for the 5 Comprehension Question Types: Upper and Lower Quartile

A comparison of upper and lower quartile mean percentage of errors is shown in Figure 3.2.[3] Standard error bars have not been included for reasons of readability, however,

[3] Note that a histogram would be, strictly speaking, more appropriate for categorical data, however, it is felt that this method illustrates the relationship between information types both within and across groups in a more salient manner

standard deviations for mean scores on question types for the upper and lower quartiles can be found in Table 3.4.

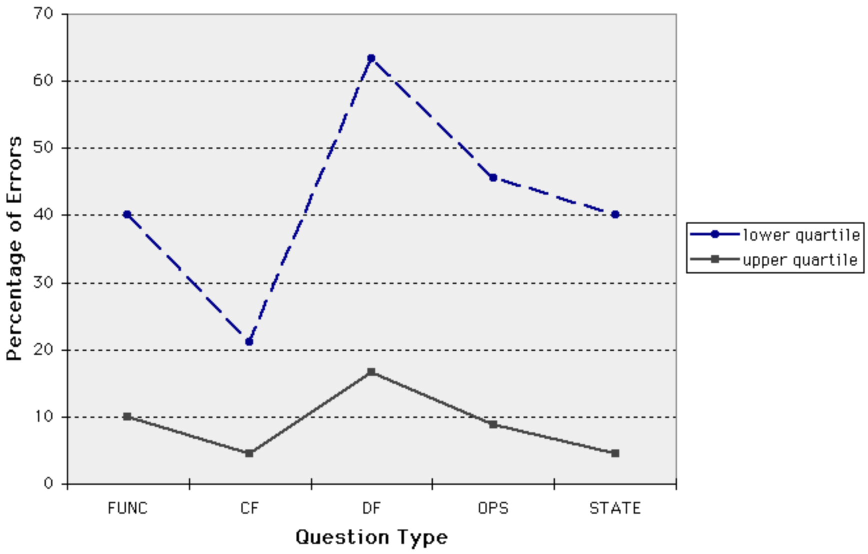


Figure 3.2: Percentage of Errors by Information Category (Quartiles)

The two quartiles show roughly the same response pattern. Compared to upper quartile subjects, lower quartile subjects make relatively more errors on data flow questions compared to other question types. On control flow, in contrast, lower quartile subjects' performance approaches that of the upper quartile subjects (Table 3.4).

A one-way ANOVA for repeated measures performed on each group (lower, upper quartile) revealed a highly significant difference in the number of errors between question types for the lower quartile (F(4,68)=7.99, p $<$ .001), and a significant difference for the upper quartile (F(4,68)=2.70, p $<$ .04).

None of the post-hoc pairwise comparisons between question types were significant for the upper quartile. For the lower quartile, control flow and data questions had the lowest and highest number of errors respectively. Therefore, the means on control flow questions and data flow questions were compared against the means of all other question types, again using the Bonferonni adjustment. Both comparisons were significant (control flow comparison: t=5.41, p $<$ .001, data flow comparison: t=-4.92, p $<$ .001).

3.3.3 Correlation between Prior Programming Experience and Performance

Measures of previous programming experience were correlated with performance on the experiment. Given that the questionnaire was relatively simple, much of the information collected was not deemed to be discriminating enough for further testing. However, two measures were chosen for investigation: the number of languages known, and self-ratings of "expert" for knowledge of any programming language.[4]

There was a highly significant correlation between number of languages known and scores on functional questions (r_s = .33, p < .005).

Likewise, a self-rating of "expert" correlated positively with total score (r_s = .27, p < .03) and with scores on functional questions (r_s = .32, p < .005).

Neither measure (number of languages known or expert rating in a language) correlated significantly, either positively or negatively, with any of the other question types.

3.3.4 Program Summaries

Analysis of the program summaries presented a number of problems, related primarily to the fact that the methodology used in previous studies was not reported in sufficient detail to allow it to be replicated. A new methodology was required, in particular, a detailed coding scheme for analysing program summary statements. The methodology which was developed is presented in Chapter 8, along with a full discussion of the replicability issue. The results of the analysis for this study are provided in Chapter 9.

3.4 Discussion

The results of the experiment are surprising in many ways: participants scored highest on control flow and operations questions, while data flow and function questions were the most problematic. The results were broadly similar for the upper and lower quartiles: control flow questions showed the lowest rate of errors, with data flow showing the highest. Therefore, a "procedural bias" seemed to manifest itself as much in this experiment as in previous experiments which used procedural languages.

[4] Later versions of the questionnaire, *e.g.* the one used in Chapter 7, collected more finely-grained information, allowing for a number of other correlations.

Furthermore, when examining scores within quartiles from the upper and lower quartile subjects separately, there were no significant differences between question types for the upper quartile. On the other hand, for the lower quartile subjects, control flow errors were significantly lower than all other types, while data flow errors were significantly higher, suggesting that programming skill is an additional factor.

At first sight, these results seem to provide evidence for the procedural mental representation postulated by Pennington, at least in the initial stages, and exhibited by novices (Corritore and Wiedenbeck, 1991).

From an educational point of view, findings such as these seem to suggest that the choice of teaching language (particularly, the first language) isn't as important as might be thought: it doesn't seem to influence the type of information on which people focus to a great extent. Seen from the perspective of curriculum planning, this is reassuring: choice of language will not "damage" students in some way, or prevent them from reaching a general understanding about programming which goes beyond the language in which a program is implemented. From the representational point of view, the results are more complex: there seem to be other factors at work apart from the process of attending first and foremost to the particular information that is highlighted by the representation in question.

Given the scale and the scope of the experiment, it seems wise to consider possible alternative explanations for the results observed, and the implications these might have. These are discussed in the following sections.

3.4.1 Impurities of Prolog

As discussed in Section 1, the advantages for novices of Prolog's declarative paradigm and its novel conceptualisation of what defines a program were much touted. However, empirical work caused novice difficulties with Prolog to be well documented. These difficulties may result in part from the fact that a declarative reading of Prolog is not always sufficient to produce well formed and efficient programs.

For example, students beginning Prolog are often introduced to the "family database" in the first week, with a very neat declarative reading of the relationships which hold between family members. Shortly after this, they begin to learn that programming in a declarative way does not always produce the desired results, and that, for example, incorrect clause placement in one's first recursive program to find a person's ancestors may cause the program to loop indefinitely. The focus thus shifts to control flow, or how to lay out programs in such a way that things happen in the desired order. *Cuts* only reinforce this emphasis on control flow. The lack of explicit control flow in the program may have a paradoxical effect: rather than allowing students to not worry about procedural details, it may force them to focus on it even more and to highlight it as a central issue so as to avoid unwanted behaviours and outcomes. Therefore, Prolog novices may already have expended much effort on understanding control flow to the point where it is not problematic for them.

Data flow questions are a different matter. In one scenario, one could claim that data flow is in some ways easier to capture: arguments are explicit, and named in all clauses, and one can follow the flow of data by tracing argument names through the program. Apart from "assert" and "retract" operations, inputs and outputs are clearly visible. However, this would not explain the difficulties with data flow which were observed.

The control flow argument (that because control flow is difficult, novices spend more time getting to grips with it) breaks down when applied to data flow: if it is similarly difficult, then why would one observe low rates of errors for one and not the other?

A speculative answer to this question concerns the interrelatedness of data and control flow. Firstly, the control flow in the programs used in the study was quite straightforward: no cuts were used, and execution required almost no use of backtracking. The different types of recursion used in the programs (*e.g.* non-tail recursion, embedded recursion) were the only aspects of control flow which were potentially problematic, and admittedly, recursion has long been recognised as being conceptually very difficult for novices to master.

When discussing data flow earlier, it was stated that difficulties with data flow manifested themselves in unification, or a lack of understanding about how data structures matched with each other. Misconceptions about unification will be exacerbated when programs are recursive, as recursive programs often involve extensive unification in order to build up an output list. In fact, George (1996) contends that many difficulties commonly thought to be associated with recursion are in fact a result of other difficulties, among them, misunderstandings about variable updating. He gives an example of 'delayed invocation update' in which students erroneously think that the return of control to a suspended process causes a variable's value to be changed to the parameter value of a previous invocation.

In the experiment reported here, it may be that the design of the programs and questions were such that, on the one hand, the control flow questions were tapping into general sequencing information about reasonably simple execution patterns, rather than specifics about the working of recursion. On the other hand, the data flow questions may have been tapping into the difficulties with understanding recursion, via questions which asked about unification at various points in the recursive process, thus leading to a high frequency of "data flow" errors.

3.4.2 The Interaction between Prolog Structure and Information Types

In addition to Prolog's "impurities" or difficulties, there is an issue of how the surface structure of the Prolog language might interact with information types. Green (1989) first coined the term "cognitive dimensions" to refer to a framework for categorising various cognitive aspects of a wide range of notations, including programming languages. In an application of cognitive dimensions to Prolog, Green (pear) makes a number of points which are applicable to program comprehension and/or construction.

One of the cognitive dimensions most relevant to Prolog is *role expressiveness*, which, as Green and Petre (1996) put it, describes how easy it is to answer the question, "What is this bit for?". Role-expressiveness will influence the degree of difficulty involved in breaking a program down into its component parts and determining the relationships between parts. Role-expressiveness is presumed to be high when a notation includes features such as meaningful identifiers, beacons, and grouping mechanisms (with the latter including modularity and secondary notation). Green starts from the premise that comprehending a program involves parsing the program back into its "cognitive components", and draws on theories of natural language parsing to argue that parsing relies on lexical cues.

According to Green, programming in Prolog involves combining a relatively limited set of notational elements in a variety of ways, as opposed to other programming languages which have a larger initial vocabulary. The result is that programs with very different behaviour may look remarkably similar, because they contain the same basic elements, arranged differently (a point which will be considered again in Chapter 9).

This implies that the lack of lexical cues, such as control flow keywords or the use of indenting, will make it harder to parse a Prolog program than a similar program in another language, a hypothesis which was borne out in an unpublished study by Green. The implication for comprehension is that Prolog may be based on a different semantic model of programming than Pascal or Basic, but Prolog's surface structure does not make these differences salient in a way which allows for easy retrieval.

Prolog's lack of role expressiveness would explain why it is generally found to be difficult to understand, but there may also be an interaction between role expressiveness and information types. Given that information types are a way of describing information which is present in the code (but may need to be inferred), then it may be possible to link various "roles" with the information types they signal. One might hypothesise, firstly, that languages containing role expressive features which can be mapped to a given information category should facilitate the retrieval of that information type from the program, and secondly, that information types with the greatest number of role distinctions (within limits, of course) in a single information category may likely facilitate retrieval. Given that Prolog will have a very limited number of roles per category (or perhaps none at all in some cases), then one would expect to find that the extraction of information types from a Prolog program is difficult (and perhaps more so, according to the information type). The idea of matching roles to information types is purely speculative however, and would require further thought and testing.

3.4.3 Procedural Tainting

Another possible explanation for the results obtained is in terms of "procedural tainting": only four of the seventy four students had not learned any other language than Prolog, and for the other seventy, at least one of the languages learned had been procedural.

Furthermore, for those students who knew more than one language, those who learned Prolog before a procedural language were quite rare, if not non-existent[5]. Certainly all

[5] It is impossible to tell with precision, as students who reported that they taught themselves a language did not record *when* that language had been learned.

students who had learned Prolog in secondary school were also learning a procedural programming language concurrently.

These factors may have led students to regard Prolog through a procedural filter, and to focus on procedural information as a priority. The fact that novices bring a certain amount of unwanted baggage from other languages with them to new languages is well documented (an account of this can be found in (van Someren, 1990a), particularly with respect to transferring from Pascal to Prolog). It may be that since students are primarily exposed to procedural languages, they bring this bias with them to Prolog, attempting to map Prolog onto a procedural superstructure, with the result that this information is looked for as a first step.

Unfortunately, finding subjects who have learned a declarative language but not a procedural one is extremely difficult, and will probably become more so.

3.4.4 Teaching Practices

Another issue is the question of teaching practices, or what the dominant culture, in this case the university, decides is important. In a post-experiment questionnaire administered to students who took part in a later study on program comprehension and information types (Good and Brna, 1998a), students remarked on what they were *taught* to attend to, and the types of responses that lecturers were looking for on open-ended questions, or in program comments. Students taking joint degrees in computer science and artificial intelligence (68% of those taking part in the study), noted that they were taught in computer science courses to explain exactly *how* a program worked, thus slanting their perception towards a procedural view of the program. Prolog teaching may not be immune from this either: students can (and do) use meaningful predicate and variable names to fudge a reasonable description of the program, therefore they are often encouraged to provide evidence of in-depth understanding by explaining the *how* of the program in addition to the *what*.

3.4.5 Novice Prior Experience

When looking at novice programmers in a particular language, it is worth remembering that they often have some experience in other languages. Although the self-report questionnaire provided only a glimpse of subjects' prior knowledge, it is interesting that both the number of languages known and subjects' self-ratings of "expert" in any language correlated positively with both their overall score, and their score on functional questions. This suggests that subjects with more experience were better able to infer the function of the program. These results could also lead to speculation as to the relationship between level of knowledge and direction of processing (*e.g.* perhaps a bottom-up strategy is being used, but only subjects with more programming experience reach the stage of integrating low-level information into a more abstract, functional view), but more investigation would be needed in order to test this hypothesis.

3.4.6 Ease of Answering

One issue which does not seem to have been considered by either Pennington or Corritore and Wiedenbeck is the relative ease with which each of the questions can be answered. It may be that questions are easier to answer not because the information required to answer the questions happens to have been stored as the dominant mental representation, but simply because reconstructing the answer to the question from the mental representation of the program (whatever it may be), requires fewer steps.

It seems likely that one's mental representation of a program is not a single entity, but a loose bag of concepts and ideas in some form, and that answering a question, or trying to write the program from memory, involves a reconstruction process comprising a variable number of steps, an idea which is shared by diSessa (1988).

This issue is slightly different for the current study, due to differences in methodology: Corritore and Wiedenbeck's study was very much dependent on memory, with students required to use what they had remembered about the program to try and answer questions about the program. The present study, where the program was always in view, was more focused on the search aspect: students could search through the program to identify the relevant part(s) of the program with respect to the question. However, this may in fact make the problem easier to address: it is more straightforward to identify the necessary reasoning steps on an external representation than it is to hypothesise what might be the state of the student's mental representation and the steps needed to formulate an answer to a given question.

The most pertinent example here is that of *operations*, which according to Pennington

(1987b) correspond to a single statement or less. Answering a yes/no question on operations would seem to entail identifying the appropriate line of code. Questions which focus on data flow in a program may involve identifying several points in the code where a data object has gone through a transformation, and attempting to reconstruct the entire transformation process in a coherent manner, a process which seems cognitively much more intensive. Therefore, questions which have low rates of errors may reflect more about the complexity of the steps involved in answering the question.

3.4.7 What is Programming?

This is a fundamental question which is at the heart of this research. The question centres on the nature of programming, or more importantly, the ways in which programming is conceptualised. Do people, particularly novices, perceive programs as active entities which dictate how to *do things*, and in what order to do them? Or do they look at programs as being descriptions of an overall goal, with data being transformed in order to fulfill this goal?

Pair (1990) would maintain the former, claiming that most people equate programming with describing calculations. He goes on to say that programming for most beginners involves mental execution, or a mental image of all of the calculations involved, rather than abstract function definitions. Likewise, research by Eisenberg *et al.* (1987) on procedures in Scheme, which are in fact "first-class objects", suggests that novices were very reluctant to consider procedures as objects. Rather, they saw them as active entities which are ready to operate on static data, or as the authors so dramatically put it, "as bundled-up 'computational energy' waiting to be unleashed" (Eisenberg *et al.*, 1987, p. 20). Although this desire to see procedures as active entities caused comprehension problems for the novices, it may be a deep-seated one, with its antecedents in everyday life.

This point is nicely illustrated by Pennington (1987b), who describes a recipe, normally thought of as a set of procedural instructions complete, in some cases, with parallelisation ("while the custard bakes, make the raspberry coulis ..."), in terms of data flow. In the case of a recipe for fettucini carbonara, the cheese moves out of the refrigerator, is grated, and divided into two parts. One part goes to the table, while one goes into an egg mixture. Likewise, the noodles go into a pot of boiling, salted water, etc. (one can imagine a description of the egg transformation from raw to beaten to cooked ...). The point about this is that while a data flow description of cooking is an *accurate* description of the changes in the ingredients over time, it is not necessarily a *useful* one for the cook.

The same may be true in programming: while data objects obviously undergo transformation, from a cognitive point of view, it makes more sense to think in terms of the actions which accomplish this. The benefits in doing so may be so great that changing the representation of a program (*e.g.* from procedural to declarative) will not change the conceptualisation of what a program is. Research on query languages shows that using a procedural language allows subjects to write difficult queries more easily than with a nonprocedural language (Welty and Stemple, 1981). There is also evidence to support the view that requiring people to think procedurally can help them to solve algebra word problems (Soloway and Ehrlich, 1982), although it is not clear whether procedural thinking refers to using a procedural paradigm or simply the act of programming itself. Furthermore, this effect has not been consistently observed (Olson *et al.*, 1987).

3.5 Chapter Summary

This chapter reported on a study which investigated the relationship between programming paradigm and comprehension, using the concept of information types in order to measure comprehension. The study used a methodology very similar to that used by Corritore and Wiedenbeck (1991), but looked at Prolog, a declarative language, in the place of a procedural language. It was shown that even when a non-procedural language is used, a procedural bias still seems to exist, with students showing lower error rates for procedural and operations questions. Reasons for why this might be so were discussed, including particular features of the non-procedural language, issues of teaching context and experience, prior knowledge, cognitive factors, and questions about the nature of programming itself. An analysis of the program summary data, obtained by asking students to write a free-form account of the program, is presented in Chapter 8, and suggests that novice comprehension is more subtle than their answers to the comprehension questions might imply.

One issue not considered in the chapter was the "match" between the language and the information being represented. While it might be assumed that procedural languages highlight procedural information in some way, it is not clear which types of information are highlighted by Prolog, as there is no direct match between information types and declarativeness. Better results could likely be achieved by investigating languages which are more closely tied to one or more information types: this issue is described in the following chapter.

Chapter 4

The Match-Mismatch Conjecture and Visual Programming Languages

4.1 Introduction

The experiment reported in Chapter 3 used information types to look at a language with an underlying declarative paradigm. In Chapter 6, we will examine data flow and control flow visual programming languages (VPLs) using the match-mismatch conjecture. The two studies have similar objectives in that they aim to shed light on the role played by the programming paradigm in comprehension. This goal of this short chapter is to provide a methodological and theoretical bridge between the experiments by explaining shifts in the methodology and focus of the studies, their empirical basis and underlying theory, and any consequences arising from these decisions. The chapter following this one will review the literature relevant to the experiment to be described in Chapter 6 in more detail, and describe the design of the materials.

The Prolog study described in Chapter 3 highlighted a number of experimental and methodological issues, one of the primary ones being that generic claims for the benefits of a language do not necessarily lead to testable hypotheses. When investigating the ways in which the declarative paradigm (embodied here in Prolog) might influence the extraction of information types during program comprehension, it appeared that claims about Prolog's benefits on the one hand, and results from empirical studies of Prolog on the other, led to very different hypotheses, and that these hypotheses were both difficult to reconcile and to refine to a degree suitable for testing.

Furthermore, it became clear that it is difficult to compare results from two studies (the Prolog study and the study by Corritore and Wiedenbeck (1991)), even if they share a very similar methodology. Unless the two experiments are administered by the same person who pays careful attention to minimising the differences between studies, they are likely to differ in ways which, however subtle, will have some effect on the results. In addition, a difference in the overall focus of the study may lead to changes in the procedure. In this case, the subject of interest moved from the programmer's mental representation of a program and a verification of the two-stage theory of program comprehension to an interest in the effect of the representation on program comprehension. Not using exactly the same experimental materials will lead to a reinterpretation of the principles the materials are designed to embody, for example, deciding exactly what data flow *is* in practical terms, and how it can best be measured.

To summarise these points:

- there is a need for testable hypotheses about information extraction from programming languages, and these hypotheses are difficult to derive from general knowledge about a language;
- there are problems with comparing results across studies;
- a main focus of interest in this research is in the effect of a programming paradigm on comprehension.

Given the above, it seems sensible to work within a framework which allows for the formulation of hypotheses relating more directly to representation and comprehension, and in which within-study comparisons of representations can be made. The match-mismatch conjecture, put forward by Gilmore and Green (1984), offers a way of addressing these needs.

The conjecture, which will be explained in detail in Section 4.2 along with related work, has certain requirements. The primary condition is the use of two (or more) relatively constrained languages which vary only along the dimension of interest, so as to avoid possible confounds. Given that it would be virtually impossible to find two commercially available languages which meet this criterion, micro-languages can be designed and implemented, a solution which has been used in previous studies (Green, 1977; Gilmore and Green, 1984). Section 4.3 discusses the use of micro-languages with respect to the ultimate aim of providing comprehension support for novices, and considers how the requirements of the experiment and the aims of the support system might be reconciled.

4.2 The Match-Mismatch Conjecture

4.2.1 Development and Previous Work

The match-mismatch conjecture first appeared in a paper by Gilmore and Green (1984), where it was claimed that "the structure of a notation affects the ease with which information can be extracted both from the printed page and from recall" (p. 47). Green *et al.* (1991) provide a restatement of the hypothesis as follows: "the conjecture is simply that performance is best when the form of the information required by the question matches the form of the program" (p. 125). The general concept that fitting the representation to the task is beneficial for problem solving is widely accepted, and many common examples of the match (or otherwise) between representation and task are nicely illustrated by Norman (1993). Although the idea may seem intuitively obvious, it has little explanatory or predictive power when expressed at this level of abstraction, and must therefore be defined in operational terms. This has been done in practice by Green and his colleagues, whose empirical work has tended to focus on precise notations and comprehension tasks.

The match-mismatch conjecture grew out of work by Green (1977) on different representations of conditional statements in procedural micro-languages. Gilmore and Green (1984) extended this work to investigate various competing hypotheses about program comprehension, namely 1) that some languages are universally "cognitively unwieldy", and would hence lead to comprehension difficulties across tasks; 2) that programs are translated into a unique mental representation, and that some languages make this translation process easier than others, or 3) that the mental operations required by certain tasks are made easier or harder by some notations (the so-called match-mismatch effect).

Gilmore and Green (1984) investigated two types of textual notations: *procedural* notations, which can be thought of as having an "if then" structure, and *declarative* notations, which they feel are more similar to production systems. Based on Green's previous work on procedural languages, they hypothesised that procedural notations facilitate access to sequential information (*e.g.* "when action X is performed, what might the next action be?"), while declarative notations highlight circumstantial information (*e.g.* "in what circumstances will action X be performed?"). They further hypothesised that matched pairings, *i.e.* a procedural program with a sequential question or a declarative program with a circumstantial question, will lead to better performance than unmatched pairings. This hypothesis was largely supported by the data, at least for the questions they considered.

The match-mismatch conjecture was then extended to visual languages in an experiment which looked at textual and visual representations of procedural and declarative micro-languages, pairing them again with sequential or circumstantial questions (Green *et al.*, 1991; Green and Petre, 1992). A match-mismatch effect appears to have been found, both overall and for the textual and graphical languages.

Moher *et al.* (1993) replicated this study using Petri Nets as VPLs. They found that the match-mismatch held for the textual representation, but not for the graphical representation, suggesting that there are perhaps other factors at work.

In a parallel line of research, the concept of "cognitive fit" has been investigated by Vessey and her colleagues in a number of domains (Vessey and Weber, 1986; Vessey, 1991; Sinha and Vessey, 1992). Sinha and Vessey describe cognitive fit as an emergent property of a model for problem solving. The model "views problem solving as the outcome of the relationship between the problem (or external) representation and the problem-solving task, which are characterized for the purposes of this analysis by the type of information each emphasizes" (Sinha and Vessey, 1992, p. 369). A match between the information highlighted in the representation and task leads to the use of similarly matching problem solving processes (and a matching mental representation). In the mismatch case, the mental representation will be formed either on the representation or on the task. In either case, mental transformations of the information will be necessary in order to solve the task.

The cognitive fit hypothesis has been applied to various domains: Vessey (1991) compared the use of tables and graphs for symbolic versus spatial reasoning, based on the hypothesis that tables support analytic processes while graphs support perceptual processes. Support for the cognitive fit theory was found based on analysing the studies reported in the tables and graphs literature.

In the psychology of programming field, Vessey and Weber (1986) carried out an experiment similar to that of Gilmore and Green (1984), looking at circumstantial and sequential questions using structured English, decision tables and decision trees. Decision trees were best for circumstantial questions, while decision trees and structured English were equally good for sequential questions. Sinha and Vessey (1992) later applied the notion of cognitive fit to the domain of recursion and iteration. Using Lisp and Pascal as the programming languages in their investigation, they found the language itself had a much stronger effect on subjects' performance than did cognitive fit. This study differs from the match-mismatch studies of Green and his colleagues in that it looked at coding rather than comprehension: it may be that this change in scenario introduces a number of other variables which dilute the match-mismatch or cognitive fit effect.

The concept of cognitive fit is interesting, firstly, because it has been extended to consider not only the fit between the problem representation and task, but also between these factors and the problem solving tool. Secondly, it has looked at both comprehension and construction tasks in various fields, *e.g.* program understanding and program coding.

4.2.2 Match-Mismatch, Information Types and Paradigms

Chapter 3 showed that looking at differences in language paradigms through different information types 'lenses' is not straightforward. New paradigms tend to be accompanied by general claims for the benefits of the language, rather than claims which are specific, and more importantly, testable. These general claims do not relate to information types in ways which allow predictions to be made: more detailed statements are necessary.

One possible way of tackling this problem is to investigate paradigms which could be said to relate more directly to information types. Pairing a paradigm with its corresponding information type should make it possible to formulate more specific hypotheses as to the relative benefits of the pairing. Furthermore, looking at two paradigms in the same study avoids the difficulties of comparing results from two separate studies (as was the case with the Corritore and Wiedenbeck (1991) study and the study reported in Chapter 3). Within-study comparisons of programming paradigm could eliminate to a large extent extraneous variables, and allow one to ensure that conditions are similar for both groups.

Looking at the match between paradigm and information type is effectively a practical restatement of the match-mismatch conjecture, although it makes various assumptions about the nature of the relationship between paradigm and information type. Pennington's work on information types stemmed from her notion of *program abstractions* (Pennington, 1987b). Program abstractions are created by analysing a sample program in terms of one type of information. The abstractions can be represented graphically, with the resulting representation highlighting one set of relationships present in the program while obscuring others. These other relationships are nonetheless derivable, *e.g.* by looking for repeated data object names, data flow can be inferred from a control flow abstraction. Pennington stresses that the abstractions are based on formal analyses developed by computer scientists.

In order to compare two paradigms based on information types, it is necessary to make the assumption that a given paradigm acts in some way as a program abstraction, in other words, that a paradigm highlights the information type corresponding to it. This is very much an assumption, even if it appears to make intuitive sense. Therefore, in a language based on control flow, it is assumed that control flow information will dominate over other types of information. Likewise, a language based on the data flow paradigm is assumed to make data flow information easier to access. Of the five information types, those which seem to be most amenable to a comparative study are control flow and data flow: functional languages would appear to rely on a definition of function in terms of inputs and outputs, while the information types definition describes function as a high level account of the program's overall goals. Similarly, the information types definition of state does not necessarily correspond to the definition of state as embodied in a paradigm such as a state transition machine. Finally, operations could be contrasted with a higher-level representation of information, but it is not immediately obvious which paradigm could provide a higher-level counterpart to a representation of operations, or for that matter, which existing paradigm would best embody operations.

It was therefore decided to develop an experiment to test the match-mismatch conjecture using representations of data flow and control flow paradigms: this experiment is described in Chapter 6. The conjecture was tested on visual micro-languages, and the justification for doing so is described in the next section.

4.3 Visual Micro-Languages, Match-Mismatch and Novice Comprehension Support

The last section described the match-mismatch conjecture, and mentioned the fact that examinations of the conjecture have almost invariably used micro-languages or, in the case of the studies reported in (Green *et al.*, 1991; Green and Petre, 1992), a notational subset of a pre-existing language (two alternative graphical representations available in LabVIEW).

The reasons for doing so are quite clear: finding two commercial languages which vary only on the dimension(s) of interest would likely be impossible. Furthermore, even if they did exist, it would be difficult to find a subject population with equal prior experience of each language. Circumventing this by using completely novice subjects would mean teaching them the two languages, again, a prospect which is not feasible in a typically short experimental timescale. In addition, many properties of a full-scale language are not necessary in an experimental setting, and may in fact be a hindrance. Therefore, it was decided to develop micro-languages for experimental use, as had been done in the past. Criticism has been levelled at micro-languages for being too far removed from full-scale languages to allow for any comparison, however, this is a conscious trade-off: experimental setups necessarily lose in realism in the hope that control over extraneous variables present in real-life situations will allow the experiment to establish a causal relationship between independent and dependent variables.

The micro-languages which were designed for the research reported in this thesis will be described in detail in Chapter 5. The languages were designed with a dual purpose in mind: firstly, to be used to test the match-mismatch conjecture, and secondly, to be used as the basis for testing out ideas for novice comprehension support. One important implication of this dual usage was the decision to make the languages visual, rather than textual.

Visual programming languages are an interesting subject of study in their own right, particularly given the gulf between the claims and the empirical results. In addition to this primary interest, it will be argued that the properties of visual programming languages may make them more suitable to the addition of support of the kind envisaged in this thesis than textual programming languages. This section first defines visual programming languages and visual programming, looking at some of the associated claims and empirical work, before going on to describe why visual programming languages might provide a good basis for information types support.

4.3.1 Visual Programming and Visual Programming Languages Defined

The terms "visual language" and "visual programming" have been used and misused, referring to a variety of environments and activities. At their most contentious, they have been used to describe Microsoft's Visual Basic and Visual C++, neither of which would be considered to be VPLs by purists, given that the underlying languages are textual.

Some authors use "visual programming" in its widest sense, such as Shu, who defines it as "the use of meaningful graphic representations in the process of programming" (Shu, 1988, p. 9). This definition includes environments allowing the visualisation of programs, execution, data, information and system design, and languages which process visual information, support visual interactions and allow programming with visual expressions (Shu, 1988). Others prefer a more restricted meaning: Myers (1986) uses the term to describe any system allowing a user to specify a program in more than one dimension. He excludes textual languages from this definition by virtue of the fact that they are processed by a compiler as a one-dimensional stream.[1] For the purposes of this thesis, "visual programming" will be taken to mean the use of visual expressions in the process of program construction and modification. Visual expressions are defined as graphics, drawings and/or icons which must be meaningful, executable and involve some notion of control and/or data flow.

According to this definition, the following will not be considered to be visual programming: the use of visual program specification and/or design tools, because they are

[1] Although it is not obvious to the author that his definition, as it stands, explicitly excludes them.

used *before* writing the program, software visualisation tools and graphical tracers, because they are used *after* writing the program, programming languages for building graphical user interfaces (GUIs), programs for visualising data or information about data (*e.g.* graphical database query languages) or programming languages for handling visual information, *i.e.* images.

4.3.2 VPLs: Claims and Empirical Evidence

The belief that graphical representations can have a positive effect on programming and learning to program is a widely held one. Claims have often been made that visual programming languages are at least a positive step forward, if not downright revolutionary, and this belief is evidenced by a plethora of visual programming languages and environments. The claims which have been made for visual programming are often rather optimistic, and seem to be preoccupied with a hypothesised "brain underuse", for example:

> "Various reasons have been cited for the interest in visual programming. Many of them pertain to the better use of the right half of the brain, which is needlessly at rest and underutilized for the purpose of computing." (Shu, 1988, p. 1)
>
> "Visual programming languages allow a programmer to use a conceptual model that is close to her own mental model." (Golin, 1991, p. 5)
>
> "The human visual system and human visual information processing is clearly optimized for multi-dimensional data. Computer programs, however, are presented in a one-dimensional textual form, not utilizing the full power of the brain." (Myers, 1986, p. 60)

In a more realistic, *i.e.* verifiable, vein, claims have been made that visual programming languages will be easier to use than their textual counterparts because visual representations support forward and backward reasoning (Anjaneyulu and Anderson, 1992; Trafton and Reiser, 1991), act as a memory aid (Merrill *et al.*, 1993), and make certain structures, *e.g.* control and/or data flow, more apparent (Cunniff and Taylor, 1987).

However, experimental studies to ascertain the benefits of VPLs have met with mixed results. For example, studies comparing a visual language with a textual one were carried out by Anjaneyulu and Anderson (1992), who found no significant differences in performance between the two, and Pandey and Burnett (1993), who found that program generation in the visual language Forms/3 was “at least as easy” as in the text-based OSU-APL and Pascal languages. Whitley (1997) has provided the most thorough summary of empirical studies using VPLs to date, and carefully considers the evidence both for and against them. Although she calls for more experimental work in order to provide evidence of the benefits of visual programming languages, one reason a clear picture may not yet have emerged with respect to visual programming languages has to do with the scope of the conjectures. Some researchers seem to want to make the claim that visual languages are simply “better” than textual languages, without considering differences across users, across tasks, and the interaction between the two. It is likely though that these and other factors play a role, and the graphical nature of visual programming cannot be considered in isolation.

However, one might also question whether what is missing from the field are more finely-grained studies, or whether visual programming languages will simply fail to show promise. The next section takes a different look at VPLs: rather than asking whether they are “good” in and of themselves, it asks whether VPLs might be well suited to providing the basis for the forms of novice comprehension support envisaged in this thesis.

4.3.3 VPLs: Possible Advantages for Support

One point which has often been made, perhaps less by VPL designers than by those interested in their cognitive properties, is that there is no such thing as a purely textual or purely graphical programming language. Fitter and Green (1979) made the more general distinction between perceptual and symbolic coding, pointing out that a purely symbolic or purely perceptual notation would not be very usable. Thus, textual languages are not simply a string of symbols: they make use of layout, indenting and other spatial mechanisms by which information is conveyed. In the same way, visual languages contain text, such as keywords, variable names, program names, etc. Rather than existing in separate camps, textual and graphical languages could more accurately be depicted as occupying distinct locations on textual and spatial continua.

The focus on the spatial characteristics of graphical notations is evidenced in Petre and Green's concept of 'secondary notation' (Petre and Green, 1992, 1993). They describe it as "the use of layout and perceptual cues which are not formally part of the notation (elements such as adjacency, clustering, white space, labelling, and so on) to clarify information (such as structure, function or relationships) or to give hints to the reader" (Petre and Green, 1993, p. 57), and postulate that it might be the main distinguishing characteristic of graphical representations. Raymond (1991) also provides a very interesting view of the specific case of layout, using Goodman's (1976) distinction between notational and analog languages. According to this theory, a language is considered to be a notation if it exhibits a number of properties, one of which is finite syntactic differentiation. Non-notational languages violate one or more of the properties. For example, one of the features of analog languages, a particular class of non-notational languages, is that they do not exhibit syntactic differentiation, in other words, they are syntactically dense. Notational systems can be characterised by discreteness, while analog systems are characterised by density. Raymond maintains that a visual programming language's "visualness" derives from its layout, which is an analog (*i.e.* syntactically dense) property.

However, the textual/spatial distinction doesn't seem to capture all of the differences between textual and visual languages: textual languages obviously contain text, but graphical languages contain more than simply spatial layout, by definition they contain graphical symbols as well, for example, icons, nodes and arcs. And although textual languages may make use of spatial layout, they tend not to contain graphical symbols. This is not to imply that previous authors have ignored this, simply that it hasn't received much attention, perhaps because it seems so evident.

This does suggest that rather than describing textual and visual languages as overlapping entities, with text and spatial layout as shared characteristics, it might be more accurate to see graphical languages as a superset of textual languages, containing graphical characteristics (*e.g.* icons, nodes, arcs) in addition to textual and spatial ones.

This recharacterisation is of interest from the point of view of deciding whether textual or graphical languages might provide a better platform for novice program comprehension support based on information types. If textual languages contain text and spatial layout, while graphical languages contain graphical symbols, text and spatial layout, then graphical languages will provide not only more dimensions, but more relationships between dimensions. As such, the number of different types of information they could potentially represent is greater. Given that the goal is to use a single base representation on which various types of support can be overlaid (not simultaneously of course, but through the use of selective hiding or highlighting), then the number of different dimensions is of great relevance.

The graphical, textual and spatial characteristics of a VPL could be used to express different types of information at any given time. However, another advantage is in allowing for redundant recoding (Fitter and Green, 1979), in other words, expressing the same information in more than one way. The example the authors give is the use of indentation to represent the nestedness of conditional structures in the program. The program code provides this information through the use of keywords, *e.g.* `begin`, `end`, `if`, `then` and `else`, and therefore, although indenting is not strictly necessary, it makes a big difference in terms of program readability. In a graphical language, redundant recoding, represented symbolically and spatially above, can also be represented graphically. The same keywords and spatial layout may be used, but now, for example, conditional constructs may also be signaled by an icon of a distinct shape (and usually colour) containing the conditional keywords. Given the difficulties which novices have in picking out the relevant information in a program, one might argue that redundant recoding is even more important.

In some ways, hypothesising that graphical representations offer more possibilities for expressing information may seem to run counter to the theory of specificity put forward by Stenning and Oberlander (1995). According to these authors, many graphical representations exhibit *specificity* in that they allow the expression of some, but not all, abstractions. In a situation where several models are possible, a single graphical representation can typically only represent one of these (*e.g.* imagine a representation of the sentence "my dog's fur is either snow white or covered in mud"). However, special conventions can be introduced into the representation for handling abstraction or indeterminacy. Graphical representations differ in this respect from non-graphical languages such as natural or logical languages (or computer languages for that matter) where abstraction and indeterminacy are more easily expressed. The benefit of weak expressiveness is its cognitive tractability, and the authors maintain that this, rather than the visual nature of graphical representations per se, makes graphical representations relatively easy to process.

However, VPLs are different from many other types of graphical representations considered by Stenning and Oberlander (1995), in that they tend to be based on semantic networks. The authors consider semantic networks to fall at the most linguistic end of the graphical continuum: as such, they enforce few specificities. Visual programming languages bear little resemblance to pictorial representations: their origins have tended to be textual languages, and early versions have sometimes been little more than textual languages encased in boxes. They use graphics to convey abstract ideas such as recursion, iteration, conditionals, etc. and as such, make heavy use of abstraction. Although their corresponding high level of expressivity may explain to some extent the mixed results which have been observed in empirical studies of visual programming languages, it does not weaken the argument that graphical representations could be of use in representing different types of information for novice program comprehension: if anything, it suggests that their powers of abstraction are similar to that of a textual programming language, and that, in addition, they possess additional dimensions onto which various types of information can be mapped.

4.4 Chapter Summary

This chapter covered various issues pertaining to the relationship between the experiments reported in Chapters 3 and 6. It described why the idea of information types on its own makes it difficult to look at the issue of paradigm, and why a methodology which allows for comparison between paradigms within a single study is more appropriate. It discussed the match-mismatch conjecture within this context, looking at prior empirical work, before going on to examine the consequences following on from this choice: namely the development and use of micro-languages. It discussed the additional role which micro-languages could play in testing ideas about program comprehension support, and why graphical languages might provide a better basis for novice comprehension support than would textual languages.

Chapter 5

Control and Data Flow Visual Programming Languages

5.1 Introduction

This chapter defines the notion of data flow and control flow paradigms, describes the characteristics of both types of language, and summarises the differences between them. It then charts the development of control flow and data flow visual programming languages from a historical point of view, providing examples of each. This is followed by a short discussion of empirical studies which have focused on these paradigms.

The second part of the chapter outlines the design of the micro visual programming languages used in the experiments described in this thesis. In the first VPL experiment, described in Chapter 6, the languages were designed around the *type* of information they provided (control or data flow) and the *format* of that information (either trees or graphs). Later experiments, one of which is described in Chapter 7, concentrated solely on the informational aspects of the programs, and used graph-based data and control flow languages. These languages differed slightly from the graph languages in the first experiment, partly as a result of feedback from the first experiment, and partly due to advances in locally available technology: the revised languages will be described in Section 5.6.2.

Before defining data and control flow paradigms, it is worth mentioning that representations are rarely purely data flow or control flow. For one thing, the concepts are interrelated: the flow of data is often governed by some notion of control, either in the form of conditionals or iterative structures, and program control habitually involves the utilisation and transformation of data objects. Furthermore, visual programming languages must be executable, therefore, they cannot exclude information for the sake of purity. However, even though each representation contains elements of the other, one type of information will predominate. Hence, *data flow* representations will be taken to mean representations which *highlight* data flow over control flow, rather than representations which include only data flow information.

5.2 Control Flow Languages

5.2.1 A Definition of the Control Flow Paradigm

Imperative, or control flow, languages were the first programming languages, and were based on the von Neumann model. According to Agerwala and Arvind (1982), this model has two main features: a global addressable memory and an instruction counter. The instruction counter provides the machine with a sequence of instructions to execute, acting as a single locus of control. The memory is a vast collection of storage locations which hold program and data objects, and its contents are updated by program instructions during execution. Wadge and Ashcroft (1985) describe data as being fetched one by one from memory and sent to the CPU. A computation is performed and data are then returned to their locations. Data is therefore normally "at rest".

The development of imperative languages was heavily influenced by the von Neumann architecture. Statements representing commands (or imperatives) are executed sequentially. Variables are used to represent memory locations, and assignment acts to change the values of variables. Therefore, programmers using imperative languages must necessarily concern themselves with issues of control such as memory allocation and variable declaration.

A distinction is sometimes made between procedural languages and imperative languages, *e.g.* Jenkins *et al.* (1986) maintain that procedural languages supplement imperative languages with means for expressing control constructs such as selection and iteration. They state further that procedural languages offer the power of a von Neumann architecture without the difficulty of having to specify the details of each particular computer. Many commonly used languages today are procedural languages, such as Pascal and C.

However, the terms *procedural* and *imperative* are used interchangeably in many programming textbooks, and where distinctions are made, they are often not consistent across texts. Given the limited subset of programming concepts which will be covered by the experiments described in this thesis, the distinction is not important, therefore *imperative*, *procedural* and *control flow* paradigms will be taken to refer to the same concept.

5.2.2 Graphical Representations of Control Flow

Control flow representations, in the form of flowcharts, have a long history. In fact, Chapin (1970) cites von Neumann as being the "intellectual father" of flowcharting. Given this history, it is understandable that flowcharts were designed around imperative programming languages as an aid for understanding control flow. Furthermore, their early appearance in the history of programming accounts for the fact that, rather than being executable representations of the program, they were designed to accompany textual programs, and to serve as a support mechanism for design or debugging, or as documentation.

There are many formalisms for flowcharts. These result, in part, from the efforts of companies to "make their mark" by developing conventions which were best suited to their own purposes, and which distinguished them from their competitors at the same time. However, most flowcharts share some general characteristics, and have been well documented in texts such as (Chapin, 1970).

Flowcharts consist of nodes linked together by arcs. Arcs represent the flow of control between nodes. As flow of control occurs in a sequential manner, there is only ever one arc between two nodes. Nodes represent tests and actions on data. Test nodes contain questions which can be of a boolean or a case variety. Boolean tests will contain two output arcs (depending on whether the test is true or false), while case tests will have as many output arcs as there are cases. Data in flowcharts is represented textually, usually as variables within nodes, rather than by any graphical symbol. There is an obvious temporal sequence to control flow, as only one arc can be followed at any time. An example of a simple flowchart can be seen in Figure 5.1.

Flowchart conventions have varied over time because of advances in software architecture which in turn changed the nature of programming. Researchers in the field

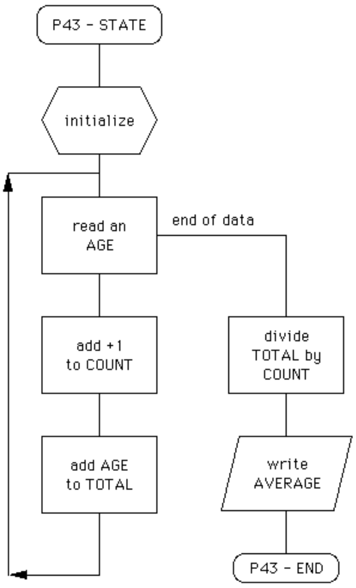


Figure 5.1: An Example of a Flowchart, from Chapin (1970)

attempted to provide improved, and more efficient notations for flowcharts, *e.g.* Nassi and Shneiderman (1973). Rather than using arrows to connect boxes, they redesigned flowcharts so as to embed a series of processes within a single box. However, they were still envisaged as aids to programming, rather than programs in their own right.

5.3 Data Flow Languages

5.3.1 A Definition of the Data Flow Paradigm

Data flow languages are in stark contrast to the traditional von Neumann approach in that they share few of the latter's characteristics. They are defined by Davis and Keller (1982) as any applicative language which is based on the idea of data flowing between function entities.

From a technical, implementational point of view, Agerwala and Arvind (1982) state that data flow can be distinguished from control flow in that it has neither a global updatable memory nor a single instruction counter. The lack of a global memory means that data flow models deal with values, rather than addresses. The instruction counter is not needed as instructions are enabled when all of the required input values are available. Thus there are no sequencing constraints other than the ones imposed by data dependencies in the algorithm. Similarly, the lack of instruction counter means that there is no single locus of control, rather there are many locally controlled events.

In the data flow model, computation can be represented as a directed graph, with each node in the graph representing a function. Nodes are linked by arcs, which represent the flow of data between functions, and hence, the data dependencies between nodes. Tokens are units of data which flow along arcs from the output port of the node which produced it to the input port of the node which requires the value. When a node has all the values it requires, it becomes enabled and can then fire. It therefore consumes the data input(s) arriving on its input arc(s), executes (absorbing the inputs), and places an output token on each output arc. These tokens are then sent to other operators which require these values. Therefore, there are as many arcs linking nodes as there are objects which are produced, used and/or transformed by the nodes.

An example of a data flow graph for calculating $X^2 - 2 * X + 3$ can be seen in Figure 5.2. The "phantom" nodes (X and Result) indicate input from and output to other programs.

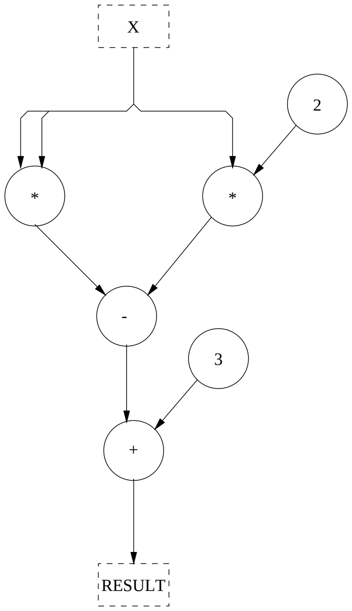


Figure 5.2: An Example of a Data Flow Graph, from Davis and Keller (1982)

Dennis (1986) summarises data flow execution succinctly in the following rules:

1. an node is enabled *iff* there is a token on each of its input arcs;

2. an enabled node may be fired. This determines the next state of the computation;

3. when a node fires, a token is removed from each of its input links, and a token is placed on each of its output links.

There are no other side-effects and no sequencing constraints apart from those which, as mentioned above, result from the data dependencies of the program. Ackerman (1982) describes a number of additional properties of a data flow representation:

- Locality of effect, in other words, the effects are limited in scope. Davis and Keller (1982) make the point that subprograms can therefore be understood *in vacuo*, without the need for information about the program's environment.

- An "equivalence of instruction scheduling constraints with data dependencies" (Ackerman, 1982, p. 15), meaning that program control is synonymous with data dependencies. A node fires when the input data it requires is available, rather than because it is instructed to do so by a centralised control mechanism.

- A "single assignment" convention, meaning that a variable may be assigned a value only once in that part of the program in which it plays a role.

- A lack of history which results from a lack of state variables. As no data is retained between invocations, the procedures do not "remember" in some sense. This can be problematic for computations which rely on values other than the current input values.

In contrast to the idea of "resting" data in the von Neumann approach, which are called from memory only when needed and then replaced, data are dynamic, and are processed while in motion, *i.e.* while they flow through a data network (Wadge and Ashcroft, 1985).

Given the definition of data flow, *i.e.* tokens of data flowing through a graph composed of nodes and arcs, it is difficult to dissociate a data flow language from its graphical representation. In fact, Agerwala and Arvind (1982) state that data flow languages were defined at the outset as graphical languages. Likewise, an article by Davis and Keller (1982) explored the advantages of graphical representations for data flow programs, with a view to "dispensing entirely with the text and viewing the graph itself as a program" (p. 27).

5.3.2 Augmented Data Flow

A *pure* data flow model is defined as one in which there are no added control flow constructs. There is no specified execution order and a node fires when all of its inputs are available. Pure data flow models are suitable for some types of computation, however, in many cases, it is necessary to introduce the idea of "selective routing of data tokens" (Davis and Keller, 1982), in other words, to include conditional structures, a common feature of procedural languages. This involves "augmenting" the pure data flow model. In the case of conditionals a two-step process is used, whereby a boolean token produced by a node which implements a decision procedure is passed to one of two selective routing nodes, *selectors* or *distributors*. These are described in (Davis and Keller, 1982) as follows: *selectors* accept a true or false data token, and use it to determine which of two inputs will be propagated along the output arc. *Distributors* use a true or false token to pick an output arc on which to place their data. Similar structures can also be found in other data flow models: Dennis (1986) talks of *T-gates*, which pass an input through if the value is true (and absorb it if the value is false), and *switches*, which pass their input onto the output arc which is designated by the control value. Similarly, many of the data flow visual programming languages in use today have either implemented variations on selectors and distributors, or invented special nodes to deal with decision procedures.

Likewise, control mechanisms such as iteration and recursion can be implemented in data flow languages in various ways. For example, recursion can be implemented as a directed acyclic graph, with iterative constructs expressed in terms of tail recursion (Dennis, 1986). One node in the graph, referred to in some models as the "apply" node, but usually having the same name as the entire graph, will cause a new copy of the program graph to be created. Davis and Keller (1982) caution that the implementation of the program should ensure that these expansions of a node into a subgraph do not carry on infinitely, although this applies equally to recursive programs in textual languages. Iteration has also been added to many data flow languages, often using a special construct, such as Show and Tell's iteration box (Kimura *et al.*, 1990).

5.4 Historical Development of Data and Control Flow

From a historical point of view, control flow and data flow graphical representations developed in very different ways.

Control flow based languages were the first dominant form of programming paradigm. The technology of the time dictated that programming languages be expressed textually. Control flow representations, or flowcharts, developed in conjunction with textual, control flow languages. The primary purpose of the flowchart was to supplement a textual language and to serve as an aid for understanding during various programming processes, such as design, debugging, etc.

On the other hand, data flow graphical representations developed much later, probably around the late 1970s. Davis and Keller (1982) made the point that the very nature of data flow made it amenable to being represented exclusively by graphs. Data flow representations were designed as actual programs, rather than to be used as documentation or as a comprehension aid for a textual program. Thus, data flow languages are, almost by definition, graphical languages: Lucid (Wadge and Ashcroft, 1985) is a rare example of a textual data flow language.

Interestingly enough, the few commercial visual programming languages widely in use today are based on the data flow paradigm, such as LabVIEW (Santori, 1990) and Prograph (Cox and Pietrzykowski, 1990). Control flow representations do not seem to have made the transition from textual language aids to stand-alone visual programming languages, as evidenced by the lack of commercially available control flow VPLs.

5.5 Empirical Studies of Representations of Control and Data Flow

The historical differences between control flow and data flow graphical representations have also carried over into the empirical studies which have examined their use. Studies of flowcharts have investigated their utility as an aid to constructing or comprehending a textual program, while data flow studies have investigated graphical data flow representations as languages in their own right.

The question under investigation in flowchart studies has typically been, “Does the flowchart provide support for program comprehension?" where the program was, for example, a Pascal or Basic program. Data flow studies, in contrast, have tended to concentrate on the graphical/textual comparison, although there are admittedly few studies which have been carried out, and the data flow issue has often been peripheral to the study's main aims.

This section describes some of the empirical studies which have been carried out with graphical control and data flow representations, before concluding with a summary of the differences between the two and some cautions on the conclusions which can be drawn from these studies.

5.5.1 Empirical Studies of Visual Control Flow Languages

This section considers two types of control flow studies: those in which a graphical control flow representation was used in conjunction with a textual language for some part of the programming process, and those where the control flow representation took the form of a visual programming language and, hence, was used as the sole source of information.

Flowchart Studies

There is a wealth of studies on flowcharts, investigating their usefulness as an aid to a language, and also comparing them to other forms of representation, such as pseudocode, design languages, etc. The results of these studies have been mixed.

A study by Ramsey *et al.* (1983) compared flowcharts against program design languages for producing a program design and for translating a design into an implementation in PL/1. They concluded that program design languages were superior to flowcharts, based on the quality of the designs produced. However, they found no differences in program comprehension or in the properties of the implementation (including their quality). The authors concluded that flowcharts may have an adverse effect in the design phase as they force designers to adopt space saving measures such as abbreviations, thus compromising readability.

A series of studies by Shneiderman *et al.* (1977) are often cited as damning evidence against flowcharts, and have been given particular credence since they investigated the use of flowcharts in conjunction with several aspects of programming: composition, comprehension, debugging and modification. The authors claimed that producing or using flowcharts in addition to a program listing did not offer significant advantages in any of these circumstances. However, their results seem cursory in some ways given the number of experiments they carried out and the large amounts of data which they must have generated. There may well be issues lurking in the data which have not been directly addressed, for example, familiarity: in a study requiring the use of flowcharts by two groups of subjects, one which normally used them and another which didn't, they found that the groups not used to using flowcharts performed worse when they were given a flowchart. However, the group which had used flowcharts in the past performed substantially better when they were allowed to use flowcharts (both when compared to their score without flowcharts, and to the non flowchart group's score without flowcharts).

Other authors, such as Scanlan (1989), have stated that this work may also suffer from methodological flaws which make it difficult to draw clear-cut conclusions, and certainly to make the sorts of generalisations which Shneiderman *et al.* did. Brooke and Duncan (1980) point out that because flowcharts were used in parallel with program listings in Shneiderman *et al*'s studies, it is not clear whether subjects actually consulted the flowcharts in all cases, particularly given the fact that subjects were experienced programmers who may have been content simply with the program listing.

A study carried out by Scanlan (1989) aimed to address some of these methodological problems. A comparison of structural flowcharts and pseudocode showed significant advantages for flowcharts, particularly as the algorithms increased in complexity, a finding which echoed that of Wright and Reid (1973). Scanlan's conclusions are very much pro graphics, but again, he sees flowcharts as "graphical documentation", rather than as executable program representations *per se*.

However, methodological difficulties do not in themselves seem to explain the differences between findings, at least for program debugging. Brooke and Duncan (1980) found no differences between a flowchart and a program listing for the correct identification of bugs, a finding similar to Gilmore and Smith (1984), who compared flowcharts with program listings and Bowles structure diagrams. Gilmore and Smith suggest that flowchart utility is not a clear-cut issue, and provide an interesting framework for determining performance in studies of this type which takes into account the features of the program itself, the context (*i.e.* task factors), and the programmer's individual characteristics.

Studies of Control Flow Visual Programming Languages

FPL FPL (First Programming Language) is a procedural, visual programming language developed at Columbia University, where it has been used, in tandem with Pascal, as a teaching tool for novice programmers. Programs are represented through the spatial arrangement of eleven different icons representing actions. The authors maintain that FPL "provides a visual map of the program that directly emphasizes its logical structure" (Cunniff and Taylor, 1987, p. 116).

A study of program comprehension was carried out to compare FPL with Pascal. Comprehension questions were designed to test for the ability to recognise simple structures, control flow and input/output, and to make simple evaluations based on the latter two.

Reaction time and accuracy of response were measured, and it was found that FPL reaction times were significantly faster than Pascal. This difference was most marked in questions dealing with the evaluation of control flow and/or input and output. Accuracy was also better using the FPL program segments. Although control flow errors were the most frequent type of error in both languages, fewer occurred with FPL than with Pascal.

The authors conclude with a speculative claim that graphical representations lead to the creation of mental images and support multiply linked representations, which in turn accounts for a rapid reaction time.

R-charts R-charts appear to be one of the few procedurally based visual programming languages used on a wide scale, at least in the former Soviet Union. The representation is in fact a sort of 'visual template' which can be overlaid onto pre-existing languages such as C/C++, Pascal, Assembler, Fortran, and other procedural languages. Ushakov and Velbitskiy (1993) also made the point that the R-chart notation is more compact than many graphical alternatives, which is an advantage in the screen real estate stakes.

R-charts have a very simple syntax based on a series of vertical arrows which denote transition from one state to the next. Text is added to the arrows, with conditions shown above the arrows, and statements (actions) below them.

Figure 5.3 shows an example of a standard control structure and its equivalent in R-charts notation.

```
if (expr) statement
        else if (expr) statement
                else statement
```

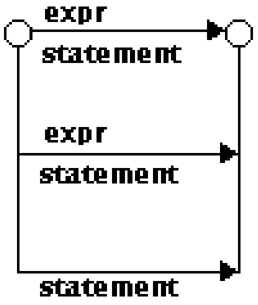


Figure 5.3: An Example of an R-Chart Control Structure, from Ushakov and Velbitskiy (1993)

Ushakov and Velbitskiy (1993) report that R-charts have been studied empirically, and that good results were reported, particularly with students and novice programmers, who showed improved understanding and shorter program development time. Unfortunately, the full results of this study were reported in a Russian language PhD, which makes it particularly difficult to obtain more details.

5.5.2 Empirical Studies of Visual Data Flow Languages

DRLP

Anjaneyulu and Anderson (1992) developed a visual data flow language (DRLP, or Dataflow Representation Language for Programming) which they consider to be a "visual isomorph" of LISP. The language has various types of nodes (input, function, predicate, etc.) through which data flows.

A study was carried out on subjects with little or no programming experience to compare learning and problem solving with DRLP versus LISP (with subjects in the latter condition having access to a structure editor).

The authors found little in the way of significant differences between the two groups. The DRLP group worked its way faster through one of the three chapters which both groups studied, and averaged significantly fewer iterations on the accompanying programming exercises. Time taken to complete a post-test and mean score were not significant.

The authors found that the DRLP group had few errors in the category relating to LISP syntax, an observation which hardly seems surprising. In their defence, the syntax/semantics distinction is sometimes hard to make: 'use of variables' comes under the heading of syntax, which may mean that DRLP is effectively shielding subjects from a recognisedly difficult semantic concept. This notwithstanding, the authors felt that DRLP did not have an impact on the conceptual difficulties associated with the LISP functions, a finding echoed by Carroll *et al.* (1980).

Overall, this study does lead to speculation about the extent to which the results are due to the graphical nature of the language, or whether eliminating LISP syntax and allowing functions to be defined in order of evaluation have in fact played a more important role.

LabVIEW Green *et al.* (1991) performed a detailed study on program comprehensibility which compared LabVIEW, a visual data flow language, with a textual notation. The study included three tasks:

- Question answering, involving conditional structures expressed in four notations: text or graphics crossed with sequential or circumstantial.
- Program comparison, in which subjects were presented with two programs in one of the four different notations and asked to say whether they were the same program or not.
- Tachistoscopic program recognition in which, given a choice of two program specifications and a pair of programs (both textual or graphical), subjects had to decide which specification matched which program.

Green *et al.* put forth various hypotheses, some relating to the structure of the program and the match-mismatch conjecture (Gilmore and Green, 1984) and described in detail in Chapter 4, and others relating specifically to the comparison between text and graphics. The match-mismatch conjecture was not supported, while the text/graphics comparison showed that question answering using the graphical notation took significantly longer than using the textual notation. They had also hoped that, in the tachistoscopic program recognition task, answers would be based on cross-referenced reasoning, *i.e.* reasoning about the relationships between program structure and domain structure. It turned out that subjects often based their decisions on a single cue. Thus, although they were not able to show that graphical representations promote recognition of program structure, their results did confirm the finding of Cunniff and Taylor (1987) that they do aid recognition of individual elements.

5.5.3 Comparing Control Flow and Data Flow Empirical Studies

As can be seen from the previous sections, it is extremely difficult to draw any unequivocal conclusions about data flow and control visual programming languages. The diversity of tasks and of subject populations makes comparisons unwise, even within paradigms.

Likewise, differences in the development of graphical representations of data and control flow, and the resulting differences in empirical studies, make cross-paradigm comparisons unworkable. For example, research on the utility of flowcharts led Shneiderman *et al.* (1977) to make the provocative comment that "since the detailed flowchart may be merely an alternative representation of the syntax of a program, it should not be helpful to programmers familiar with a programming language. Having a French recipe in addition to an English version of the same recipe would not be helpful to a cook knowledgeable in both languages" (p. 381). One possible objection to this statement is that French and English recipes, much like textual and visual programs, will not necessarily be informationally equivalent due to differences in representation and underlying 'culture'. This objection notwithstanding, it is clear that the aim of most flowchart studies was to investigate the utility of flowcharts in conjunction with a textual language, not as languages in their own right. It seems unjust to rule out control flow visual programming languages, or even visual programming languages generally, on the grounds that studies of flowcharts as an aid to textual programs were inconclusive.

Furthermore, the relation of some flowchart studies to programming is not always straightforward. While some of the studies were designed to investigate whether flowcharts were of use in the programming process, either during design, debugging, etc., others compared flowcharts in a more general way with graphical and non-graphical methods of representing information. Despite this, many of the materials used, both in programming and non-programming studies, are quite similar.

For example, a study by Wright and Reid (1973) used a decision process to contrast different alternatives for “expressing the outcomes of complex contingencies”, comparing “bureaucratic style prose”, to logical trees (which the authors call “algorithms”), short sentences or tables. An example of these contingencies, expressed in short sentence form, is as follows:

```
Where only time is limited
   travel by rocket.

Where only cost is limited
   travel by satellite if journey more than 10 orbs.
   travel by astrobus if journey less than 10 orbs.
```

Scanlan’s study comparing structured flowcharts with pseudocode for programming tasks, described above, used short programs which were very similar to Wright and Reid’s “complex contingencies”. Likewise, the study by Green *et al.* (1991) on LabVIEW, also described above, used problems of the same type.

It is understandable to some degree that it is difficult to distinguish between what counts as a program and what doesn’t: in Wright and Reid’s description of the bureaucratic prose version, one can see the parallel between qualifying clauses such as, “If …then …, unless …in which case …” and traditional programming control constructs such as “If …then …, else if …then…”. Programs are in many ways no more than a complex set of instructions. However, the programs described above are programs only in a limited sense: boolean values are passed through the program, but there are no variables, no data values being changed or updated, and there are no iterative constructs, all of which are common features of most programs, even in the relatively small, simple programs which novices will have encountered.

Fitter and Green (1979) make the point, in discussing possible reasons for the differing claims for flowcharts at the time, that the flowchart studies with positive results tended to use flowcharts which were effectively decision trees, with only one path from root to leaf. On the other hand, the flowcharts used in studies with less positive results expressed loops and jumps, thus resembling networks rather than trees. In order to understand what is happening at a given point in representations of this type, it is necessary to search through multiple possible paths, and to ensure that the context has been understood, including any preconditions. This sounds very like the criticisms levelled at procedural languages in general, and it may be that the results of unsuccessful flowchart studies are in fact due more to the underlying paradigm than to the representation used.

In any case, in order to fully investigate the issue of paradigm in visual programming languages, more studies are necessary, particularly ones which compare paradigms within the same study, even if this means that the studies are small and of only limited generalisability. The next two chapters report on studies of this kind, while the following sections describe the development of the micro-languages which were used in these studies.

5.6 The Design and Development of the Micro-Languages

In order to test hypotheses about the relationship between different visual programming language paradigms and task, a number of "micro-language" variations were developed. Micro-languages are, as their name implies, small languages which contain a subset of the functionality of a full-scale language. They are not always executable, and are usually designed for a specific purpose. Their main advantage in an experimental setting is that they allow the experimenter to control for extraneous variables. In the case of the experiments reported in this thesis, finding two full-scale languages, one based on the data flow paradigm, and one based on the control flow paradigm, which were identically matched on features other than the ones the experimenter wished to manipulate would have been impossible.

The main requirement for the languages to be used in the experiments was that they should be as simple as possible. There were many reasons for this: firstly, the experiments were to be carried out with novice programmers. Given the short timescale, the subjects needed to be able to learn the languages relatively quickly. In addition, the number of constructs in the language needed to be manageable and not cause memory problems over the course of the experiment. Finally, keeping the languages simple allows one to ensure that extraneous variables have not been introduced.

Furthermore, the distinctions between paradigms should be as salient as possible, namely:

- for the control flow language:
    - arcs should indicate flow of control;
    - only one arc should be followed at any given time;
    - multiple assignment can occur;
    - side-effects are possible.
- for the data flow language:
    - arcs should indicate the flow of data;
    - multiple, simultaneous paths are possible;
    - only single assignment is allowed;
    - no side-effects should occur.

In the follow descriptions of the micro-language development, "version 1" refers to the languages used in the experiment described in Chapter 6, while "version 2" refers to those used in the experiment reported in Chapter 7. The general characteristics of the micro-languages will first be described, followed by a definition of each type of node comprising the language. The descriptions are, for the most part, similar to the descriptions given to subjects in the experiment, with additional comments added as necessary. They are followed by an example program showing how the nodes combine to form a program.

5.6.1 Development of the Micro Languages – Version 1

The experiment described in Chapter 6 used four micro-languages in total: a control flow graph, a control flow tree, a data flow graph, and a data flow tree.[1]

All of the representations began with a line of text giving the name of the program, and the program's inputs and outputs. In the control flow versions, the variable names were purposely non-descriptive, although the second line of the program provided further information as to their type and role:

```
position_3 - inputs(A,B), outputs(C)
            A: Element, B:List, C:Position of Element
```

In the data flow versions, the inputs were described as arguments, *e.g.* :

```
position_3 - inputs(Element, List), outputs(Position)
```

This distinction between versions was made because variables are not used at intermediate positions in data flow programs: subjects must refer to the beginning of the program to ascertain the names of data objects. In the control flow version, variable names appear throughout the program, and it was felt that control flow subjects would be at an unfair advantage if those variable names explicitly mentioned the type and/or role of the data object in the program.

Control Flow

Control Flow Graph Control flow graphs are representations which use the arcs between nodes to represent the program's control flow. Recursion is represented by a node in the graph which effectively acts as a recursive call to the program.

Tables 5.1 & 5.2 show each node used in the control flow graph, along with a short description of its function.

[1] In terms of data flow, trees are something of a misnomer in the sense that any operation requiring more than one data object at the outset will have more than one root. A more correct name for the data flow trees would be *acyclic* graphs (as distinguished from *cyclic* graphs, such as the data and control flow graphs used in the experiment). However, the similarity between the two terms proved to be rather confusing, therefore the terms *tree* and *graph* were maintained.

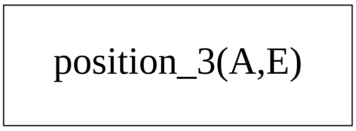
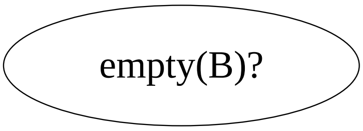
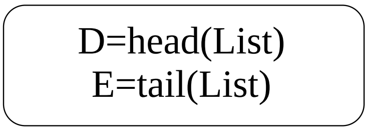
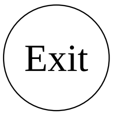
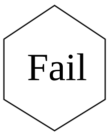
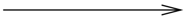
Shape	Explanation
	A **program node** represents an entire program. It occurs as a call to a program from within a program (if both nodes have the same name, this expresses recursion).
	A **test node**, in the shape of an ellipse, contains a binary choice question. The node always has two output arcs, corresponding to either a "true" or a "false" value. These arcs effectively direct the program's flow of control. Examples of test nodes are '`> ?`' and '`= ?`'.
	A **binding node** is a rounded box containing all variable bindings. Note that variables can be bound to values or to the result of an operation such as addition, multiplication, selecting the head of the list, etc. In the example, `D=head(List)`, `E=tail(List)`, `D` will take the value of the head of the list, while `E` takes the value of the tail of the list.
	An **exit node** indicates successful termination of the particular call to the program.
	A **fail node** indicates unsuccessful termination of the program.
	A **solid line** indicates the flow of control from one node to the next. Only one arc will connect two given nodes.

Table 5.1: Control Flow Graph Nodes: Version 1

Shape	Explanation
- - - - - - - - ->	A **dotted line** indicates flow of control between programs (as opposed to nodes).

Table 5.2: Control Flow Graph Nodes: Version 1 (continued)

Figure 5.4 shows an example of a control flow graph for the `position` program.

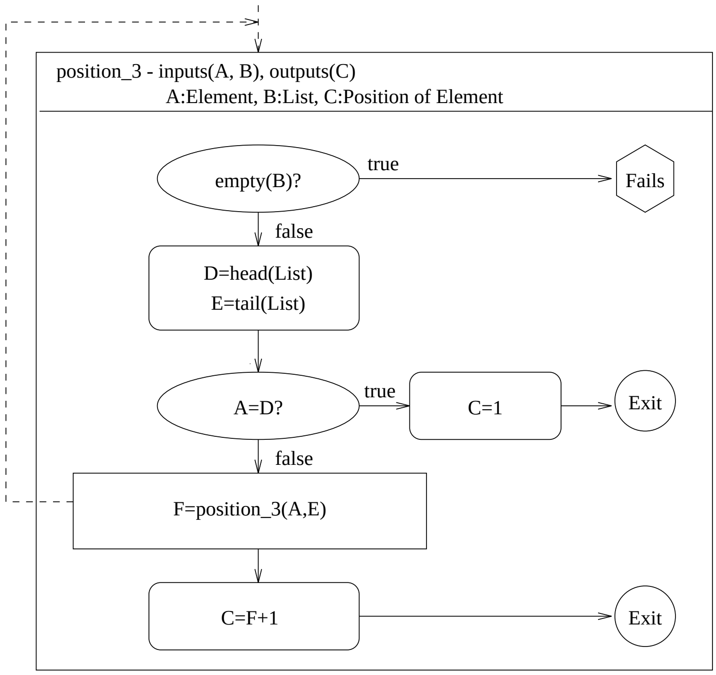


Figure 5.4: Control Flow Graph Representation for `position`

Control Flow Tree Control flow trees differ from graphs in several respects:

- By definition, there are no loops in the graph (and hence no dotted lines between programs): each level of the representation represents a call to the program;
- As each level of the tree represents a separate call, variable renaming does not occur across levels;
- There are only three types of nodes: program nodes, and fail and exit nodes. Program nodes contain all actions and bindings which occur during that particular call.

The entities making up the control flow tree are described in Table 5.3.

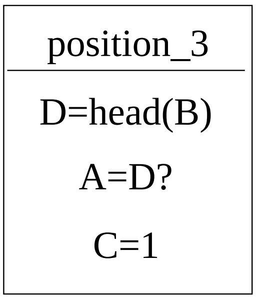
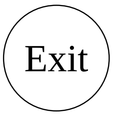
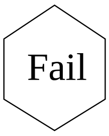
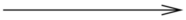
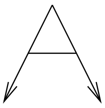
Shape	Explanation
	A **program node** represents a single call to a program (named on the first line). If it is called by a program with the same name, this indicates recursion. At each level in the tree, the program node contains all of the program events for that particular call, namely, tests and variable binding.
	**Exit node:** see Table 5.2.
	**Fail node:** see Table 5.2.
	A **solid line** indicates the flow of control from one level (*i.e.* one invocation) to the next.
	**"And" arcs** indicate that both branches of the tree must succeed for execution to succeed.

Table 5.3: Control Flow Tree Nodes: Version 1

Figure 5.5 shows the control flow tree representation for the `position` program.

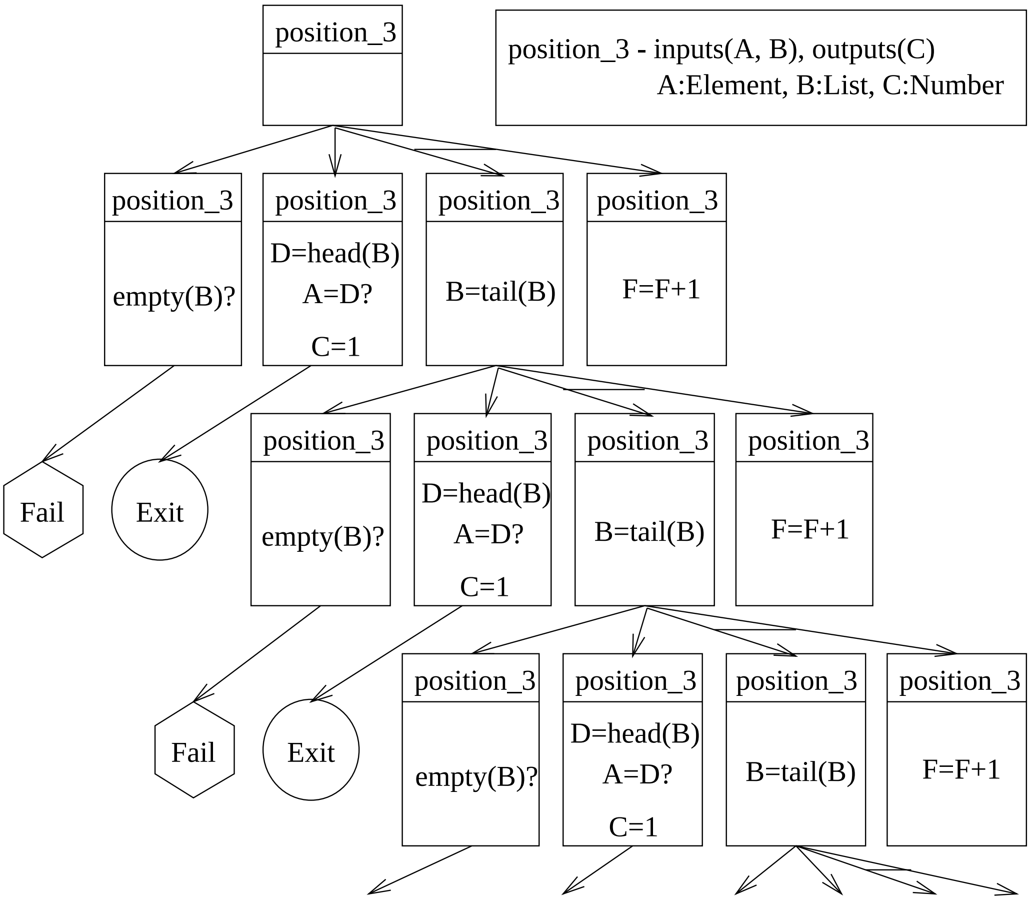


Figure 5.5: Control Flow Tree Representation for `position`

Data Flow

The basic model of data flow comprises function nodes (similar to the action nodes in the control flow language), which compute a value. However, arcs now denote flows of data, with data travelling along the arcs in the form of tokens. Thus, instead of one arc joining nodes, there are as many arcs as there are data objects required by that node.

Control constructs such as "if ... then ... else" are implemented in the form of a test box which effectively combines Davis and Keller's notion of selectors and distributors with the test which produces the boolean output that the selectors and distributors require.

Data Flow Graph The data flow language comprises the entities shown in Table 5.4.

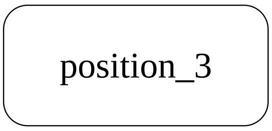
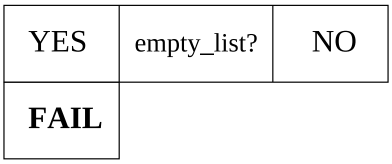
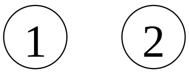
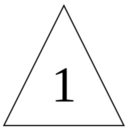
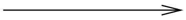
Shape	Explanation
	**Function nodes** indicate a call to a program or to a particular action, *e.g.* call `position` recursively or take the head of a list.
	**Test nodes** contain a test in the middle and YES and NO boxes on either side. Data *required* by the test flows into the middle, and data *affected* by the test flows into and out of the YES or NO sections of the node. If a data token is not output as a result of the test, a small box attached to the bottom of the YES or NO box indicates exit from, or, in this case, failure of, the program.
	**Ports** are shown at the entrance to programs, allowing the inputs and outputs to be identified.
	A **new object node** indicates the creation of a new data object. Its value is shown in the node.
	A **solid line** indicates data flow. Each line represents a path along which a different data token can flow.

Table 5.4: Data Flow Graph Nodes: Version 1

Figure 5.6 show how these nodes fit together to form the data flow graph version of the `position` program.

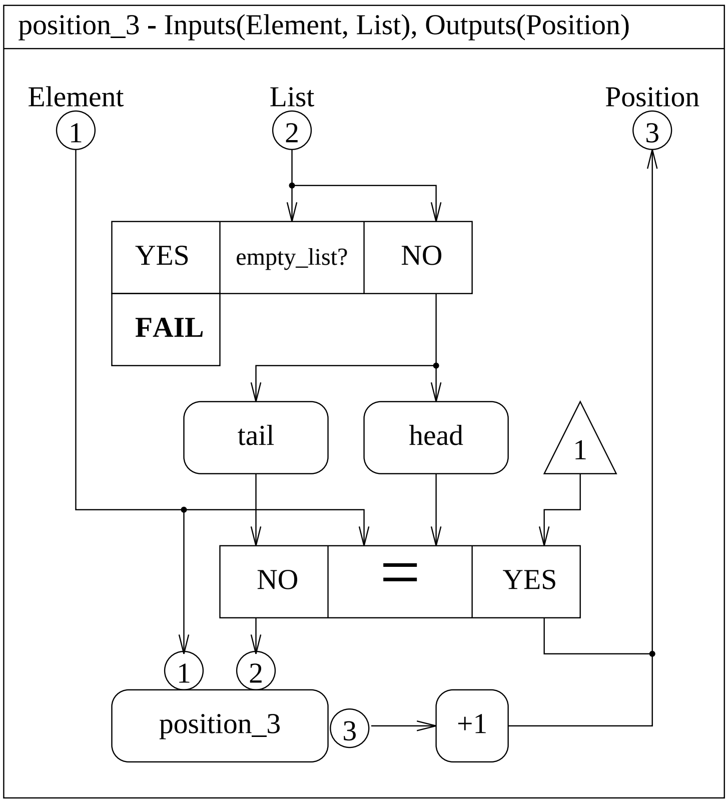


Figure 5.6: Data Flow Graph Representation for `position`

Data Flow Tree The data flow tree representation is essentially an "unfolded" version of the data graph. Instead of being represented as a node in a graph of the same name, the graph is unfolded into a number of copies of the program, so that data flows through the first call to the recursive program, and continues to flow down the representations of subsequent calls, shown below the initial calls as dotted outlines.

The nodes used in the data flow tree are essentially similar to the data flow graph, and are described in Table 5.5.

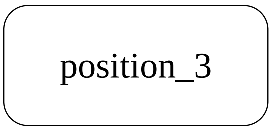
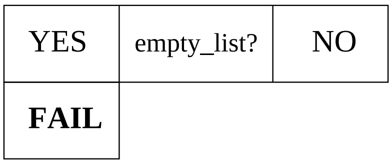
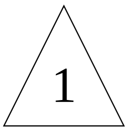
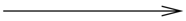
Shape	Explanation
	**Function nodes:** see Table 5.4.
	**Test nodes:** see Table 5.4.
	**New object node:** see Table 5.4.
	A **solid line** indicates data flow. Each line represents a path along which a different data token can flow.
- - - - - - - -	**Dotted outlines of objects**: nodes and arcs which are dotted indicate that the call to the program may continue, depending on the data provided.

Table 5.5: Data Flow Tree Nodes: Version 1

Figure 5.7 shows the data flow tree version of the `position` program.

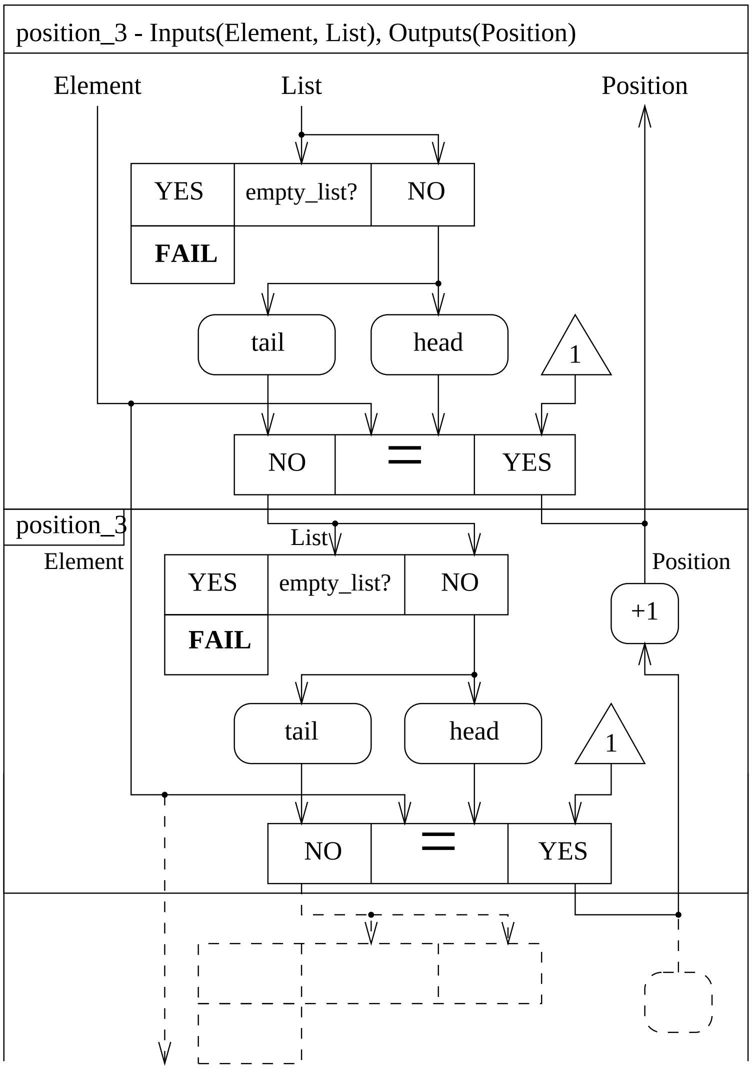


Figure 5.7: Data Flow Tree Representation for `position`

5.6.2 Development of the Micro Languages – Version 2

This section describes the second version of the data flow and control flow micro languages.

The languages differ in several respects. From a purely syntactic point of view, the languages make use of colour. The first version of the languages were implemented in Hypercard which, at the time, had difficulties integrating colour with other desired functionalities. The version 2 languages were implemented in Macromedia Director, which provides not only facilities for colour, but also serves as a potential platform on which further features could be added. For example, the representations could be animated, allowing experiments on program comprehension and/or debugging to be carried out with a dynamic representation. Furthermore, the basic language definition could be extended so as to allow program construction experiments to be conducted in addition to comprehension experiments.

In version 2, only two languages were created: a control flow graph and a data flow graph. It was decided to focus on these two types of representation as they were the most likely candidates to act as the basis for an actual visual programming language: as discussed in (Good and Brna, 1996a), the tree representations have unwanted viscosity and diffuseness, two of the 'cognitive dimensions' identified by Green (1989).

Changes to the actual nodes will be discussed under the control and data flow headings below.

Control Flow

The control flow language is similar to the first version of the control flow graph described in Section 5.6.1 in that both have `program nodes`, `test nodes` and `exit` (or `stop`) `nodes`. However, the language has moved away from elements which previously gave it an air of "visual Prolog" by eliminating `fail nodes`. Finally, the concept of the `binding node` was generalised to an `action node` to encompass any type of action which occurs in a program (usually as a result of a test).

The nodes used in the control flow language are described in Table 5.6.

Figure 5.8 shows how the nodes fit together to form the `passes` program.

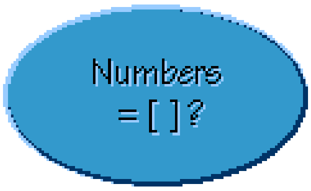
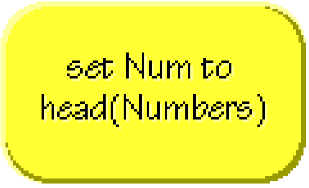
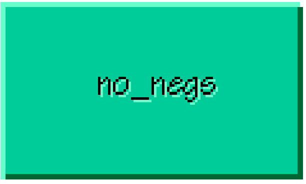
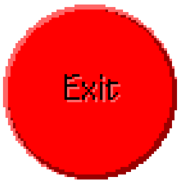
Shape	Explanation
	A **test node** tests whether a certain condition holds, for example, whether two values are equal to each other, or whether a list is empty (as in this example). A test always finishes with a question mark, and a test node has a true arc and a false arc directing the flow of control. The following tests are used in the experiment: • = (equals) • > (greater than) • < (less than)
	An **action node** performs an action, for example, assigning a value to a variable. The following actions are used in the experiment: • set X to Y - sets a variable X to Y, where Y can be a value, or the result of the following operations: – tail(A) - returns the tail, (*i.e.* all elements except the first one) of list A. – head(A) - returns the head (the first element) of list A. – - (subtraction) – * (multiplication) – + (addition) • print X - prints a variable • join X to Y - inserts element X into the front of the list Y.
	A **program node** represents an entire program. When a program node appears in a graph, it calls a program. If the program node has the same name as the graph it's in, it means that it is calling the program recursively. If it doesn't have the same name, then it is calling another program.
	An **exit** node indicates that the program terminates successfully.

Table 5.6: Control Flow Nodes: Version 2

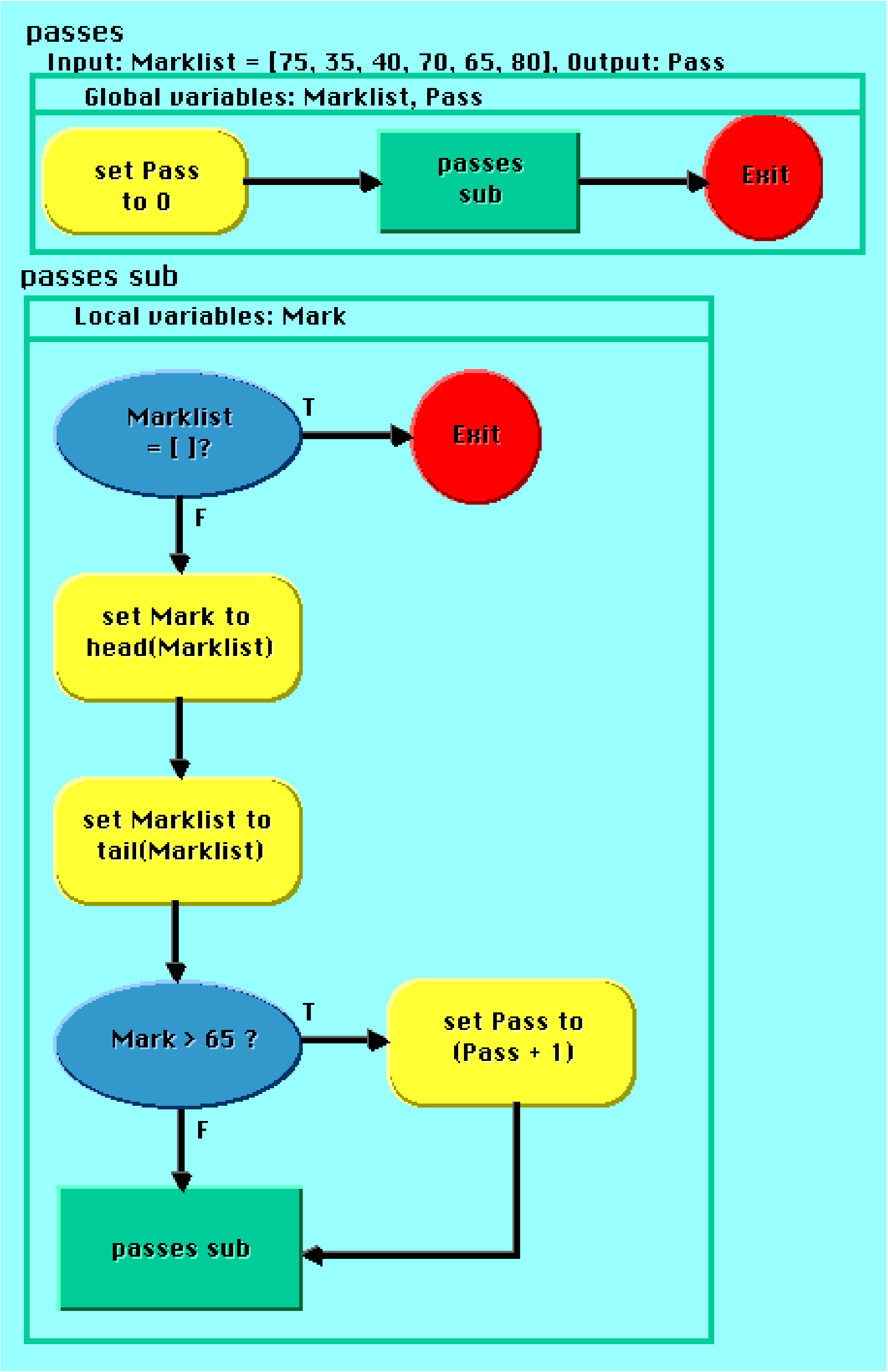

passes
Input: Marklist = [75, 35, 40, 70, 65, 80], Output: Pass
Global variables: Marklist, Pass
set Pass to 0
passes sub
Exit
passes sub
Local variables: Mark
Marklist = [ ]?
T
Exit
F
set Mark to head(Marklist)
set Marklist to tail(Marklist)
Mark > 65 ?
T
set Pass to (Pass + 1)
F
passes sub

Figure 5.8: The passes Program: Control Flow Version

Data Flow

The data flow language underwent more changes relative to the first version of the data flow graph described in Section 5.6.1. The changes were designed to add more generality to the language and more clarity, based on feedback from the experiment using the first version.

The changes can be summarised as follows:

- The concept of `function node` from version 1 was separated into `action nodes` and `program nodes`. This is to distinguish between the act of calling a function and calling a program (this also eliminates unnecessary differences with the control flow representation);
- Program input and output are now represented as explicit nodes;
- The composite `test node` from version 1 was separated into a `test node` followed by either a `selector` or `distributor` node. The latter served to direct the flow of data depending on the result of the test. It was decided to use this two stage process, described in (Davis and Keller, 1982), as subjects found the working of the composite node difficult to understand.
- the `new object node` was subsumed under the `function node`, as a particular instance of a general function.

Tables 5.7, 5.8 and 5.9 illustrate and describe the nodes used in the data flow language.

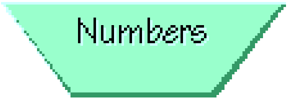
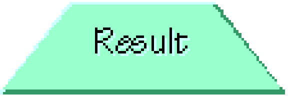
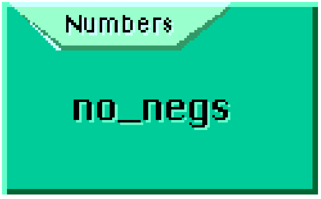
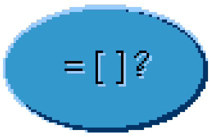
Shape	Explanation
	An **input node** shows the input to a program (one node per input).
	Likewise an **output node** shows the output of the program (again, one node per output).
	A **program node** represents an entire program. When a program node appears in a graph, it calls a program. If the program node has the same name as the graph it's in, it is calling the program recursively. It if doesn't have the same name, then it is calling another program.
	A **test node** tests whether a certain condition holds, for example, whether two values are equal to each other, or whether a list is empty (as in this example). A test always finishes with a question mark. The result of a test will either be a value of true or false. The following tests are used in the experiment: • = (equals) • > (greater than) • < (less than) Test nodes are always associated with two types of nodes: Selectors and Distributors.

Table 5.7: Data Flow Nodes: Version 2

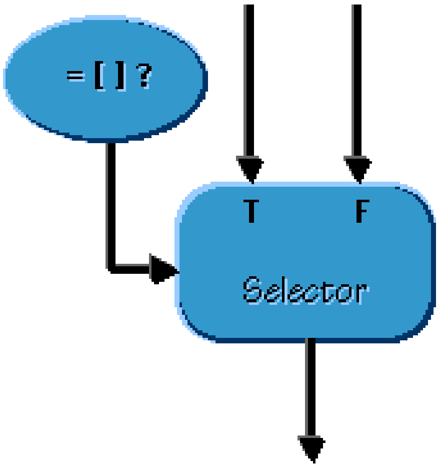
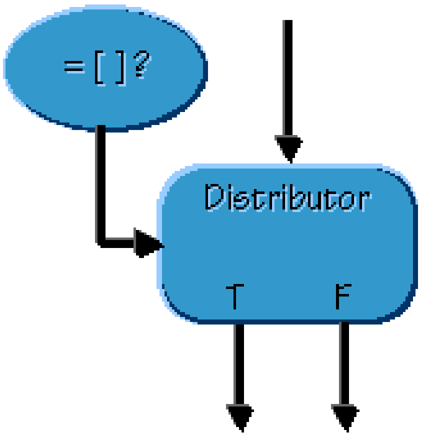
Shape	Explanation
	**Selectors** have an input arc on the side of the node. This represents the result of the test (and so can be either true or false). Selectors have two other inputs at the top of the node and one output arc at the bottom of the node. If the result of the test is true, the input arc marked "T" will be chosen, and the data on that arc (for example, a number or a list) will flow through to the output arc. If the result of the test is false, the input arc marked "F" will be chosen, and that data will flow through to the output arc.
	**Distributors** also have an input arc on the side of the node. Distributors have only one other input arc at the top of the node, but two output arcs at the bottom. If the result of the test is true, the data on the input arc will flow through to the output arc marked "T". If the result of the test is false, the data on the input arc will flow through the node and out along the output arc marked "F".

Table 5.8: Data Flow Nodes: Version 2 (continued)

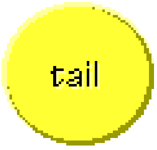
Shape	Explanation
	A **function node** performs an action, for example, splitting a list and only letting the tail (*i.e.* every element except the first one) pass through. The following actions are used in the experiment: • tail - splits the input list, taking off the first element and only letting the tail pass through. • head - splits the input list, discarding the tail and only letting the head pass through. • print - prints a value. • join - takes two inputs, an element and a list, and inserts the element into the front of the list. • subtract - takes two numbers and subtracts the element on the right from the one on the left. • * - multiplies two numbers together. • `[ ]` - creates an empty list (which is then usually used to build up the results of a recursive call). • + 1 - adds 1 to the input number.

Table 5.9: Data Flow Nodes: Version 2 (continued)

Figure 5.9 shows a data flow version of the `passes` program.

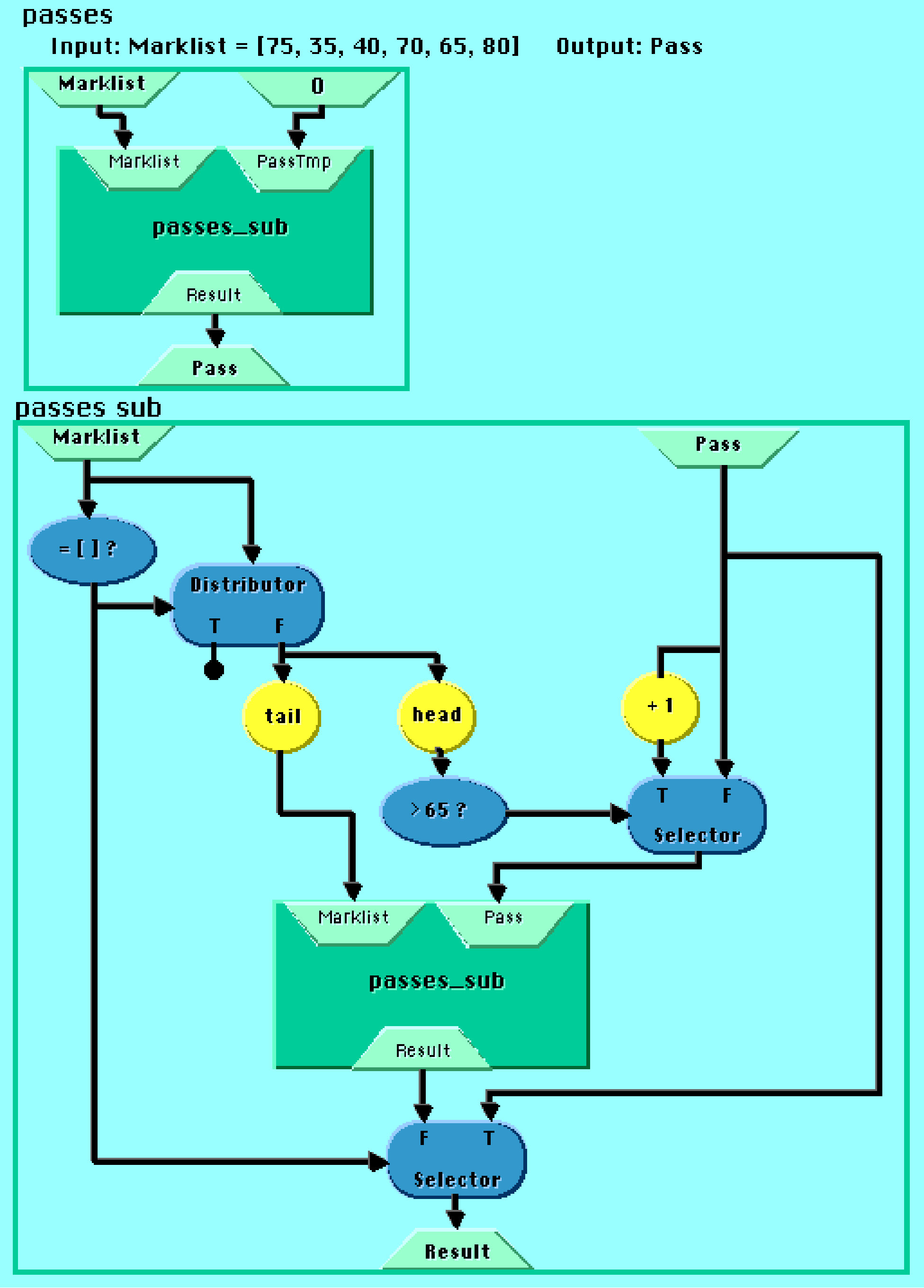


Figure 5.9: The passes Program: Data Flow Version

5.7 Chapter Summary

This chapter defined control flow and data flow paradigms. It described graphical representations of each and explained that, for historical reasons, graphical representations of control flow (*e.g.* flowcharts) have typically been developed to act as accompaniments to textual programs, in contrast to graphical data flow languages. It then described empirical work which examined control flow and data flow representations. In a discussion of this research, it was cautioned against using flowchart studies as evidence for rejecting control flow VPLs out of hand.

Finally, the chapter described the development of two versions of control and data flow micro VPLs. The next two chapters describe the experiments in which the micro-languages were used.

Chapter 6

A Preliminary Study on Control Flow and Data Flow Visual Programming Languages

6.1 Introduction

This chapter reports on a study which tested the match-mismatch conjecture (described in Chapter 4) using two micro visual programming languages based on the data and control flow paradigms (described in Chapter 5). Green (1997) provides an interesting description of the match-mismatch conjecture as follows:

> Extracting information about a program is correspondingly easy when the information matches the notation and hard when there is a mismatch. A good parallel is swimming upstream or downstream: sequential information is easy to determine from a Basic program, because one is swimming downstream, but hard to determine from say an event-driven program, because one is trying to swim in the opposite direction from the language. (Green, 1997, pp. 1-2)

In terms of data flow and control flow languages, this conjecture would predict that data flow languages will highlight data flow, therefore tasks requiring data flow information will be comparatively easier to perform than if one were using a control flow language. Conversely, control flow languages will make control flow more salient and facilitate tasks requiring control flow information.

The investigation of paradigm relates to content, in other words, to a decision about *what* information to represent. However, that choice does not completely dictate the syntactic representation: the same semantics can be represented in various ways. There exist a number of taxonomies of external representations based, for example, on the empirical study of how people classify representations (Lohse *et al.*, 1994), or on the spatial properties of the representation (Engelhardt *et al.*, 1996), and Blackwell and Engelhardt (1998) attempt to bring this work together in a "taxonomy of diagram taxonomies." In the visual programming domain, Shu (1988) makes the distinction between diagrammatic systems (*e.g.* charts and graphs), iconic systems, and tables or form based systems. The distinction is useful, but does not seem to have predictive power in the sense of helping one determine which type of representation might be best for particular tasks, situations, etc.

Indeed, it is often not clear what the consequences of a syntactic choice might be, or why one representation is easier to use or more effective than another one. Although it could be argued that differences between representations are due to the fact that the representations are *informationally* but not *computationally* equivalent (Larkin and Simon, 1987), this distinction is not always useful in the sense that the computational properties of the diagram may not necessarily lie with the diagram itself, but in the interaction between the user and the diagram.

In the programming domain, a number of early studies looked at the effect of information presentation in design and comprehension tasks, and the results of a number of these studies were presented in Chapter 5 (Ramsey *et al.*, 1983; Scanlan, 1989; Wright and Reid, 1973). The wide range of results obtained from these studies points to the importance of the particular task and setting in determining performance.

In the study described here, it was decided to further investigate the relationship between representation and problem solving by representing programs in either tree or graph form. This particular choice of representations was directly related to the use of recursive programs in the experiment. Recursive problems were used as the focus of this study because, firstly, the psychology of programming literature documents the difficulty involved in both learning and using recursion (Anderson *et al.*, 1988; Bhuiyan *et al.*, 1991, 1992; Kahney, 1989; Kurland and Pea, 1983; Pirolli and Anderson, 1985; Pirolli, 1986), with one of the major problems identified being one of seeing recursion as "looping". Secondly, the nature of recursion is such that difficulty is not a function of size: small, compact programs are often more difficult to understand, and it was felt that a graphical representation of such programs would provide a good opportunity for observing the types of difficulty discussed in (Green *et al.*, 1991; Petre and Green, 1993)

Various hypotheses were entertained as to the effect that trees and graphs might have on the comprehension of recursive programs: trees may represent a larger search space in the sense that each recursive call, rather than being represented implicitly as a "loop" in the graph, generates an additional level in the tree. Thus, even though much of the information contained in the tree is a replication of other levels, recovering from erroneous paths may be more time-consuming than for graphs. On the other hand, graphs (and particularly cyclic graphs such as the ones used in the study), while being more compact visually, lack the advantage of trees in that they do not provide a visual record of states previously visited in the form of a path from the root to the current node. This is particularly relevant in the case of recursion, where representing it as a node in a cyclic graph does little to distinguish it from iteration.

Thus, this chapter investigates two graphical ways of abstracting information in programs and two specific ways of presenting this information. The hypotheses which relate these issues to task performance are stated in the next section.

Finally, the study was designed so as to gather qualitative data on how people navigate through diagrams. Although claims have been made for visual programming languages (Shu, 1988; Golin, 1991), they have not always been supported by empirical evidence. Green *et al.* (1991) and Green and Petre (1992) highlight some of the difficulties involved in using graphical representations of programs; this study provides further information on the types of strategies used and the misunderstandings which occur when attempting to use a visual representation of a program.

6.1.1 Hypotheses

Two main hypotheses were investigated in this study: the match-mismatch hypothesis and the presentation hypothesis.

The match-mismatch hypothesis was formulated as follows: when the question type (data flow or control flow) requires information which is highlighted by the diagram (again, data flow or control flow) then performance will be enhanced, as measured by a lower response time and fewer errors. Conversely, a mismatch between the question type and the information content of the diagram (*e.g.* a data flow question coupled with a control flow diagram) should result in a decrease in performance, as manifested by a greater number of errors and an increase in time taken to complete the task.[1]

The second hypothesis concerned the presentation type, *i.e.* the particular way in which the information content is externally represented. In this study, two types of presentation were investigated, trees and graphs. It was expected that diagram type would affect performance, with one or the other favouring problem solving, measured again in terms of speed and accuracy. However, this was a bidirectional hypothesis: as described above: there were opposing tensions as to the advantages of each type of diagram, and it was unclear how these would interact. In addition, the author is unaware of similar studies which compare these two types of presentation in the area of programming and thus empirical or experimental results on which to base predictions were not available.

The final objective of the study concerned the possible interaction between information type and presentation type, and the effect, if any, of combining a particular type of representation (data or control) with its physical manifestation (tree or graph).

6.2 Method

6.2.1 Design

The experiment was a two-factor randomised mixed design, with each factor having two levels. The independent variables were the information content of the diagram (in the form of control flow or data flow diagrams), which was a between-subjects factor, and presentation type (in the form of trees or graphs), which was a within-subjects variable. Subjects were randomly assigned to either the control flow or the data flow condition, and the presentation order of trees and graphs was counterbalanced across and within subjects so as to control for order effects.

[1] Note that in a previous report of this study (Good, 1996), the terms "congruent" and "incongruent" were used to refer to situations of "match" and "mismatch". "Congruent" and "incongruent" should be taken to be equivalent to "match" and "mismatch" respectively.

Although a pure within subjects design has the advantage of minimising the effect of any individual peculiarities, doing so would have made the experiment overly long, a factor which would probably have outweighed any potential benefits. Additionally, a between subjects design was unfeasible given the small number of subjects in the pilot study.

6.2.2 Subjects

Ten subjects took part in the experiment. All except one were MSc students in either Artificial Intelligence or Cognitive Science at the University of Edinburgh.[2] All ten subjects had taken, or were in the eighth week of, a ten-week introductory Prolog course which included extensive use of recursive programs. Subjects were chosen from this course as it was important that they had some knowledge of recursive constructs in order to complete the experiment.

6.2.3 Materials

The materials used consisted of a paper based pre-test, a Hypercard stack which incorporated the practice materials and the experimental stimulus, and a screen recording utility to record the subjects' interactions with the system.[3] The pre-test, programs used, and corresponding comprehension questions are shown in Appendix B.

The Pre-Test

The paper and pencil pre-test consisted of five multiple choice questions designed to test subjects' understanding of both tail and non-tail recursion. This was to ensure that subjects' knowledge would be sufficient to allow them to take part in the experiment. Given that all subjects knew Prolog, the questions were based on short Prolog programs. The questions required subjects to recognise correct versions of a particular program, determine the behaviour of a standard but unnamed piece of code, correct a piece of buggy code, determine and distinguish between the effects of two different types of recursive list building, and predict the invocation order of lines of code, given a particular query.

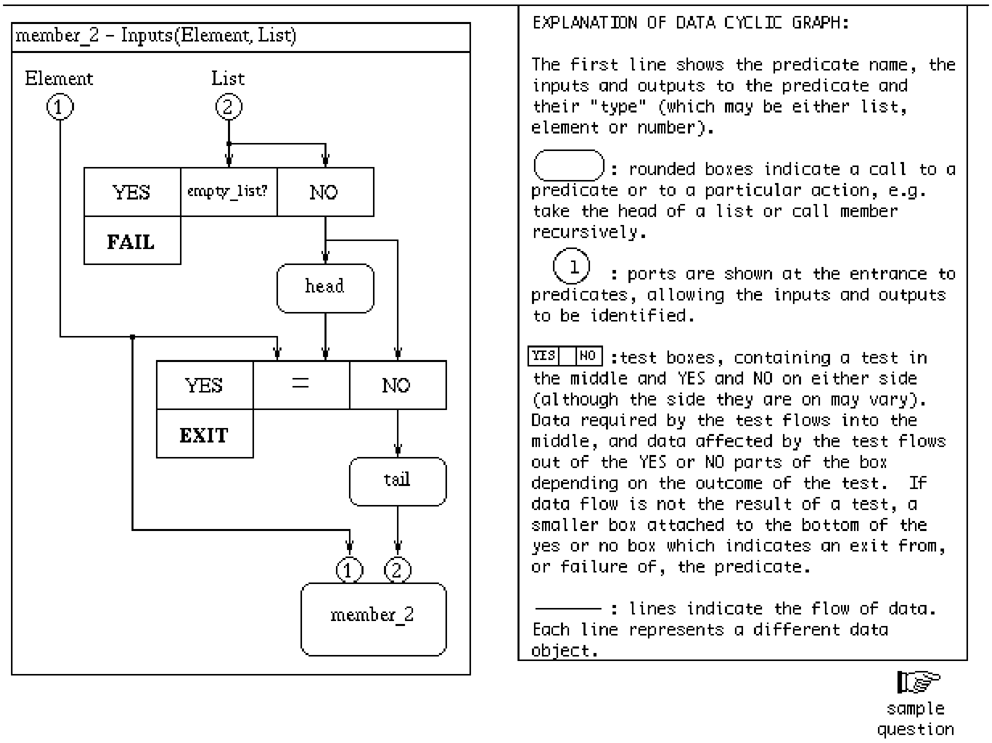
[2] The remaining subject had already completed the MSc course in Artificial Intelligence.

[3] Farallon Inc.'s 'ScreenRecorder' utility, part of the 'MediaTracks' package.

In addition to the recursion questions, the pre-test also included a question on the subjects' knowledge of various programming languages, *i.e.* which languages they had learned, how long they had used them, and to what extent (*e.g.* "took a course on it", "used extensively in my job", "used off and on").

The Experiment

The experiment was implemented as a Macintosh Hypercard stack, and consisted of a familiarisation stage and the experimental trials. The first screen showed a short description of the various stages of the experiment. This was followed by the familiarisation session, which began with a screen showing a graph diagram, and a description of each of the component parts of the diagram in terms of their shape, meaning and function (the data flow graph version is shown in Figure 6.1). The subsequent screen showed the same diagram and a sample question similar to those asked in the experiment proper. This was repeated for tree diagrams, *i.e.* each subject saw a tree diagram accompanied by an explanation of its component parts, followed by a sample question accompanied by the same tree diagram.



EXPLANATION OF DATA CYCLIC GRAPH:

The first line shows the predicate name, the inputs and outputs to the predicate and their "type" (which may be either list, element or number).

▢ : rounded boxes indicate a call to a predicate or to a particular action, e.g. take the head of a list or call member recursively.

① : ports are shown at the entrance to predicates, allowing the inputs and outputs to be identified.

YES | NO :test boxes, containing a test in the middle and YES and NO on either side (although the side they are on may vary). Data required by the test flows into the middle, and data affected by the test flows out of the YES or NO parts of the box depending on the outcome of the test. If data flow is not the result of a test, a smaller box attached to the bottom of the yes or no box which indicates an exit from, or failure of, the predicate.

——— : lines indicate the flow of data. Each line represents a different data object.

Figure 6.1: Practice Session: An Explanation of the Data Flow Graph Language

The experimental trials followed the practice questions: after a short textual description of a program in terms of input, output and a concrete example of both, subjects were presented with either a tree or a graph representation of the program code along with a multiple choice question. The question either matched or "mismatched" the diagram. The next stimulus showed the same diagram and a new question: if the first question matched the information highlighted by the representation, the second question did not, and vice versa. This sequence of textual description, followed by diagram plus question, then the same diagram plus a new question, was repeated for a second program. Thus each subject answered a total of four questions, two per diagram.

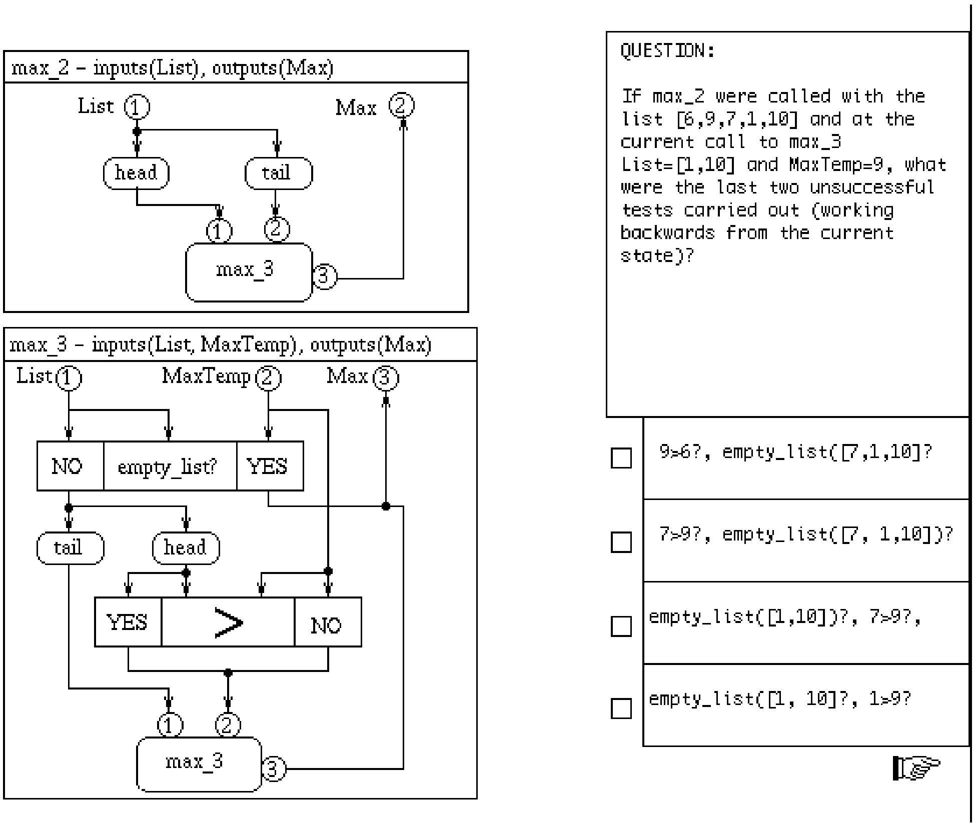
Trials differed for control flow and data flow groups in that the control flow group used a control flow tree and a graph, while data flow subjects used a data flow tree and a graph. For both groups, the order of presentation of tree and graph was counterbalanced, as was the order in which the problems were presented (both groups received the same questions: a data flow question and a control flow question for each diagram).

The programs used were `position` and `max`. The four versions of the `position` program, control flow graph, control flow tree, data flow graph, and data flow tree, were shown in Figures 5.4, 5.5, 5.6, 5.7 respectively (Chapter 5). Corresponding versions of the `max` program are shown in Appendix B. The `position` program takes an element, and a list containing that element, and returns the position of the element in the list, while `max` takes a list of numbers and recursively scans the list to find the maximum number. The problems were chosen because, although they are not uncommon, they both have particular properties: `max` involves the swapping of values between variables, while a non-tail recursive version of `position` was selected as results from Kurland and Pea (1983) show that performance on tail recursive problems do not allow one to distinguish between a correct model of recursion and a model of recursion as iteration.

Figure 6.2 shows one screen of the experimental setup, in which a data flow graph for `max` is presented with a control flow (mismatched) question.

6.2.4 Procedure

Subjects were first given the pre-test and allowed to spend as much time on it as they wished.



QUESTION:

If max_2 were called with the list [6,9,7,1,10] and at the current call to max_3 List=[1,10] and MaxTemp=9, what were the last two unsuccessful tests carried out (working backwards from the current state)?

□ 9>6?, empty_list([7,1,10]?

□ 7>9?, empty_list([7, 1,10])?

□ empty_list([1,10])?, 7>9?,

□ empty_list([1, 10]?, 1>9?

Figure 6.2: Data Flow Graph for the `max` Program with a Control Flow Question

Subjects then worked through the experiment, again at their own pace. All subject/system interactions were logged and time-stamped using the ScreenRecorder utility.

After finishing the experiment, subjects were asked to view a replay of the screen recording of their session and talk the experimenter through it, *i.e.* on the basis of the mouse movements, explain what they were in the process of doing at any given time, a technique used previously by Cox (1996). Subjects used the ScreenRecorder playback facilities to fast forward and/or pause the recording as needed, and this part of the experiment was audio taped.

6.3 Results

The study was designed so as to allow for the examination of two main effects, information match-mismatch and presentation type, and the interaction between the two. Following a brief discussion of the pre-test results, this section looks at results in terms of the main effects and their interaction, and then goes on to discuss some qualitative results from the retrospective protocol analysis.

6.3.1 Pre-test

The mean score on the pre-test was 3.9 (out of 5), with a standard deviation of 0.88. With the exception of one question, which all subjects answered correctly, incorrect responses were evenly distributed over questions and multiple choice options. The means of both groups were largely similar (control flow group: 4.0; data flow group: 3.8).

6.3.2 Programming Experience

Subjects' self-report questionnaires allowed crude measures of previous programming experience to be collected. Overall, subjects reporting knowing an average of 3.2 languages, a figure which was identical for both groups.

All subjects knew Prolog, with the next most popular languages being C and Pascal, followed by C++ and Fortran.

In terms of number of years of experience, subjects had used C for the longest periods of time, followed by Basic and Assembler.

6.3.3 Practice Session

All subjects saw two questions during the practice session (one accompanied by a tree diagram, and one by a graph diagram). Although these questions were not part of the experiment per se, the results were compared so as to ensure that there were no major differences between the data and control flow groups, and are summarised below:

Condition	**Mean Response Latency (in seconds)**	**Accuracy**
Data Flow	259.8	70% correct
Control Flow	274.2	80% correct

Table 6.1: Results of Practice Questions

6.3.4 Information Match-Mismatch

The match-mismatch hypothesis, *i.e.* the hypothesis that performance would be improved when the information required by the task matched that present in the diagram, was examined in terms of both response latency and error rates.[4]

Each question was scored simply as either correct (1 point) or incorrect (0 points), and it was expected that the error rate (in terms of the percentage of incorrect answers) would be lower in congruent situations. Overall, the error rate for congruent questions was 50%, as opposed to 35% for incongruent situations. This difference between conditions goes in the opposite direction to that predicted (*i.e.* the error rate was lower when the information type and question were *not* congruent). As the accuracy data was deemed to be ordinal rather than interval, non-parametric statistics were employed. Using a Wilcoxon signed ranks test, the difference between congruent and incongruent conditions was not significant (T^{+}=24, N=8, ns).

Mean response latency was 182 seconds for congruent questions and 165.5 seconds for incongruent questions, a trend which again goes in the opposite direction to that predicted. The time data was screened to ensure that requirements for parametric statistics were met. The difference between conditions was not significant (t-test, t=.59, ns).

Figure 6.3 shows the plotted means for speed and error rates (with 95% confidence intervals).

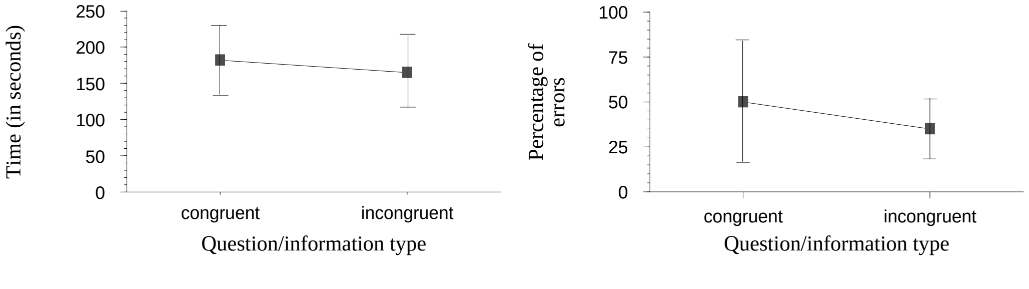


Figure 6.3: Response Latency and Percentage of Errors for Congruent and Incongruent Questions

Thus, the null hypothesis, that congruency between question type and information

[4] These were correlated to check for speed-accuracy trade-offs, however the correlation was not significant (Spearman rank-order correlation coefficient, r_s=.051, ns).

content in the diagram does not lead to differences in performance, could not be rejected. In fact, there was a trend for congruent situations to be associated with a longer solution time and a greater number of errors.

6.3.5 Data Flow *vs.* Control Flow

Analysing the data in terms of whether the questions are congruent or incongruent with the information highlighted by the representation obscures the issue of the *type* of information being investigated. In other words, the relatively high performance in cases where the representation does not match the task may in fact be due to one of the representations in question being generally better for all of the situations studied. To test this hypothesis, the data was reexamined in order to look at the accuracy and response latency on control flow and data flow questions for both groups. The results are shown in Table 6.2.

Condition	**Question Types**		**Overall**
	CF	DF	
CONTROL FLOW			
Mean	158.30	163.10	160.70
Std Dev	90.67	51.11	54.23
DATA FLOW			
Mean	167.90	205.90	186.90
Std Dev	86.82	28.78	49.40

Table 6.2: Mean Response Latency (in seconds) per Question Type

When the data flow and control flow conditions are compared, it can be seen that the response latency for both types of question is higher for the data flow group, as is response latency overall. These differences are not however significant.

Likewise, response accuracy shows the same trend, with subjects in the control flow condition showing more accurate responses overall (Table 6.3). Again, these differences are not significant.

Overall, accuracy rates for control flow questions are higher than for data flow questions across both groups, showing that the apparent "incongruency effect" is in fact largely unidirectional.

Condition	Question Types		Overall
	CF	DF	
Control Flow			
Mean	.70	.60	.65
Std Dev	.45	.22	.22
Data Flow			
Mean	.70	.30	.50
Std Dev	.27	.45	.31

Table 6.3: Proportion of Correct Responses per Question Type

6.3.6 Presentation Type

The second hypothesis concerned the effect of varying presentation type on performance. Each subject received two questions accompanied by graphs and two accompanied by trees. Their scores on each were compared. The error rate for tree questions was 55% as opposed to 30% for graph questions. This difference was not significant (Wilcoxon signed ranks test, T^{+}=24, N=7, ns).

In terms of speed, the mean time taken to answer questions using trees was 198.9 seconds, as opposed to 148.7 for questions using graphs. Again, this result is not statistically significant (t-test, t=1.277, ns) due to large variation between subjects and within groups.

Figure 6.4 shows the plotted means for speed and error rates (with 95% confidence intervals).

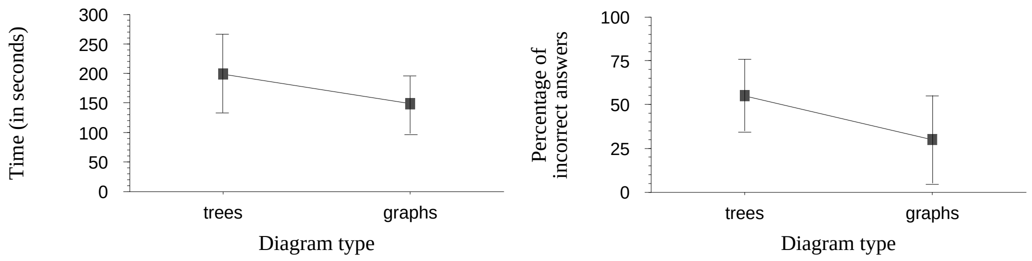


Figure 6.4: Response Latency and Percentage of Errors for Trees versus Graphs

In summary, trends in the results suggest that graphs are associated with more accurate responses and an improved response time, although the data do not reach statistical significance.

Breakdown of Presentation Type by Paradigm

It was decided to subdivide trees and graphs further by *type*: control flow tree, data flow tree, control flow graph and data flow graph. This was done in order to investigate whether the effects lay more with one particular type of tree or graph, rather than across both types equally. The mean response latency for trees and graphs by condition is shown in Table 6.4.

Condition	**Question Types**	
	Graphs	Trees
CONTROL FLOW		
Mean	128.10	193.30
Std Dev	40.69	117.09
DATA FLOW		
Mean	169.30	204.50
Std Dev	87.39	69.78

Table 6.4: Mean Response Latency (in seconds) per Group per Presentation Type

The difference in time between groups is more marked for the graph representation as compared to the tree representation (41 seconds and 11.2 seconds respectively). Within groups comparisons show that for the control flow group, the difference between graphs and trees is almost double that of the data flow group (65.2 seconds as opposed to 35.2 seconds). None of these results were statistically significant, although the trend indicates that while graphs are associated with the lowest response latencies, control flow graphs in particular show the lowest latency overall.

This is further reinforced by the accuracy data shown in Table 6.5, where it can be seen that control flow graphs show the highest accuracy, while for the data flow group, accuracy remains the same over both types of presentation.

A Mann-Whitney U test comparing scores across conditions for the graph question was not significant (U=4.5, p < .10).

A Wilcoxon Signed-Ranks Test comparing tree and graph scores within groups was significant for the control flow group (T^{+}=15, N=5, p < .05).

Condition	Question Types	
	Graphs	Trees
CONTROL FLOW		
Mean	.90	.40
Std Dev	.22	.22
DATA FLOW		
Mean	.50	.50
Std Dev	.35	.35

Table 6.5: Response Accuracy (Proportion) per Group per Presentation Type

6.3.7 Interaction between Main Effects

An analysis of variance on the time data was run to determine the significance, if any, of the main effects (group, presentation type and congruency) and their interactions. None of the main effects or interactions were significant due to low sample sizes and large amounts of variance. Nevertheless, rank ordering the size of the effects suggests that the effects of presentation type were more consistent than the effects of information congruency.

6.3.8 Qualitative results

This section presents the qualitative data obtained from the retrospective verbal protocols. The verbal protocols were taken by audiotaping subjects as they explained their screen recording to the experimenter after the experiment. The analysis of the protocol looked at two types of information: the *strategies* used by subjects as they reasoned using the diagram, and the *misunderstandings*[5] which arose. This categorisation is similar to Mulholland's (1994) with the exception that two of Mulholland's three categories, information types and strategies, were collapsed into one on the assumption that information types are implicit in the strategies used by subjects. Additionally, apart from strategies involving the control flow, data flow and goal of the program, which are applicable to a large number of programming activities, the remaining categories were devised specifically for this study.

Note that this is an initial attempt at classifying protocol data from a study of this

[5] Like Mulholland (1994), the term *misunderstanding* will be used rather than *misconception* as it is believed that problems observed were due in part to an unfamiliarity with the notation as opposed to any deep-seated conceptual difficulties.

kind, therefore the method is not overly sophisticated: the categorisation was developed iteratively on the basis of Mulholland's (1994) scheme. When examining the protocols, only those utterances which were instances of a strategy or misunderstanding were classified (*i.e.* some utterances were left unclassified). Classification was carried out by the author only.

The strategies and misunderstandings are defined below, followed by a table indicating the number of occurrences of each type per condition. Further discussion of the merits of this type of analysis can be found in Section 6.4.

Strategies

CF: Reasoning about the control flow of the diagram.

CORRES: Matching something described in the question (often in the form of hypothesised behaviour) to a particular part in the diagram which represents it (as opposed to, say, starting at the beginning of the diagram and working to that point).

CV: "Code visualisation" is an attempt to mentally visualise the corresponding textual representation of the program code.

DF: Reasoning about the data flow of the diagram.

GOAL: Reasoning about the goal of all or part of the program.

META: Rather than using the diagram to reason about the problem, subjects use information from the following sources, or a combination thereof: description of program behaviour, example of program behaviour, program names (which were meaningful in the study and hence led to expectations about the node's behaviour), and information contained in the answers to the multiple choice questions.

META_PART: This is similar to META but applies only to a subset of the diagram, *e.g.* using knowledge of a particular procedure to predict the meaning or value of a particular part of the diagram.

Misunderstandings

Misunderstandings include erroneous inferences, or incorrect navigational activities. However, a number of the misunderstandings described are derived from the incorrect use of a correct strategy, *e.g.* CORRESM, METAM and META_PARTM.

CORRESM: A variation on CORRES which involves making an incorrect correspondence between a situation described in the question and the diagram itself, *e.g.* equating part of the execution described in the question with the wrong level in a tree diagram.

DFM: The subject does not understand the origin of a particular data object, or how the object arrived at a particular node in the diagram.

FMM: Subjects report that they forget the meaning of a particular construct, *e.g.* that a triangular shape denotes the creation of a data object. This does not in itself describe the action which then followed: either the meaning was correctly guessed, or the concept was ignored (ICM) or misinterpreted (WMM).

ICM: Subjects do not take a particular diagrammatic construct into account when reasoning with the diagram, which is especially important if it has an effect on the data, *e.g.* a node which takes the first element off a list and returns the rest of the list, as opposed to returning the list intact.

METAM & META_PARTM: These are erroneous versions of the META and META_PART strategies described earlier. Although these strategies are used here, the inferences made on the basis of the various sources are incorrect.

MISUNDM: This is an underspecified category in the same sense as FMM, as it is unclear what actions follow from it. Although the subject reported that the meaning of a particular object was not understood, he/she did not go on to say whether the object was ignored or its meaning reinterpreted.

NOCORRESM: The subject is unable to determine, on the basis of a situation given in the question, the corresponding point in the diagram.

SKIPM: Subjects are at a point in the diagram and skip to another, incorrect, point. This may arise, for example, from an incorrect evaluation of a choice point, which causes them to jump to the wrong node.

STARTM: Investigation of the diagram starts at an arbitrary point rather than at the start, with the subject ignoring all processing occurring before that point. For example, in a diagram which depicts one procedure calling another recursive procedure, the first procedure, which transforms the input data in some way, is ignored.

WMM: The wrong meaning is attributed to a construct, or the construct is assumed to have a behaviour which it does not have. This may be due in some cases to short-term memory limitations (*e.g.* the subject forgets the meaning of the construct and attributes an erroneous one) rather than to a deeper misunderstanding. If so, it may be possible to reduce these errors by a change in experimental design.

Table 6.6 shows the frequency of occurrence of each strategy in total and per group (control or data), while Table 6.7 shows the frequency of occurrence of each type of misunderstanding in total and per group.

Strategy	**Group**		**Total**
	Control Flow	**Data Flow**	
CF	6	4	10
CV	1	4	5
CORRES	14	3	17
DF	4	3	7
GOAL	3	0	3
META	5	5	10
META_PART	2	0	2
Total	33	19	52

Table 6.6: Strategies Reported in the Protocol

Looking at the breakdown per group, the control flow (CF) group showed more reporting of strategies than the data flow (DF) group: 33 as opposed to 19. At the same time, the reporting of misunderstandings per group was similar (18 CF group, 22 DF group). In the light of the report from many DF subjects that they found the data flow diagrams very confusing and difficult to follow, it may be that DF subjects were less able to articulate a strategy, whereas CF subjects could proceed more methodically and report on strategies used.

Misunderstanding	**Group**		**Total**
	Control Flow	**Data Flow**	
CORRESM	4	0	4
DFM	0	1	1
FMM	0	2	2
ICM	3	1	4
META_PARTM	2	0	2
METAM	0	5	5
MISUNDM	0	4	4
NOCORRESM	4	0	4
SKIPM	2	1	3
STARTM	1	2	3
WMM	2	6	8
Total	18	22	40

Table 6.7: Misunderstandings Reported in the Protocol

The results also show that the types of misunderstandings reported varied between groups. While CF subjects reported more difficulty in matching up parts of the question with parts of the representation (4 occurrences each of CORRESM and NOCORRESM in the CF group), DF subjects reported no occurrences of either. The DF group reported more difficulty with attributing meaning to concepts (FMM, MISUNDM, WMM) rather than navigational difficulties.

However, it is difficult to make a clear distinction between errors which might be termed "errors of reasoning" and those which would be considered as "errors of navigation". In other words, is DFM, a misunderstanding about the origin and trajectory of a particular data object, due to erroneous reasoning about how the program deals with data objects, or is it simply a case of having got lost while tracing through the diagram? For this reason, and in the absence of data which could lend further support to these issues, it was decided not to attempt to classify error types into higher level groupings or to perform more sophisticated analyses on the data.

One speculative comment is worth making, nonetheless: the CORRES strategy stands out as being one which is used much more by the control flow group than the data flow group, and which accounts for 42% of the strategies used by the control flow group. In addition, as noted above, CORRESM and NOCORRESM errors appear only in the control flow group. This may indicate that different types of task activity are occurring across groups: it would appear that control flow subjects spend time trying to map parts of the task to the representation, an activity which data flow subjects engage in less frequently. It may be that rather than scanning the entire representation in an attempt to understand it, control flow subjects are attempting to focus on only those parts of the representation which they feel are necessary for the task. In other words, control flow subjects may be using an "as-needed" rather than a "systematic" strategy (Littman *et al.*, 1986) (but with better results than in the original study ...). A study focusing on diagram navigation, with concurrent verbalisation, might allow possible differences in strategy, and the role played the representation in facilitating them, to be elucidated.

In brief, the qualitative analysis resulted in a classification of the strategies used, and the difficulties involved in diagram navigation and attribution of meaning to the various diagrammatic constructs. In addition, it provided some indication that the difficulties experienced by subjects vary according to the type of information present in the diagram. This issue requires further investigation however.

6.3.9 Summary

The following points provide a summary of the main findings of the study:

- With respect to the hypothesis that performance is comparatively better if there is a match between the information content of the representation and the information required by the question, the data did not permit the null hypothesis (that there will be no difference between congruent and incongruent representations) to be rejected;
- There is a (non-significant) trend for some representations to enhance performance more than others, regardless of the task. Control flow representations were associated with the lowest times overall and on both types of task (data flow and control flow), as well as the lowest error rates overall.
- With regard to the representation format hypothesis, although there was a trend overall for graphs to lead to more accurate responses and a faster reply time, the difference was not significant.
- However, an examination of trees and graphs according to type (data flow or control flow), showed that overall, control flow graphs were associated with the

lowest response latencies and error rates. The difference between control flow graphs and control flow trees (within group) was significant.

- Finally, a qualitative analysis of the strategies and misunderstandings involved in visual program comprehension and diagram navigation showed differences between groups, with the control flow group experiencing more difficulties in matching parts of the task to parts of the representation, and the data flow group having more conceptual difficulties.

6.4 Discussion

This section looks at the results not only in terms of how they can be accounted for on a theoretical basis, but also considers the influence of elements within the experiment.

As shown above, the match-mismatch hypothesis was not upheld. This hypothesis predicted that task performance would be relatively better if the information required by the question was of the same type as that highlighted by the diagram (compared to a situation where the information required is obscured by the diagram). In fact, the match-mismatch issue seems to mask an issue of "control flow supremacy", where best performance is associated with control flow tasks and control flow representations, regardless of the task. Thus, the issue of semantics per se seems to take precedence over the semantic-task match.

On the other hand, the results from comparing trees and graphs show that the syntax of the representation does not have the same degree of influence: although graphs proved more beneficial than trees, syntax on its own was not significant. Syntax and semantics did interact though, in that control flow graphs were most effective for problem solving, both in terms of speed and accuracy.

These results seem to suggest that one particular type of representation is best in all situations (at least in the situations examined here), a finding which goes against previous work looking at the match between task and representation (Gilmore and Green, 1984) or the notion of cognitive fit (Vessey, 1991).

However, it seems more likely that neither situation tells the whole story: no one representation is probably best for everything, but on the other hand, the match-mismatch conjecture is only one factor in determining an effective representation for problem-solving. In some cases, other factors may interact to diminish, or even override, its effect. For example, Sinha and Vessey (1992) applied the notion of cognitive fit to the domain of recursion and iteration in programming, using Lisp and Pascal as the programming languages in their investigation. They found the language itself had a much stronger effect on subjects' performance than did cognitive fit. Similarly, in the current study, other factors may have contributed to the results obtained.

The following sections will firstly make some general methodological comments, and go on to discuss the main experimental findings, starting with general diagram use, and followed by a consideration of some of the factors which may have contributed to the results obtained, including familiarity and previous experience, the relationship between data and control flow, the conceptualisation of programming, and elements of the experimental setup which may have contributed to the results obtained.

6.4.1 Questions of Methodology: Screen Recording

By recording subjects' mouse movements over the course of the experiment and asking them to comment on the recording after the event, the intention was to obtain 1) data on the method used by subjects to search through the diagram and 2) some indication of the inference processes they were engaged in at particular points.

However, the technique proved to be only partially successful as a cueing mechanism, as once most subjects saw a particular diagram on the screen, they began to use the mouse in real time to demonstrate their problem solving to the experimenter, rather than following the pre-recorded mouse movements. For those subjects who did try to coordinate their explanations with the pre-recorded mouse movements, there was evidence that this process was very difficult, judging by the number of requests to either stop the recording, reduce the playback speed, or rewind the recording. This suggests that visually following the mouse movements, attempting to recall what was happening at that particular time *and* verbalising this to the experimenter may simply be too demanding, in which case, the methodology should be refined for future experiments.

6.4.2 General Diagram Use

In terms of general observations of diagram use, there was some confirmation of earlier findings, particularly the idea of familiarity and the need for training. Green *et al.* (1991) and Green and Petre (1992) dispute the idea of "graphical superlativism" whereby visual programming languages are "cognitively natural" and hence superior to textual languages, and stress the importance of training and experience in using graphical notations. This was particularly evident in the case of the only subject to obtain correct answers to all questions, despite the fact that his score on the Prolog pre-test for recursive skills was one of the lowest. The subject's verbal protocol showed a good understanding of how to navigate through the representation and where to look for relevant information, with no errors of interpretation reported, and extensive use of what Petre and Green (1992) term "secondary notation". That subject later reported that he used graphical software engineering representations in his work and felt comfortable constructing and using many different types of representations, including data flow diagrams and entity relationship diagrams.

Green *et al.* also found that "cues imply meaning", in other words, that if a graphical element was visible, it was expected to have some relevance. This phenomenon did not occur in the present study. On the contrary, some subjects seemed to ignore constructs present in the diagram (and necessary in order to correctly answer the question), including any computation which occurred as a result of that construct. Reasons why this might be so are discussed in Section 6.4.5.

6.4.3 Familiarity and Previous Experience

One explanation for the lack of support for the match-mismatch hypothesis may be that subjects were more familiar both with control flow based languages, and with graphical representations of control flow, and that a familiarity effect therefore overshadowed the match-mismatch effect.

In terms of language familiarity, procedural, control flow based, textual languages were very important historically, and continue to be extremely widely used today, therefore this paradigm is likely to have been more familiar to subjects. Indeed, the data on previous programming experience shows that apart from Prolog, which all subjects were familiar with (all were taking or had taken a course on it), procedural languages were both best known by subjects in terms of years of experience, and known by the greatest number of subjects.

Additionally, flowcharts have often been used to illustrate the workings of textual, procedural languages, and the experiment described here showed an effect of "control flow supremacy" which was most marked with respect to graphs. Although the control flow diagrams were not strictly speaking flowcharts, the graphs in particular borrowed heavily from the flowchart tradition, and subjects may have been familiar with diagrams of this type. Unfortunately, the experiment did not collect data which could support or refute this claim.

In any case, it may be that the subjects were more familiar with this type of layout, both from a semantic and a syntactic point of view. One way of investigating the familiarity issue might be to use a variation of the ER taxonomy task (Cox and Brna, 1993), which required subjects to sort and label a series of 87 graphical items (maps, sketches, tables, tree diagrams, graphs, etc.). This task could be adapted more to the programming domain by using a selection of graphical representations of programs, including control and data flow graphs and trees of the type used in the current study. Subjects could then be asked to rate or group the representations in terms of their familiarity, and the results could be correlated with subsequent performance measures.

Anecdotal evidence suggests, however, that the distinctions of interest to the experimenter may be too fine-grained for the subjects. When running the experiment described in Chapter 7, the representations used were explained to the subjects in a post-experiment debriefing. Many in the data flow group erroneously jumped to the conclusion that the representations they had used were traditional flowcharts.

Finally, time constraints make it difficult to include more than a minimum of pre-tests: the fact that the experiment requires students to learn a new (albeit "micro") programming language before answering comprehension questions makes for a relatively long and intensive experiment, and the number of pre-experiment activities which can reasonably be included is limited.

The issues of familiarity and prior experience have links to the way in which one conceives of programming, an issue which was introduced in Chapter 3. It was noted there that programs may popularly be conceived of as active entities which input data objects and, through a series of processes, output new data objects. Separating the idea of conceptualisation from the familiarity issue is not an easy task: do people think about programs as acting on data because they have been exposed to that type of paradigm, or does the issue go beyond prior experience? Assuming a continued move away from procedural languages in the future, it will be interesting to see whether novice programmers who were initially exposed to a nonprocedural language (*e.g.* an object-oriented language) will have a different conception of what programming involves.

6.4.4 The Data/Control Flow Relationship

A major factor which may have played a role in the results obtained is the interrelatedness of data and control flow. Data flow and control flow are intertwined: data flow diagrams necessarily comprise control flow information and vice versa. Thus, it may be that, for example, more data flow information was present in the control flow diagrams (in the sense that less inference was needed to access it), while less control flow information was present in the data flow diagrams.

One way to investigate this issue would be to devise a characterisation of the cognitive load in terms of the procedural steps involved in searching the representations and performing the necessary inference upon the data obtained from them. This information could then be incorporated into a cognitive model, a point which is discussed further in Chapter 10.

However, even if it is possible to determine the amount of information of varying types present in a given diagram, it may be that the nature of both the information and the task leads us to think in terms of *degrees* of match rather than a more categorically defined notion of match *vs.* mismatch, or congruency *vs.* incongruency. In other words, because the information being considered is interrelated, and because the task is such that inference (as opposed to simple read-off) is almost always necessary regardless of whether the situation is congruent or incongruent, it may be more useful to rank representations in terms of their efficacy rather than simply categorise them.

6.4.5 The Use of Additional Cues

The use of "additional cues" in the experimental setting may have influenced the results and led to the use of meta-reasoning. Subjects in the experiment had access to much information about the program in addition to the graphical representation. First, they were presented with a textual description of the program in terms of input, output and a concrete example of both. Additionally, the programs themselves were reasonably common, and the program names were meaningful (`max` and `position`). By the time subjects actually viewed the representations of the programs, they may well have formulated *a priori* ideas about what the programs did and how they worked, based for example on expectations stemming from the program's name, its described behaviour, examples, or the answers proposed. This may have led them to a less thorough investigation of the graphical representation, using it more as part of a hypothesis verification process than anything else. This is similar to the situation reported by (Adelson, 1984), in which the availability of program code gave subjects an additional source of information on which to base their understanding of a program, which were meant to have been based on an accompanying flowchart describing either the functional or procedural aspects of the program.

This in itself doesn't necessarily explain why differences between conditions were observed, both in terms of speed and accuracy. However, the familiarity issue may play a role here, making more familiar representations easier to search. The size of the search space is also relevant, and one would expect that representations with fewer nodes and less repetition, or which are more "terse" in Green's cognitive dimensions parlance (Green, 1989), would be searched more quickly, as seems to have been the case.

Possible solutions to this problem of meta-reasoning include the use of non-meaningful code names and less textual information about the code, or the use of buggy programs, and focusing the task on comprehension with a view to debugging rather than solely on comprehension. This does not in itself preclude testing the match-mismatch hypothesis: *e.g.* questions about the effects of a faulty control flow representation on subsequent data flow could be devised. The potential benefit of this approach is that it involves a realistic setting: part of a programmer's job often involves debugging code written by other programmers.

6.4.6 The Nature of the Task

One further element which is related in part to the use of additional cues stems from the nature of the tasks used. The multiple choice questions were quite difficult, while the distractors contained various subtleties, embodying such misconceptions as the “off by one” error. Many questions required both forwards and backwards mental execution of the program using hypothetical data. This scenario is very different from tasks which simply require subjects to locate information which is available in read-off form in a table or graph. Therefore, subjects may again have relied not so much on the representation as on their own mental execution of the program, the general workings of which were surmised from the textual description of the program rather from the program itself.

6.5 Conclusions

The present study has suggested that the relationship between the task and the information required by the task is not a simple one. It is difficult to imagine that a given representation can simply be deemed to be the “right” one for the task in the absence of information on factors such as the features of the representation (what information it contains and in what format), the nature of the task, the match between the two, and the user of the representation, in terms of both experience with the representation and cognitive style preferences.

Trends have, however, been elucidated, pointing again to a control flow bias, with control flow questions being easier to answer, and control flow representation seemingly easier to use. The issue of representation format points to graphs as being superior to trees.

This section attempts to place this research in a wider context and consider the implications of this experiment for further work.

One point about the research reported here is that in some ways it addresses ease of learning, rather than ease of use. Given the short time scale of the study, data is not available on the use of the representations following the initial learning stages. A representation may be difficult to learn but prove useful once it has been mastered.

For example, one might argue that the data flow representation is the more "complete" representation in the sense that it portrays data flow explicitly, but has also been augmented to include control flow constructs. By contrast, in the control flow representation, data flow must be inferred.

On the other hand, the very completeness of the data flow representation may also make it more complex: it certainly requires more nodes to represent the same program. Furthermore, data flow representations require following as many arcs as there are data objects, thus necessitating, in the case of more than one data object, a parallel search of the diagram, rather than the serial search afforded by the control flow diagram. This in turn places more of a demand on memory as placekeeping operations must occur for some arcs while search is being directed down others. These operations may make data flow diagrams intrinsically harder to use unless the program consists solely of transformations on a single data object.

The question arises as to whether it is better to start novices off on representations which are familiar and easy to use, or whether it is worth investing more effort at the outset in order to reap more benefits in a not-too-distant future (assuming that this is indeed the case for data flow representations).

There are obvious difficulties with conducting long term studies using micro languages, however, one way of addressing this question is to look at the representations in a broader context, and to focus on how they affect other types of understanding than simply data and control flow. From a practical point of view, this could be achieved by attempting to integrate the methodology used by Pennington (1987b) and Corritore and Wiedenbeck (1991) with that of Gilmore and Green (1984). This would allow us to look at the ways in which the representations interact with various types of information, and also to obtain a more open-ended measure of students' general program understanding.

In addition, the issue of "cognitive style preference" is worthy of further investigation. Although familiarity with one type of diagram may facilitate performance using that diagram, some persons may simply perform better using graphical, as opposed to sentential, representations. The idea that "graphical readership" is an acquired skill (Petre and Green, 1993) is not disputed, however, there is evidence to support the claim that some individuals learn better than others from graphical teaching methods and that their skills in a given domain improve to a significantly greater extent when taught using a graphical teaching method than do those of “less diagrammatic” reasoners (Cox *et al.*, 1994; Stenning *et al.*, 1995). How this issue relates to program comprehension will be investigated in the next experiment.

6.6 Chapter Summary

This chapter provided an initial investigation of control and data flow visual programming languages using the match-mismatch conjecture. Rather than finding that a match between representation and task facilitated performance, a “control flow supremacy” effect was uncovered: the error rate was lowest for control flow questions regardless of the representation used, and furthermore, control flow representations were associated with rapid response rates on both types of question. In terms of information presentation (trees *vs.* graphs), control flow graphs were associated with best performance. Finally, an initial qualitative description of strategies and misunderstandings was carried out based on retrospective verbal protocols. Possible explanations for the “control flow supremacy” effect were discussed, focusing on issues of familiarity and prior experience.

Chapter 7

Data Flow and Control Flow Visual Programming Languages: A Comparison

7.1 Introduction

Chapter 3 described an experiment using an information types approach which resulted in a "control flow supremacy" effect: despite using a language based on a declarative paradigm in the experiment, subjects still performed relatively better on questions requiring control flow and low-level operations information than on those requiring functional or data flow information. Another experiment, reported in Chapter 6, looked at the relationship between visual representations of control and data flow and task using the match-mismatch conjecture. Again, the results showed a trend for control flow *representations* to be associated with lower error rates and faster responses for both types of question, and for control flow *questions* to be answered more quickly and accurately regardless of the representation.

The experiment described in this chapter brings together the 'match-mismatch conjecture' and 'information types' strands of research in order to investigate the issue of representation and task once again. At first sight, this may not seem like a wise move, given that both of the previous experiments seem to suggest that a control flow effect is masking any potential effects of representation-task match. However, it is argued here that doing so may allow the advantages of the 'match-mismatch' and the information types approach to be combined, while eliminating some of their disadvantages in the process.

7.1.1 Advantages of the Combined Methodology

Combining the methodology of the information types studies with that of the match-mismatch conjecture studies has the potential to capitalise on the positive aspects of each type of study.

On its own, the match-mismatch conjecture is stated in a way that allows for the formulation of relatively focused and precise hypotheses about the relationship between notation and task. However, it provides information about only a subset of program comprehension, namely, in a cross comparison, about the two types of information hypothesised to be obscured or highlighted by each of the two representations. For example, given a match-mismatch study involving control flow and data flow tasks and notations, the conjecture covers the relationship between data flow and control flow representations and tasks only: other information types which may be important to comprehension, such as function or operations, cannot be considered. The hypothesis is more precise, but the scope is more limited.

Information types studies, in contrast, encompass a broader range of information about a program (typically, five types). However, hypotheses do not follow as obviously from the concept of information types as they do from the match-mismatch conjecture. They embody a particular way of slicing up, or abstracting from, a program at an informational level, but need to be combined with predictive hypotheses. In the past, information types have been used in attempts to ascertain the mental representations of programmers of various levels, but this does not mean that they cannot be applied to other research questions, such as the interaction between task and representation.

Studying a range of information types, rather than only those which are directly relevant to the match-mismatch conjecture, leads to a further question: what is the relationship between a representation and tasks requiring information types not directly highlighted or obscured by the representation? There may be no relationship between them, however, descriptions of programming paradigms suggest otherwise, given that different types of programming information are interrelated, at least from a theoretical point of view. For example, the semantics of so-called applicative, functional languages are claimed to be closely related to data flow graphs (Dennis, 1986). According to Davis and Keller (1982), data flow languages are in fact a subset of functional languages therefore, it could be hypothesised that functional tasks are also facilitated by data flow representations.

The way in which the paradigm is *represented* may also lead to hypotheses about the interrelationship between information types: when describing data flow, Dennis (1986) makes the point that a node firing defines the "next state" of the computation. Given that any node can fire as soon as its inputs are available, nodes in data flow graphs can potentially fire in parallel. Therefore, it seems likely that there will not be the same well-defined, or at least easily localisable, notion of state as in a control flow representation. In the same way, control flow graphs essentially represent a series of sequential operations, with low-level operations implemented at a node level. Data flow, on the other hand, sometimes requires more than one node to perform a single operation and the nodes may not be spatially contiguous. Combining the match-mismatch conjecture with the information types approach should allow these theoretical claims to be tested: if interrelationships between information types do exist, *e.g.* information types X and Y are related, then representations which highlight information type X should make tasks requiring associated information type Y easier to perform. This is in effect an extension of the match-mismatch hypothesis to *groups* of related information.

Although the match-mismatch conjecture and the concept of information types provide some theoretical guidelines for experimentation, neither give hard and fast rules for the way in which evidence of comprehension should be elicited from subjects. Question answering[1] was used to test the match-mismatch conjecture in a study of data flow visual programming languages (Green *et al.*, 1991), however, alternative dependent measures could presumably be used, for example, a debugging task. Likewise, information types could be elicited through a number of different means. However, combining some of the elicitation methods used in the information types studies with those of the match-mismatch studies should enable access to a broad range of information: using multiple choice questions (as in the match-mismatch conjecture study) rather than binary choice questions (as in the information types studies) will in principle allow finer-grained information about students' misconceptions to be collected. Looking at response latency in addition to accuracy, as was the case for the match-mismatch studies, should also provide further insight into the nature of the comprehension process. Finally, requir-

[1] presumably multiple choice, although this is not clear from (Green *et al.*, 1991).

ing subjects to write a summary of the program (information types studies) allows for investigation of the differential effect of notation on general program comprehension and the way in which that information is communicated to other parties. The open-endedness of the summary request ensures that the experimenter is not forcing his/her view of program comprehension on the subjects, and provides a useful contrast with the more focused multiple choice questions.

From the design point of view, developing an experiment which incorporates both strands of research is relatively straightforward. From a theoretical point of view, doing so highlights the tension between internal and external representations, and leads to quite different hypotheses. According to the match-mismatch conjecture, performance on data flow questions should be better for subjects using the data flow VPL rather than the control flow one. Likewise, subjects using the control flow VPL should perform better on control flow questions than their data flow counterparts. The conjecture does not predict performance on other types of questions. However, based on previous information types studies with novices, including the study reported in Chapter 3, performance is predicted to be best on procedurally based, control flow type questions regardless of the language. The study reported in Chapter 6 would also predict better performance in the control flow condition as a result of the observed "control flow supremacy" effect. These hypotheses are listed in the next section.

An additional aim of the study reported here was to investigate the issue of "graphical skill" and its relation to comprehension performance with visual programming languages. In other words, just as previous programming experience tends to correlate positively with performance, do measures of graphical ability correlate with visual programming skill? Previous investigations of visual programming languages (Cunniff and Taylor, 1987; Petre *et al.*, 1995) have used the paperfolding test, part of the Kit of Factor-Referenced Cognitive Tests (Ekstrom *et al.*, 1976), but this test appears to measure the ability to mentally manipulate objects in 3-D space, a skill which does not seem necessary for reasoning with a visual program. It was felt that the pathfinding test, part of the same set of tests, might be a more appropriate measure, as it requires subjects to trace a path from a starting point to an end point, a behaviour which is also required in tracing the execution of a visual program. In order to investigate the utility of these measures, both of these tests were included in this experiment.

7.1.2 Hypotheses

The hypotheses were stated as follows. According to the "control flow supremacy" hypothesis:

- overall accuracy should be higher for the control flow group compared to the data flow group.
- overall response latency should be lower for the control flow group as compared to the data flow group.
- time taken to "learn" the micro-language should be lower for the control flow group compared to the data flow group, as should time taken to inspect each program.

According to the match-mismatch hypothesis:

- response *latency* on control flow questions should be lower for the control flow group as compared to the data flow group, and vice versa for data flow questions.
- response *accuracy* on control flow questions should be should be higher for the control flow group as compared to the data flow group, and vice versa for data flow questions.

The following sections report on a study which investigated the above hypotheses. The study was in fact the second of two identical studies which focused on these hypotheses. Unfortunately in the first study, despite subjects' random assignment to conditions, it was determined later that the data flow group was the more "skilled" group in terms of all pre-test measures: pathfinding test, paperfolding test, Prolog pre-test and overall programming experience (number of languages known, length of time known, and self-rating in terms of level of expertise). Although none of these differences on its own was significant, the combined differences were felt to suggest that the samples had not been drawn from the same population, therefore only the second study will be considered here.

7.2 Method

7.2.1 Design

The experiment was a two-factor randomised mixed design. The independent variables were the visual programming language paradigm (either data flow or control flow), a between-subjects factor, and question type, a within-subjects factor with five levels: functional, data flow, control flow, operations and state. Subjects were randomly assigned to either the control flow or the data flow condition, and the presentation order of the programs and questions used was randomised so as to control for order effects.

7.2.2 Subjects

Twenty two subjects took part in the experiment. All subjects were in the first term of the second year of a four year course in computer studies at Napier University, and had been taught C^{++} and COBOL. Subjects had some knowledge of recursion, primarily at a theoretical level.

In the results described below, data from twenty subjects are considered: data from two of the subjects had to be discarded as they experienced difficulties during the experiment, which were manifested either in the form of missing data, or by asking the experimenter questions at points which would have adversely affected the timing data.[2]

7.2.3 Materials

The experiment included two pre-tests: the paperfolding test and the pathfinding test, both part of the Kit of Factor-Referenced Cognitive Tests (Ekstrom *et al.*, 1976). The pathfinding test is usually administered in two parts, each of which lasts seven minutes, and contains items of increasing difficulty. Because of time constraints, a split half version of the pathfinding test was devised using odd items from one part and even items from the other, thus limiting the total time needed to administer the test to seven minutes. Both tests are shown in Appendix C.

[2] An examination of the students' programming self-report questionnaires showed that neither had taken the first year course (one was a foreign student while the other had transferred from a technical course). Thus both may have had insufficient knowledge of recursion, taught in the first year, to understand the programs).

A self-report questionnaire was also devised to determine subjects' prior programming experience and familiarity with various programming languages (shown in Appendix C). This questionnaire had previously been used in the unreported study, but was modified slightly so as to allow for a finer-grained assessment of programming skill. In addition to asking for details about how subjects had learned the language in question (school, university, self-taught or used in job), it also asked them to rate their level of programming expertise for each language on a scale of 1 to 5, regardless of how they had originally learned the language (in the original questionnaire, subjects were only asked to rate their knowledge if the language was self-taught).

The experimental setup was implemented in Macromedia Director and ran on a Macintosh. Five programs were used (the first being a practice program). The programs were chosen because they were recursive and they included reasonably complex passing of values between arguments. However, they were relatively short programs (equivalent to approximately 10-12 lines of code in a textual language). In choosing the programs, two issues were considered: program names needed to be meaningful in some real-world domain (*e.g.* a program about taxi scheduling or choosing a basketball team), as one of the issues of interest was the level at which subjects would choose to describe the program in their free-form summary (*i.e.* whether they described it at a domain, program, or detailed level). As a result, nonsensical or minimalist names (as used in the study described in Chapter 3) could not be used. Additionally, it was necessary for program names to be relatively unmeaningful at a programming level, so as to avoid a situation similar to that described in Chapter 6 where subjects made use of the program name to infer details about the program's function and behaviour rather than relying solely on the graphical representation of the program.

The first screen of the program allowed the experimenter to choose between the data or control flow versions of the experiment. The next screen was the introductory screen for participants, explaining the overall structure of the experimental session. The following screens provided an introduction to the language: each successive screen introduced a new node, with an accompanying text describing the node (Figure 7.1 shows a screen with several control flow nodes and the accompanying explanation for the most recently introduced node).

Following this, a series of screens showed a sample program, explaining each stage in the program, and how each node contributed to producing the overall program output. Figure 7.2 shows a screen describing the workings of a simple data flow program.

The last part of the practice session gave subjects a simple example program and required them to first study it. The next screen showed a blank text box and asked subjects to type in a summary of the program. The program was not visible on this screen. The subsequent five screens showed a series of multiple choice questions (one per screen), with the program shown alongside the questions.

The experiment itself consisted of four programs, which were displayed (in a randomly generated order) as in the practice session above: each program was followed by a screen asking subjects to write a summary of the program. The summary screen was followed by five multiple-choice comprehension questions, corresponding to the five information types. These questions were presented in random order, one per screen.

Control flow and data flow versions of the `passes` program can be found in Figures 5.8 and 5.9 respectively (both of which are in Chapter 5). Finally, Figure 7.3 shows one of the comprehension questions (in this case, the function question) with accompanying multiple choice answers. Note that for all questions, there is only one correct answer (in the example shown, this is the answer selected).

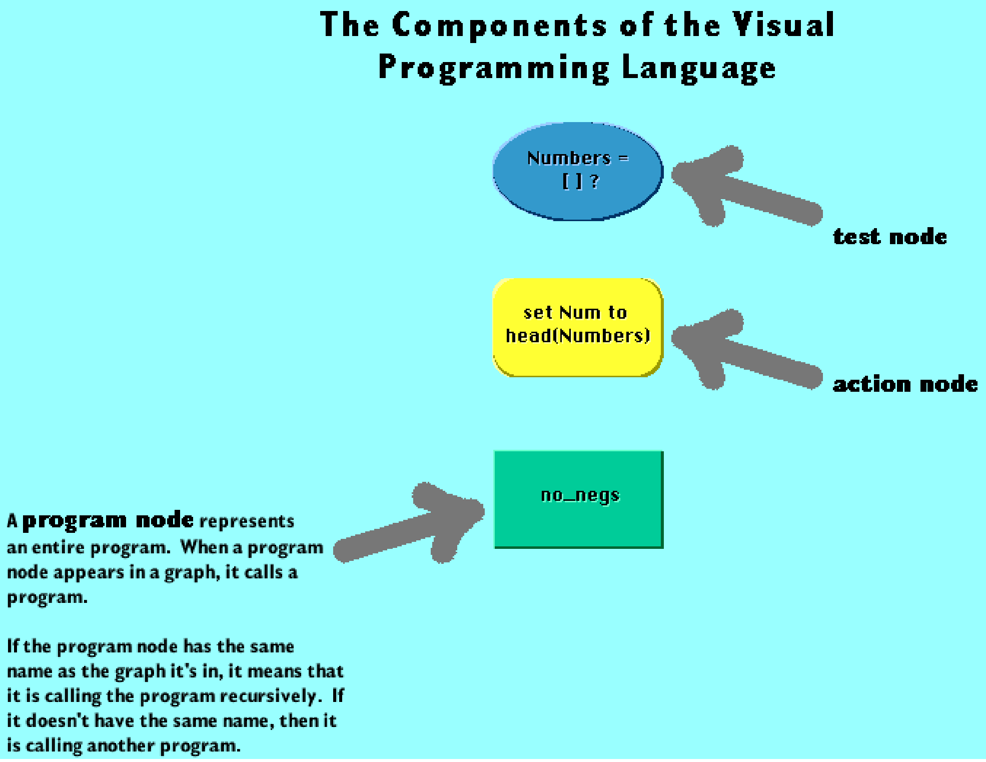
The data and control flow versions of all programs used in the experiment, along with the accompanying multiple choice questions[3], can be found in Appendix C.

The experimental program was designed so that it automatically generated a log of each subject's session, including time spent on each screen, the contents of the program summary field, and responses to each multiple choice question. It also marked the responses for correctness and recorded this information in the log file.

7.2.4 Procedure

Subjects were run individually, and each experimental session lasted between 1 and 1.5 hours.

[3] Previous instantiations of the "five types" methodology used binary choice questions, including (Pennington, 1987b), (Corritore and Wiedenbeck, 1991), and the experiment reported in Chapter 3. Given the 50% chance of guessing the right answer, and the fact that wrong answers do not provide much information on the types of misunderstandings which students might harbour, it was decided to develop multiple choice questions for the experiment.

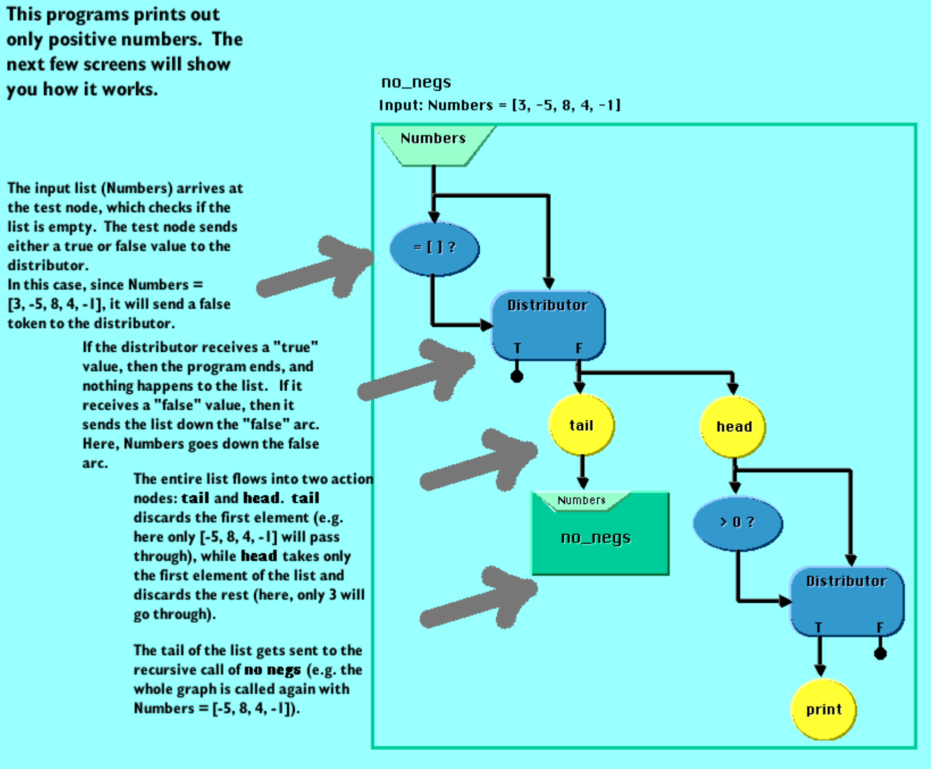


The Components of the Visual Programming Language

A **program node** represents an entire program. When a program node appears in a graph, it calls a program.

If the program node has the same name as the graph it's in, it means that it is calling the program recursively. If it doesn't have the same name, then it is calling another program.

Figure 7.1: Explanation of Control Flow Nodes in the Practice Session



This programs prints out only positive numbers. The next few screens will show you how it works.

no_negs
Input: Numbers = [3, -5, 8, 4, -1]

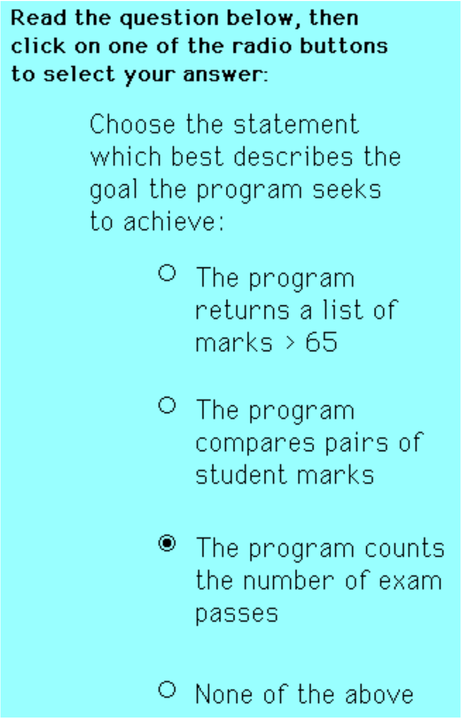
The input list (Numbers) arrives at the test node, which checks if the list is empty. The test node sends either a true or false value to the distributor.
In this case, since Numbers = [3, -5, 8, 4, -1], it will send a false token to the distributor.

If the distributor receives a "true" value, then the program ends, and nothing happens to the list. If it receives a "false" value, then it sends the list down the "false" arc. Here, Numbers goes down the false arc.

The entire list flows into two action nodes: **tail** and **head**. **tail** discards the first element (e.g. here only [-5, 8, 4, -1] will pass through), while **head** takes only the first element of the list and discards the rest (here, only 3 will go through).

The tail of the list gets sent to the recursive call of **no negs** (e.g. the whole graph is called again with Numbers = [-5, 8, 4, -1]).

Figure 7.2: Explanation of a Data Flow Program in the Practice Session



Read the question below, then click on one of the radio buttons to select your answer:

Choose the statement which best describes the goal the program seeks to achieve:

○ The program returns a list of marks > 65

○ The program compares pairs of student marks

● The program counts the number of exam passes

○ None of the above

Figure 7.3: The `passes` Program: Function Question

Subjects were first given two paper and pencil pre-tests: the paperfolding and pathfinding tests (the latter being a split half version of the full test). Both tests were timed. Following this, subjects were asked to fill out the self-report questionnaire on their prior programming knowledge.

Subjects then moved to the computer and were told to work through the experiment at their own pace. Points where questions could be asked were indicated on the screen. Subjects read through the description of either the data flow or control flow language depending on the condition. The description spanned several screens, and explained each component of the language, then showed how the individual components fit together to form a program. Subjects were able to move both backward and forward through the explanations. After asking questions of clarification (if any) to the experimenter, subjects worked through a practice problem sequence which was identical in structure to those used in the experiment: a short program (slightly less complex than those used in the experiment itself), followed by a request to summarise the program (the program was not visible at this time), and five comprehension questions, one per question type, which were shown one to a screen and accompanied by the program. Following another opportunity to ask questions of clarification, subjects worked through

Condition	**Pre-tests**	
	Pathfinding (out of 16)	**Paperfolding (out of 20)**
CONTROL FLOW		
Mean	7.6	13
Std Dev	4.58	3.86
DATA FLOW		
Mean	6.5	11.1
Std Dev	3.66	4.25

Table 7.1: Mean Scores per Condition on Pre-tests

the four problems.

7.3 Results

7.3.1 Pathfinding and Paperfolding Pre-tests

Mean scores on the pre-tests per group are shown in Table 7.1.

Although the mean scores for both tests were higher for subjects in the control flow group, the differences were not significant.

Correlations were computed between the pre-tests and between the pre-tests and experimental scores. A summary of these correlations follows:

- **Correlations between pre-tests:** there was a significant correlation between the pathfinding and paperfolding pre-tests overall (r_s = .55, p < .02), and also for the control flow group (r_s = .76, p < .01).

- **Correlations between pre-tests and overall scores:** the only correlation between a pre-test and overall performance was for the data flow group, where there was a positive correlation between the paperfolding test and overall score (r_s = .73, p < .02).

- **Correlations between pre-tests and information types items:** there was a positive correlation overall between the paperfolding test and accuracy measures on state questions (r_s = .49, p < .03). There was also a significant negative correlation, for the control flow group, between response latency on control flow questions and paperfolding scores (r_s = -.76, p < .02).

7.3.2 Self-Report Questionnaire on Programming Experience

Subjects reported knowing between 2 and 8 languages, with the mean being 3.95 overall (3.7 for the control flow group, and 4.2 for the data flow group).

All subjects knew C^{++} and COBOL as a result of their course. Basic and Pascal were the next most popular languages both overall and per group.

60% of the subjects, both overall and per group, had learned a language in secondary school, with the average number of courses taken being 1.1 in the control flow group and 1 in the data flow group.

50% of both control and data flow subjects had taught themselves a language. Of those, the mean number of self-taught languages was 1.4 in the control flow group and 2.2 in the data flow group.

When asked to rate their level of expertise on a scale of 1 to 5 (1 = novice and 5 = expert), 80% of control flow subjects reported having at least intermediate knowledge of one or more languages, compared to 70% in the data flow group. The number of languages known at this level was equal for both groups (20 languages per group). Only one subject (in the control flow group) rated himself as having expert programming knowledge.

A summary of correlations between prior programming experience and other measures is shown below. For each, it is indicated whether these correlations apply to all subjects or to the control or data flow group only.

1. **Correlations with languages learned in school:** the number of languages which were learned in school (prior to starting university), correlates both with the total number of languages known (overall: $r_s = .55$, p < .02, control flow group: $r_s = .66$, p < .04), and also with the subject's rating of him/herself as having intermediate/expert knowledge (a rating of 4 on the scale described above) of one or more languages (overall: $r_s = .46$, p < .05).

2. **Correlations with number of languages known:** the number of languages known correlates with subjects' self-rating of programming knowledge at an intermediate level (overall: $r_s = .52$, p < .02), and at an intermediate/expert level

(overall: r_s = .80, $p < .001$; control flow group: r_s = .92, $p < .001$); data flow group: r_s = .77, $p < .01$). It also correlates with subjects' scores on functional questions (overall: r_s = .58, $p < .008$; control flow group: r_s = .67, $p < .04$).

3. **Correlations between level of expertise and experimental measures:** a positive correlation between subjects' self-rating of intermediate/expert programming knowledge and functional questions approached significance (overall: r_s = .44, p = .05).

In short, breadth of knowledge seems to be associated with depth of knowledge (as evidenced by correlations between number of languages known and level of expertise reported). The number of languages known and a relatively high level of knowledge (intermediate/expert) is also associated with high scores on functional questions.

7.3.3 Language Study Time/Program Inspection Time

A measure was taken of the total time needed to "learn" the programming language, in other words, how much time subjects spent reading and studying the explanations of the language components during the practice session. The mean language study time was 285.19 seconds for the control flow group, and 463.54 for the data flow group.[4] This difference was highly significant (t = -3.81, $p < .005$).

Given that the data flow notation was slightly more complex, it required more explanation than did the control flow notation (16 screens of information as opposed to 13). The times above were corrected for by working out a 'mean time per screen', which was 21.94 seconds for the control flow group and 28.97 for the data flow group. Note that this method is not fool-proof, given that subjects could go back and forth across screens as many times as they wished. However, it seems preferable to incorporate some measure of correction rather than simply use raw scores. The difference between the corrected scores was significant (t= -2.17, $p < .025$).

Program inspection time refers to the mean time which subjects spent studying each program. For the control flow group, this was 108.58 seconds, while it was 155.41 seconds for the data flow group. This difference was significant (t= -1.77, $p < .05$).

[4] As language study time for one of the data flow subjects was missing, the mean group time was substituted.

Finally, time taken to write the program summary was checked: although there is no reason to expect this to differ across groups, there may have been a possibility that subjects used the program summary time to reflect on and consolidate their knowledge about the program following the program inspection phase, and that, for example, a short inspection time may have been followed by a longer summary writing time. The mean time program summary writing time was 312.50 seconds for the control flow group, and 320.13 for the data flow group (t = -.11, ns). Furthermore, there was a strong positive correlation between program inspection time and program summary time for both groups (r_s = .87, p < .001 for the control flow group, and r_s = .85, p < .002 for the data flow group), suggesting that subjects were not using an "inspect quickly, and reflect while writing the summary" strategy.[5]

7.3.4 Response Latency and Accuracy

- **A Note on Analysis Methodology**

  During the experiment, each subject inspected four programs and answered five questions about each program (each question representing one of the information types). Thus subjects answered four questions per information type. The questions were multiple choice, with one correct answer and three distractor items (one of the items was always "None of the above", which was the correct response in 25% of cases).

  Ideally, it would have been possible to make use of a previously validated set of questions when performing the experiment, but these do not exist. Alternatively, a set of questions could have been developed and validated prior to the experiment. This, however, is very time-consuming, as it effectively involves running an entire experiment with the sole aim of testing the questions. Instead, it was decided to perform a qualitative item analysis on the questions used in order to determine which questions were the best indicators of performance, and which were potentially contributing only "noise" to the data. Although many formal item analysis techniques exist, most were felt to go beyond the scope of the item validation required for this experiment. Nonetheless, there are many occasions when simple item analyses such as the one used in this study would benefit the

[5] The analysis of program summaries themselves can be found in Chapter 8.

psychology of programming field.

When performing an item analysis, it is worth remembering that item analysis is distinct from item selection, and the former does not determine the latter. In other words, item analysis techniques may allow items to be rated according to some criteria, such as difficulty, but, apart from items which are clearly pathological (*i.e.* for which no one chooses the correct response), they do not provide hard and fast guidelines for choosing to use an item or not. For example, a well constructed but easy item may be appropriate as a practice item but not as an item in the main experiment.

In examining the questions used in this experiment, overall item difficulty was first considered. Anastasi (1988) suggests that in order to achieve maximum differentiation between test takers, items should be chosen at the .50 difficulty level (*i.e.* 50% of persons taking the test answer the item correctly). Questions which produce floor or ceiling level effects allow little differentiation. In addition, Anastasi suggests that for multiple choice items, the desired proportion of correct answers should be set slightly higher to compensate for guessing.

Multiple choice questions also require analysis of the distractor items used in conjunction with the correct answer in order to determine whether they are functioning correctly. All distractors should preferably attract responses (Thorndike and Hagen, 1977), but it should be ensured that distractors do not distract the best subjects (Kline, 1986).

Items were selected for further consideration based on the following criteria:

- the item received at least 35% and not more than 75% of the responses (*i.e.* around a mean of 50% but with a slight upward adjustment to account for guessing (Anastasi, 1988));
- not more than one distractor received 0% of the replies;
- no distractor attracted more responses than the correct option.

As subjects answered four items per question type (data flow, control flow, function, operations and state), it was decided that this method would only be feasible if at least two items for each question type met the above criteria. This was the case for all information types, with the data flow type having three questions meet the criteria. Therefore, only these questions are considered in the analysis. Note

Condition	**Question Types**					**Overall**
	FUNC	CF	DF	OPS	STA	
CONTROL FLOW						
Mean	31.41	32.77	35.50	17.60	34.31	30.79
Std Dev	9.79	7.26	22.57	10.88	10.96	9.22
DATA FLOW						
Mean	38.24	39.71	44.49	21.66	47.63	38.91
Std Dev	24.98	22.92	16.06	7.90	19.59	9.37

Table 7.2: Mean Response Latencies (in seconds) and Standard Deviations for the 5 Comprehension Question Types

Condition	**Question Types**					**Overall**
	FUNC	CF	DF	OPS	STA	
CONTROL FLOW						
Mean	.50	.75	.57	.75	.65	.64
Std Dev	.41	.26	.39	.26	.41	.18
DATA FLOW						
Mean	.65	.40	.77	.60	.45	.59
Std Dev	.24	.32	.22	.39	.37	.12

Table 7.3: Proportion of Correct Responses and Mean Scores and Standard Deviations for the 5 Comprehension Question Types

that the item analysis was carried out for all subjects as a whole, and therefore should not favour one group over the other.

The mean response latency for all questions was 30.79 seconds for the control flow group and 38.91 seconds for the data flow group. This difference was significant (t= -1.95, $p < .04$).

The overall mean number of correct responses (out of a total of 11) was 7 for the control flow group, and 6.5 for the data flow group (Mann-Whitney U test, U= 41.5, ns).

Response latency and accuracy were further partitioned by question type. Mean response latency (and standard deviation) for each group, both overall and per question type, are shown in Table 7.2.

For response accuracy, the proportion of correct answers for each group overall and per question is shown in Table 7.3.

Figure 7.4 shows the mean response latency per group and per question type. Although response times for both groups follow the same general pattern, times for the data flow group are consistently higher.[6]

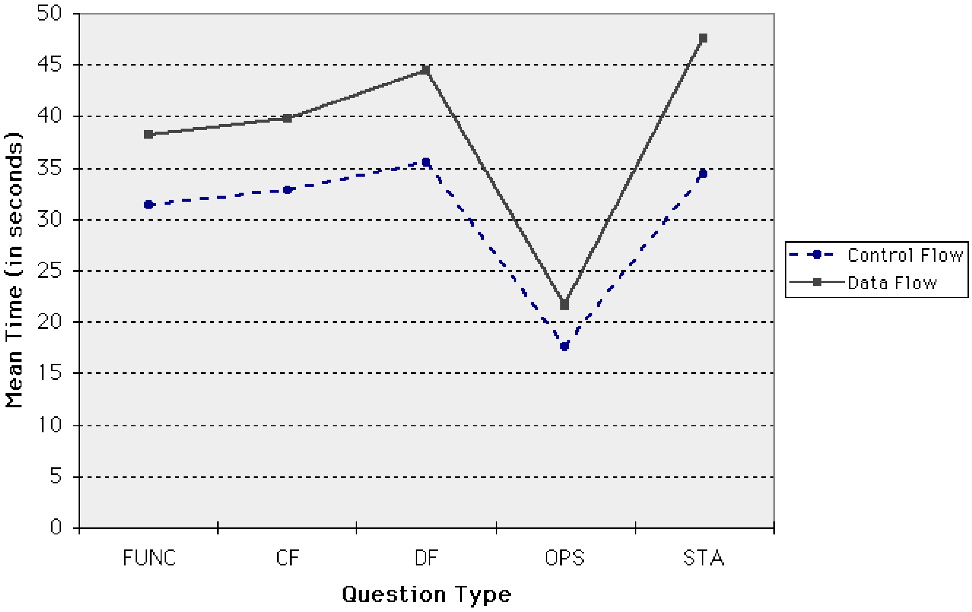


Figure 7.4: Mean Response Latency per Group and per Question Type

Figure 7.5 shows the percentage of correct responses per group and per question type. In contrast with the latency data, there is a marked differential pattern of response between the control and data flow group, with data flow subjects scoring comparatively higher on functional and data flow questions, and control flow subjects performing better on control flow, operations and state questions.

A mixed design ANOVA with group as a 2 level between-subjects factor and question type as a 5 level, repeated-measures, factor showed a main effect for group which approached significance (F=4.22, df(1,18), p = .055), and a highly significant effect for question type (F=5.96, df(4,72), $p < .001$), but no group by question type interaction (F=.24, df(4,72), ns).

As the accuracy data was deemed to be ordinal, rather than interval, parametric tests could not be used. As there is no non-parametric equivalent for mixed, repeated measures ANOVAs, the accuracy data could not be tested in the same way.

[6] Strictly speaking, a bar chart would be a more appropriate way of representing this data, as it is categorical rather than continuous. However, it is felt that the differences between groups across the information categories are more visually salient when represented in this way.

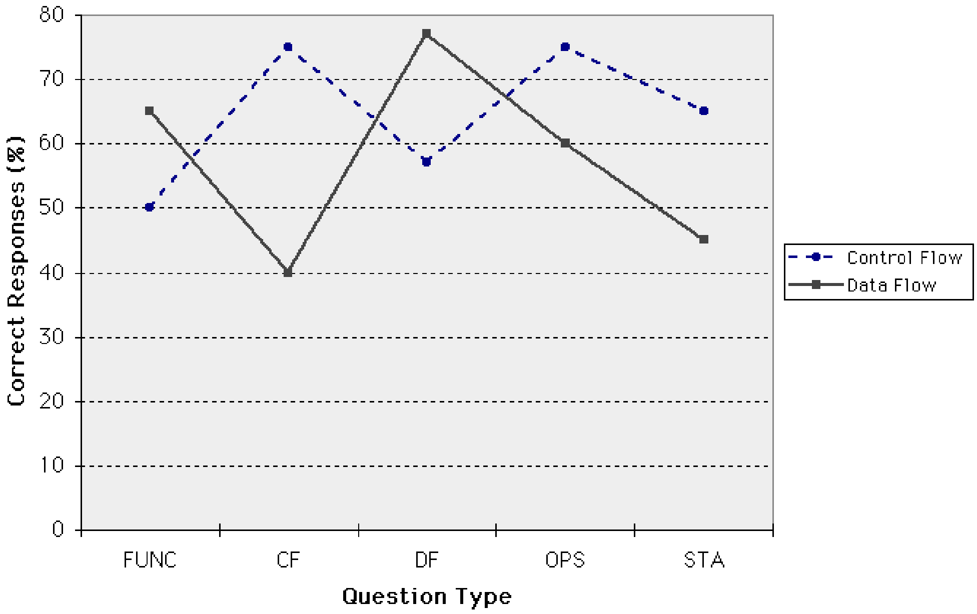


Figure 7.5: Percentage of Correct Responses per Group and per Question Type

7.3.5 Correct Responses Only

A further analysis involved taking into consideration the mean times for correct responses only, to ensure that the latency data was not being adversely affected by quick, but incorrect, responses. The response latency for correct responses only is 28.43 seconds for the control flow group and 38.88 seconds for the data flow group, which is roughly similar to the overall response latency reported above, and which, again, is significant (t=-2.04, $p < .03$).

An examination of response latencies by question type shows largely the same pattern for correct responses as for all responses. However, further analysis could not be performed because of the number of missing cases (*i.e.* subjects who had no correct answers on the questions for a particular information type).

7.3.6 The Match-Mismatch Hypothesis

The application of Gilmore and Green's match-mismatch hypothesis to this experimental situation predicts that performance on a data flow question should be better when the subject is using the data flow VPL than the control flow VPL and vice versa. In order to test this hypothesis, response latency and accuracy for the control flow and data flow questions only were considered. This information is shown graphically in Figure 7.6.

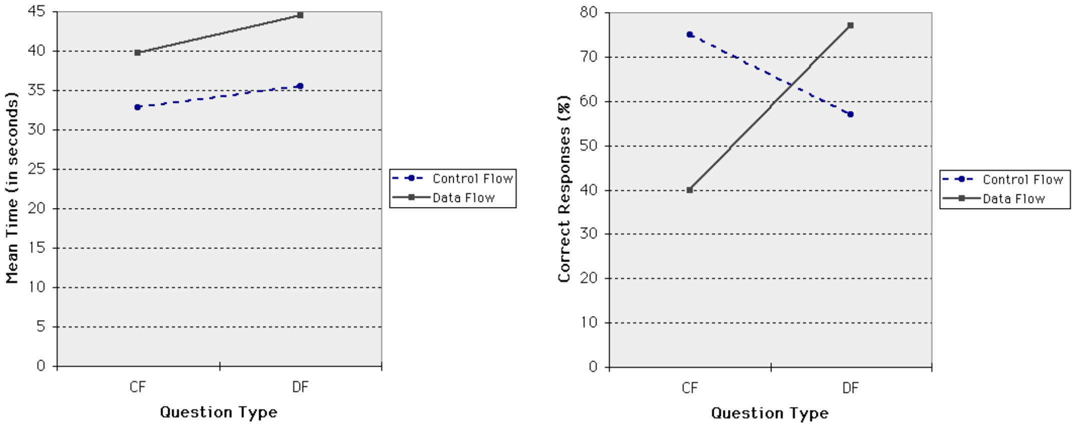


Figure 7.6: Match-Mismatch: Latency and Accuracy for Data Flow and Control Flow Questions

As can be seen, no match-mismatch effect occurs relative to latency: latency for the data flow group is slower regardless of the question type (as for all types of questions: see Figure 7.4). However, a match-mismatch effect can be observed in the accuracy data, an effect which is strongest for the data flow group. The significance of this effect was tested, and the results are shown below. Because the data are treated as ordinal, an overall test of significance could not be carried out, as there is no non-parametric equivalent for a mixed, repeated measures ANOVA.

According to the match-mismatch hypothesis, mean response accuracy will be:

- higher for the control flow group relative to the data flow group on control flow questions (**supported: Mann-Whitney U test, U= 22.5, p $<$ .04**);
- higher for the data flow group relative to the control flow group on data flow questions (**not supported: Mann-Whitney U test, U= 35.5, ns**);
- higher for the control flow group on control flow questions relative to data flow questions (**not supported: Wilcoxon signed-ranks test, T^{+}= 25.5, ns**);
- higher for the data flow group on data flow questions relative to control flow questions (**supported: Wilcoxon signed-ranks test, T^{+}= 50.5, p $<$ .01**);

Therefore, the latency data provided no evidence for the match-mismatch hypothesis (which is to be expected given the pattern shown in Figure 7.6), while the accuracy data suggested a match-mismatch effect which was, for some of the comparisons at least, significant.

7.3.7 The Grouped Match-Mismatch Hypothesis

As described in the introduction to this chapter, different types of programming information are often thought to be associated with each other, although these associations have usually been theoretically based rather than empirically tested. Information types which are not explicitly represented in the notation, but are linked with the information which is, could be said to be grouped information types. This leads to the "grouped match-mismatch hypothesis": if task X requires information X, and if information X is associated with information highlighted by the representation (information Y), then performance should be better as compared to performance on tasks requiring non-associated information (*e.g.* information Z and task Z).

This hypothesis was tested by comparing combined scores on data flow and functional questions across groups, and combined scores on control flow, operations and state questions (again across groups).

Table 7.4 shows the results for each.

Condition	**Grouped Question Types**	
	FUNC/DF	CF/OPS/STA
Control Flow		
Mean Score	.54	.72
Std Dev	.25	.21
Data Flow		
Mean Score	.72	.48
Std Dev	.10	.24

Table 7.4: Proportion of Correct Responses and Standard Deviations for Combined Question Types

This grouped match-mismatch effect is illustrated in Figure 7.7.

Mann-Whitney U tests performed on the data were significant for the control flow/operations/state grouping (U= 23.5, p < .03) and approached significance for the functional/data flow grouping (U = 30, p = .065).

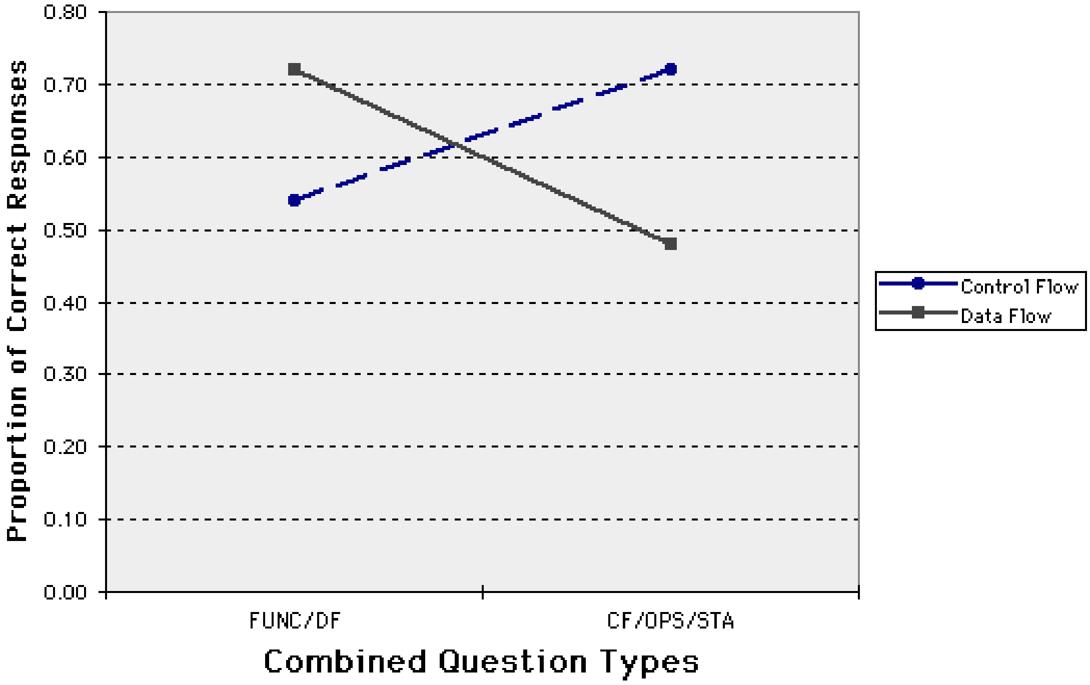


Figure 7.7: Proportion of Correct Scores on Grouped Question Types (Control and Data Flow Groups)

7.3.8 Summary of Results

The results of the experiment can be summarised as follows:

Pre-Tests: there was a positive correlation between pre-tests overall and for the control flow group, and some evidence of positive correlations between the paper-folding test and the experimental measures;

Programming Self-Report Questionnaire: the number of languages and level of knowledge correlates positively with subjects' scores on function questions (an effect also found in the experiment reported in Chapter 3);

Language Study/Program Inspection: data flow subjects spent significantly longer learning the micro-language and inspecting the experimental programs than did the control flow subjects;

Response Latency and Accuracy: the mean response latency was significantly lower for the control flow condition compared to the data flow condition. Response latency for correct responses only showed much the same pattern. There was no significant difference in overall accuracy however;

The Match-Mismatch Conjecture: the match-mismatch conjecture was supported to some extent, although not with respect to latency. For accuracy, scores on data flow questions were significantly higher than control flow questions in the data flow condition, while scores on control flow questions were higher for the control flow group when compared with scores on control flow questions for the data flow group;

The Grouped Match-Mismatch Conjecture: in terms of response patterns across individual information types, control flow subjects scored significantly higher on control flow, operations and state questions, while data flow subjects scored more highly (although not significantly) on data flow and function questions.

7.4 Discussion

Many interesting findings arose from the study reported in this chapter. This section considers them in more detail under the following headings:

- Visual Programming: what skills are involved?;
- Overall results and individual information types;
- Match-mismatch and beyond;
- Implications for teaching and design of VPLS;

Before doing so however, it considers some methodological issues.

7.4.1 Questions of Methodology

One methodological decision which may be questioned is that of taking measures of both response time and error rate. Green (1977) notes that according to Poulton (1965), subjects can be persuaded to keep either their speed or accuracy constant, meaning that the effects of that variable will appear in the other, making it much more sensitive. Previous tests of the match-mismatch conjecture used this approach, considering time as the variable of interest. Certainly, from a statistical point of view, this approach would have made sense: given that there is no non-parametric equivalent of a mixed ANOVA with repeated measures, an overall measure of the match-mismatch effect could not be calculated. It is likely that if parametric statistics had been used, an effect would have been found, however, it was necessary instead to look at differences between individual variables rather than an overall effect.

Nevertheless, it can be argued that examining both time and accuracy allowed an interesting effect to be uncovered: namely the fact that a match-mismatch effect occurs at an accuracy level, but that a "control flow supremacy" effect occurs at a latency level. In other words, questions requiring information highlighted by the paradigm may be answered more accurately than when the information must be inferred, but the time taken to answer these and all of the other questions depends on the paradigm itself, rather than on the interaction between task and paradigm. This effect would be interesting to explore further.

7.4.2 Visual Programming: What Skills are Involved?

One of the aims of the study was to investigate factors which might be involved in subjects' success (or otherwise) in using visual programming languages. Visual programming skill might be seen as having two main components: programming skill and graphical readership/aptitude skills. This skill distinction maps roughly onto a semantic/syntactic distinction, with programming skill involving the semantics of the programming domain, and graphical skill corresponding to the syntactic elements of the representation.

Part of the investigation described here centred on trying to ascertain, firstly, whether programming and diagrammatic skill can be measured, and, secondly, whether either of these correlated positively with success in using the visual language. The investigation techniques are fairly crude, but do give some initial pointers to issues which could be followed up.

Graphical Skill

As mentioned in the introduction to this chapter, one of the issues of interest in this study was to look at measures of graphical skill, with the ultimate aim of determining which skills might predict performance with a VPL. The paperfolding task, used in prior studies (Cunniff and Taylor, 1987; Petre *et al.*, 1995) appears to measure a subject's ability to mentally simulate the manipulation of objects in 3-D space, a skill that does not at first sight seem necessary for 2-D visual programming. Although it correlates to some extent with experimental measures, the overall pattern is not clear-cut.

The pathfinding test was chosen as a possible alternative to the paperfolding test. It appears to be more representative of the activities involved in tracing the execution of a visual program of the node and arc variety, namely, following a path through a quite complex representation.[7] Nevertheless, the pathfinding test did not correlate in a meaningful way with experimental measures.

Does this mean that measures of graphical reasoning are never good indicators of subsequent visual programming skill? Probably not, but it does call for a finer-grained approach to try and identify the skills which comprise visual programming skill, and then to find or possibly even develop tests which measure these abilities.

However, before discarding tests such as pathfinding and paperfolding, it seems worthwhile to study extreme scores, *e.g.* considering scores from only the top and bottom quartiles. The scores reported in this study do show some variation, but there are few extreme scores. The paperfolding and pathfinding scores from the first, unreported, study were substantially higher, but again, did not contain sufficient variation. Unfortunately, cross study comparisons cannot be made due to other, extraneous differences between subjects (namely, large differences in the courses being taken, the objectives of the course, and subjects' general academic background).

Programming Skill

The study also attempted to determine whether programming skill correlates positively with overall performance. Programming skill was measured via a self-report questionnaire, a less desirable option than actual performance measures, but in this case, the only viable one. Information such as marks in programming courses were not obtained for ethical reasons, while it was not possible to include a programming pre-test (such as the one used in the unreported experiment) because of time constraints.

[7] Even if it could be argued that the pathfinding test requires only one path to be followed, unlike most data flow programs.

Nevertheless, a positive correlation was found between prior programming experience, both in terms of breadth and depth of experience, and scores on functional questions. The fact that it does not correlate with overall accuracy and response latency suggests that one should not read too much into the correlation. On the other hand, the fact that the correlation involves functional questions rather than other types of questions is probably noteworthy: previous studies claim that a functional understanding of the program is associated with greater expertise and/or deeper levels of comprehension (Adelson, 1984; Pennington, 1987b; Corritore and Wiedenbeck, 1991).

7.4.3 Overall Results and Individual Information Types

The overall results of the study are intriguing: data flow subjects take longer to learn the micro-language (compared to control flow subjects learning the control flow micro-language), and they spend more time studying the programs. Although their responses follow the same pattern as for control flow subjects in that questions that are answered quickly for control flow subjects are also answered comparatively quickly by data flow subjects, data flow subjects have slower response times, not only overall, but on every type of question. This extra time does not seem to result in improved performance, in fact, performance of the data flow group is slightly worse than the control flow group.

One question which could be asked was whether the programs used in the experiment were biased toward control flow, *i.e.* they were more easily represented in a control flow manner. This issue arose when deciding how to represent recursion: for some programs, the programmer assisting with language design felt that control flow representations were more amenable to representing recursion as tail recursion, while data flow representations were more appropriate for representing recursion as non-tail recursion. However, for the programs in question, it was decided to use a single type of recursion in both representations, and to bias it toward the data flow representation, *i.e.* to use non-tail recursion for both representations. As the results show, this did not have the effect of improving the performance of the data flow subjects relative to their control flow counterparts.

On the basis of the results, it could be concluded that data flow is not really a contender in control flow/data flow comparisons, given that it leads to longer response times for slightly less accurate responses.

However, accuracy, while not significantly different overall, shows a markedly different pattern when examined at the level of individual information types: a match-mismatch effect occurs, with subjects in the data flow group scoring higher on data flow questions as compared to the control flow group (and also to the data flow group's scores on control flow questions). Similarly, subjects in the control flow group score higher on control flow questions relative to the data flow group (and to their own scores on data flow questions). Thus, focusing only on overall performance ignores interesting differences which are occurring at an information types level.

7.4.4 Match-Mismatch and Beyond

As mentioned above, match-mismatch effects occurred in the accuracy data, some of them significant. This seems to indicate that representations of control and data flow paradigms, at least the ones used in this study, do in fact highlight control and data flow information respectively, an assumption which was made in the introduction to this chapter. This is an important issue, as visual programming languages differ from the program abstractions discussed by Pennington (1987b) in that they must, by definition, be executable. Therefore, although they might highlight some types of information by virtue of the fact that they embody a particular paradigm, they cannot exclude information which might be considered irrelevant in a "pure" abstraction. The requirement for completeness could well have clashed with the requirement for highlighting, but it does not look like this was the case.

In addition to this simple match-mismatch effect, a *grouped* match-mismatch effect also seems to occur, operating at the level of combined information types. Information thought to be associated with data flow or control flow information, either from a theoretical or a representational point of view, did in practice seem to be highlighted by the representation with which it was thought to be associated. Control flow representations showed a procedural grouping, including low-level operations and state information. Data flow representations appeared to highlight higher-level concepts such as function. Therefore, the control flow group scored highly on control flow, operations and state questions as compared to the data flow group, who scored more highly on data flow and functional questions.

An objection could be made to this interpretation which cannot be discounted. The point was made in Section 7.1 that the functional information type was related to data flow from a theoretical point of view, while state and operations were related to control flow from a representational point of view. Just as it was necessary to question the relationship between paradigm and abstraction, one must also query that between question type and paradigm, in particular, whether functional questions relate to the functional paradigm in any meaningful way, or whether they simply share the same name. Functional questions are concerned with the overall goals of the program. Functional paradigms are based around the idea of evaluating functions which use input values as arguments and whose value is the result of the computation. In that sense, it is difficult to say that function questions are effectively tapping into the functional information associated with a data flow paradigm. This distinction does not seem to operate in the same way for the other information types: state and low-level operations do appear to be information types which are both explicit in the control flow representation, and whose definitions correspond to the type of information required by the corresponding question. One hypothesis is that representational associations are more salient than semantic associations.

In the data flow/function association, an alternative explanation may be that the more difficult and unfamiliar the representation, the more time must be invested in understanding it. This in turn leads to a more complete understanding of the program, and hence, to improved performance on abstract questions such as the functional ones. This point is returned to in Chapter 8.

7.4.5 Implications for Teaching and Design

The findings of this study have a number of implications, both from the point of view of teaching visual programming, and of designing visual programming languages.

Firstly, the way in which a program is represented does make a difference to program comprehension. Data flow representations favour data flow comprehension, likewise for control flow representations and control flow comprehension. There was an obvious procedural bias evident in the languages known by subjects, which was probably reinforced by the course of studies in which they were engaged (a practical, mainstream computing course, with little emphasis on alternative or experimental paradigms). This bias may have favoured response latency for the control flow representation, but the pattern of response accuracy shows that the representation does have an effect.

The implications are encouraging for teachers of computing who wish to expose their students to different paradigms. This study suggests that changes in paradigm lead to changes in the information which is accessed from the representation. Changing paradigm may in itself be a useful teaching aid when addressing the issue of program comprehension.

Furthermore, the study seems to indicate that these effects still occur when there is considerable "procedural bias", in other words, when students have been exposed either solely or primarily to procedural programming languages, and/or been taught from a procedural perspective. Thus, students may not necessarily be "harmed" by initial exposure to procedural languages, as Wells and Kurtz (1989) and others have feared.

Note that the topic of interest here is a representational issue related to program comprehension by novice programmers. The author is not suggesting that paradigm in any way influences program design, in the sense that learning a particular paradigm will cause students to think and to write programs in that paradigm. It has already been suggested that programmers, at least experts, design programs in a personal pseudo language and can switch mentally between paradigms as required (Petre, 1996).

In terms of language design, the basic implication is that there is a mapping between tasks requiring information of a particular type, and language paradigms which highlight that information. Additionally, the relationship between the semantic and representational levels goes beyond a single type of represented information, and encompasses information which is semantically related to the information being represented. This offers an opportunity which can be capitalised on when designing a representation.

7.5 Chapter Summary

This chapter reported on an experiment which compared small control and data flow VPLs. The study used a combination of the match-mismatch conjecture and the information types methodology in order to investigate the effect of paradigm on comprehension, both in terms of the information purported to be highlighted by the representation, and in terms of general comprehension. The results were split across the hypotheses in the sense that control flow subjects learned the languages and performed all tasks more quickly than data flow subjects, regardless of the match between paradigm and task (control flow supremacy hypothesis). However, the accuracy scores, largely similar overall across groups, showed differential patterns of response for the question types corresponding to the representation in question (match-mismatch hypothesis). In addition, a "grouped match-mismatch effect" was found whereby tasks requiring information types thought to be related to the information type highlighted by the representation also appeared to be facilitated.

One aspect of the data analysis which has not been covered in this chapter is that of the program summaries. As mentioned in Chapter 3, it was necessary to develop a new methodology for analysing this data. Chapter 9 presents the analysis of the program summary data, and shows that it lends further support to the findings discussed in this chapter.

Chapter 8

A New Methodology for Program Summary Analysis

8.1 Introduction

Program summaries have played an important role in the information types methodology, and data of this type was collected by Pennington (1987b), Corritore and Wiedenbeck (1991) and in the experiments described in Chapters 3 and 7.

A program summary is a free-form account of a program which a subject produces after studying the program. Instructions given to subjects have tended to be relatively non-directive, leaving the content essentially up to them. Although Pennington does not provide details of the instructions she gave to her subjects, Corritore reported in a personal communication that her instructions were "functionally oriented". The experiment reported in Chapter 3 followed Corritore's lead and asked subjects to "write a short summary of what this program does". Later experiments, *e.g.* the one described in Chapter 7, used a more open-ended wording, asking subjects the following: "Now write a short summary of the program you just saw." The lack of explicit guidelines for the content of the summary allows for wide scope and variation in the responses.

The advantages of the program summaries which are produced is that they are valuable sources of rich data. Furthermore, the program summary methodology neatly circumvents the problems of 'false positive' results often associated with binary choice questions, and the difficulties associated with developing sensitive and reliable multiple choice questions and corresponding distractor items. Program summaries allow subjects to express their view of a program, using their own words, at their chosen level of abstraction and providing as much (or as little) detail as they feel is necessary.

As always, the price to pay for rich data is the difficulty of analysing it: quantitative statistics are not always appropriate, and qualitative methods must be devised. There are numerous ways of analysing written texts of this type, however, it is not simply a case of identifying the 'correct' method in the same way one chooses the right statistical test, particularly when the semantic content of the text is of interest. Given differences in research aims between studies, it is almost impossible to choose a scheme 'off the shelf' as it were, and Bakeman and Gottman (1997) are of the opinion that borrowing a pre-existing analysis scheme is akin to "wearing someone else's underwear". Although this may be stretching the point, it does suggest that analysis schemes are not universal: they are both content and context sensitive, and must be developed through what is often a lengthy, iterative process.

The complexity of the analysis is related to the *replicability* of the analysis. Rich, complex data may lead to complex analysis schemes: extra care must be taken to ensure that these schemes are in fact understandable and usable, and are no more complex than is necessary for the purpose of the analysis. Replicability can also be compromised by schemes which are ill-defined. Schemes which are not fully worked out and/or which are not accompanied by explicit instructions enabling them to be used by persons other than the original researcher are not of much use: it is impossible to compare results reliably.

Previous analyses of program summaries by Pennington (1987b), and Corritore and Wiedenbeck (1991) have suffered from some of these problems. Pennington carried out her analysis by dividing the summaries into statements, and classifying the statements as being of one of three types: *data flow*, *control flow*, or *functional*. The data was used to supplement that obtained from the multiple choice questions. Although this may be a worthwhile aim, reports of program summary analyses have been sketchy, with little in the way of a methodology or instructions for performing the analysis. This has rendered previous analyses unreplicable, and therefore made it impossible to compare results across studies.

In an attempt to overcome these difficulties, this chapter first describes the methodology used in previous experiments (Pennington, 1987b; Corritore and Wiedenbeck, 1991), and looks critically at the difficulties which were encountered when attempting to reapply the scheme. Following this, it considers other methods for analysing program summary data, and proposes a new analysis scheme based on a program hierarchy. The advantages and disadvantages of the scheme are then discussed.

8.2 Pennington's Methodology for Program Summary Analysis

Analysing program summaries as a way of measuring program comprehension can be traced to an experiment carried out by Pennington (1987b). In addition to answering binary choice questions about a program of moderate length, subjects were also asked to write a summary of the program. Although the exact wording of the request is not given, it is more than likely that the instructions were brief and non-directive with respect to the type of information the summary should contain.

Summaries were requested at two points during the experiment: firstly after a 45 minute study period, and again after having carried out a modification to the program. Pennington hoped to compare the two in order to see how their focus changed over time. Unfortunately, she was unable to do so, as rather than writing a second summary, subjects tended to simply refer to their earlier summaries, and then describe the modification they had just carried out. This is an interesting example of how a change in context resulted in a change in behaviour even though the task remained the same.

Pennington performed two analyses on the program summaries, classifying each statement by both *information type* and *level of detail.* The results of Pennington's analysis can be found in Chapter 2, the focus here is on the methods used in the analysis, described in the following two sections.

8.2.1 Information Type Analysis

Pennington states that the information types investigated *included* procedural, data flow, and function statements. The other categories used in the program comprehension tests, namely operations and state, do not seem to have been used: no results were reported for them in any case. Why they were omitted from the analysis is not discussed. Pennington's definition of each category is very brief, and expressed primarily through examples. She defines the three categories as follows:

- "**procedural statements** include statements of process, ordering, and conditional program actions." (Pennington, 1987b, p. 332);
- "**data flow statements** also include statements about data structures" (Pennington, 1987b, p. 332);
- **functional statements** are not defined by Pennington, merely illustrated with an example.

Pennington provides the following summary excerpts to illustrate each type of statement, all from (Pennington, 1987b, p. 332):

Procedural: "after this, the program will read in the cable file, comparing against the previous point of cable file, then on equal condition compares against the internal table ...if found, will read the tray-area-point file for matching point-area. In this read if found, will create a type-point-index record. If not found, will read another cable record."

Data flow: "the tray-point file and the tray-area file are combined to create a tray-area-point file in Phase 1 of the program. Phase 2 tables information from the type-code file in working storage. The parameter file, cables file, and the tray-area-point file are then used to create a temporary-exceed-index file and a point-index file."

Functional: "the program is computing area for cable accesses throughout a building. The amount of area per hold is first determined and then a table for cables and diameters is loaded. Next a cable file is read to accumulate the sum of the cables' diameters going through each hole."

8.2.2 Level of Detail Analysis

In terms of level of detail in program summaries, Pennington defined four levels:

detailed : references to a program's operations and variables;

program : references to a program's "procedural blocks";

domain : references to real world objects;

vague : statements with no specific referents.

Pennington uses the example summary segments above as illustrations of the level of detail: the procedural summary is the most *detailed*, the data flow summary is described at the *program* level, the functional summary is described at the *domain* level, and an example of a vague statement is, "this program reads and writes a lot of files." (Pennington, 1987b, p. 333).

Little detail is provided about the analysis: the above definitions comprise the description of the scheme, while the coding process itself it not mentioned.

8.3 Analysing Analysis Schemes

At a joint workshop between researchers at the Computer Based Learning Unit of the University of Leeds, the Department of Artificial Intelligence (University of Edinburgh), and the Human Communication Research Centre (also University of Edinburgh) on the analysis of educational dialogue, a preliminary *desiderata* for coding schemes was drawn up.

The list was based on a large-scale exercise which involved coding a range of educational dialogues using various schemes and comparing the results. The hope is that this list will continue to be developed and fleshed out so as to ultimately provide a useful tool for judging the relative utility of a coding scheme, and also for comparing two or more schemes.

In the meantime, it was decided to select the relevant points from this list[1], to revise and restructure it, and to use it to analyse Pennington's two coding schemes, along with the scheme which is proposed in later sections. The main changes to the initial list were to separate the various questions into those concerning the scheme itself and those concerning the application of the scheme. Also, at the workshop, a separate list was drawn up which covered comments arising from practical experience: these were transformed into questions which could be asked of a scheme.

Note that this approach is firmly interdisciplinary in that it is taking ideas and methods from discourse analysis and applying them in the psychology of programming field. As

[1] Given that the framework was designed to look at coding schemes for analysing dialogue arising in educational settings, not all of the points are applicable.

such, it should not be read as a criticism of work which has previously been carried out or as implying that the work is faulty or incomplete in some way. The aim is that its application will highlight the need to consider the methodological issues involved in devising quantitative analyses for current and future work.

Furthermore, the scheme is being applied to what is essentially a reconstruction of Pennington's program summary analysis based on the information contained in (Pennington, 1987b). It may well be that Pennington's analysis has been spelled out more thoroughly in other documents, and therefore it should be borne in mind that it is the reconstructed program summary analysis scheme which is being contrasted with the scheme proposed in Section 8.6.

The revised list of questions is shown below:

Analysis of the Scheme

- What is the theoretical background of the scheme?
- What are the aims of the scheme, in other words, what hypotheses does it allow one to test, either in actual fact, or potentially?
- To what domain is the scheme applicable?
- Does the scheme come with a coding manual which describes how to apply it (thus ensuring an improved chance of reliability)?
- Is the scheme accompanied by an example transcript?
- Categories:
  - How many categories does the scheme contain?
  - Is there a hierarchical structure to the scheme (*i.e.* categories at different levels)?
  - If there are different levels, are they interdependent?
  - What is the level of granularity of the categories (*e.g.* segment, sentence, utterance, paragraph)?
  - Are the categories orthogonal?
  - Are the categories mutually exclusive?

  - Are coding examples provided for each category in the scheme?

- Does the scheme account for non-ideal behaviour?

Analysis of the Application of the Scheme

- Is hindsight allowed in coding or must a dialogue/discourse be coded sequentially?
- How many coders were used?
- Were the coders trained?
- If more than one, was the reliability reported?
- Categories:
  - Is the number of categories manageable? (*i.e.* do some categories end up being unused because there are so many of them?)
  - Is it straightforward to make a category decision? Is there any decision process provided for doing so (*e.g.* a decision tree?)
  - Does the scheme include default categories which are misused (*i.e.* they become "bucket" categories)?
- Is the scheme accompanied by analysis tools?
- If there is ambiguity in applying the scheme, is it due to the scheme or to the real world?

8.4 A Critical Analysis of Pennington's Methodology

This section describes the application of the categories outlined above to Pennington's scheme. Each of the questions is restated, and its applicability to Pennington's scheme is discussed.

Pennington's classification consisted of two separate passes over the text, considering statements firstly in terms of information types, and then in terms of level of detail. Some of the questions apply to the scheme in general, such as its theoretical basis, while others, such as the number of categories in the scheme, can only be answered by looking at the information types scheme and the level of detail scheme in isolation. If the latter is the case, it will be indicated below.

Analysis of the Scheme

What is the theoretical background of the scheme?

The scheme is part of a study which investigates the nature of programmer's mental representations. It is based on theories of text comprehension elaborated by van Dijk and Kintsch (1983) and applied to program comprehension by mapping structures in text comprehension theory onto a program's organisational structure (*e.g.* plan knowledge).

What are the aims of the scheme, in other words, what hypotheses does it allow one to test, either in actual fact, or potentially?

Pennington used the scheme to investigate programmers' mental representations, both in terms of their nature and in terms of how they might change over time as a result of a change in the programmer's task goals (although it proved to be difficult to determine the latter via the program summaries: see Section 8.2 for a discussion of this).

To what domain is the scheme applicable?

Program comprehension.

Does the scheme come with a coding manual which describes how to apply it?

No. This is unfortunate, as it would seem to be necessary in order to replicate Pennington's results. An operational definition of the entire coding scheme in the form of a coding manual would allow one to answer such questions as what should be done first, how to proceed, whether coding should be sequential and/or whether one can go back and change things, at what level (if applicable) the coding should start, and if more than one scheme is being applied, whether one should be applied before the other, etc.

Is the scheme provided with an example transcript?

Not an entire program summary, but summary excerpts of a few sentences in length are provided.

Categories:

- ***How many categories does the scheme contain?***

  Information Types: 3

  Level of Detail: 4

- ***Is there a hierarchical structure to the scheme (i.e. categories at different levels)?***

  No. There is no subcategorisation in either classification.

- ***If there are different levels, are they interdependent?***

  As mentioned above, neither classification has levels, however, the two classifications are interdependent to some extent. Pennington states that a relationship between the two was observed in that a majority of procedural statements were expressed in terms of program objects, while functional statements were expressed in terms of real world objects. This relationship between classifications is somewhat unclear nonetheless, as not all types of statements can be expressed at every level of detail. For example, a basketball team (domain) could also be described as a list of numbers (program or detailed, depending on how one reads Pennington's classification). However, when the object being described is a program, references will almost always be made in program terms, as it is hard to imagine which domain terms could be used to refer to a program in general, or how a reference to a program could be couched in detailed terms (the latter is usually reserved for the internals of the program). Likewise, a procedural statement will likely be described at a detailed level, as the low-level details of a program often cannot be described in domain terms.

- ***What is the level of granularity of the categories (e.g. segment, sentence, utterance, paragraph)?***

  Pennington describes the classification in terms of 'statements' but unfortunately does not define the term. It could be assumed that it is the smallest segment of text to which one category can unequivocally be applied. On the other hand, the statements may have been larger, with more than one code being applicable, and the code representing the "best fit' to the data being the one that was applied.

  With respect to the level of detail analysis, the granularity of classification was unclear: did the classification refer to program actions, to program objects, or both? It is likely to be both, but again, this is not stated.

- ***Are the categories orthogonal?***

  It would be more appropriate to say that the two *classifications* (information types and level of detail) are orthogonal.

- ***Are the categories mutually exclusive?***

  Information Types: This is very difficult to ascertain without practical experience and will therefore be addressed in the section on the application of the scheme. At first glance, the categories do appear to be mutually exclusive: certainly, notions of function, control flow and data flow are common parlance, and their definitions do make clear the distinction between each.

  Level of Detail: Yes, these appear to be mutually exclusive.

- ***Are coding examples provided for each category in the scheme?***

  Yes, for both schemes, although these are very vague. A marked-up summary segment is not provided, rather Pennington shows segments which "include" a majority of statements of a particular category.

Does the scheme account for non-ideal behaviour?

No, the scheme does not consider errors.

Analysis of the Application of the Scheme

Is hindsight allowed in coding or must a dialogue/discourse be coded sequentially?

Not mentioned.

How many coders were used?

It is not stated whether Pennington herself classified the statements or whether an independent rater was used. Likewise, there is no mention of there having been more than one rater. It is quite likely that Pennington herself acted as the rater.

Were the coders trained?

Again, it is not clear, but assuming that Pennington both developed the schemes and coded the summaries, the question is not applicable.

If more than one, was the reliability reported?

No report on reliability (but reliability cannot be measured if only one coder is used).

Categories:

- ***Is the number of categories manageable? (i.e. do some categories end up being unused because there are so many of them?)***

  There are only 3 information types categories and 4 level of detail categories, therefore this isn't a problem.

- ***Is it straightforward to make a category decision? Is there any decision process provided for doing so (e.g. a decision tree?)***

  Category distinctions are very hard to make in both classifications, and no decision process is provided.

  As mentioned above, Pennington's category definitions are sketchy, and based essentially on examples which "include" these categories. Also, it is not clear why only a subset of the information types categories from the comprehension questions were used in classifying the summaries. Without a more precise definition of each category, and further examples, it is impossible to apply the categories to other data.

  Obviously, terms such as *data flow* and *control flow* certainly have some meaning within a computer science context, and are used frequently with, one hopes, a shared definition. One can assume that they denote concepts which are agreed to exist at some abstract level. However, classifying statements requires precise, operational definitions of the terms, and unfortunately, these are not provided.

  Clear definitions are particularly important in the case of borderline statements and ambiguous cases. Such a case arises in one of the examples provided by Pennington. The following sentence is classified as *procedural*:

  > If not found, will read another cable record.

  while the statement below is classified as a *function* statement (or at least it is taken from a summary which Pennington says "contains many function statements" ...).

  > Next a cable file is read to accumulate the sum of the cables' diameters going through each hole.

It is difficult to distinguish between these statements. It may be that it hinges on viewing the program as an active or a passive agent: in the procedural case, the program is active, while in the functional case, the data is simply read in. This is only conjecture, as the information provided in the paper does not allow one to make this distinction. This does not mean that the distinction is invalid: it simply suggests that more precise definitions are necessary, along with a number of explicitly marked up examples. Rather than giving excerpts of program summaries, it would be helpful to provide several complete, coded, program summaries. Finally, a decision process, such as a flowchart or a decision tree, would facilitate coding.

- ***Does the scheme include default categories which are misused (i.e. they become "bucket" categories)?***:
  This is difficult to judge without having managed to apply the scheme, but there is no default category, and it seems unlikely that any particular category would be overapplied.

Is the scheme accompanied by analysis tools?

No, or at least it does not appear to be.

If there is ambiguity in applying the scheme, is it due to the scheme or to the real world?

Information Types: There is considerable ambiguity in applying the scheme, probably due both to the scheme itself (lack of clear definitions) and to the real world (interrelatedness of the different categories of information in programming).

Level of Detail: This classification does not seem to suffer from the same sort of ambiguity.

Summary

The primary criticism which can be levelled at Pennington's analysis is the lack of detail. This is not to say that the analysis scheme itself is necessarily inadequate in some way, but the absence of detail does not allow this to be ascertained. In fairness, the program summaries are only one source of data from the experiment rather than the main focus, which may explain to some extent why little information is provided about their analysis. On the other hand, the distinct lack of information as to how the program summary analysis was carried out makes it impossible to replicate.

8.5 Alternative Schemes

Several attempts were made to apply Pennington's scheme using one rater, two raters and groups of raters. In the latter cases, reliability proved to be extremely low. This was likely due to the lack of information and detail as to the coding scheme and its application. Furthermore, the programs used in the current study were much shorter than those used by Pennington, which has an effect on the way in which statements are classified (*e.g.* a description of a piece of code which produces an average may be considered quite low-level in a very large program, but when the program is only 10-15 lines long, the description may in fact represent the program's main function). This factor may also have had a negative effect on attempts to apply the scheme.

However, Pennington's distinctions between information types are very pertinent, particularly within the context of comparative studies: they have the potential to shed light on the question of whether differences in language (or representational style) lead to differences in subjects' descriptions of their understanding of a program.

In the course of trying to carry out analyses of the program summaries, it became clear that a new scheme was necessary. A number of ideas were tried out, and rejected for various reasons: they are described in the sections below.

8.5.1 Analysis by Summary

An initial attempt was made to characterise an entire program summary rather than looking at individual statements, again using the following information types: function, data or control. It was thought that this technique might be viable based on the assumption that the program summaries produced in the studies described here would be more coherent, and represent more of a 'finished product' than those analysed in the study by Corritore and Wiedenbeck (1991). As Corritore and Wiedenbeck did not allow subjects to view the program they had studied when answering the comprehension questions, the program summary may have played the role of an *aide mémoire*, with subjects jotting down anything of potential use in answering the questions. In the studies described in Chapters 3 and 7, the programs were shown alongside the comprehension questions, therefore subjects did not need to rely on the program summary to answer the questions. It was felt that this situation might allow subjects to 'coordinate' their summaries, thus producing a more coherent account of the program, rather than simply a summary of things that might later prove to be useful.

The primary disadvantage of looking at the entire summary is that it is too coarse-grained: it applies a single category to summaries which may vary considerably in length and detail, and which contain varying percentages of the three information types. Furthermore, it suffers from the same problems as Pennington's analysis: namely, the categories are too ill-defined to be applied reliably and consistently.

8.5.2 Linguistic Analyses

Two further analyses were carried out which focused on the linguistic structure of the program summary rather than on the content: the identification of cue phrases, and 'keywords in context'. Both involved taking a lower-level, bottom-up approach and investigating the way in which program summaries were structured.

Cue phrases are defined as "phrases whose function is to link spans of discourse together" (Knott and Dale, 1994, p. 45). Examples of cue phrases are *firstly, after, or else, moreover.* It was hoped that looking at the occurrence of cue phrases in the summaries would lead to the discovery of differential patterns of use between the two groups. The analysis involved identifying and counting the cue phrases in each summary.

Although interesting, analysing program summaries in terms of cue phrases presented a number of problems: firstly, they focus on very limited segments of the program summary, therefore, they would need to be used in conjunction with another type of analysis. Secondly, not all of the cue phrases are meaning independent in the programming domain. For example, many common cue phrases are also common control keywords in programming languages, *e.g. if, then, else, otherwise.* This confusion meant that the analysis could, for some cue phrases, be mistaken for a semantic analysis, when this was not the intention.

The "keywords in context" indices provided a listing of each word in the summary, surrounded by its context, *i.e.* the words which preceded and followed it. Reframing the data in this way was interesting in two respects: it highlighted the repeated use of particular words of interest, *e.g.* data objects and verbs describing program actions, and by showing them in their context, allowed us to check how subjects were using expressions. Keywords in context provided an initial way of detecting patterns of language use so as to begin to think about what they might mean in terms of an information types classification: this is discussed in the next section.

8.6 Proposal for a New Scheme

As discussed above, it is not feasible to apply Pennington's scheme as it stands, at least not on the basis of the information provided in (Pennington, 1987b). However, it is important to be able to classify program summaries according to their informational content: they provide useful information, perhaps about the way programmers mentally conceptualise a program, but certainly about how they choose to express their understanding of a program, and the possible role of program representation in the process.

Therefore, two new schemes for program summary analysis are proposed. The classification is similar to Pennington's in that it depends on two passes through the summaries: one based on *information types* and the other based on *object descriptions.* The information types classification is a more finely-grained and fully specified refinement of Pennington's scheme.

The object classification is essentially a more restricted version of Pennington's levels of detail: it was decided to focus solely on data objects within the program, as describing program events in terms of level of detail was felt to entail an unwanted overlap with the information types classification. For example, if one is describing a program action, it is difficult to differentiate between describing that action in procedural terms (information types) and program terms (levels of description). Furthermore, looking at object descriptions allows one to focus on those objects which can be described in various ways (*e.g.* a basketball team, a list of heights, or a list of numbers) and to investigate the ways in which subjects choose to describe program objects.

8.6.1 Information Types Classification

The information types classification is used to code summary statements on the basis of the information types they contain. In the sections below, the categories which make up the classification are first described, followed by a short discussion of the relationships between categories, and the way in which they fit together to form a program summary.

Information Types Categories: Descriptions and Examples

The information types classification comprises eleven categories, described below with examples of each. Note that the examples are taken from actual transcripts, and spelling and punctuation have not been corrected. In some cases, segments preceding or following the segment of interest have been included to provide context and aid understanding (shown in square brackets).

function: the overall aim of the program, described succinctly. In the case of the short programs used in the experiments described here, one function code (and more infrequently, two) will be sufficient to describe the program goals.

> – The program is selecting all players over a certain height and allowing then to join the yeam.
>
> – The program checks the heights of a list of basketball player to see who is over 180cm tall.
>
> – The program calculates the differences between the input distances ...

actions: events occurring in the program which are described at a lower level than function (*i.e.* they refer to events within the program), but at a higher level than operations (described below). For example, an action may involve a small group of nodes rather than one node only. Alternatively, it may be described as operating over a series of inputs, or describe actions in non-specific ways, *e.g.* describing tests in general, rather than the exact tests being carried out.

> – This sub-program checks each individual element of this list ...
>
> – 'Sun Span' is then worked out.
>
> – ... comparing each of the other elements ...

– The program makes two checks ...

– ...they are stripped in turn out of the list.

operations: small-scale events which occur in the program, such as tests, assignment, etc. Operations usually correspond to one node in a VPL, or one line of textual code.

– ...then the program sets the height to head(height) ...

– ...then it increments the counter by 1 ...

– A selector checks to see if the set is equal to [ ] ie 0 ...

– the head value and the previuse head value are then subtracted.

state-high: a high-level definition of the notion of state. Pennington didn't include a state category, but it was felt that one was necessary in order to account for statements which describe the current state of a program when a condition has been met (and upon which an action is dependent). State-high differs from state-low in terms of granularity: the former describes an event at a more abstract level than the latter (which usually describes the direct result of a test on a single data object). The relationship between the two is akin to the relationship between actions and operations.

– Once all the elements have been processed ...

– [The program goes through a set] until it finds 5 heights greater than 180 ...

– [The program takes a set and goes through it adding one to a counter (originally set at 0)] if the value being examined at the current iteration is greater than 65.

– [The program continues] until there are no player left unchecked in the list

state-low: a lower-level version of the category described above. State-low usually relates to a test condition being met, or not met, and upon which an operation depends. Again, this category was felt to be necessary as many summary statements describe not only the tests in the programs, or the operations following the tests, but also the results of the tests, in other words, the state of a particular data object within the program.

- If the head is greater than 180 ...
- If the head is greater than 65 ...
- ... when the test is empty is true ...
- ... if empty distances (eg[]) ...
- If the height test is true ...

data: inputs and outputs to programs, data flow through programs, and descriptions of data objects and data states.

- The program accepted a list of numbers indicating sunhours.
- ... it then passes a list of heights to a sub-program ...
- ... the heights over the height are sent ot the team ...
- ... the final result of the program is the number of players over the height 180.
- ... all heights are entered into an empty set and sent out as results.

control: information having to do with program control structures and with sequencing, *e.g.* recursion, calls to subprograms, stopping conditions.

- ... and the sub-program is called recursively.
- ... the nested recursions begin to unwind.
- It exits the program and goes back to the main program ...

elaborate: further information about a process/event/data object which has already been described. This also includes examples.

- [If the current mark is above a certain pass level] (65 in this case) ...
- [The head(numbers) is assigned to one variable] (which I'll call mark) ...
- ... [the head value and the previuse head value are then subtracted] and the difference taken.

meta: statements about the participant's own reasoning processes, *e.g.* "I'm not sure what this does".

- .... I can't remember!
- Dhoo! forgot where that route went!!!

> ... [and then joins it to the other value it would if created if he had done what i just said] (complicated).

unclear: statements which cannot be coded because their meaning is ambiguous or uninterpretable. This category is not synonymous with an 'error' category however: statements which are not correct within the program, *e.g.* "the program produces a list" when it in fact produces a number, but which can be categorised as being of a particular type are classified.

> – [If the height is greater than 180, 1 is added to the counter] and the height is recorded. *It is not clear here whether 'recorded' means 'printed', 'added to a list', 'assigned to a variable' ...*
> – [This program takes a number of sunny hours] and determines whether the amount of sunny hours is hi or lo. *The program (which is also the subject of the descriptions below) in fact calculates the range between the highest and lowest numbers.*
> – The program is listing how many hours of sun there was only when the sun was High.
> – [Then the results are sent to a set through a selector process] to make sure the results are HiLn ...

incomplete: statements which cannot be coded because they are incomplete. Statements which fall into this category tend to be unfinished sentences (examples were not felt to be necessary here). Note that this only occurred with the Prolog experiment, where subjects were timed.

Information categories are related to each other in terms of level of granularity, which can be envisaged as follows: at the top level, the program can be described in terms of a small number of functions (in some cases, just one, given that the programs being examined were quite small). At a finer level of granularity, these *functions* are accomplished by a series of *actions*. The actions may be dependent on certain conditions, represented by *state-high* nodes. At an even finer level of granularity, the actions themselves are implemented in the program by *operations* which, in a VPL, often correspond to a single node. Likewise, *state-low* nodes describe the state of a data object, usually just after a test.

One point worth noting is that the information types classification does not include the classification of erroneous statements: statements which are not correct within the context of the program but can still be classified according to an information type are categorised as such.

The Coding Process

The short examples shown above were designed to give a flavour of the way in which statements are coded: the full coding process in described in the coding manual (Appendix D).

A simple coding environment, using Word macros, has been designed to facilitate the coding of both the information types classification and the object classification, which will be described below.

The program summaries are represented in table cells, with one summary segment per row. The classification system is represented as a panel of buttons (see Figure 8.1), with each button representing a category, and categories grouped together as appropriate. The button icons do not bear a direct relation to the category, but they were useful in visually distinguishing between categories when coding, and so were maintained.

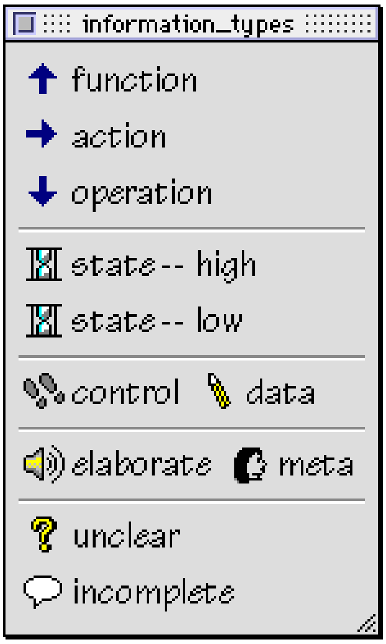
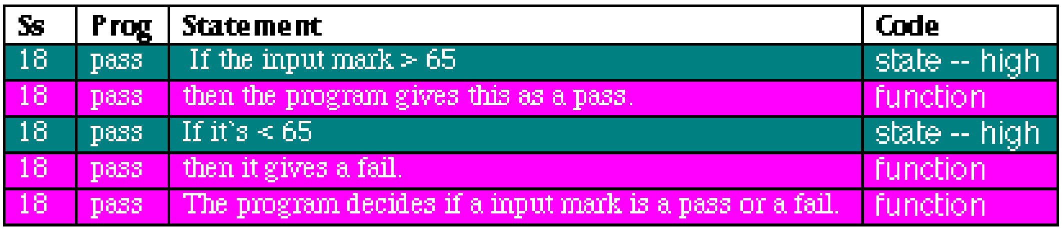


Figure 8.1: Coding Panel for 'Information Types' Classification

With the cursor on the line to be coded (or the first line if one is coding sequentially), coding consists of clicking on the button with the desired code. That code is then placed in the cell next to the program segment, the entire row is shaded in a colour corresponding to that code, and the next row is highlighted, ready for coding. Coding an already coded statement overwrites any prior code, so recoding is simple.

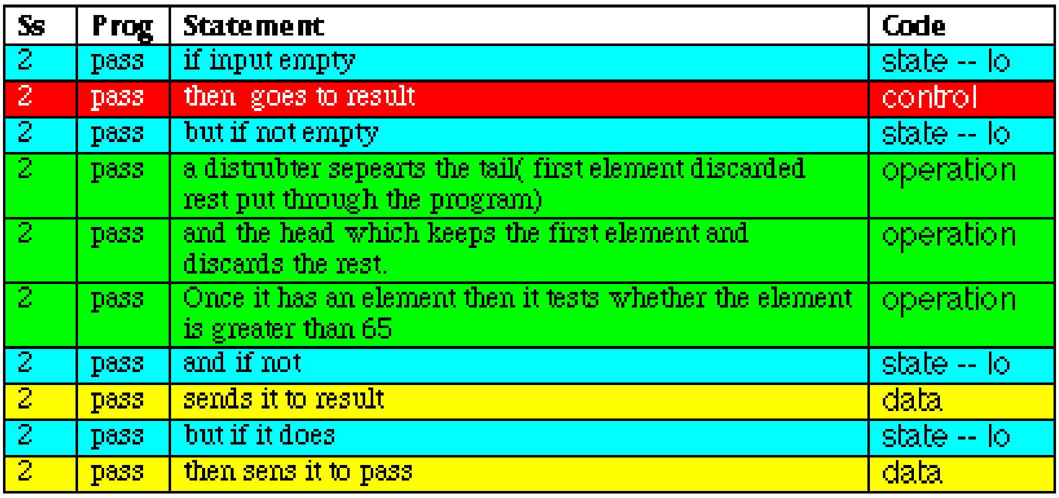
The colour shading is quite useful as a descriptive aid, and allows two summaries to be compared easily. For example, Figure 8.2 shows a relatively high level program summary, with the program being described primarily in terms of its function and high level states.



Ss	Prog	Statement	Code
18	pass	If the input mark > 65	state -- high
18	pass	then the program gives this as a pass.	function
18	pass	If it's < 65	state -- high
18	pass	then it gives a fail.	function
18	pass	The program decides if a input mark is a pass or a fail.	function

Figure 8.2: A High Level Program Summary

Figure 8.3 shows a program summary which contains more low-level state and operations statements, with some data flow statements at the end.



Ss	Prog	Statement	Code
2	pass	if input empty	state -- lo
2	pass	then goes to result	control
2	pass	but if not empty	state -- lo
2	pass	a distrubter sepearts the tail(first element discarded rest put through the program)	operation
2	pass	and the head which keeps the first element and discards the rest.	operation
2	pass	Once it has an element then it tests whether the element is greater than 65	operation
2	pass	and if not	state -- lo
2	pass	sends it to result	data
2	pass	but if it does	state -- lo
2	pass	then sens it to pass	data

Figure 8.3: A Low Level Program Summary

In addition to the information types and object description palettes, a palette was also designed to carry out basic coding and segmenting actions, such as splitting segments, inserting/deleting new lines, etc.

8.6.2 Object Descriptions

The aim of this classification is to look at the way in which objects are described. The basic question being asked is, "How do subjects, when not constrained by specific instructions, choose to describe objects present in the program?". The most interesting cases are those in which there is a choice of levels at which the object can be described. For example, an input to the program could be described as *a list of numbers*, or alternatively, as *a series of basketball player's heights.*

There are various points to note about classifying objects:

- some objects cannot be classified at more than one level. For example, a program is, by definition, a program specific object: it is hard to imagine it as a description of something existing in the real world domain (although it is in some cases possible, *e.g.* a calculator).
- similarly, objects introduced within the program (*i.e.* not inputs or outputs), and which have a *raison d'être* only within a program, cannot be classified in domain terms. An example is a counter: it is used only within the program to keep track of objects across iterations or recursive calls.

The main distinction being made is between program objects and domain objects. Pennington's program and detailed classifications have been collapsed into one "program" classification, while other finer-grained distinctions have been introduced and are described below.

Object Categories: Descriptions and Examples

The object classification comprises seven categories, which are described below with examples. Note again that the examples are taken from actual transcripts, and spelling and punctuation have not been changed.

program only: refers to items which occur only in the program domain, and which would not have a meaning in another context, for example, a counter. A useful question for distinguishing this type of object is to ask oneself whether the object would be required if one were solving the problem using paper and pencil rather than a computer.

- This program initially sets *a counter* to zero ...
- ...then *the counter* is not incremented and the sub-program is called again.

program: an object, which could be described at various levels, described in program terms. Program terms refer to the use of any program specific data structure (*e.g.* a list) or variable (either indicated by the lack of an article, or the word in quotes, capitalisation, etc.)

- ...checking first whether *the list* is empty or not ...
- If *'Hight'* is then equel to or less than 180 'Sub Team' is run again.
- If *the current height variable* is above ...

program – real-world: object descriptions using terminology which is valid in both real-world and program domains, *e.g.* results, numbers, values (the latter only if it isn't being used to describe the value of a variable). The term *real-world* is being used in contrast to *domain* as follows: domain terminology refers to the problem domain, *e.g.* basketball players' heights, exam marks, distances between cities. All of these entities could be represented in the real-world as numbers. These numbers are not specific to a problem domain (as seen here, they cut across all of the domains), but they not specific to programming either (if one were adding up exam marks by hand, one would also be manipulating numbers). Therefore, a reference to *numbers* would be classified as a program – real-world description, while *exam marks* would be classified as a domain description. Originally, a 'real-world' category was also created (similar to the domain/program – domain distinction), but it turns out to be difficult to determine whether the object is being described in a way which is completely free of the program context. therefore it was felt to be safer to have only a 'program – real world' category.

- The program takes *2 numbers* ...
- The program gives out *the 5 highest values* that were input to the program.

– It then passes *a list containing numbers* to a sub program which first extracts . . .

program – domain: object descriptions which contain a mixture of program and problem domain references, *e.g. a list of marks* (note that care must be taken to ensure that domain references are not in fact being used as variable names), or a reference which is equally valid in the program and the problem domains (*e.g.* differences)

– *The first height* becomes . . .

– This is processing *a list of marks* . . .

– . . . it then passes *a list of heights* to a sub-program.

domain: an object which is described in domain terms rather than by its representation within the program, *e.g. a mark, a distance, sunny days.*

– This program checks *a basketball players height* from [the list given].

– This program takes *a number of sunny hours* and . . .

– This program calculate *the number of studends* who passed . . .

indirect reference: an anaphoric reference to a data object. These references can be matched to the object by referring back in the program summary, thus another option would be to count them as two instances of the same category. However, this has the unwanted effect of inflating the category in question.

– . . . *they* are stripped in turn out of [the list].

– . . . if *it* is then the program returns to the main program.

– . . . and adds 1 to *it*.

unclear: statements which are ambiguous and cannot be coded, either because the statement itself is unclear, or because the object which is being referred to cannot be identified.

– . . . then *the amount of passes* is incremented by 1.

– . . . is sent ti *the pass marker* . . .

– . . . [the head goes into] *a folder*.

Although not all of the categories are related, some do have links between them. Program and domain categories could be referred to as 'pure' categories in the sense that they refer to one level of description only. Program – real-world and program-domain are amalgamates of pure categories. Program only is a special case: unlike the categories just mentioned, it is used for objects which are inherently linked to the program domain and hence cannot be described at other levels. Finally, indirect reference and unclear statements are independent categories in the sense that they are not linked to the others in any obvious way.

Coding Examples

The coding process is described in a short coding manual in Appendix D, however, examples of the coding process are shown here.

The environment used for coding is identical to the one described for the information types classification, with the exception of the panel of buttons used (shown in Figure 8.4). This panel contains buttons corresponding to the object description categories, rather than the information types.

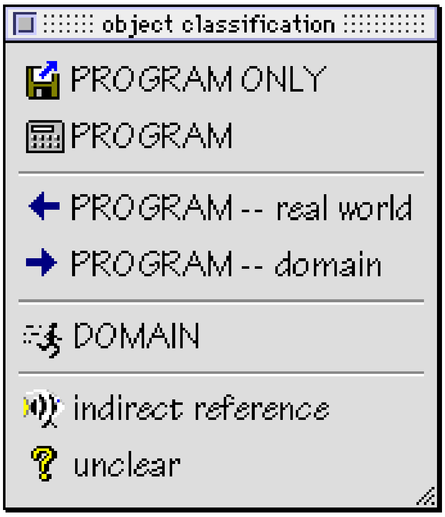
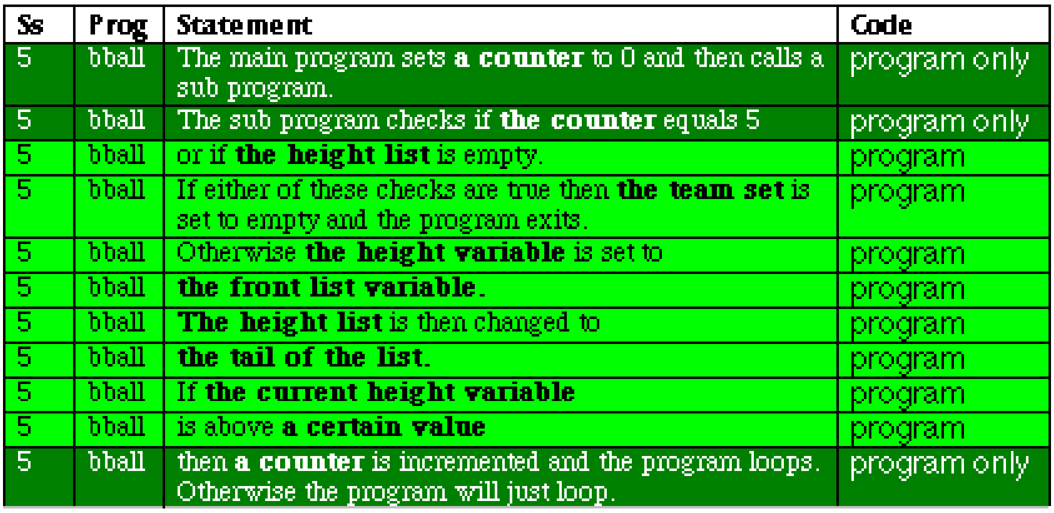


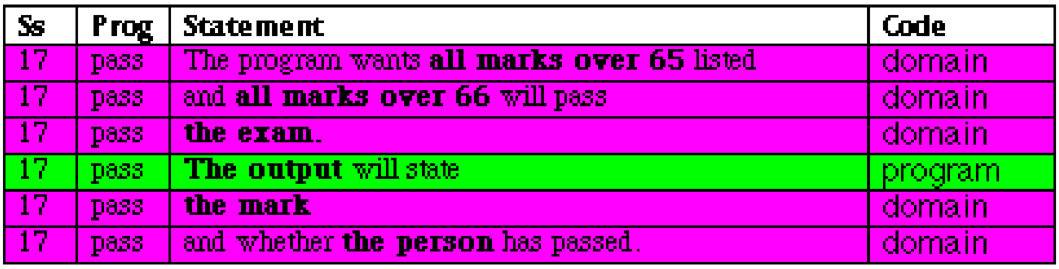
Figure 8.4: Coding Panel for 'Object Description' Classification

As mentioned in conjunction with the information types classification, colour shading is quite useful as a descriptive aid, and allows two summaries to be compared easily. For example, Figure 8.5 shows a summary which is primarily focused on program descriptions, while Figure 8.6 shows a summary which is largely domain oriented.



Ss	Prog	Statement	Code
5	bball	The main program sets **a counter** to 0 and then calls a sub program.	program only
5	bball	The sub program checks if **the counter** equals 5	program only
5	bball	or if **the height list** is empty.	program
5	bball	If either of these checks are true then **the team set** is set to empty and the program exits.	program
5	bball	Otherwise **the height variable** is set to	program
5	bball	**the front list variable**.	program
5	bball	**The height list** is then changed to	program
5	bball	**the tail of the list**.	program
5	bball	If **the current height variable**	program
5	bball	is above **a certain value**	program
5	bball	then **a counter** is incremented and the program loops. Otherwise the program will just loop.	program only

Figure 8.5: A Summary containing mainly Program Statements



Ss	Prog	Statement	Code
17	pass	The program wants **all marks over 65** listed	domain
17	pass	and **all marks over 66** will pass	domain
17	pass	**the exam**.	domain
17	pass	**The output** will state	program
17	pass	**the mark**	domain
17	pass	and whether **the person** has passed.	domain

Figure 8.6: A Summary containing mainly Domain Statements

8.7 A Critical Analysis of the Scheme

This section presents a critical analysis of both new classifications by applying the list of points described in Section 8.3. As this chapter does not cover the application of the scheme, only the scheme itself will be analysed here: the application of the scheme will be analysed in Chapter 9.

What is the theoretical background of the scheme?

The scheme is part of a study which investigates VPL program comprehension by novices, comparing the data flow and control flow paradigms. The scheme is not based directly on a theory as there is, to the author's knowledge, no theory of novice reasoning with data flow and control flow VPLs. The scheme is however based on Pennington's work on information types (Pennington, 1987b).

What are the aims of the scheme, in other words, what hypotheses does it allow one to test, either in actual fact, or potentially?

Information Types: This scheme aims to look at the ways in which novice programmers communicate their understanding of a program, by classifying their summary statements according to the types of information they highlight. The scheme should be particularly useful when carrying out cross-comparisons of language paradigms or representations.

Object Description: The scheme aims to investigate the level of abstraction at which novices choose to describe the data objects in programs which they have studied. Again, the scheme should be most informative in the context of comparative studies.

To what domain is the scheme applicable?

Program comprehension.

Does the scheme come with a coding manual which describes how to apply it?

Yes, see Appendix D.

Is the scheme provided with an example transcript?

Yes, transcripts are shown in Appendix E.

Categories:

- ***How many categories does the scheme contain?***

  Information Types: 11

  Object Description: 7

- ***Is there a hierarchical structure to the scheme (i.e. categories at different levels)?***

  Information Types: There is no hierarchical structure in the sense that one category would be a subclass of another, however, there are differences in the level of granularity between some categories.

  Object Description: No.

- ***If there are different levels, are they interdependent?***

  Information Types: Yes, on a semantic level (*e.g.* 'actions' are higher-level descriptions of 'operations', and 'state-high' is related to 'state-low' in obvious ways), but this does not have repercussions for coding, in the sense that there are no nested categories.

  Object Description: Not applicable.

- ***What is the level of granularity of the categories (e.g. segment, sentence, utterance, paragraph)?***

  Information Types: Segments, which correspond to what Bales (1951) terms 'units to be scored' in Interaction Process Analysis, an early language coding system whose methods have been well-tested. They are represented textually as single, simple sentences with a subject and predicate (either of which may be implied).

  Object Description: Segments, which are defined as a piece of the program summary which contains exactly one reference to a data object. Note that the difference in the definition of segment means that program summaries must be segmented twice: once for each classification.

- ***Are the categories orthogonal?***

  Again, it would be more appropriate to say that the two classifications (information types and level of detail) are orthogonal.

- ***Are the categories mutually exclusive?***

  Information Types: Yes, although again, this distinction becomes more clear once coding is attempted.

  Object Description: Yes (see above comment).

- ***Are coding examples provided for each category in the scheme?***

  Yes.

Does the scheme account for non-ideal behaviour?

No, the scheme does not consider errors.

8.8 Chapter Summary

This chapter discussed a reconstruction of Pennington's original classification of program summary statements according to information types and level of detail. It analysed the classifications used in the scheme in relation to a number of criteria, and went on to propose two new classifications which focus on information types and object descriptions. The classifications are based on Pennington's, but are more complete and well-defined. These schemes were also analysed using the same criteria, which allows for some comparisons to be made.

Both classifications are aimed at coding essentially the same types of information in the same domain, however, the reasons for doing so differ: Pennington *et al.* are interested in mental representations, while the focus of the studies described in this thesis is the effect of notations and representations on descriptions of programs.

As compared to Pennington's scheme, the schemes proposed in this thesis provide more concise definitions of categories, including specific types of elements which are classified under each heading, more extensive examples, completely coded transcripts, and a coding manual. Furthermore, the relationship between the schemes is more clearly spelled out.

The next chapter looks at the application of these new schemes, at the results obtained, and also looks critically at the scheme itself, allowing it to be compared with Pennington's scheme on the issue of application.

Chapter 9

Results from the Program Summary Analysis

9.1 Introduction

Program summaries provide an important source of data in the so-called 'information types' studies, a point which was discussed in detail in Chapter 8. Producing a summary allows subjects to describe a program in their own words, so summaries can supplement more narrowly focused questions in order to provide a fuller account of program understanding. Typically, program summaries have been analysed along two dimensions: the *information* contained in the summary statement and the *level of detail* (Pennington, 1987b; Corritore and Wiedenbeck, 1991). Although these categorisations do not map directly onto the five information types used in the program comprehension questions, parallels can be drawn between the information types and the program summary analysis in such a way that the results from each task can either provide confirmation of the other, or highlight differences in information type use/extraction across tasks.

A number of factors will undoubtedly influence the production of program summaries. Corritore and Wiedenbeck showed that program length may affect the use of information types in the summary, but there are surely others, such as the task, the nature of the program, the paradigm, the programming notation, level of understanding, expertise, etc., all of which seem interesting avenues to explore. These types of investigations are well suited to the methodology of comparative studies, where it is easier to control for other factors and focus on the issue of interest.

One of the major problems associated with program summary data is that the methodology used for analysing it has been ill-defined and, as a result, difficult to apply. This led to the development of a new analysis scheme, as described in Chapter 8, and this chapter provides both a critical look at the application of the scheme, and an overview of the results obtained.

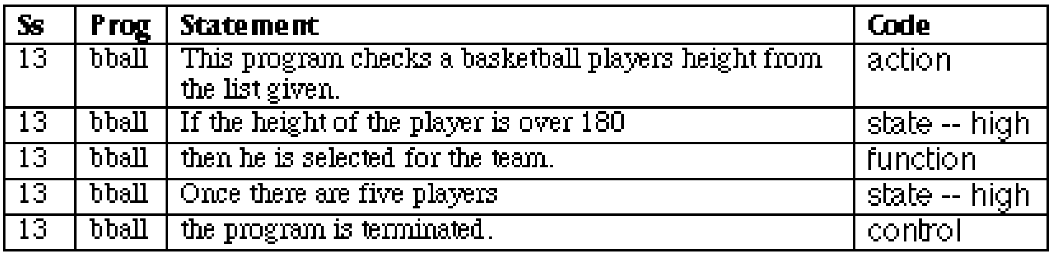
A disadvantage of proposing a new approach is that it makes it difficult to compare results with previous studies. However, it could be argued, firstly, that an underspecified methodology also makes comparisons problematic, as there is likely to be wide variation in the way in which it is applied. Secondly, comparisons between studies are fraught with difficulty in any case, as very often there will be other factors, in addition to the factors of interest, which result in unwanted differences between the studies. This is certainly the case for the studies reported here (the Prolog study and the VPL study), where differences in factors such as whether the experiment was timed or not and whether the program was visible during the summary writing process are likely to have had an effect on the summary produced. Therefore, any comparisons between the studies reported here and previous studies will be made cautiously and with the understanding that they are speculative. The only reliable comparisons which will be made and described in detail are those between the control flow and data flow languages in the VPL study.

This chapter has three main parts: the first provides a walkthrough of a coding example in order to give the reader a feel for what is involved in coding a summary. The second part looks at the application of the scheme, and analyses it according to the framework put forward in Chapter 8, Section 8.3. The final part describes the results of the program summary analysis for the Prolog experiment described in Chapter 3, then looks at the data from the VPL study in Chapter 6. The chapter concludes with a general discussion of the results and the main issues arising from the analysis.

9.2 Program Summary Analysis: An Example

This section provides a brief description of how a statement is coded, so as to provide a fuller understanding of the critical analysis of the scheme in the next section.

Figure 9.1 shows a short program summary.



Ss	Prog	Statement	Code
13	bball	This program checks a basketball players height from the list given.	action
13	bball	If the height of the player is over 180	state -- high
13	bball	then he is selected for the team.	function
13	bball	Once there are five players	state -- high
13	bball	the program is terminated.	control

Figure 9.1: An Example Program Summary

Line 1 shows a typical action statement: it describes something that happens in the program, but is not specific enough to allow it to be coded as an operation. For example, "checking a basketball player's height to see if it is greater than 180" would be an example of an operation, while this is an instance of a non-specific processing action involving the input list.

Line 2 describes the outcome of an operation, in other words, it describes the state of a particular data object whose value has just been tested. It is classified as a high-level description, while a corresponding low-level description of the same event would be, for example, "if the number $>$ 180".

Line 3 describes in essence the overall goal of the program, which is to select basketball players for a team.

Line 4 again describes the state of a data object in high level terms (a state-low equivalent would be "if the counter = 5 test is true ...").

Line 5 describes program termination. Any action such as recursion, iteration, control passing to a subprogram, or the program stopping, failing or exiting would be classified as such.

9.3 A Critical Analysis of the Scheme Application

This section applies the questions described in Section 8.3 of Chapter 8 to the *application* of the coding scheme (an analysis of the scheme itself having been carried out in Chapter 9).

Is hindsight allowed in coding or must a dialogue/discourse be coded sequentially?

Hindsight is allowed, and even encouraged. Ensuring reliability within categories seems more important than enforcing a strict sequential coding order.

How many coders were used?

One.

This sheds some doubt on the reliability of the scheme, and the issue of inter-rater reliability must be addressed in the future. However, the application of the scheme to data from different experiments provides initial evidence of its feasibility. Both analyses rely on a low-level, content based investigation of the summaries. The fact that they consider the semantics of the domain requires coders to have an in-depth understanding of the programs which the subjects studied, the different ways in which the programs were represented, the variable names used and the domains represented. Coders need to understand basic programming in general, but also the ways in which programming culture might influence subjects' descriptions. For example, students from Edinburgh University, who had been taught Prolog, consistently referred to lists as lists, while students from Napier University, who had been taught COBOL and C^{++}, referred to lists as 'sets', which, if taken in the mathematical sense, are not the same thing. However, they were treated as such for the sake of the analysis, as it was clear that it was a matter of terminology rather than misunderstanding.

A further example from the object description scheme illustrates the subtleties of coding. The following three sentences, taken from the summaries, all use the word results:

- ...the number of *results* greater than 65.
- ...one is added to *the results* which is initially set at 0.
- ...it then outputs *the results*.

In the first sentence, the subject is referring to a program which looks at exam results, therefore *results* is being used in a domain context. In the second sentence, the subject is referring to a data node in the data flow representation which is called 'results': this would be classified as a 'program' reference. In the third, the subject is referring to the overall results, or output, of the program, and this would be classified as 'program – real-world'.

In many cases, the distinction between a domain object, say, ***a height***, and its variable name within the program, ***Height***, is very difficult to make, and requires attention to very subtle indicators of use.

The need for in-depth knowledge in order to carry out coding differs from other schemes, for example, those which require coders to mark conversational 'moves' in dialogue (Carletta *et al.*, 1997), where a thorough understanding of the domain is not as crucial and/or the 'domain' in question is limited (in this case, two versions of a simple map consisting of a route via a small number of landmarks).

This requirement means that although training a coder would be entirely possible, it would also be quite time-consuming.

Were the coders trained?

Not applicable, given that the author developed the scheme and coded the segments.

However, if additional coders were to be used, they would require extensive training to apply the scheme reliably.

If more than one, was the reliability reported?

N/A.

Categories:

- ***Is the number of categories manageable? (i.e. do some categories end up being unused because there are so many of them?)***

  There are 11 information types categories and 7 object description categories. This allows for finer-grained distinctions than does Pennington's scheme, but the number of categories is not overwhelming (previous versions of the scheme used a number of subcategories in each category, resulting in over 50 categories, and were substantially pared down with the goal of manageability in mind).

- ***Is it straightforward to make a category decision? Is there any decision process provided for doing so (e.g. a decision tree?)***

Category distinctions are not straightforward (see above for examples). However, it is felt that this is due to the nature of the coding task, *e.g.* difficulties in disambiguating the use of similar or identical words, rather than with flaws in the coding scheme itself.

In addition to a description of the categories, with examples, a decision tree is provided for both classifications (see Appendix D).

- ***Does the scheme include default categories which are misused (i.e. they become "bucket" categories)?***:

  Information Types: It does not appear to be the case. The 'unclear' category accounts for 1.82% of statements in the Prolog study, and in the VPL study, 4.07% and 1.36% for the control and data flow groups respectively. All other statements were classifiable in other categories.

  Object Description: Likewise. This scheme was not applied to the Prolog data, however, the 'unclear' category accounted for 0.39% and 1.03% of statements in the VPL study for the control and data flow groups respectively.

Is the scheme accompanied by analysis tools?

Yes, it is accompanied by a coding tool, implemented in Word, which incorporates a series of coding panels which allow one to code (and recode) a scheme by pressing the button corresponding to the desired code.

If there is ambiguity in applying the scheme, is it due to the scheme or to the real world?

Information Types: There is ambiguity involved due to the fact that the different information categories are interrelated in a program.

Object Description: Again, there is some ambiguity involved, as a result of trying to determine the level at which subjects are using object descriptors: it may be that it is not clear to the subjects themselves, and that they are moving conceptually between the problem domain and the task domain.

Summary

The scheme aims to provide a more comprehensive, fine-grained methodology for analysing program summaries. This inevitably means that it has become more detailed than Pennington's original scheme. The benefit of this increased detail is a corresponding increase in clarity. However, it may paradoxically exchange old difficulties in coding for new ones: whereas before, distinctions were hard to make because the scheme did not specify them very well, now the distinctions may be so fine-grained that they are tedious to make.

However, it is difficult to see how the problem can be avoided if one wishes to carry out this type of analysis, as the aims of the scheme make coding a skilled process. It was already mentioned that coding relies on in-depth and quite precise knowledge about the programming domain and the task domain, in contrast with other schemes which allow one to code on the surface structures of the text.

Coding schemes are sometimes criticised for not being usable 'off-the-shelf' as it were, in other words, because they require substantial knowledge, practice and experience on the part of the coder. This is perhaps unjust: for example, one expects a person marking exam scripts to have an in-depth knowledge of the domain, the questions being asked, and in many cases, the marker receives specific guidelines for marking. It follows that the type of coding described here, which is dependent in the same way on a detailed understanding of the content of the subject's reply and the domains to which it relates would have the same requirements.

On the plus side, the scheme appears at first sight to be widely applicable. It was initially devised on the basis of the control flow VPL data, and then applied to the data flow VPL summaries. Finally, the scheme was applied to the data from the Prolog experiments. Thus, it has been shown that the scheme can be used with very different paradigms (control flow, data flow, declarative) and representations (graphical, textual). An obvious question is whether the scheme will scale up beyond the simple programs which have been used in these experiments: for the moment, that question cannot be answered.

9.4 Prolog Experiment

This section describes an analysis of the program summary data from the Prolog experiment described in Chapter 3. Given the points made in the introduction, it is difficult to compare the data obtained from this study with previous studies. However, the results are interesting in themselves, and furthermore, they provide support for the applicability of the scheme.

Before describing the results, it should be noted that the data from both the Prolog and the VPL studies has large amounts of variance. This is likely to be inherent in the nature of the task, as writing a program summary is very different from answering more conventional experimental questions. Firstly, subjects were not given specific instructions as to what to put in their summaries, and secondly, there is no *right* answer to the request for a program summary.

9.4.1 Word Count

The mean length of the summaries was 19.01 words. The relatively short length of the summaries may be due to the fact that subjects had a fixed time period in which to study the program, answer the comprehension questions, and write the program summary. The request for the program summary was the last item on the page, which may have limited even further the time they had to devote to the summary.

9.4.2 Information Types Classification

Program summaries were first segmented according to the definition of a segment given in Chapter 8 and then categorised.

The mean number of segments per summary was 2.38. Again, this is quite low, probably due to time constraints. The data cannot be compared with previous studies (Pennington, 1987b; Corritore and Wiedenbeck, 1991), as this information was not reported and furthermore, the segmentation may well have been carried out differently.

A Methodological Aside

Using the information types classification with the Prolog data showed up an important difference between Prolog and other types of languages such as the data and control flow languages described later in this chapter.

In many languages, actions and operations on data are represented explicitly as such, even in languages which are not traditional, procedural languages. For example, in the data flow language investigated in this thesis, an operation to take the head of the list is represented as a single node, marked "head". In the control flow language, this operation can also be traced to one node, which usually assigns a value to a variable at the same time as taking the head, *e.g.* "set Distance to head(Distances)".

However, in Prolog, taking the head of the list is not so much an explicit action as a way of representing the data. In the example below:

```
adjust_sub([X|Xs], Y, [Z|Zs]):-
            Z is X + Y,
            adjust_sub(Xs, X, Zs).
```

The expression `[X|Xs]` is effectively taking the head of the list, simply by virtue of the way the list is represented, but this is very different from an operation as defined in most languages. It is difficult to know therefore, when speaking about Prolog, if references to "taking the head of the list" should be considered as *data* or as *operations*. Certainly, this representation has the same effect as a node which explicitly splits the list, but it is more implicit, and resides in the data structure itself. In fact, the same program can be written for novices as such:

```
adjust_sub(List, Y, [Z|Zs]):-
            List = [X|Xs],
            Z is X + Y,
            adjust_sub(Xs, X, Zs).
```

in order to make this operation more explicit, and more like an operation in other languages.

Events such as these, which are directly linked to data structure representation, fall somewhere between the *data* and *operations* categories. In the same way that operations can be described at a more abstract level in terms of *actions*, Prolog gives rise to a higher level description of program events which could be classified as either *data* or *actions*: *data_actions* as it were.

Classifying these events as data could lead to the criticism that the analysis is skewed towards the data category simply because subjects have no other way of describing the actions and operations in the program. On the other hand, creating new categories such as *data_operation* and *data_action* should be avoided unless a definite need can be identified. Therefore, an initial analysis was carried out by classifying statements twice in order to determine whether this makes a difference to the overall results: firstly, including these types of statements as *data*, and secondly, considering them as either *operations* or *actions*. Table 9.1 shows the result of these classifications, giving the mean proportion of information types category statements, with standard deviations. Columns 2 and 3 show the results of classifying *data_operations* and *data_actions* as *data*, and columns 4 and 5 show this data reclassified as *operations* and *actions*.

Category	**Data Class.**		**Op/Act Reclass.**	
	Mean	**Std Dev**	**Mean**	**Std Dev**
function	29.58	22.40		
data	27.71	17.84	18.40	15.83
action	15.52	13.26	19.03	15.45
elaborate	6.30	9.09		
incomplete	5.13	9.54		
control	4.98	8.61		
meta	3.05	13.70		
state-high	2.76	5.27		
unclear	1.82	4.51		
state-low	1.65	4.17		
operation	1.50	3.36	7.31	9.77

Table 9.1: Mean Proportion of Information Types Statements

As can be seen in the column entitled, "Data Class.", functional statements predominate, followed by data flow statements. Action statements are also reasonably well-represented in program summaries. All other types of statements occur relatively infrequently (less than 7%).

Figure 9.2 represents the data oriented classification (columns 2 and 3 of Table 9.1).

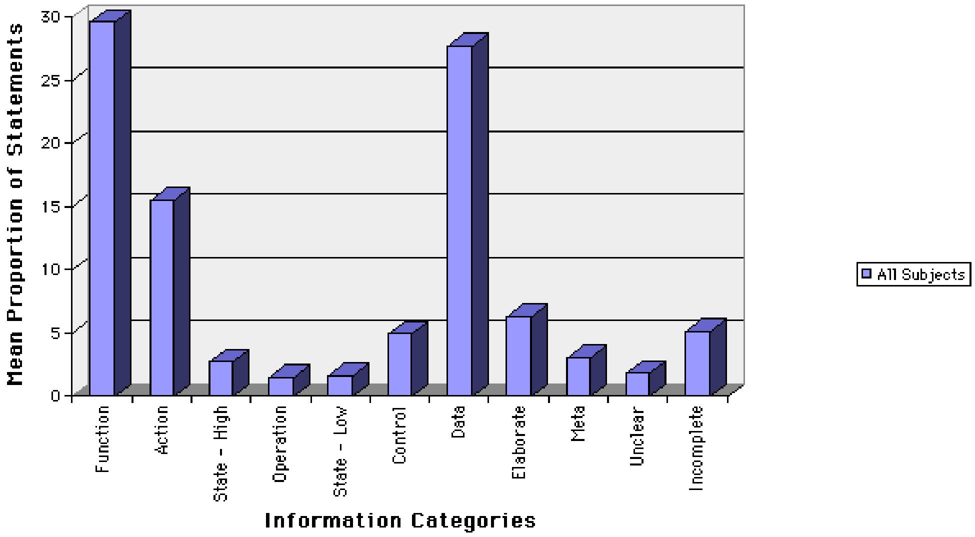


Figure 9.2: Information Statement Categories: Data Categorisation

A repeated measures ANOVA with statement type as an 11 level factor was highly significant (F=53.51, df(10,730), $p < .001$).

Figure 9.3 shows the reclassification of this data (Table 9.1, columns 4 and 5), with *data_operations* and *data_action* statements classified as *operations* and *actions* respectively. It can be seen that functional statements still predominate, but the reclassification has the effect of more or less equalising action and data flow statements. The reclassification affects the operations statements to a much greater degree than the action statements.

Again, a repeated measures ANOVA with statement type as an 11 level factor was highly significant (F=38.85, df(10,730), $p < .001$).

9.4.3 Internal Validity of Program Summaries

Correlations were carried out between the different types of summary statements in order to determine whether certain types had a tendency to co-occur. The correlations were performed for each of the classifications described in Table 9.1 to ensure that they held for both categorisations of data flow, operations and action statements (this was the case). Rather than provide an exhaustive table of all possible correlations, only those which are significant at $p < .05$ or higher will be described in the paragraph

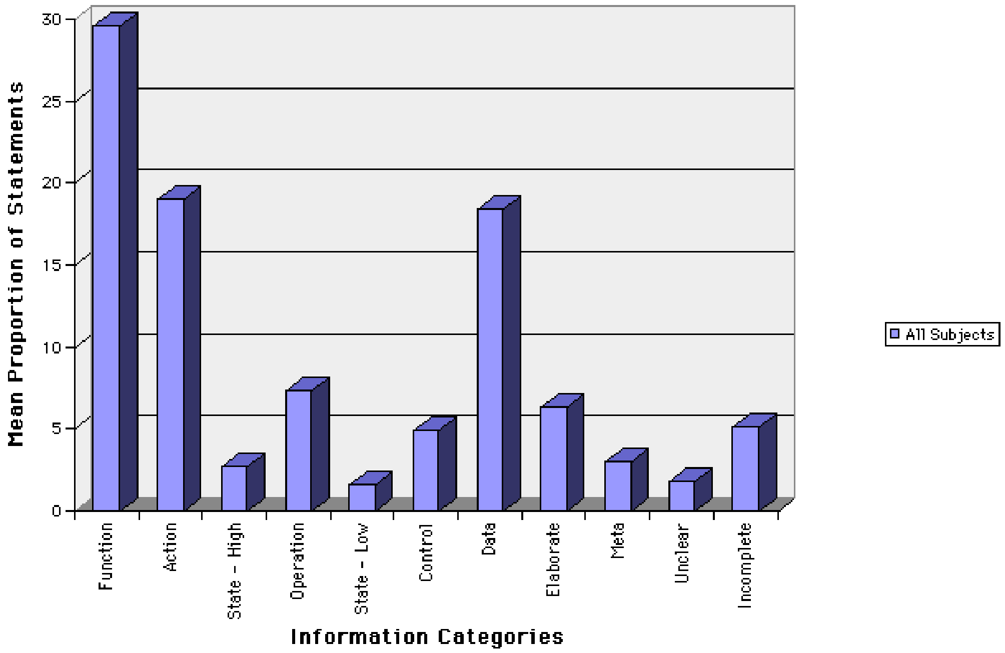


Figure 9.3: Information Statement Categories: Op/Act Categorisation

below.

Functional statements correlated negatively with all statement types apart from data flow statements. Data flow statements correlated negatively with action and control flow statements. Both state-high and state-low statements correlated positively with operations and control flow statements. Finally, control flow and operations statements correlative positively.

9.4.4 Relationship between Program Summaries and Comprehension Questions

In order to investigate possible relationships between the two tasks used in the experiment, *i.e.* the binary choice questions and the program summaries, correlations were performed between statement types and question scores. This was done in order to investigate whether, for example, high scores on a particular information type question correlated with the frequency of that type of statement in the program summaries. Three significant results were observed: a positive correlation between operations questions and function statements (r_s = .29, p < .02), between control flow questions and operations statements (r_s = .27, p < .03), and a highly significant positive correlation between function questions and elaboration statements (r_s = .45, p < .001).

The correlation between summary length and overall score was checked, and was significant (r_s = .24, p < .04). It had previously been hypothesised that short summaries might be associated both with low overall scores (as subjects have difficulty in formulating their understanding of the program), and with high scores (as they understand the program well enough to produce a less verbose, high-level account rather than a blow-by-blow description), thus producing a bell shaped curve. This was not borne out however: instead, it seems that high scores are associated with longer summaries, and vice versa.

On a related note, the relationship between total score (on the binary choice questions) and the different types of statements present in the program summaries (function, data, operations, action, control, state-high and state-low) was investigated. The only significant correlation was between total score and the percentage of 'elaboration' statements, which are statements which provide further information about something already present in the summary, or concrete examples of the program's behaviour (r_s = .24, p < .04).

9.4.5 Object Description Classification

The object description classification was not carried out on the program summaries from the Prolog experiment as non-meaningful program and variable names were used in order to avoid giving clues to subjects about the program's functionality. However, this meant that subjects did not have the option of describing objects in domain terms, and were essentially restricted to program only, program, and program – real-world terms. As the main question of interest with respect to object descriptions was whether subjects choose to use domain or program terms, it was decided not to classify the summaries using this scheme.

9.4.6 Discussion and Implications

The most striking result from the Prolog data is the high proportion of what might be termed 'high-level' statements, particularly functional summary statements (almost

30%). Functional statements are followed by data flow and action statements in terms of frequency (in different proportions depending on the way in which *data_actions* are classified). On the other hand, statements relating to control flow, low-level operations and state occur relatively infrequently.

These results differ from those of similar experiments in two respects. Firstly, this study appears to be the only information types study to find that functional statements were the dominant type of statement, and conversely, that procedural (*i.e.* control flow and operations) statements were quite rare.

Secondly, unlike the other studies, the results of the program summary analysis do not map neatly onto the results from the binary choice questions used in the experiment. The lowest rate of errors on the binary choice questions occurred for control flow, operations and state questions, with functional and data flow questions having the highest rate of errors (30.6% and 43.2% respectively). This is in contrast both to previous research and to the VPL study reported in Section 9.5, where the multiple choice question and program summary data complement each other quite well.

These results are quite intriguing, and the next sections consider why they might have occurred. When the results from the comprehension questions were considered in Chapter 3, the discussion focused on an examination of some of the features of the Prolog language that may have contributed to the results observed. Likewise, the following sections hypothesise that the patterns of information present in the program summaries may revolve more around the features of the Prolog language than the underlying declarative paradigm. Before discussing this, it is worth noting that it is often difficult to distinguish between what necessarily follows from a particular paradigm in terms of the semantic and representational features of the language, and what is optional. Furthermore, it is sometimes difficult to separate out the semantic and syntactic features of the language, and distinguish between them, as the discussion will show.

The following sections consider firstly hypotheses relating to the frequency of occurrence of various information types in Prolog program summaries, and finish with some thoughts on what makes for a 'good' program summary.

Action Statements: Familiarity?

As reasons for the occurrence of each of the three dominant statement types (functional, action and data) will be considered, this section looks at action statements. This is more for the sake of completeness, so as to cover all three information types, rather than any theoretical implications, therefore, this section will be brief.

Distinguishing between *operations* or *actions* often revolves around the scope of the event, in other words, whether it is described as being applied to one data object (*e.g.* "it takes the first element of the list, adds it the counter, and then recurses on the tail of the list") or to all objects (*e.g.* "it adds the value of each successive element to the counter").

The high proportion of action statements suggests that subjects had a good general understanding of recursion, and did not feel it was necessary to discuss how the program acted on each element of the list, understanding that until the base case is reached, the action will be the same for all list elements. Given that recursion is Prolog's principal control structure, in contrast to other languages, and therefore that subjects would have already had much exposure to recursive programs, this is not surprising. Data from the VPL study, in which subjects had only a theoretical understanding of recursion, showed much more evidence of descriptions of the program behaviour at an operations level, suggesting that they were 'feeling their way through' the recursive construct for each successive element in the data structure.

Dataflow Statements: Explicitness of Data?

One explanation for the high frequency of data flow statements has to do with the explicitness of Prolog's data representation: data is represented very clearly in each predicate as arguments, which can either be input or output arguments (or both, depending on the circumstances in which the predicate is called). Because data objects are more visible components of programs than they may be in other languages, data flow through a program can be traced relatively easily. This focus on data objects seems to be reflected in descriptions of Prolog programs, with many of the statements classified as 'data' statements describing the program's inputs and outputs.

Dataflow Statements: is Prolog really a Data *Structure* Language?

Although Prolog is considered to be a declarative language, and some have suggested that Prolog is in many respects a data flow language, this study suggests that Prolog might more accurately be described as a data *structure* language. Section 9.4.2 pointed out that what might normally be considered actions and operations in other languages are, in Prolog, embedded in the way in which particular data structures are represented. The following example was used earlier in the chapter to show that taking the first element from a list is an operation which is embedded in the data representation `[X|Xs]`.

```
adjust_sub([X|Xs], Y, [Z|Zs]):-
             Z is X + Y,
             adjust_sub(Xs, X, Zs).
```

Using the same example, it can be seen that the operation of adding an element to a list is expressed as `[Z|Zs]`, in other words, in exactly the same way as the operation to take the head of a list (`[X|Xs]`). The way in which these data structure representations function within the program will depend on their instantiation, and on other events which occur both before and after the execution of that particular line of code. This example serves to show that the distinctions in other programming languages between what might be considered data on the one hand, and actions or operations on the other hand, are blurred because of a shared representation in Prolog.

It seems quite likely that this feature of Prolog leads to the *data_action* and *data_operations* descriptions which were observed. Certainly, Bergantz and Hassell (1991), who used an adapted information types methodology to examine verbal protocols of professional Prolog programmers, felt the need to introduce a new information type: that of *data structure*, in order to account for subjects' initial understanding of Prolog.

Functional Statements: Time Limitations or Diffuseness?

Time Limitations One practical explanation for the preponderance of functional statements is the limited time which subjects had to reply, as opposed to the VPL study, where subjects could take as much time as they needed to formulate their summaries.

It seems plausible that given time constraints, subjects would opt to provide a concise, high-level view of the program, as they simply would not have time to give a low-level sequential account of the program's execution. On a related note, this may explain why functional accounts tend to be associated with expertise: given that experts are often asked to examine programs of moderate length or longer, say 200 lines or more, it seems unlikely that they would be able to give a blow-by-blow account of the program, and may therefore resort to shorter, functional accounts.

On the other hand, an explanation based on time constraints would not explain the results obtained by Pennington (1987b) and Corritore and Wiedenbeck (1991): both imposed time limits on their subjects, and neither observed high levels of functional statements. It is more likely that the explanation lies elsewhere.

Diffuseness When discussing the fact that there was no difference, visually, between taking the head of a list and adding an item to a list, it was noted that a correct interpretation depended on actions which may have preceded or followed the line of code in question. This suggests that Prolog is in some way "diffuse". Unfortunately, the term "diffuseness" is not used here according to the definition given in Green and Petre (1996), where it is used in conjunction with "terseness" to describe the number of symbols or graphical entities which a notation requires to achieve a particular aim. According to Green and Petre's definition, Prolog would be considered to be quite terse: Prolog programs tend on average to be shorter than equivalent programs written in procedural languages.

The term 'diffuse' is used here to describe the fact that events in Prolog which might be seen as functionally related, or related in terms of execution, are often dispersed in the program text. Events may appear in the program which cannot actually take place until an event further in the program causes the first event's data structure to become instantiated.

An example will hopefully make this clearer: one might describe the program shown above by saying that a list is split into head and tail, `Z` is calculated and added to a list, and `adjust_sub` is called again with the tail of the input list, a common description. The code ordering in this description would be: beginning of line 1, line 2 (at which point `Z` becomes instantiated), end of line 1, line 3. It becomes apparent that Prolog does not lend itself to straightforward sequential descriptions of a program, proceeding line by line. Thus, it may well be that comprehending a program which is diffuse in this manner requires more cognitive effort than understanding one in which the actions are represented in a logical fashion in the program text. Rather than a line by line reading of the program, a program synthesis stage may be required, which might then lead to a higher level, abstract description of the program, *i.e.* a functional description. It is interesting to speculate as to whether spelling out each variable instantiation explicitly, as was shown for the input argument of the `adjust_sub` predicate in Section 9.4.2, would, because it allows a more sequential reading of the program, result in *fewer* functional descriptions. In any case, the issue of diffuseness is discussed in relation to the data flow VPL later in this chapter, where the same phenomenon occurred.

Task Factors

One question which has not yet been answered is why the results from the program summary data do not correlate well with those from the binary choice questions. Various tentative hypotheses might be put forward to explain this: Chapter 3 described possible reasons why the results from the information types questions were quite similar to those obtained with procedural languages, despite Prolog's obvious differences. One hypothesis was that the questions might not be tapping into the information types they were designed to uncover, for reasons having to do both with the design of the questions, the interrelatedness of various information types, and the particular programs chosen. This factor might also explain the lack of correlation with the information types statements in the program summaries.

However, it is also felt, as a general rule, that binary choice questions may be neither sensitive nor realistic indicators of program comprehension: there is a 50% chance, with each one, of guessing the correct answer. Certainly, Pennington recognised their potential lack of ecological validity, stating that, "The ability of programmers to answer our comprehension questions is a limited indicator of success at more goal directed programming tasks." (Pennington, 1987a, p. 112). Although multiple choice questions were used in later experiments, the quest for realistic tasks could be taken further. For example, one could investigate whether Prolog provides access to function and data flow information in more realistic situations which require that information, such as debugging or maintenance.

This brings up the issue of what any given task might *require* in terms of information. With respect to the program summaries, it may be that the subject feels that the type of information asked about in the comprehension questions does not have any relevance in a summary of the program. In other words, it is one thing to be able to answer questions about program operations with relative ease, but it does not necessarily follow that the same person will feel it is important to include operations information in his/her program summary. The issue of how the subject's perception of the task of summary writing, including its goal, purpose, intended audience, etc., might influence his/her performance is considered in detail in (Good and Brna, 1998a).

What should be in a Summary?

The discussion above leads nicely to questions about what a summary *should* contain. The evidence from the Prolog experiment seems to suggest that the presence of any one information type in program summaries is not associated with an increased level of understanding. Rather, the length of the summary seems to be important, as is the tendency to provide examples or restatements of events which have already been described. In fact, there is a significant correlation between the length of the summary and percentage of elaboration statements (r_s = .39, p < .001).

Data such as this does not provide support for a two-stage model of comprehension, whereby programs are first understood in terms of low-level operations before a functionally oriented view is developed in the second stage. If this were the case, one would expect program summaries with high levels of data flow and functional statements to correlate well with performance on comprehension. Instead, it seems that subjects who score highly on the comprehension questions provide explanations which contain both a generic description of the program (at whatever level of abstraction they choose) and a concrete example showing how the program transforms its inputs into outputs ("elaboration" statement).

9.5 Visual Programming Experiment

This section presents an analysis of the program summary data obtained in the experiment described in Chapter 6. As the experiment compared the performance of two groups, the control flow group and the data flow group, the analysis described here will allow us to look at how the paradigm might influence the way subjects communicate their understanding of the program.

9.5.1 Word Count

Before being normalised for the between groups comparison, a word count was taken of each summary. The mean length of the control flow summaries was 70.91 words, as compared to 48.85 words for the data flow summaries. Because of the high degree of variance (control flow standard deviation: 42.46, data flow standard deviation: 24.43), the difference was not significant (t-test(unrelated), t= 1.42, ns).

9.5.2 Information Types Classification

As explained in Chapter 8, the information types classification consisted of 11 categories (although only 10 were used here, as the 'incomplete' category was not necessary for the data from this experiment). Program summaries were first segmented and then categorised.

The mean number of segments per summary was 9.5 for the control flow group and 7 for the data flow group, a difference which was not significant (t-test(unrelated), t= 1.14, ns). Given the number of words and number of segments for each group, this means that segments of the data flow group were slightly shorter in length than those of the control flow group (7.46 words per segment for the control flow group and 6.99 for the data flow group).

Table 9.2 shows the mean proportion of information types category statements, with standard deviations, for the control and data flow groups.

Summaries from the data flow group contain higher proportions of function, action, state-high, and data flow information types than do the control flow group. The control flow group's summaries contain higher proportions of operation, state-low, and control

Category	Control Flow		Data Flow	
	Mean	Std Dev	Mean	Std Dev
function	11.62	19.02	20.93	31.73
action	7.10	8.64	9.10	7.39
operation	30.22	17.37	15.67	12.68
state-high	6.22	8.07	8.23	7.36
state-low	12.93	10.81	10.04	7.53
data	13.10	13.45	24.68	12.97
control	14.10	9.68	5.33	4.22
elaborate	.49	.79	3.61	5.17
meta	.15	.49	1.05	2.57
unclear	4.07	7.18	1.36	2.26

Table 9.2: Mean Proportion of Information Types Statements per Group

flow statements. In terms of other types of statements, data flow subjects tend to use more elaboration statements (often in the form of examples), and more meta-statements, while control flow subjects made more statements which were judged to be unclear.

Figure 9.4 shows this information graphically.

A mixed design ANOVA for repeated measures with groups as a 2 level between-subjects factor and statement type as a 10 level, repeated measures, factor, showed a significant effect for statement type (F=7.19, df(9,162), p < .001), and an almost significant group by statement interaction (F=1.94, df(9,162), p = .05). Post-hoc pairwise comparisons were made using the Bonferonni adjustment. Four comparisons were made, therefore, only probabilities of less than .0125 were considered to be significant.

Table 9.3 summarises the results of the unrelated t-tests: only the control flow comparison was significant.

Statement Type	Result		
function	t= -.80	p= .219	ns
operations	t= 2.14	p= .023	ns
data	t= -1.96	p= .033	ns
control	t= 2.62	p= .011	sig

Table 9.3: Pairwise Comparisons of Statement Types across Data Flow and Control Flow Groups: Results of unrelated t-tests

One issue of interest when investigating the occurrence of information types in program summaries is the *level of abstraction* of the information type (where, for example,

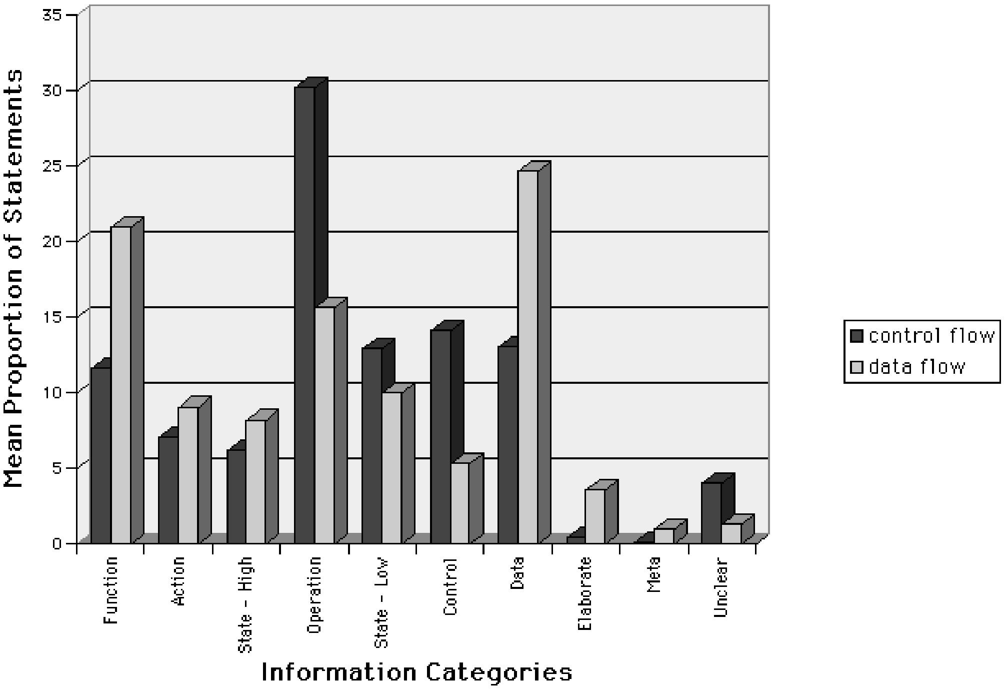


Figure 9.4: Information Statement Categories per Group

Condition	Info-High	Info-Low
Control Flow		
Proportion	30.94	57.24
Std Dev	29.51	36.35
Data Flow		
Proportion	53.83	31.04
Std Dev	21.62	21.13

Table 9.4: Proportion of High and Low Information Statements per Group

function and data flow are considered to be higher level abstractions than operations and control flow).

In order to examine differences in abstraction levels across groups, two composite measures were devised: *info-high*, made up of function, data and state-high statements, and *info-low*, made up of operations, control and state-low statements.[1] The mean proportion of info-high and info-low statements per group is shown in Table 9.4.

This information is shown graphically in Figure 9.5.

A mixed design ANOVA with groups as a 2 level between-subjects factor and statement

[1] 'Action' statements were not included as they would appear to fall somewhere between the two.

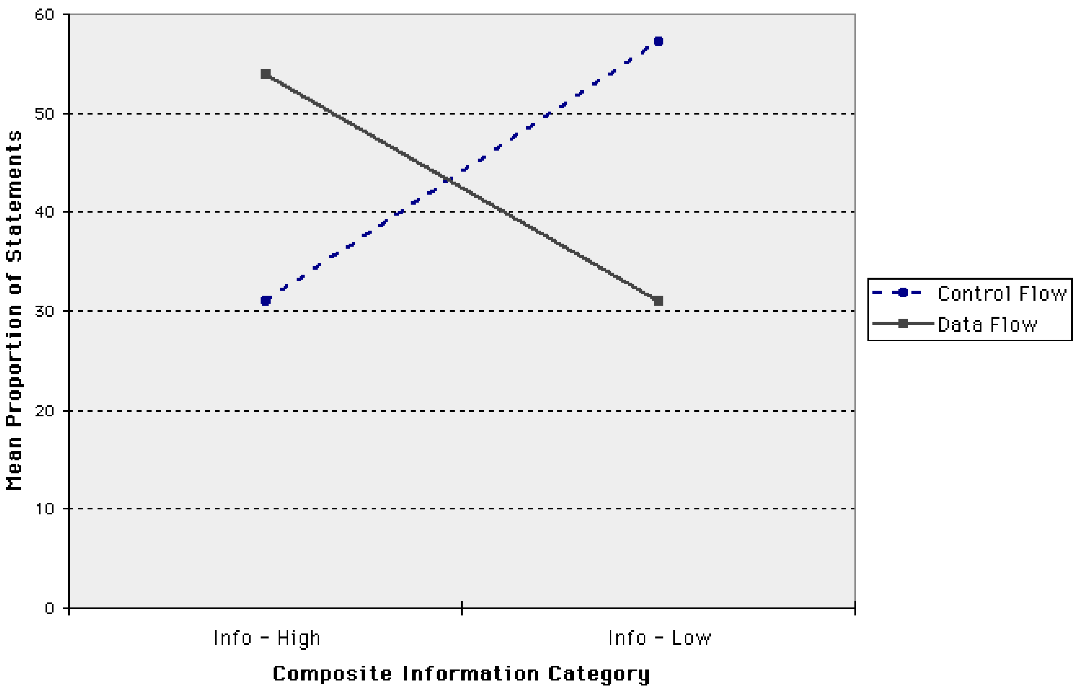


Figure 9.5: 'High' and 'Low' Information Statements per Group

type (info-high, info-low) as a 2 level, repeated measures factor, showed an interaction for group and statement type which approached significance (F=4.05, df(1,18), p = .06), but no main effect for group (F=.42, df(1,18), p = .53), or for statement type (F=.02, df(1,18), p = .89).

9.5.3 Internal Validity of Program Summaries

One question which arose was the extent to which summaries were internally consistent, in other words, whether the usage of particular types of summary statements correlated with others. The results of correlating information types statements is shown in Table 9.5. It can be seen that so-called high-level statements (function, data) do not correlate positively with each other. Low-level statements (state-low, operations, control) correlate positively with each other, and negatively with high-level statements.

9.5.4 Object Description Classification

In order to classify summaries according to the way in which program objects were described, the summaries were resegmented in such a way that one data object occurred

	Func	**Data**	**Control**	**Op**	**Sta-Low**	**Action**
Sta-High	r_s = -.45 p < .05	r_s = .34 ns	r_s = .15 ns	r_s = -.04 ns	r_s = -.04 ns	r_s = .55 p < .02
Func	-	r_s = .10 ns	r_s = -.82 p < .001	r_s = -.55 p < .02	r_s = -.61 p < .005	r_s = .01 ns
Data	-	-	r_s = -.52 p < .02	r_s = -.67 p < .001	r_s = -.44 p < .05	r_s = .72 p < .001
Ctrl	-	-	-	r_s = .85 p < .001	r_s = .80 p < .001	r_s = -.32 ns
Op	-	-	-	-	r_s = .70 p < .001	r_s = -.47 p < .04
Sta-Low	-	-	-	-	-	r_s = -.37 ns

Table 9.5: Correlations between Statement Types

in each segment.

The mean number of segments per group was 11.6 for the control flow group and 8.4 for the data flow group. This difference was not significant (t-test, t= 1.29, ns).

Table 9.6 shows the mean proportion of object description category statements, with standard deviations, for the control and data flow groups.

Category	**Control Flow**		**Data Flow**	
	Mean	**Std Dev**	**Mean**	**Std Dev**
program only	4.07	3.75	3.81	3.62
program	46.93	28.15	33.84	22.15
program – real-world	11.21	8.52	18.55	7.28
program – domain	4.52	3.75	5.29	7.77
domain	22.78	30.74	20.46	15.08
indirect	10.09	7.09	17.02	8.56
unclear	.39	.84	1.03	3.24

Table 9.6: Mean Proportion of Object Description Statements per Group

Summaries from the data flow group contain higher proportions of program – real-world, program – domain and indirect statements than do the control flow group. The control flow group's summaries contain higher proportions of program only, program and domain statements. Finally, data flow subjects made more references to objects which were judged to be unclear than did control flow subjects. Figure 9.6 shows this information graphically.

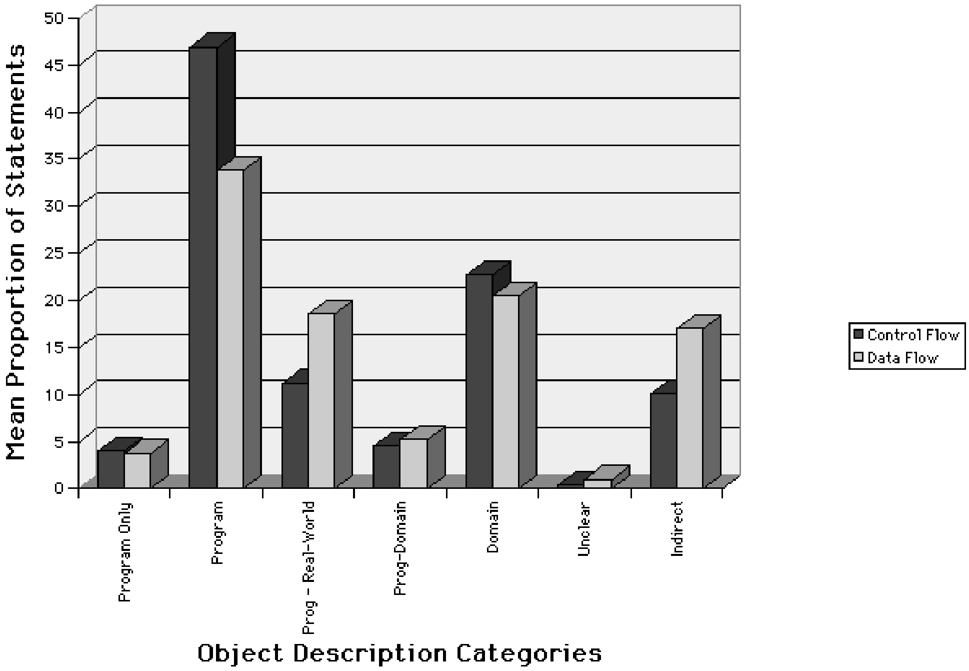


Figure 9.6: Object Description Statements per Group

	Object-High	**Object-Low**
CONTROL FLOW		
Proportion	27.30	51.00
Std Dev	29.92	30.47
DATA FLOW		
Proportion	25.75	37.65
Std Dev	21.17	22.89

Table 9.7: Proportion of High and Low Object Description Statements per Group

A mixed design ANOVA with groups as a 2 level between-subjects factor and statement type as a 7 level, repeated measures factor, showed a significant effect for statement type (F=15.83, df(6,108), $p < .001$), but no group by statement interaction.

Finally, two composite measures were devised: *object-high*, made up of domain and program – domain statements, and *object-low*, made up of program and program only statements.[2] The mean proportion of statements per group is shown in Table 9.7, and graphically in Figure 9.7.

A mixed design ANOVA with groups as a 2 level between-subjects factor and statement

[2] Again, program – real-world statements were not considered, as it is not clear where exactly they fall on the high – low continuum.

type (object-high, object-low) as a 2 level, repeated measures factor, showed a significant main effect for group (F=11.94, df(1,18), p < .005), but no effect for statement type (F=2.34, df(1,18), p = .143) or group by statement interaction (F=.26, df(1,18), p = .618).

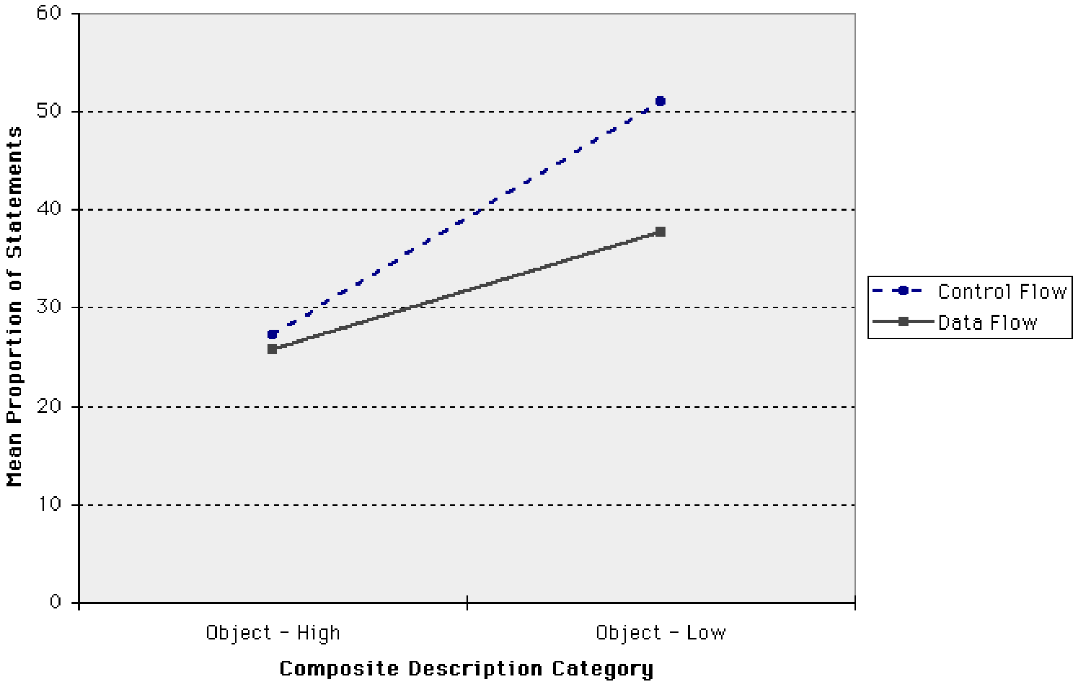


Figure 9.7: ‘High’ and ‘Low’ Object Description Statements per Group

9.5.5 Relationship between Information Types and Object Descriptions

When looking at the relationship between information type and level of detail, Pennington (1987b) stated that there was a tendency for functional descriptions to be described in domain terms, while procedural statements were expressed in terms of program objects. This suggests that descriptions at high levels of abstraction will use more domain based terminology, while lower-level descriptions will use program specific terminology. In order to test this, the level of information type (high or low) and the object description level (again, high or low) were correlated. The results, shown in Table 9.8, clearly support Pennington’s hypothesis overall and for the control flow group: there is a strong positive correlation between information types statements and object description statements of the same level of abstraction, and a strong negative correlation when the levels of the two types of statements do not match.

However, this trend is not found in the data flow group: apart from a significant positive correlation between high level information types statements and high level object descriptions, no significant correlations were observed.

GROUP	CATEGORY	Info-Low	Info-High
Overall	Object-Low	.71 p < .001	-.76 p < .001
	Object-High	-.64 p < .002	.68 p < .001
Control Flow	Object-Low	.77 p < .009	-.81 p < .005
	Object-High	-.75 p < .02	.72 p < .02
Data Flow	Object-Low	.54 ns	-.61 ns
	Object-High	-.50 ns	.67 p < .04

Table 9.8: Mean Proportion of Object Description Statements per Group

9.5.6 Relationship between Program Summaries and Comprehension Questions

In addition to exploring the relationship between statements within the summary, described in the previous section, correlations were investigated between statement types and multiple choice question scores. This was done in order to determine whether high scores on a particular information type question correlated with the frequency of that type of statement in the program summaries. There were, by and large, very few significant results, either overall or per group.

The correlation between summary length and overall score was checked, and was not significant (neither overall nor per group). It was hypothesised, as with the Prolog study, that short summaries might correlate positively with low multiple choice question scores, as evidence that subjects had problems formulating their understanding of a program, and with high scores, showing that they were producing a higher level, less verbose account of the program. A scatterplot showed that this was not the case.

The relationship between total score (on the multiple choice questions) and the level of description was investigated, checking correlations between total score and high and low level object and information descriptions. Again, total score did not correlate with any of the measures, neither overall nor per group, apart from a positive correlation, for the data flow group, between low-level information types statements and overall score (r_s = .66, p < .04).

These descriptions (high-object, low-object, high-info, low-info) were then correlated with scores on each of the five information types multiple choice questions (function, data, operations, control, state). Overall, scores on operations questions correlated negatively with both info-high and object-high summary statements (for both: r_s = -.45, p < .05). Operations scores also correlated positively with object-low statements (r_s = .56, p < .01).

No significant correlations were observed for the control flow group, however, the data flow group had a positive correlation between control flow scores and low-info statements (r_s = .71, p < .02) and a negative correlation between functional statements and low-info statements (r_s = -.65, p < .05).

Finally, scores on each of the five information types multiple choice questions (function, data, operations, control, state) were correlated with the different types of statements present in the program summaries (function, data, operations, action, control, state-high and state-lo). Again, very few clear-cut results emerged from these correlations.

9.5.7 Summary of Results

The main results of the program summary analysis for the VPL study are as follows:

- Program summaries from the data flow group were shorter than those of the control flow group by approximately 30%.
- The information types contained in the program summaries varied between groups: statements in the data flow summaries were more likely to be high-level statements (function, data flow, state-high), while those in the control flow summaries were low-level (operations, control, state-low).
- The object descriptions showed a different trend: control flow subjects tended to describe objects in programming terms, while data flow subjects, rather than

using more domain based terms, seemed to make more references to objects in non-domain, real-world terms, or to refer to them indirectly.

- There was a strong correlation between level of information and level of description overall, with high-level descriptions of objects correlating positively with the use of high-level information types, and vice versa.
- From a methodological point of view, it was shown that the categorisations of information types and object descriptions developed in Chapter 8 could be applied to the program summary data, and yielded results which were consistent with the data from the multiple choice comprehension questions also used in the experiment.

9.5.8 Discussion and Implications

The following sections examine some of the issues arising from the analysis in more detail, considering the effect of paradigm on comprehension, along with the issue of diffuseness and how it influences information type use and program comprehension.

Paradigm and Comprehension

The results of the program summary analysis provide support in many ways for the quantitative results discussed in Chapter 7. In terms of the information types present in the program, subjects in the data flow group used a higher number of functional and data flow statements, while descriptions of state were also written at a more abstract level. Control flow subjects, on the other hand, provided more detailed program summaries, with many more descriptions of low-level operations, state, and more mentions of control flow. Overall, data flow subjects produced shorter, more higher level, abstract accounts of the program, which placed more emphasis on the data flow relationships in the program. Control flow representations seemed to promote a focus on the lower-level workings of the program, on the operations which the program carries out, and on the control structures embodied in the program. Why this might be so is considered in a later section.

In any case, these results suggest that programming paradigm does influence one's understanding of a program. What is particularly interesting about this result is that it occurred with visual, rather than textual, programming languages. One could imagine an experiment using two textual languages containing very specific keywords: it might be expected that, for each group, a higher occurrence of paradigm specific keywords would be found in subjects' program summaries. In contrast, the VPLs used here contained very few keywords of this type (the 'set' operation springs to mind as being the most distinguishing feature between the control and data flow paradigms). Instead, events were conveyed through the combination of text, icons and spatial layout, and this was reflected in the program summaries, in the sense that subjects had to devise their own textual descriptions of the program. The most salient example of this was with respect to the flow of data through the program: it seemed to be relatively clear to subjects in the control flow condition that arcs represented flow of control, while data flow subjects envisaged their arcs as "sending" data to various places.

Only one subject in the control flow condition mentioned the data flow across programs, stating that:

> it [the program] then passes a list of heights to a sub-program.

Most of the control flow summaries described the movement of the locus of control through the program, but without accompanying data:

> – . . . and the program goes back to the start of the subprogram.
> – The program passes sub is called.
> – The program part 'Sub Pass' is then run.
> – 'Sub Pass' is ended as is the whole program.

In contrast, the data flow summaries convey the notion of 'active' data, and in almost all cases, talk of 'sending' data to various points in the program:

> – . . . the heights over the height are sent ot the team.
> – . . . a signal is sent to a counter.
> – . . . while the tail is sent to the distributor.
> – . . . and then sends the tail to another copy of itself.

The next section looks at the relationship between the information types and object descriptions.

Information and Detail: How are they Related?

Subjects' descriptions of objects did not show the same clear-cut trends as did their use of information types. Control flow subjects did tend to describe objects more frequently in program based terms, but data flow subjects did not, in contrast, use more domain descriptions than the control flow group. They did use more program – real-world descriptions and slightly more program – domain descriptions. One interesting difference was the use of 'indirect' references (10% in the control flow group as opposed to 17% in the data flow group). At first sight, the difference does not seem to have much credence, but it may well be related to one of the purported advantages of data flow programming for novices, namely the lack of intermediate variable names. Control flow subjects have a new object name to use each time a 'set' operation occurs, while data flow subjects only have available the initial input and output, which may lead to more indirect object references, and possibly more non-domain, real-world references to things such as 'numbers' or 'answers'.

When comparing results across conditions, there are few marked differences in object descriptions, suggesting that data flow may lead to a more abstract, functional account overall, but it does not lead to more abstract, domain based descriptions of objects.

This provides an interesting twist to Pennington's claim that functional descriptions are couched in domain terms, and procedural descriptions in program terms. It appears that the two are related for the control flow subjects: there are very strong correlations between the level of abstraction of information type statement and the object description level, in that high level information types statements correlate positively with high level object descriptions, and vice versa. However, the situation was not observed for the data flow group: only high level object descriptions correlated positively with high level information types statements, with the remaining correlations being non-significant. It is interesting that Pennington first postulated the relationship between level of information and level of description in the context of a procedural language, and that the results from this study showed support for this idea only within the context of the control flow language.

Mixed Program Summaries

In Pennington's investigation of the behaviour of expert programmers (Pennington, 1987b), she divided program summaries into three types: *program level summaries*, containing mainly operational and program level statements, *cross-referenced summaries*, containing a more even distribution over operations, program and domain levels, and *domain summaries*, containing a majority of domain and vague statements. She maintained that best performance was associated with cross-referenced summaries. Notwithstanding possible problems with Pennington's classification, discussed in Chapter 2, the results from the VPL study do not seem to provide support for Pennington's claim, in the sense that cross-referenced summaries containing functional or data flow statements do not occur.

When examining the types of statements which make up a program summary, it was seen that high-level statements, *i.e.* data flow, functional and state-high statements, do not correlate positively amongst themselves or with low-level statements. On the other hand, low-level statements such as control, operations and state-low correlate positively with each other, and negatively with high-level statements, suggesting that other subjects write summaries containing a mixture of low-level statement types, but few high-level statements.

Looking at this at the level of individual information types, statements correlated negatively with other types of statements. This suggests that subjects who use functional statements tend to write a summary containing essentially functional statements, a finding which is consistent with the Prolog study discussed earlier in the chapter, and also with the findings of Bergantz and Hassell (1991).

Diffuseness Revisited

Given the evidence from the VPL study, it could be concluded that control flow representations highlight control flow, and data flow representations highlight data flow. Furthermore, since control flow and program actions and operations are related, they are also highlighted, and given that data flow and function are related, function is also highlighted. In other words, each paradigm highlights a certain type (or even a 'group of types') of information over the other, and that the consequences of this will be evident both in tasks requiring the information in question (comprehension questions), and in situations which require one to communicate about the program (the program summary). Certainly, the results observed appear to be consistent with what is known about different programming paradigms and their purported strengths and weaknesses, however, it is worthwhile looking in more depth at why the results and trends may have occurred.

This section does just that, following the lead of the investigation of Prolog's representational properties, described earlier in this chapter. When considering Prolog, it was hypothesised that some of the program summary results, namely the high level descriptions of the programs, might be due to the representational properties of the Prolog language, which may not necessarily be a logical consequence of the paradigm itself.

Apart from the obvious differences between the data and control flow VPLs, namely that the arcs between nodes represent the flow of data in the former case, and the flow of control in the latter, the data flow representations can be distinguished from the control flow representations by their 'diffuseness'.

When discussing diffuseness with respect to Prolog, it was noted that the term was not being used in the same way as in (Green and Petre, 1996), where it is considered to be one of a series of *cognitive dimensions*, and denotes the number of symbols or graphical objects used in a particular language. Prolog's diffuseness referred to its non-linearity, and a lack of grouping of events which might logically be related. This section will argue that one of the differences between the control flow and the data flow VPL used in the study reported here is that the data flow language is doubly diffuse, both in the sense used in (Green and Petre, 1996), and in the Prolog sense.

Firstly, when investigating low-level operations in the control flow and data flow VPLs, it becomes apparent that operations which can be accomplished in one node in the control flow representation may require two or more nodes in the data flow representation.

This can be illustrated by referring back to Figures 5.8 and 5.9 in Chapter 5. The first major event to occur in the `passes sub` program is a check to ensure that the list has some elements in it. If so, the list is split into head (first element) and tail (the remaining list). In the control flow version, these events require three nodes, while in the data flow version, they require four. This may not seem like a crucial difference, but when considering the arcs between nodes, the control flow version requires two arcs, while the data flow requires no less than six. If Green and Petre are including arcs in their definition of 'graphical entities', then it is clear that the languages are very different in terms of their level of diffuseness/terseness.

Diffuseness in the Prolog sense, *i.e.* the lack of grouping of functionally linked events, or a lack of execution sequencing, is also evident in the data flow language. Again, looking at the programs in Figures 5.8 and 5.9, it can be seen that if a particular mark is above 65, a counter is incremented. In the control flow program, the test box (`Mark > 65?`) is immediately followed by an operation which increments the counter (`set Pass to (Pass + 1)`). In the data flow version, a `true` token gets sent to the selector, which has the effect of allowing the `pass` token to pass through the `+1` node, thus incrementing it by one. There are examples of events which are even more spatially and temporally disjointed, *e.g.* tracing the exact effect of the '`= [ ]?`' test throughout the program, where understanding the event requires one to locate nodes in disparate parts of the representation, and to trace backwards in time to events which have occurred but whose outputs were waiting to be used by the most recently activated node. Reconstructing data flow program actions does not involve top-bottom, left-right scanning of the control flow version (a typical diagram searching sequence for speakers of languages which read from left to right Winn (1993)), and is in many ways an exacerbated example of the scanning involved in reconstructing the actions of the Prolog program examined earlier in this chapter.

This spatial diffuseness in data flow representations may be due in part to the parallel nature of data flow programs. For example, a node may produce two outputs, or tokens, each of which travels along a distinct arc to two different nodes, which are spatially distant from each other. The tokens then remain suspended on those arcs until the arrival of the other inputs required by the node in question. Once a node has the necessary inputs, it will fire and produce data outputs. From a cognitive point of view, this means that the user needs to maintain a mental list of those arcs with tokens waiting on them, and update this list each time an event occurs which causes any flow of data tokens to other nodes.

The two types of diffuseness present in the data flow representation (both the number of symbols and the spatial diffuseness), show up in differences in data flow program summaries, when compared to control flow summaries for the same program. Control flow, which requires only one arc to be followed from node to node at any one time, in a straightforward manner with little backtracking, seems to contribute to a "...and then ...and then ...and then ..." style of program summary. In contrast, activities which happen in parallel in a data flow representation, perhaps requiring a combination of nodes, which may be spatially distant, tend to be grouped together into a higher-level description in a typical data flow summary. A frequent occurrence of this phenomenon is when a list is split into head and tail. The sequential nature of control flow seems to encourage accounts which follow the program through its execution, for example:

> ...the height variable is set to the front list variable. The height list is then changed to the tail of the list.

whereas the data flow account is much more succinct:

> ...the set is split into its head and tail aspects.

or even:

> The entered set are split up.

One conjecture that should be entertained in light of the above discussion, and which also concurs with the Prolog results, is that the diffuseness of a representation might provide a different type of 'useful awkwardness' than the one described by Petre *et al.* (1998) with respect to multiple representations. The process of bringing together information which is spatially diffuse, and which requires backtracking during a simulation of the execution process in order to understand the program (which might explain the longer time taken by the data flow subjects to inspect programs), has the possible benefit of requiring subjects to "chunk" together nodes which accomplish the same goal, and to describe them at a higher level of abstraction. Obviously, there are likely to be limits on the amount of diffuseness which is beneficial: one can easily appreciate that a proliferation of symbols in a representation which makes no use of secondary notation to group together functionally related elements will rapidly become unmanageable, to say the least.

However, diffuseness does seem like an interesting issue to explore further, as a paradigm can be represented in various ways, and there is no reason to believe that diffuseness is inherent to data flow paradigms. It seems in any case, on the basis of the evidence presented in this chapter, that it spans textual and graphical languages. One way of corroborating what is currently only speculation about diffuseness would be to control for that aspect, by creating data and control flow representations which are equally diffuse, to see if this affects the nature of the program summaries.

The relationship between the production of high-level program summaries and task behaviour on more realistic tasks would also be worth investigating. Although there is no reason to believe that a high-level summary had an effect on the comprehension questions task (as evidenced by a lack of correlation between the two), it was argued that these questions were perhaps not realistic measures of comprehension. It would be more interesting to look at how these types of summaries might correlate with behaviour on, for example, debugging tasks.

Trends and Significance: Individual Differences?

Trends in the program summary analysis were quite clear-cut, showing differences between the control and data flow groups in terms of the way in which they communicate their knowledge of a program which were in line with the initial hypotheses. However, few of the results were statistically significant, due to a large amount of variance.

This suggests that there may be individual differences playing a role. Certainly, a perusal of the results suggests that individuals often have a "personal summary style", with particular linguistic conventions used frequently. For example, some subjects, regardless of group, always begin their summary with a description of the program inputs. Others describe the program in terms of what it creates or outputs. One subject often started summaries by stating the overall goal of the program, and then describing how the program accomplished its goal, leading to summaries with a pattern of "This program does X. It does this by < a series of actions >".

However, the issue of individual differences may also be linked with the tasks the subjects are required to do. Although the overall results of the VPL study showed differences between groups which were consistent across tasks, when the results are looked at on an individual level, results from the two tasks do not correlate very well.

This suggests that individuals may be responding to task demands in different ways, a point which was touched upon when discussing the Prolog results. Again, in the absence of explicit instructions, subjects will likely form their own ideas about what the program summary should comprise, and there will be variations between individuals, based on factors such as what the summary might be used for, who the subject thinks the summary is being produced for, etc.

Although the issue of individual differences has not been considered in depth in this thesis, the results suggest that there is scope for doing so, particularly within the context of task demands, and the way in which the subject responds to them.

9.5.9 Chapter Summary

This chapter explored the application of the summary analysis schemes put forward in Chapter 8, and described the results of the analysis. The analysis scheme seems to show promise, in that it was able to capture differential patterns of statement use between groups and experiments. It showed quite clearly that, depending on the programming language used in the experiment, there are differences in program summaries in terms of the percentage of statements of each type. The Prolog language seems to lead to a high level of function, data flow and action statements. The data flow VPL was associated with high level statements (function, data, action, state-high), while the control flow language was associated with low level statement types (operations, control flow, state-low). This suggests that the properties of the language influence, at the very least, the way in which one communicates one's knowledge of the program. Various properties of the languages were explored which might account for these differences, *e.g.* the language's diffuseness, its representation of operations, actions and data (and their interrelatedness), and its sequentiality.

The following chapter concludes the thesis: in doing so, it considers the implications of the results of this and other chapters for the provision of novice comprehension support.

Chapter 10

Conclusions and Further Work

This final chapter summarises the primary contributions of the thesis, and relates them back to the main thesis questions put forward in Chapter 1 to examine if, how, and to what extent the questions have been addressed. It then considers briefly some work which is ongoing, and concludes with suggestions for future work.

10.1 Contributions and Findings

The contributions of this thesis can be summarised as follows:

- a critical overview of the main theories and research in program comprehension, and an analysis of how they relate to novice program comprehension support and teaching;
- a consideration of how the concept of information types can be used to investigate the influence of programming paradigm on comprehension, leading to a study using Prolog;
- a review of the characteristics of the control flow and data flow paradigms, including a description of their historical development, and a summary of empirical work;
- the development and implementation of two visual micro-languages, based on the data flow and control flow paradigms;
- an investigation of the control flow and data flow visual programming languages using the match-mismatch conjecture;

- the development of a methodology which combines the match-mismatch conjecture with the information types methodology, thereby allowing for the precision of the match-mismatch conjecture and the wider coverage of information types. This methodology was applied to a new version of the control flow and data flow visual programming languages in a third study;
- an extension of the match-mismatch conjecture to include *groups* of match (whereby tasks requiring information associated with the information highlighted by a notation are also facilitated);
- the development of a methodology for coding descriptions of programs along two dimensions, including a fully worked out coding scheme, a coding manual/description, and a computer environment for semi-automated coding;

Findings can be summarised as follows:

- an information types study using Prolog suggested that for tasks which simply require binary choice comprehension questions to be answered, results mirror those obtained when procedural languages are used: namely a predominance of procedural and operational information. However, more open-ended tasks show the opposite, with data flow and functional information figuring prominently in subjects' accounts of the program;
- results from the match-mismatch study using data flow and control flow visual programming languages showed a "representational supremacy" effect in that one representation was associated with better performance across tasks, regardless of whether the representation and task were purported to match. A discussion of these results suggested that the match between representation and task may be only one factor in determining comprehension performance, and considered issues such as representational familiarity and prior experience.
- a final study using data flow and control flow visual programming languages with a combined information types/match-mismatch methodology showed results which supported both the "control flow supremacy" hypothesis and the match-mismatch hypothesis: for time taken to complete comprehension questions, control flow representations proved superior to data flow representations. A match-mismatch

trend occurred in the accuracy data, although all interactions are not significant. In addition, a grouped match-mismatch effect was also observed, with improved performance for tasks requiring information thought to be related to the information highlighted by the representation: in the case of data flow, this was functional information, while in the case of control flow, this was operations and state. Finally, data from the program summaries supported these trends, with control flow subjects highlighting control flow and low-level operations in their summaries, and data flow subjects focusing on data flow and functional relationships.

10.2 Thesis Questions Revisited

This section relates the contributions and findings described above to the original thesis questions.

- *How do particular languages interact with the extraction of information types, particularly with respect to novices? Previous studies, which showed a predominance of low-level control flow information in the initial stages of comprehension, used control flow based languages. It is an open question whether this effect holds for different types of language, e.g. declarative, event driven.*

  Language paradigm does seem to interact with the extraction of information types by novices: a study using Prolog suggested that when comprehension is measured via question answering, results mirror those obtained when procedural languages are used: namely a predominance of procedural and operational information. However, when subjects are allowed to express their understanding in their own words, a trend in the opposite direction is observed, with data flow and functional information figuring prominently in their accounts of the program.

  A study on visual programming languages, based on either a data or control flow paradigm showed an even stronger effect: with the control flow language, performance on operations and control flow questions was best, a trend that was reflected in subjects' accounts of the program. With the data flow language, performance on data flow and functional questions was best, supported by a high frequency of these types of statements in free-form summaries of the programs.

- *Some language paradigms could be said to mirror information types, e.g. control-flow languages and control-flow information. What is the relationship between information types and languages whose underlying paradigm mirrors a particular information type? Will there be an influence on the types of information extracted from the program?*

  Results from an initial experiment using data flow and control flow visual programming languages were not clear-cut: the control flow language was associated with improved performance regardless of the task, suggesting that other factors might play a role in comprehension.

  In a later experiment, again using data flow and control flow visual languages, the relationship between information types and paradigm was more obvious: the data flow visual language seemed to highlight data flow in a way that made it accessible for answering data flow questions, and allowed it to feature prominently in summaries of the program; likewise, the control flow language highlighted control flow, with evidence found both in the data from the comprehension questions and from the program accounts.

- *How does the task interact with the information highlighted in the representation?*

  This question was addressed above. Briefly, task does interact with information highlighted by the program, but other factors, such as previous experience, also play a role. In some cases, these factors obscure the task-representation interaction, as in the experiment reported in Chapter 6. However, the effect was shown more clearly in a later experiment (Chapter 7).

- *From a methodological point of view, how can information extraction be measured most effectively, and in a way which is ecologically valid: what techniques should be used to gather and analyse the data?*

  This issue has been addressed in several chapters. There is an obvious trade-off between tightly controlled experimental conditions and more realistic settings. Binary choice questions are easy to devise and score, but their ecological validity was questioned in Chapter 3. Multiple choice questions offer a way of obtaining more precise data on possible misconceptions, but it is very difficult to develop good multiple choice questions, and again, question answering may not be an appropriate way of measuring program comprehension. Open-ended requests for information, used in obtaining program summary data, allow for rich data, but both the development of an analysis scheme and the analysis itself are very time-consuming. An even more ecologically valid method, not used in this thesis, would be to embed the comprehension activity within a realistic task. This does mean that the results obtained cannot be generalised to other situations, but on the other hand, comprehension rarely takes place outside of a task-oriented situation.

- *Can errors in comprehension be cast uniquely in terms of information types? Specifically in the case of visual programming languages, does the notation introduce difficulties on a syntactic level which cannot be accounted for by a semantic description of the language in terms of information types?*

  Work on error data was not considered in this thesis, for reasons of space. However, ongoing work, described briefly in the next section, suggests that while many errors can be described in terms of information types, it is not possible to characterise all errors in these terms. This is because it is hypothesised that information types describe the semantic level of a program, rather than the syntactic level. Errors such as navigational errors occur on a syntactic level (even if the two levels are necessarily interrelated), and these two types of error, syntactic and semantic, have different implications for support.

- *What might support for comprehension based on information types look like, and on what type of language could it be built?*

  Chapter 4 argued that a visual representation was the best underlying platform for information types support, and the next section presents some initial steps towards building a system.

10.3 Novice Program Comprehension Support: Ongoing Work

The last two thesis questions focused on error data and on ideas for the provision of novice support. Based on data collected from the experiment described in Chapter 6 and from an experiment similar to the one described in Chapter 7, but not reported in this thesis, work has begun on a system to provide novice comprehension support. The design of the system is presented fully in (Good and Brna, 1998b).

The main idea behind the system is to overlay comprehension support onto the visual programming language itself, in this case, the data flow visual VPL in Chapter 7 (although it is felt that the idea behind the support could equally be adapted to other languages). This allows the novice to continue working with the representation with which he/she is at least partially familiar, rather than having to move to an additional, unknown representation. Support is offered in the form of features, with each feature represented as a button on a panel: clicking on a button toggles its feature either on or off. When the feature is on, clicking on the relevant part of the representation will apply the particular feature. Examples of features will be described below,[1] however, the next section describes the analysis which motivated the support.

Before looking at how to provide support, it is useful to look at the features of the language for which support is being proposed. Error data from the experiments was analysed and classified, and the language was also examined in terms of Green's cognitive dimensions (1989). This allowed us to look at how errors relate to information types, and also at possible problems in terms of the representational features of the language. The results of this analysis suggested some permanent changes to the representation, while others suggested non-permanent changes, in other words, temporary support. These non-permanent changes can be further divided into changes which are either generic or task dependent. These are described below:

- *permanent changes* to the representation. These indicate "pathological" features of the representation for which it is difficult to see a benefit in any comprehension situation.

  For example, in the data flow language, all arcs denote flow of data. However, the flow of control in the program is effected by the flow of boolean tokens from test boxes to action boxes. The sole purpose of these tokens is to either activate or inhibit the action node, after which they are absorbed. In this sense, they are different from most other types of data which are input to the program, transformed, and output. However, all of the data is represented in the same way. Distinguishing between the two may make control and data flow more visually explicit, and there are a number of other examples where it would make sense to change the base representation rather than to provide support.

[1] A full listing of features can be found in (Good and Brna, 1998b).

- *non-permanent, non-task variant changes* to the representation. A non-permanent change indicates support which can be added as needed, and does not vary on a task by task basis. These *generic features* are designed to provide support for navigation through the representation, and to allow the student to off-load onto the representation some of the information which must normally be kept in memory while trying to make sense of the representation.

  For example, students often lose track of the paths they have traced and the point at which they have arrived. This is particularly important in the case of a data flow diagram, where each data object is represented by its own arc: it is often useful to trace one object to a particular point, and then break off to trace another data object up to the same point in order to capture some of the sequential nature of the program. With a *path tracing feature* on, tracing the path with the mouse button down would cause it to change colour up to the point at which the user releases the mouse button.

- *non-permanent changes* to the representation which are *task dependent.* The implication of the match-mismatch conjecture for program comprehension support is that various types of information contained in the program will be relevant in particular contexts, and so could usefully be made salient in those situations. In the case of *task based features*, each feature is designed to highlight a particular information type. The features aim to alleviate the errors and misunderstandings encountered by students as they attempt to retrieve information from the representation.

  For example, students may be at a point in the diagram and skip to another, incorrect, point, based on an incorrect evaluation of a choice point. The *evaluation feature* (one of a number of *control flow features*) would allow students to selectively display one output branch of a test box at a time, based on the hypothesised outcome of the test.

Features in the proposed support environment fall into either the second or third category, in other words, changes which can be made to the representation on an as-needed basis, either at any stage of the program comprehension process or in relation to a specific task.

Work on the support environment is very much ongoing, but will be subject to empirical testing once it has been completed.

10.4 Suggestions for Future Work

Many issues were touched upon in this thesis which could usefully be followed up. This section considers some of the more interesting ones, at least from the author's point of view, and makes suggestions for further work. These are grouped roughly under the headings of languages, programmers, and methodology.

10.4.1 Languages

Language Design and Structure

One aspect of visual programming languages which would be interesting to explore further is the design of the languages themselves. The languages investigated in this thesis were purposely kept simple, and very close to pre-existing models, but it is obvious that there is great scope for improving their design. Although the previous section touched upon an analysis of programming languages with a view to changes in design, this work could be taken further, based both on actual patterns of use, and theoretical analyses of the languages.

One way of going about this would be to continue an analysis of the languages in terms of Green's cognitive dimensions (1989). Green and his colleagues have carried out extensive work on this framework, and the idea seems to have been taken up by a number of researchers. It would be interesting to try and relate selected cognitive dimensions to the idea of information types. This idea was described briefly in Chapter 3, where the relationship between role expressiveness and information types was speculated upon. Would it be possible to design a language which was role expressive in a way that highlighted information types? More importantly, would it have any effect on comprehension?

Language Use

One strand of work missing from this thesis is a fine-grained examination of process data. This point was touched in Chapter 6 with respect to possible differences in strategy employed by control and data flow subjects, and the role of the representation in promoting one strategy over another. It would be very useful to obtain verbal protocols of subjects while they are trying to make sense of a visual programming language, and to analyse these protocols in terms of information types and errors of navigation.

Designing compelling graphical representations of programming languages is undoubtedly very enjoyable, yet it sometimes appears as if little thought is given to the extent to which the representations chosen will "make sense" to the users, and be usable by them. Exceptions to this trend are Bell *et al.* (1991) and Modugno *et al.* (1996), but more detailed work of this kind would be very beneficial.

One could even go so far as to attempt to build a cognitive model of the processes involved in navigating through a visual programming language. The model could concentrate on the procedural steps which occur when searching the representations and performing the necessary inference upon the data obtained from them. The characterisation would need to include such points as: 1) the exact information needed to answer a comprehension question 2) how that information is derived 3) how much of that information is present in the diagram in a "read-off" form and how much needs to be inferred 4) how many steps are needed in the inference process and what these steps consist of 5) the information needed by each step of the inference process, and 6) where that information is derived from (*i.e.* start at step 2 for each piece of information).

The main risk in deciding to build a cognitive model is that it can capture only a relatively limited subset of the phenomena under investigation. Furthermore, Green *et al.* (1991) make the point with respect to using cognitive modelling techniques to study visual programming languages that doing so may shift the focus away from the object of interest, in this case, from the underlying logical structure of the language to the representational format.

10.4.2 Programmers

Conceptualisation of Programming

The question, "What is programming?" was touched upon at various points in the thesis. A novel reply is provided by Weinberg (1971), who feels that, "'Programming'

– like ‘loving’ – is a single word that encompasses an infinitude of activities” (p. 121), while Green (1990) concurs, in a comparatively cautious manner, that “‘Programming’ is an exceedingly diverse activity” (p. 22). However, the question here is not about finding a normative definition for the concept of programming, but in exploring people’s conceptions of what programming is, and what it involves. The reason for doing so is to investigate whether conceptions of programming influence one’s approach to learning programming, or to learning new paradigms. The point was made several times that people, particularly novice programmers, see programs as active entities, rather than as a structure through which data flows, for example. It is an open question whether this has any effect on their ability to pick up programming concepts couched in non-procedural terms.

Graphical Skill

Another subject of interest, which unfortunately could not be explored further, concerns the nature of “graphical skill”. There is not really a satisfactory definition of what it is, and what it implies. The question of whether standardised tests such as pathfinding and paperfolding measure skills which are useful for visual programming is an open one: looking at the VPL comprehension performance of persons with extreme scores on these tests may suggest whether this work is worth taking further.

On the other hand, perhaps these tests simply do not measure the skill which we hypothesise to be of interest, *i.e.* the ability to reason on the basis of graphical representations. In a study on Hyperproof, a multimodal teaching system for first order logic, Cox *et al.* (1994) and Stenning *et al.* (1995) found significant interactions between measures of reasoning aptitude and teaching method (graphical or sentential). However, a distinguishing feature of one of the measures of aptitude, *determinate* problems, was that its premisses determined a unique (or nearly unique) model from which a number of conclusions could be drawn. The existence of a unique model made graphical representations suitable for solving these problems. To the extent that visual programming languages, with their inherent abstraction mechanisms, are very unlike graphical representations of this type, it is an open question whether a relationship between these aptitude measures and visual programming skill would be found.

It would also be worthwhile exploring the difference between graphical familiarity, the importance of which was highlighted by Petre and Green (1993), and "graphical skill": it may be that the latter can be reached through graphical familiarity (obtained in turn, for example, via through the explicit teaching of graphics), will lead to measurable graphical skill. As mentioned in Chapter 6, it would be interesting to take the ER taxonomy task, which was heavily logic based, and adapt it for use in the context of graphical representations in programming, asking subjects to group and label representations, in order to investigate whether, and how, this correlates with performance.

Perhaps the relationship between graphical skill and graphical familiarity could be probed by other measures, for example, by correlating measures on graphical pre-tests which could be said to measure graphical skill, and self-rating measures such as the taxonomy task, which tap into graphical familiarity. In any case, much more work needs to be done to untangle the skills involved in successfully reasoning with graphical representations: the types of skills are likely to be numerous, and come into play in different situations, depending both on the nature of the representation and the nature of the task.

10.4.3 Methodology

Finally, one issue which arose in different guises was the most appropriate way of obtaining sensitive and yet realistic measures of program comprehension, a familiar problem in experimental settings. An increase in ecological validity is often accompanied by a lack of control over factors which are not of direct interest. The shortcomings of measuring comprehension by having subjects answer questions about a program was already addressed, as was the difficulty of interpreting data obtained from more open-ended measurement techniques.

A search for new methods of probing comprehension should continue: the elicitation technique used by Holt *et al.* (1987) is of interest, as it allows subjects to describe their understanding in a way which is more open-ended than direct questioning, but which imposes more structure on the data than a request for a program summary. What is particularly positive here is that this structure is subject guided rather than experimenter guided. Although Holt *et al.* analysed the structure of the representations, this could be combined with an analysis focusing more on the content of the representations, for example, information types. It would be interesting to see the results.

10.5 Conclusions

This thesis looked at the issue of program comprehension by novices, investigating ways in which it might usefully be portrayed and studied, and focusing on the role which paradigm and representation play in shaping novice understanding. The idea of information types was looked at in depth, both as a methodological tool for experimentation and as the basis for support for novice comprehension.

The work carried out in this thesis has a number of implications:

- Although the studies did not aim to investigate the development of mental models of program comprehension, the results from the visual programming language study are relevant to the extent that they do not support a two-stage theory of program comprehension, whereby functional knowledge is built on the basis of procedural knowledge. Instead, it appears that some languages allow novices to derive functional knowledge without the procedural underpinnings, resulting in pseudo-expert comprehension;

- Simple question answering tasks may not be tapping into the issues of greatest interest and importance in program comprehension: their use seems to depend implicitly on the idea of a static mental representation which is accessed according to need. Therefore, answering an operations questions with ease, for example, implies that the mental representation is structured in terms of operations. Questions with best response latency/accuracy may simply require fewer cognitive operations to reconstruct an answer based on the available knowledge.

- Issues of importance in determining program comprehension likely cut across the textual/graphical distinction: diffuseness is an example of a programming language feature which is hypothesised to lead to a more high level, abstract understanding of the program, regardless of whether it is embodied in a textual or graphical language;

- Certain claims about visual programming languages require qualification, for example, the frequent assertion that VPLs makes data flow more explicit. This research showed that what is highlighted by a VPL depends, at least in part, on what it was designed to highlight, and on its underlying properties, rather than on its "visualness" per se.

To conclude, this research has focused on topics of interest in the domains of program comprehension and visual programming languages. In doing so, it has considered methodological issues relating to the way in which novice program understanding might best be studied, and gone on to describe how the results of the research might be used to provide useful support for novice comprehension. As such, it has established a preliminary basis for future work in this area.

Bibliography

Ackerman, W. B. (1982). Data flow languages. *IEEE Software*, pages 15–25.

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, **9**(4), 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.

Agerwala, T. and Arvind (1982). Data flow systems. *IEEE Computer*, pages 10–13.

Anastasi, A. (1988). *Psychological Testing*. Macmillan Publishing Company, New York, sixth edition edition.

Anderson, J. R., Pirolli, P., and Farrell, R. (1988). Learning to program recursive functions. In M. T. H. Chi, R. Glaser, and M. J. Farr, editors, *The Nature of Expertise*, pages 153–183. Lawrence Erlbaum Associates.

Anjaneyulu, K. S. R. and Anderson, J. R. (1992). The advantages of data flow diagrams for beginning programming. In C. Frasson, G. Gauthier, and G. I. McCalla, editors, *Intelligent tutoring systems: Second International Conference, ITS '92*, pages 585–592.

Bakeman, R. and Gottman, J. M. (1997). *Observing Interaction: An Introduction to Sequential Analysis, Second Edition*. Cambridge University Press, Cambridge.

Bales, R. F. (1951). *Interaction Process Analysis: A method for the study of small groups*. Addison-Wesley Press.

Baron, J., Szymanski, B., Lock, E., and Prywes, N. (1985). An argument for non-procedural languages. In R. Hernigan, B. W. Hamill, and D. M. Weintraub, editors, *The Role of Language in Problem Solving I*, pages 127–145, Amsterdam. Elsevier Science Publishers.

Bell, B., Rieman, J., and Lewis, C. (1991). Usability testing of a graphical programming system: Things we missed in a programming walkthrough. In *Proceedings of CHI-91*, pages 7–12, New Orleans, LA.

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, editors, *Lines of Thinking: Reflections on the Psychology of Thought, Volume 2*, pages 59–71. John Wiley & Sons.

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

Bhuiyan, S., Greer, J., and McCalla, G. I. (1992). Learning recursion through the use of a mental model-based programming environment. In C. Frasson, G. Gauthier, and G. I. McCalla, editors, *Intelligent tutoring systems: Second International Conference, ITS '92,*, pages pp. 50–57.

Bhuiyan, S. H., Greer, J. E., and McCalla, G. I. (1991). Characterizing, rationalizing and reifying mental models of recursion. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, pages 120–125. Lawrence Erlbaum Associates.

Blackwell, A. F. and Engelhardt, Y. (1998). A taxonomy of diagram taxonomies. In *Proceedings of Thinking with Diagrams 98: Is there a science of diagrams*, pages 60–70.

Boehm-Davis, D. A. (1988). Software comprehension. In M. Helander, editor, *Handbook of Human-Computer Interaction*, chapter 5, pages 107–121. Elsevier.

Bonar, J. and Cunningham, R. (1988). BRIDGE: An intelligent tutor for thinking about programming. In J. Self, editor, *Artificial Intelligence and Human Learning*, chapter 24, pages 391–409. Chapman and Hall.

Brooke, J. B. and Duncan, K. D. (1980). An experimental study of flowcharts as an aid to identification of procedural faults. *Ergonomics*, **23**(4), 387–399.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, **18**(6), 543–554.

Carletta, J., Isard, A., Isard, S., Kowtko, J. C., Doherty-Sneddon, G., and Anderson, A. H. (1997). The reliability of a dialogue structure coding scheme. *Computational Linguistics.*

Carroll, J. M., Thomas, J. C., and Malhotra, A. (1980). Presentation and representation in design problem-solving. *British Journal of Psychology*, **71**(1), 143–153.

Chapin, N. (1970). Flowcharting with the ANSI standard: A tutorial. *Computing Surveys*, **2**(2), 119–146.

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

Clocksin, W. F. and Mellish, C. S. (1981). *Programming in Prolog, First Edition.* Springer-Verlag, New York.

Corritore, C. L. and Wiedenbeck, S. (1991). What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, **3**(2), 199 – 222.

Cox, P. T. and Pietrzykowski, T. (1990). Using a pictorial representation to combine dataflow and object-orientation in a language independent programming mechanism. In E. P. Glinert, editor, *Visual Programming Environments, Vol I: Paradigms and Systems*, pages 313–322. IEEE Computer Society Press, Washington, D.C.

Cox, R. (1996). *Analytical Reasoning with Multiple External Representations.* Ph.D. thesis, The University of Edinburgh.

Cox, R. and Brna, P. (1993). The relationship between external representations and analytical reasoning performance: implications for the design of a learning environment. DAI Research Paper 646, Department of Artificial Intelligence, University of Edinburgh.

Cox, R., Stenning, K., and Oberlander, J. (1994). Graphical effects in learning logic: reasoning, representation and individual differences. In A. Ram and K. Eiselt, editors, *Proceedings of the 16th Annual Conference of the Cognitive Science Society*, pages 237–242. Lawrence Erlbaum & Associates.

Cunniff, N. and Taylor, R. P. (1987). Graphical vs. textual representation: An empirical study of novices' program comprehension. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 114–131, New Jersey. Ablex Publishing Corporation.

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

Davis, A. L. and Keller, R. M. (1982). Data flow program graphs. *IEEE Computer*, **15**, 26–41.

Dennis, J. B. (1986). Models of data flow computation. In *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 346–354, London. Springer-Verlag.

Dichev, C. and du Boulay, B. (1988). A data tracing system for Prolog novices. Cognitive Science Research Paper CSRP 113, University of Sussex.

diSessa, A. (1988). Knowledge in pieces. In G. Forman and P. B. Pufall, editors, *Constructivism in the Computer Age*, pages 49–70. Lawrence Erlbaum Associates, Hillsdale, NJ.

du Boulay, B. and O'Shea, T. (1981). Teaching novices programming. In M. J. Coombs and J. L. Alty, editors, *Computing Skills and the User Interface*, Computers and People Series, pages 147–200. Academic Press, London.

Eisenberg, M., Resnick, M., and Turbak, F. (1987). Understanding procedures as objects. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 14–32, New Jersey. Ablex Publishing Corporation.

Ekstrom, R. B., French, J. W., Harman, H. H., and Dermen, D. (1976). *Manual for the Kit of Factor-Referenced Cognitive Tests.* Educational Testing Service, Princeton, NJ.

Engelhardt, Y., de Bruin, J., Janssen, T., and Scha, R. (1996). The visual grammar of information graphics. In J. C. B. Damski and N. H. Narayanan, editors, *Workshop Notes Visual Representation, Reasoning and Interaction in Design, Artificial Intelligence in Design '96.*

Ennals, J. R. (1981). Prolog: An introduction for teachers. Technical Report 81/7, Department of Computing, Imperial College.

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

Gellenbeck, E. M. and Cook, C. R. (1991). An investigation of procedure and variable names as beacons during program comprehension. In S. P. Koenemann, T. G. Moher, and S. P. Robertson, editors, *Empirical Studies of Programmers: Fourth Workshop*, pages 65–81, New Jersey. Ablex Publishing Corporation.

George, C. G. (1996). *Investigating the Effectiveness of a Software Reinforced Approach to Understanding Recursion.* Ph.D. thesis, Department of Mathematical and Computing Sciences, Goldsmith's College.

Gilmore, D. J. (1990). Expert programming knowledge: A strategic approach. In J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, editors, *Psychology of Programming*, Computers and People Series, chapter 3.2, pages 223–234. Academic Press, London.

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

Gilmore, D. J. and Green, T. R. G. (1988). Programming plans and programming expertise. *The Quarterly Journal of Experimental Psychology*, **40A**(3), 423–442.

Gilmore, D. J. and Smith, H. T. (1984). An investigation of the utility of flowcharts during computer program debugging. *International Journal of Man-Machine Studies*, **20**, 357–372.

Golin, E. J. (1991). A method for the specification and parsing of visual languages. Technical Report CS-90-19, Brown University, Department of Computer Science.

Good, J. (1996). The 'right' tool for the task: An investigation of external representations, program abstractions and task requirements. In W. D. Gray and D. A. Boehm-Davis, editors, *Empirical Studies of Programmers: Sixth Workshop*, pages 77–98. Ablex Publishing Corporation.

Good, J. and Brna, P. (1996a). Novice difficulties with recursion: Do graphical representations hold the solution? In *Proceedings of the European Conference on AI in Education, Lisbon, Portugal, September 30 – October 2, 1996.*

Good, J. and Brna, P. (1996b). Scaffolding for recursion: Can visual languages help? In *IEE Colloquium on Thinking with Diagrams*, pages 7/1–7/3. IEE.

Good, J. and Brna, P. (1998a). Explaining programs: when talking to your mother can make you look smarter. In *Proceedings of the Tenth Annual Meeting of the Psychology of Programming Interest Group (PPIG-10)*, pages 61–70.

Good, J. and Brna, P. (1998b). Information types and cognitive principles in program comprehension: Towards adaptable support for novice visual programmers. In *Proceedings of the Intelligent Tutoring Systems Conference (ITS'98).*

Goodman, N. (1976). *Languages of Art: An Approach to a Theory of Symbols.* Bobbs-Merrill, Indianapolis, 2nd edition edition.

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

Green, T. R. G. (1980). Programming as a cognitive activity. In H. Smith and T. R. G. Green, editors, *Human Interaction with Computers*, pages 271–320. Academic Press.

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

Green, T. R. G. (1990). The nature of programming. In J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. gilmore, editors, *Psychology of Programming*, Computers and People Series, pages 21–44. Academic Press, London.

Green, T. R. G. (1997). Cognitive approaches to software comprehension: Results, gaps and limitations. Extended abstract of talk at workshop on Experimental Psychology in Software Comprehension Studies 97, University of Limerick, Ireland. at URL http://www.ndirect.co.uk/ thomas.green/workStuff/Papers/LimerickTal k1997/LimerickTalk.html (current on 30th January 1998).

Green, T. R. G. (to appear). Building and comprehending complex information structures: Issues in prolog programming. In P. Brna, B. du Boulay, and H. Pain, editors, *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, Cognitive Science & Technology. Ablex, Stamford, CT.

Green, T. R. G. and Petre, M. (1992). When visual programs are harder to read than textual programs. In G. C. van der Veer, M. J. Tauber, S. Bagnarola, and A. M., editors, *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)*, Rome. CUD.

Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: A cognitive dimensions framework. *Journal of Visual Languages and Computing*, **7**, 131–174.

Green, T. R. G., Sime, M. E., and Fitter, M. J. (1980). The problems the programmer faces. *Ergonomics*, **23**, 893–907.

Green, T. R. G., Petre, M., and Bellamy, R. K. E. (1991). Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. In S. P. Koenemann, T. G. Moher, and S. P. Robertson, editors, *Empirical Studies of Programmers: Fourth Workshop*, pages 121–146, New Jersey. Ablex Publishing Corporation.

Holt, R. W., Boehm-Davis, D. A., and Schultz, A. C. (1987). Mental representations of student and professional programmers. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 33–46, New Jersey. Ablex Publishing Corporation.

Jenkins, M. A., Glasgow, J. I., and McCrosky, C. D. (1986). Programming styles in Nial. *IEEE Software*, **3**(1), 46–55.

Johnson, W. L. and Soloway, E. (1985). Proust: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, **11**(3), 267–275.

Kahney, H. (1989). What do novice programmers know about recursion? In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*, pages 209–228. Lawrence Erlbaum Associates.

Kimura, T. D., Choi, J. W., and Mack, J. M. (1990). Show and tell: A visual programming language. In E. P. Glinert, editor, *Visual Programming Environments, Vol. I: Paradigms and Systems*, pages 397–404. IEEE Computer Society Press, Washington, D.C.

Kintsch, W. and van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review*, **85**, 363–394.

Kline, P. (1986). *A Handbook of Test Construction: Introduction to Psychometric Design.* Methuen & Co.

Knott, A. and Dale, R. (1994). Using linguistic phenomena to motivate a set of coherence relations. *Discourse Processes*, **18**, 35–62.

Kowalski, R. (1979). *Logic for Problem Solving.* Artificial Intelligence Series. North-Holland, New York.

Kreutzer, W. and McKenzie, B. (1991). *Programming for Artificial Intelligence: Methods, Tools and Applications.* Addison-Wesley, Sydney.

Kurland, D. M. and Pea, R. D. (1983). Children's mental models of recursive Logo programs. In *Proceedings of the 5th Annual Conference of the Cognitive Science Society*, pages 1–5, Rochester, N.Y.

Larkin, J. H. and Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, **11**, 65–99.

Letovsky, S. (1986). Cognitive processes in program comprehension. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers: First Workshop*, pages 58–79, New Jersey. Ablex Publishing Corporation.

Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E. (1986). Mental models and software maintenance. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers: First Workshop*, pages 80–98, New Jersey. Ablex Publishing Corporation.

Lohse, G. L., Biolsi, K., Walker, N., and Rueter, H. (1994). A classification of visual representations. *Communications of the ACM*, **37**(12), 36–49.

Mayer, R. E. (1988). From novice to expert. In M. Helander, editor, *Handbook of Human-Computer Interaction*, chapter 25, pages 569–580. Elsevier.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., and Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, **13**, 307–325.

Merrill, D. C. and Reiser, B. J. (1994). Scaffolding effective problem solving strategies in interactive learning environments. In L. Erlbaum, editor, *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society.*

Merrill, D. C., Reiser, B. J., Merrill, S. K., and Landes, S. (1993). Tutoring: Guided learning by doing. Technical Report 45, The Institute for the Learning Sciences, Northwestern University.

Modugno, F., Corbett, A. T., and Myers, B. A. (1996). Evaluating program representation in a demonstrational visual shell. In W. D. Gray and D. A. Boehm-Davis, editors, *Empirical Studies of Programmers: Sixth Workshop*, page ? Ablex Publishing Corporation.

Moher, T. G., Mak, D. C., Blumenthal, B., and Levanthal, L. M. (1993). Comparing the comprehensibility of textual and graphical programs: The case of petri nets. In C. Cook, J. Scholtz, and J. Spohrer, editors, *Empirical Studies of Programmers: Fifth Workshop*, pages 137–161, New Jersey. Ablex Publishing Corporation.

Moss, J. and Case, R. (to appear). Developing children's understanding of the rational numbers: A new model and an experimental curriculum. *Journal for Research in Mathematics Education.*

Mulholland, P. (1994). The effect of graphical and textual visualisation on the comprehension of Prolog execution by novices: an empirical analysis. In *Collected Papers of the Sixth Workshop of the Psychology of Programming Interest Group*, pages 18–26.

Myers, B. A. (1986). Visual programming: Programming by example, and program visualization: A taxonomy. In *CHI '86: Human Factors in Computing Systems*, pages 59–66.

Nassi, I. and Shneiderman, B. (1973). Flowchart techniques for structured programming. *SIGPLAN Notices*, **8**(8), 12–26.

Norman, D. A. (1993). *Things that make us smart: Defending human attributes in the age of the machine.* Addison-Wesley Publishing Company.

Olson, G. M., Catrambone, R., and Soloway, E. (1987). Programming and algebra word problems: A failure to transfer. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 1–13, New Jersey. Ablex Publishing Corporation.

Pain, H. and Bundy, A. (1987). What stories should we tell novice Prolog programmers. In R. Hawley, editor, *Artificial Intelligence Programming Environments.* Ellis Horwood, Chichester.

Pair, C. (1990). Programming, programming languages and programming methods. In J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, editors, *Psychology of Programming*, Computers and People Series, pages 9–19. Academic Press, London.

Pandey, R. K. and Burnett, M. M. (1993). Is it easier to write matrix manipulation programs visually or textually? an empirical study. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 344–351.

Pennington, N. (1987a). Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113, New Jersey. Ablex Publishing Corporation.

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

Pennington, N. and Grabowski, B. (1990). The tasks of programming. In J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, editors, *Psychology of Programming*, pages 45–62. Academic Press.

Perkins, D. N. and Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers: First Workshop*, pages 213–229, New Jersey. Ablex Publishing Corporation.

Petre, M. (1996). Programming paradigms and culture: implications of expert practice. In M. Woodman, editor, *Programming Language Choice: Practice and Experience*, pages 29–44. International Thomson Computer Press, London.

Petre, M. and Green, T. R. G. (1992). Requirements of graphical notations for professional users: Electronics CAD systems as a case study. *Le Travail humain*, **55**(1), 47–70.

Petre, M. and Green, T. R. G. (1993). Learning to read graphics: Some evidence that 'seeing' an information display is an acquired skill. *Journal of Visual Languages and Computing*, **4**, 55–70.

Petre, M. and Winder, R. (1988). Issues governing the suitability of programming languages for programming tasks. In D. M. Jones and R. Winder, editors, *People and Computers IV*. Cambridge University Press.

Petre, M., Price, B., Fix, V., Scholtz, J., Wiedenbeck, S., Netesin, I., and Yershov, S. (1995). Comparing program comprehension in different cultures and different representations. In *7th Workshop of the Psychology of Programming Interest Group (PPIG-7)*, page 94.

Petre, M., Blackwell, A. F., and Green, T. R. G. (1998). Cognitive questions in software visualisation. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multi-Media Experience*, pages 453–480. MIT Press.

Pirolli, P. (1986). A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction*, **2**(2), 319–355.

Pirolli, P. L. and Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, **39**(2), 240–272.

Poulton, E. C. (1965). On increasing the sensitivity of measures of performance. *Ergonomics*, pages 69–76.

Ramalingam, V. and Wiedenbeck, S. (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Proceedings of 7th Workshop on Empirical Studies of Programmers*, Alexandria, VA USA.

Ramsey, H. R., Atwood, M. E., and Van Doren, J. R. (1983). Flowcharts versus program design languages: An experimental comparison. *Communications of the ACM*, **26**(6), 445–449.

Raymond, D. R. (1991). Characterizing visual languages. In *Proc. of the 1991 IEEE Workshop on Visual Languages*, pages 176–182, Kobe, Japan.

Rist, R. (1986). Plans in programming: Definition, demonstration and development. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers: First Workshop*, pages 28–47, New Jersey. Ablex Publishing Corporation.

Robertson, S. P. and Yu, C.-C. (1990). Common cognitive representations of program code across tasks and languages. *International Journal of Man-Machine Studies*, **33**, 343–360.

Santori, M. (1990). An instrument that isn't really. *IEEE Spectrum*, **27**(8), 36–39.

Scanlan, D. A. (1989). Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, **6**(5), 28–36.

Scanlon, E. and O'Shea, T. (1988). Cognitive economy in physics reasoning: Implications for designing instructional materials. In H. Mandl and A. Lesgold, editors, *Learning Issues for Intelligent Tutoring Systems*. Springer-Verlag, New York.

Shneiderman, B. and Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, **8**(3), 219–238.

Shneiderman, B., Mayer, R., McKay, D., and Heller, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, **20**(6), 373–381.

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

Sinha, A. P. and Vessey, I. (1992). Cognitive fit: An empirical study of recursion and iteration. *IEEE Transactions on Software Engineering*, **18**, 368–379.

Soloway, E. and Ehrlich, K. (1982). Tacit programming knowledge. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pages 149–151.

Soloway, E. and Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, **SE-10**(5), 595–609.

Soloway, E. and Spohrer, J. C., editors (1989). *Studying the Novice Programmer*. Lawrence Erlbaum Associates.

Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J. (1982). What do novices know about programming? In A. Badre and B. Shneiderman, editors, *Directions in Human/Computer Interaction*, pages 27–54. Ablex.

Soloway, E., Adelson, B., and Ehrlich, B. (1988). Knowledge and processes in the comprehension of computer programs. In M. T. H. Chi, R. Glaser, and M. J. Farr, editors, *The Nature of Expertise*, pages 129–152. Lawrence Erlbaum Associates.

Soloway, E., Guzdial, M., and Hay, K. E. (1992). Programming for the rest of us. In *5th Workshop of the Psychology of Programming Interest Group (PPIG5)*, pages 23–27.

Stenning, K. and Oberlander, J. (1995). A cognitive theory of graphical and linguistic reasoning: Logic and implementation. *Cognitive Science*, **19**(1), 97–140.

Stenning, K., Cox, R., and Oberlander, J. (1995). Contrasting the cognitive effects of graphical and sentential logic teaching: reasoning, representation and individual differences. *Language and Cognitive Processes*, **10**.

Tabachneck, H. J. M., Leonardo, A. M., and Simon, H. A. (1994). How does an expert use a graph? A model of visual and verbal inferencing in economics. In *Proceedings of the 16th Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum & Associates.

Tabachnick, B. G. and Fidell, L. S. (1983). *Using Multivariate Statistics*. Harper Row, New York.

Taylor, J. (1988). Programming in Prolog: An in-depth study of problems for beginners. Cognitive Science Research Paper CSRP 111, University of Sussex.

Thorndike, R. L. and Hagen, E. P. (1977). *Measurement and Evaluation in Psychology and Education*. John Wiley & Sons.

Trafton, J. G. and Reiser, B. J. (1991). Providing natural representations to facilitate novices' understanding in a new domain: Forward and backward reasoning in programming. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society*, pages 923–927, Chicago, Illinois. Lawrence Erlbaum.

Ushakov, I. and Velbitskiy, I. (1993). Visual programming in r-technology: Concepts, systems and perspectives. In *East-West International Conference on Human-Computer Interaction: Proceedings of the EWHCI'93*, pages 71–88. Intl. Centre for Scientific and Technical Information.

van Dijk, T. A. and Kintsch, W. (1983). *Strategies of Discourse Comprehension*. Academic Press, New York.

van Someren, M. W. (1990a). Understanding students' errors with Prolog unification. *Instructional Science*, **19**, 361–376.

van Someren, M. W. (1990b). What's wrong? understanding beginners' problems with Prolog. *Instructional Science*, **19**, 257–282.

Vessey, I. (1991). Cognitive fit: A theory-based analysis of the graphs versus tables literature. *Decision Sciences*, **22**, 219–240.

Vessey, I. and Weber, R. (1986). Structured tools and conditional logic: An empirical investigation. *Communications of the ACM*, **29**(1), 48–57.

von Mayrhauser, A. and Vans, A. M. (1994). Program understanding — a survey. Technical report, Colorado State University.

Wadge, W. W. and Ashcroft, E. A. (1985). *Lucid, the Dataflow Programming Language*. Academic Press, London.

Weinberg, G. M. (1971). *The Psychology of Computer Programming*. Computer Science Series. Van Nostrand Reinhold Company, New York.

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

Wells, M. B. and Kurtz, B. L. (1989). Teaching multiple programming paradigms: A proposal for a paradigm-general pseudocode. *SIGSCE Bulletin*, **21**(1), 246–251.

Welty, C. and Stemple, D. W. (1981). Human factors comparison of a procedural and a nonprocedural query language. *ACM Transactions on Database Systems*, **6**(4), 626–649.

Whitley, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, **8**(1), 109–142.

Widowski, D. (1987). Reading, comprehending and recalling computer programs as a function of expertise. In *Proceedings of CERCLE Workshop on Complex Learning.*

Wiedenbeck, S. (1986). Processes in computer program comprehension. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers: First Workshop*, pages 48–57, New Jersey. Ablex Publishing Corporation.

Winn, W. (1993). An account of how readers search for information in diagrams. *Contemporary Educational Psychology*, **18**, 162–185.

Wright, P. and Reid, F. (1973). Written information: Some alternatives to prose for expressing the outcomes of complex contingencies. *Journal of Applied Psychology*, **57**(2), 160–166.

Appendix A

Materials: Prolog Experiment

This appendix shows the experimental packet given to subjects in the Prolog experiment (described in Chapter 3), and includes:

1. the instructions to subjects[1];
2. the experimental problems and comprehension questions;
3. the programming self-report questionnaire.

[1] As the experiment took place within a practical session, the instructions for the experiment were embedded within a description of that day's practical session.

Today's Practical

Today's practical session investigates experimental design and methodology, particularly regarding experiments with people. The aim is to give you a better idea of the issues involved in designing and running an experiment. Starting from the practical implementation of an experiment, we will see whether it is possible to uncover the design decisions underlying the experiment, and whether we can make any predictions about the expected outcome.

During the practical session, you will be asked to take part in a short experiment. The main aim in doing so is to give you first hand experience of experimental methodology and design. The experiment may also shed some light on the use of Prolog as a language for teaching programming, and help you consolidate your knowledge of Prolog as you reflect on it in different ways.

The experiment aims to investigate whether having learned Prolog (as opposed to learning another programming language, say, Lisp or Pascal) affects the way you think about programming. Of course, we could simply ask you this question directly, but as you can probably guess, doing so wouldn't necessarily give us the types of answers which might be useful: for example, the answers might not be quantifiable in a consistent way and hence will be not suitable for any sort of statistical analysis.

The experiment in question is not a "toy" experiment: we hope to collect data which can be analysed, and which will either support work which has already been done on the subject, or offer new and possibly conflicting evidence[2].

Before starting the experiment, you should be aware that:

- this is in no way a test of your Prolog knowledge, and the data collected will not be used in any form as part of your assessment on the AI1 course[3];
- you are not expected to get all of the questions right. What we're interested in here are the *types* of questions which people get wrong compared to those they get right;
- the data obtained will be both anonymous and strictly confidential.

What to do

After reading through these instructions, your demonstrator will give you a booklet to work through. On each page of the booklet (except for the last page), there will be a short Prolog program. For each program, you will be asked to answer five questions about the program and write a short summary of it. You will have five minutes to work on each page and your demonstrators will tell you when to stop work on one page and start on the next.

[2] If you are interested obtaining more information on this research, please let your demonstrator know.

[3] Please also note that the *content* of the experiment (*i.e.* Prolog) will not be examined in the degree exam; however the issues of experimental design and methodology are examinable topics.

If you finish a page before the five minutes are up, please DO NOT start working on the next one until the demonstrators tell you to do so. Also, please DO NOT go back to previous pages.

There will be a total of six programs, however, the first program will be a practice one so that you can get a feel for what you are required to do. If you have any questions after completing this practice program sheet, there will be time to ask the demonstrators before going on to the other programs.

The last page will have some questions asking you about your programming experience. Please fill this in, and then give the entire booklet to your demonstrator.

Note that it will not be possible for the demonstrators to give you details of the experimental design, as knowing the hypotheses may influence how you respond[4]. When you have handed in your booklet to the demonstrators, you will receive a summary of the issues involved in the experiment, and some questions to answer. These are designed to help you try and uncover the design and methodology information underlying the experiment. On Friday, you will receive an email explaining the design of the experiment, giving the experimental hypotheses, the variables manipulated, anticipated results and possible conclusions.

If you have any questions about these instructions, please ask the demonstrators now.

[4] For the same reason, please do not discuss your ideas on the experimental design with people who have not yet taken part in this week's practical session.

```
?- alter([i, like, my, bike], R).


1a. alter([], []).

1b. alter([X|Xs], [Y|Ys]):-
            change(X, Y),
            alter(Xs, Ys).

2a. change(my, your).
2b. change(i,you).
2c. change(me,you).
2d. change(ours,yours).
2e. change(X,X).
```

1. Will the output list contain items in the input list?

2. Does `change(X, X)` output the same value it receives as input?

3. Does the variable `X` in clause `1b` get bound to `my` before it gets bound to `i`?

4. Is `Xs` always instantiated in the recursive call (i.e. in clause `1b`)?

5. Does this program change singular nouns into plural ones?

6. Now write a short summary of what this program does.

```
?- outer([1,2,3,4,5], [1,2,3,4,5], Holder).


1a. outer([], _, []).

1b. outer([I|Is], L, [Y|Ys]):-
            inner(L, I, 0, Y),
            outer(Is, L, Ys).

2a. inner([], _, Sum, Sum).

2b. inner([J|Js], I, SumIn, Sum):-
            Temp is I * J,
            SumNext is SumIn + Temp,
            inner(Js, I, SumNext, Sum).
```

1. Does the variable `Temp` affect the value of the variable `Y`?

2. Is the variable `Temp` initially instantiated to `0`?

3. Is the variable `SumNext` calculated before the variable `Temp` is calculated?

4. When the variable `J` is instantiated to `5`, is the variable `SumIn` equal to `0`?

5. Does the program output a list containing the sums of squares of all numbers between `1` and `5`?

6. Now write a short summary of what this program does.

```
?- match([9, 12, 10, 11], C).


1a. match(L, C):-
            match_sub(L, 0, C).

2a. match_sub([], C, C).

2b. match_sub([X|Xs], C0, C):-
            X > 10,
            C1 is C0 + 1,
            match_sub(Xs, C1, C).

2c. match_sub([X|Xs], C0, C):-
            X =< 10,
            match_sub(Xs, C0, C).
```

1. Is the second argument of `match_sub` an input argument?

2. Is `C1` ever instantiated to the same value as `C0`?

3. When `X` is bound to `10`, is clause `2b` of `match_sub` executed?

4. When `X` is bound to `9`, is `C0` bound to `0`?

5. Does the program add up the numbers less than `10` in the input list?

6. Now write a short summary of what this program does.

```
?- comb([1, 4, 6, 7], [2, 3, 8], R).


1a. comb([], L, L).

1b. comb(L, [], L).

1c. comb([X|Xs], [Y|Ys], [Z|Zs]):-
            X < Y,
            Z = X,
            comb(Xs, [Y|Ys], Zs).

1d. comb([X|Xs], [Y|Ys], [Z|Zs]):-
            X >= Y,
            Z = Y,
            comb([X|Xs], Ys, Zs).
```

1. Will the list that becomes bound to the variable `R` in the query contain all elements of the input lists?

2. Is `Z` always instantiated to `X` or `Y`?

3. When `X` is bound to `4` and `Y` is bound to `3`, is clause `1c` the next successfully executed clause?

4. When the first list is empty, will the second one also be empty?

5. Does the program combine the elements of the input lists in ascending order?

6. Now write a short summary of what this program does.

```
?- adjust([1, 3, 2, 7], Res).


1a. adjust(X, R):-
            adjust_sub(X, 0, R).

2a. adjust_sub([], _, []).

2b. adjust_sub([X|Xs], Y, [Z|Zs]):-
            Z is X + Y,
            adjust_sub(Xs, X, Zs).
```

1. Is the variable `Y` always set to `0`?

2. Is `Z` initially instantiated to `0`?

3. Does the program recurse over all of the elements of the list?

4. When `X` is instantiated to `3`, is the value of `Y` equal to `2`?

5. Does this program total the numbers in the list?

6. Now write a short summary of what this program does.

```
?- a([4, 8, 3, 2], R).


1a. c([], 0).

1b. c([X|Xs], C):-
            c(Xs, CXs),
            C is CXs + 1.

2a. s([], 0).

2b. s([Y|Ys], S):-
            s(Ys, SYs),
            S is Y + SYs.

3a. a(L, A):-
            c(L, CL),
            s(L, SL),
            A is SL/CL.
```

1. When the first argument of `c` is an empty list, can the second argument have a value other than `0`?

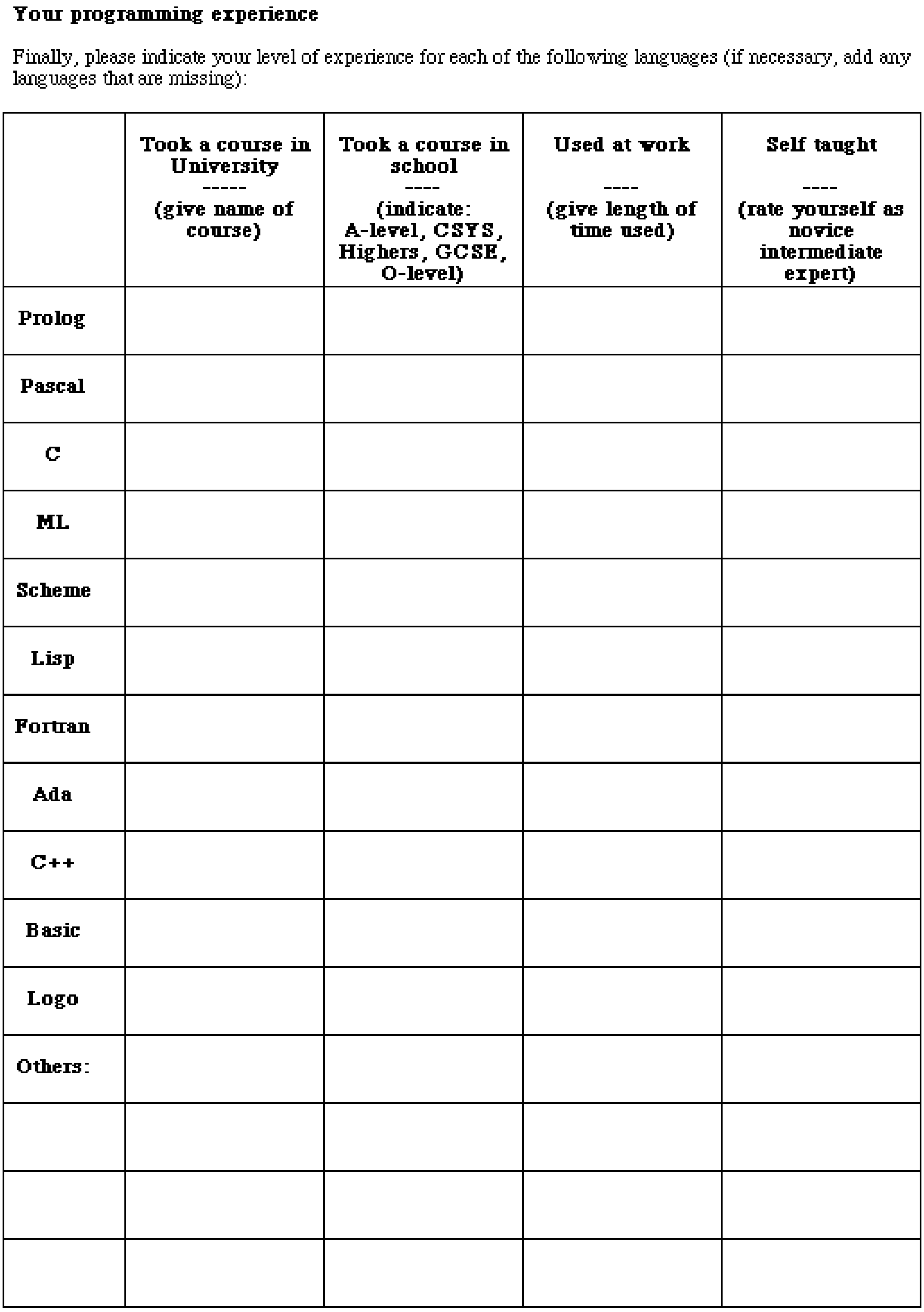
2. Is `S` the difference between `Y` and `SYs`?

3. Does the value of `X` affect the value of `C`?

4. Is there a part of this program which calculates the sum of the list of numbers?

5. Is `SL` calculated before `CL`?

6. Now write a short summary of what this program does.



Your programming experience

Finally, please indicate your level of experience for each of the following languages (if necessary, add any languages that are missing):

	Took a course in University ----- (give name of course)	Took a course in school ---- (indicate: A-level, CSYS, Highers, GCSE, O-level)	Used at work ---- (give length of time used)	Self taught ---- (rate yourself as novice intermediate expert)
Prolog				
Pascal				
C				
ML				
Scheme				
Lisp				
Fortran				
Ada				
C++				
Basic				
Logo				
Others:				

Appendix B

Materials: VPL Pilot Experiment

This appendix provides the materials used in the VPL pilot study (described in Chapter6), and includes:

1. the Prolog pre-test;
2. the `max` program in the four versions used in the experiment: control flow graph, control flow tree, data flow graph, data flow tree (the `position` program, also used, was shown in Chapter 5);
3. the comprehension questions for `max` and `position`.

B.1 Prolog Pre-Test

Questionnaire

Please complete the following questions. There is only one answer to each multiple-choice question: please circle the letter which corresponds to your answer. If you have any questions, please let me know. When you have finished, please hand these sheets in to me. Thank you.

1. Given this query: `rem_dup([d,a,r,c,r,r], X).` and the following code, in what order will the clauses of `rem_dup` be (successfully) called?

```
clause 1     rem_dup([], []).

clause 2     rem_dup([H|T], X):-
                   member(H,T), !,
                   rem_dup(T, X).

clause 3     rem_dup([H|T], [H|Final]):-
                   rem_dup(T, Final).

auxiliary     member(X, [X|_]).
predicate     member(X, [_|T]):-
                   member(X, T).
```

   (a) 3 - 3 - 2 - 3 - 2 - 2 - 1

   (b) 3 - 3 - 2 - 3 - 2 - 3 - 1

   (c) 3 - 2 - 1 - 3 - 2 - 3 - 2

   (d) 2 - 2 - 3 - 2 - 3 - 2 - 1

   (e) None of the above.

2. Given the following piece of code, what would be the value of `N` given this query: `mystery([3, [4,5,6], 2, [1]], N).` ?

```
mystery([H|T], N):-
        mystery(H, N1),
        mystery(T, N2),
        N is N1 + N2.

mystery([], 0).
mystery(X,1).
```

   (a) 21
   (b) 6
   (c) 2
   (d) 4
   (e) None of the above.

3. The predicate count_up/1 should show the following behaviour:

```
|?- count_up(3).
1
2
3

yes
```

However, there is a bug in this program which prevents it from doing so. What needs to be done to correct it?

```
line 1   count_up(0).
line 2   count_up(N):-
line 3         N1 is N-1,
line 4         write(N), nl,
line 5         count_up(N1).
```

(a) The base case (line 1) should be count_up(N) instead of count_up(0).

(b) Swap lines 5 and 3.

(c) Swap lines 5 and 4.

(d) Line 3 should be N+1 instead of N-1.

(e) None of the above.

4. Look at these two programs:

Program A

```
make_list([], []).
make_list([H|L1], [H|L2]):-
        make_list(L1, L2).
```

Program B

```
make_list(L1, L2):-
        make_list(L1, [], L2).

make_list([], X, X).
make_list([H|L1], X, L2):-
        make_list(L1, [H|X], L2).
```

If the query `make_list([a,d,r], X).` is given to both programs, what will they return?

(a) Program A will return X=[a,d,r].
Program B will return X=[a,d,r].

(b) Program A will return X=[a,d,r].
Program B will return X=[r,d,a].

(c) Program A will return X=[r,d,a].
Program B will return X=[r,d,a].

(d) Program A will return X=[r,d,a].
Program B will return X=[a,d,r].

(e) None of the above.

5. Which version of append, when given the query `append([1,2], [3,4], X).` will return `X = [1,2,3,4].`?

(a)

```
append(L, [], L).
append(L1, [H|L2], [H|L3]):-
        append(L1, L2, L3).
```

(b)

```
append([], L, L).
append([H|L1], L2, L3):-
        append(L1, L2, L3),
        NewL3 = [H|L3],
        append(L1, L2, NewL3).
```

(c)

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
        append(L1, L2, L3).
```

(d)

```
append([], L, L).
append([H|L1], L2, L3):-
        append(L1, [H|L2], L3).
```

(e) None of the above.

6. Finally, please indicate the programming languages you know, and how much experience you have with each language, e.g.

Pascal	: 3 months	- off and on
C	: 6 years	- used in my job
Prolog	: 1 year	- did the MSc course

B.2 The max Program (All Versions)

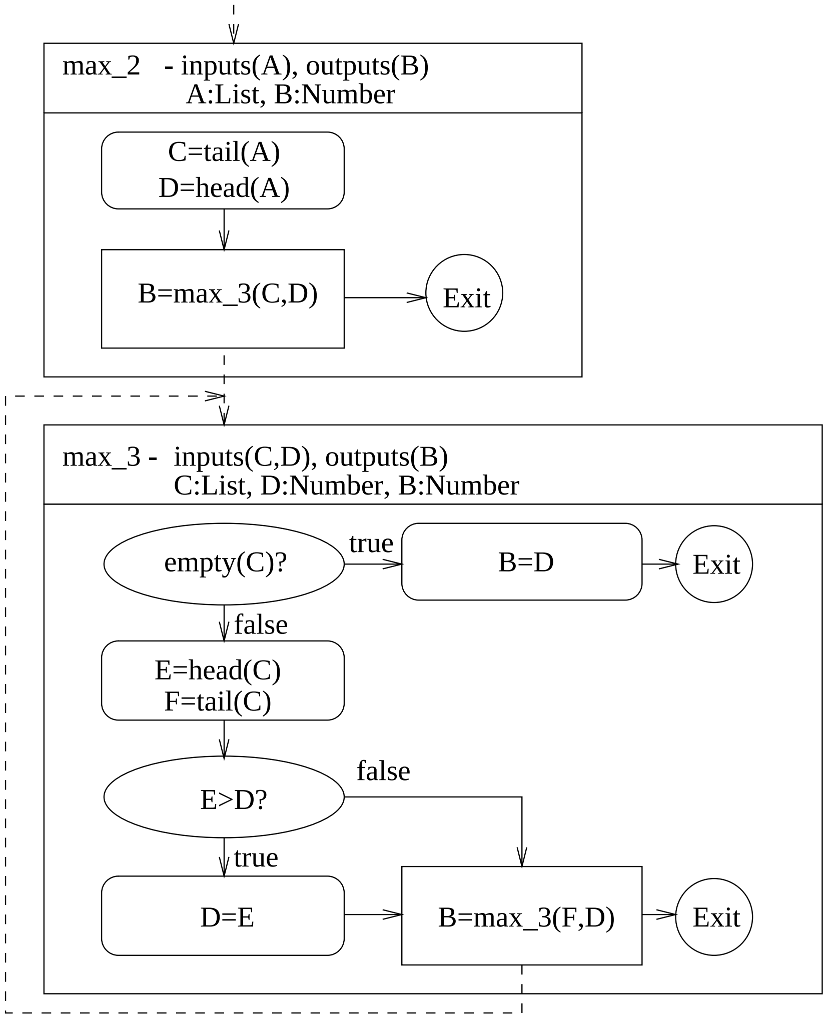


Figure B.1: Control Flow Graph Representation for max

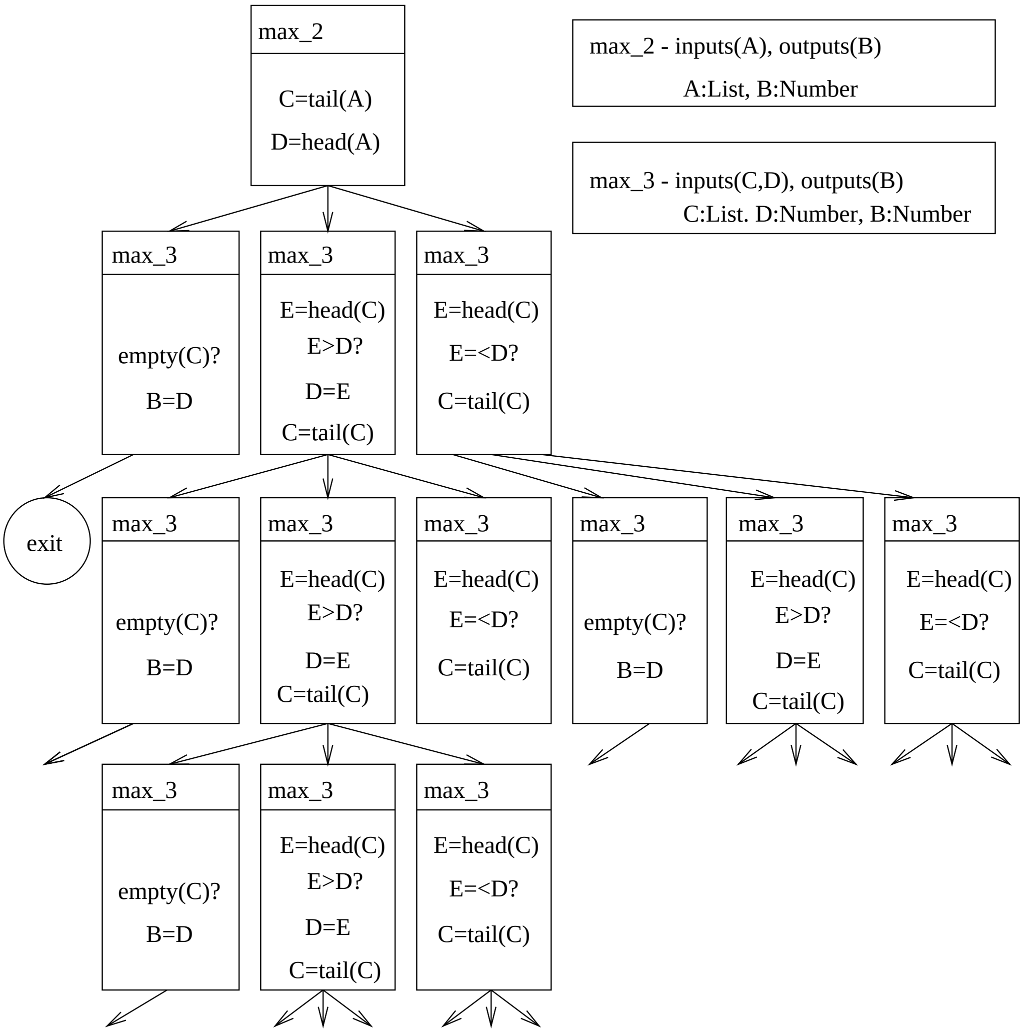


Figure B.2: Control Flow Tree Representation for `max`

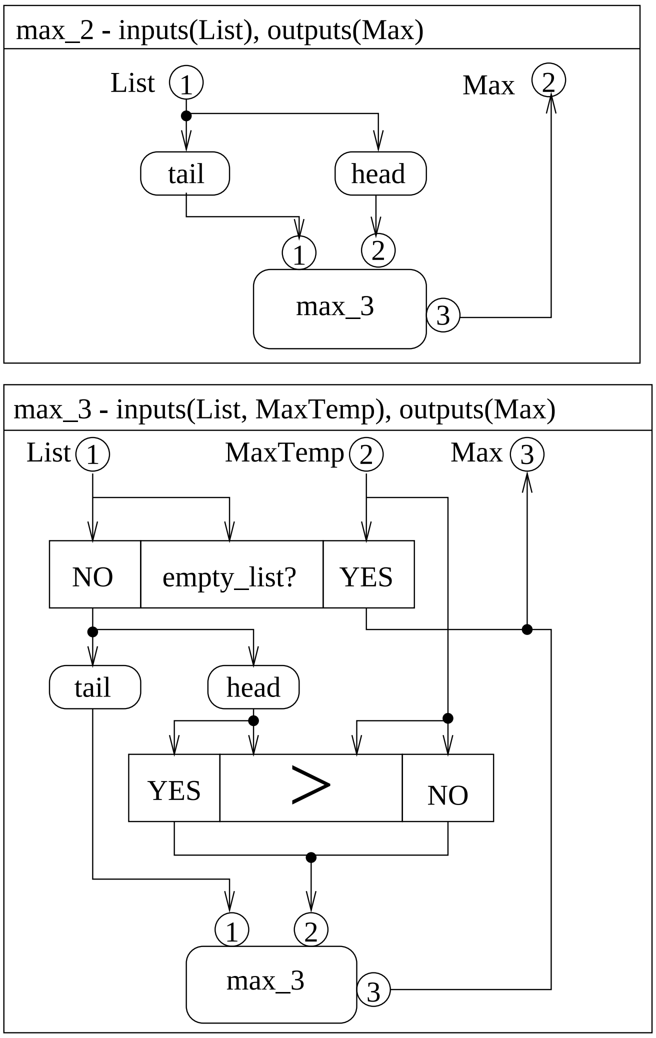


Figure B.3: Data Flow Graph Representation for `max`

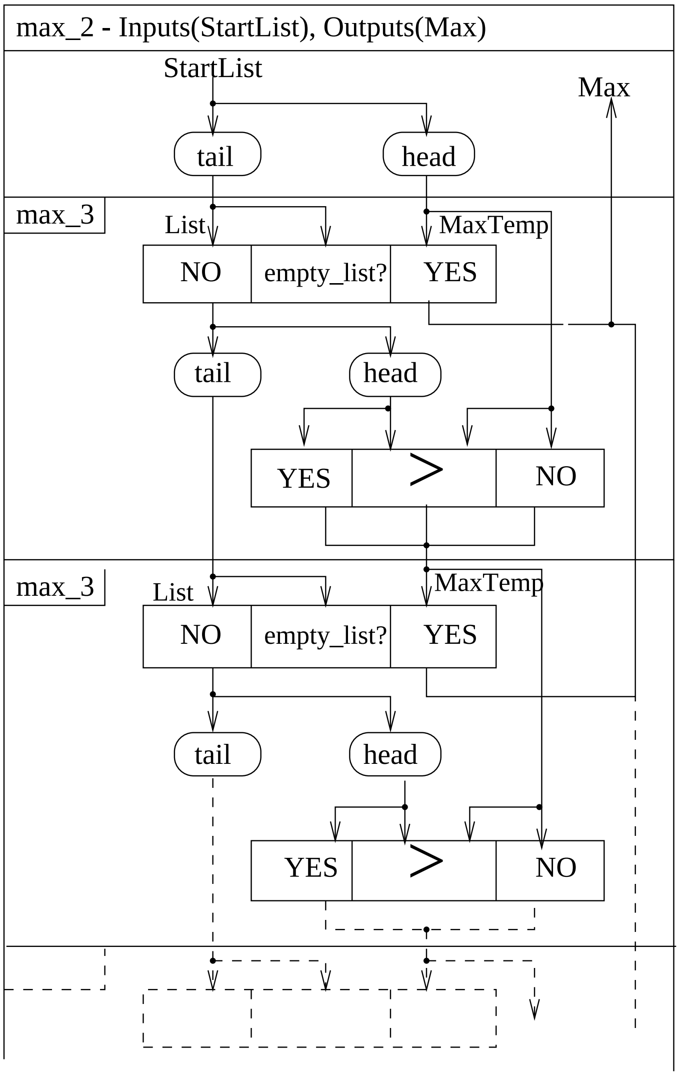


Figure B.4: Data Flow Tree Representation for `max`

B.3 Comprehension Questions

MAX: CONTROL FLOW

If max_2 were called with the list [6,9,7,1,10] and at the current call to max_3 (i.e. prior to execution of this call) List=[1,10] and MaxTemp=9, what were the last two unsuccessful tests carried out (working backwards from the current state)?

○ 9>6?, empty_list([7,1,10]?

○ 7>9?, empty_list([7, 1,10])?

○ empty_list([1,10])?, 7¿9?

○ empty_list([1, 10]?, 1¿9?

MAX: DATA FLOW

If the input to max_2 is the list [1,5,4,6,2], what will be the values of the list and MaxTemp just after three ”greater than” (>) comparisons (note that these comparisons can be either successful or unsuccessful)?

○ List=[6,2], MaxTemp=5

○ List=[6,2], MaxTemp=1

○ List=[2], MaxTemp=6

○ List=[], MaxTemp=6

Table B.1: Questions for VPL Pilot Study: `max`

POSITION: CONTROL FLOW

position_3 is given an element, e, and a list [a,x,e]. What will be the following events after the test ”e=e?” is reached?

○ position_3 is called with an empty list.

○ Position returns 1 and the call to position_3 terminates.

○ position_3 fails.

○ Position=1 for that call, and as each call to position_3 terminates,
Position is augmented by 1.

POSITION: DATA FLOW

position_3 is called with Element=g and List= [6,3,d,2,g,e]. On a subsequent call to position_3, List=[2,g,e] (before execution of that call). What is the value of Position when that particular call terminates?

○ [g,e].

○ 2

○ 2+1

○ It won't have a value.

Table B.2: Questions for VPL Pilot Study: `position`

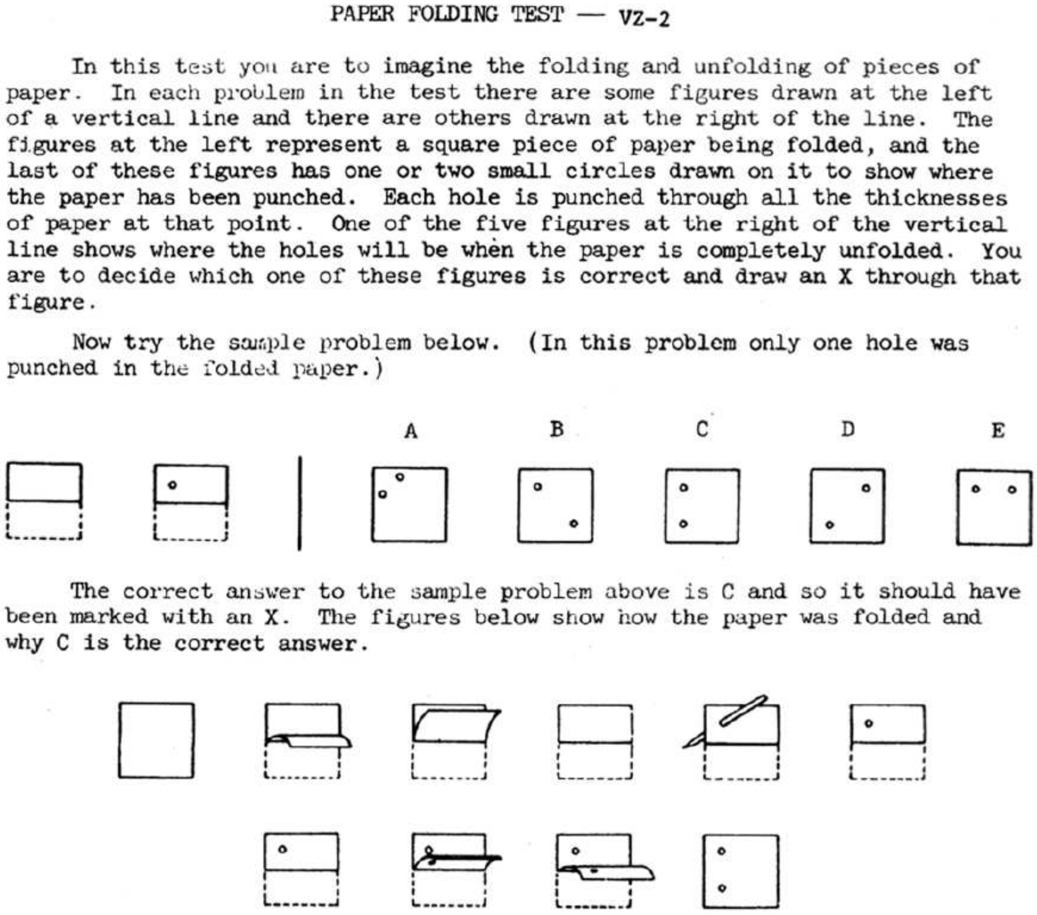
Appendix C

Materials: VPL Experiment

This appendix provides the materials used in the VPL experiment described in Chapter7, and includes:

1. Sample items from the paperfolding and pathfinding tests (Ekstrom *et al.*, 1976), used as pre-tests in the experiment;
2. Data and control flow versions of three of the four programs used: `basketball`, `distance_between` and `sunny` (the fourth program `passes`, was shown in Chapter 5).
3. Comprehension questions for all programs.
4. The programming self-report questionnaire.

C.1 Pre-Tests: Paperfolding and Pathfinding



PAPER FOLDING TEST — VZ-2

In this test you are to imagine the folding and unfolding of pieces of paper. In each problem in the test there are some figures drawn at the left of a vertical line and there are others drawn at the right of the line. The figures at the left represent a square piece of paper being folded, and the last of these figures has one or two small circles drawn on it to show where the paper has been punched. Each hole is punched through all the thicknesses of paper at that point. One of the five figures at the right of the vertical line shows where the holes will be when the paper is completely unfolded. You are to decide which one of these figures is correct and draw an X through that figure.

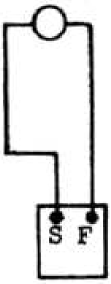
Now try the sample problem below. (In this problem only one hole was punched in the folded paper.)

The correct answer to the sample problem above is C and so it should have been marked with an X. The figures below show how the paper was folded and why C is the correct answer.

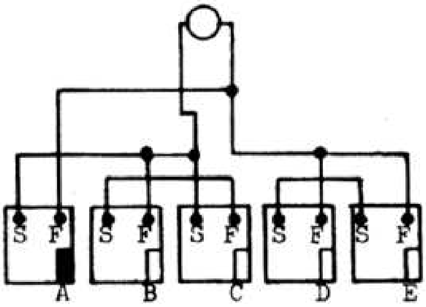
Figure C.1: Instructions and a Sample Problem from the Paper-Folding Test (Ekstrom *et al.*, 1976)

CHOOSING A PATH -- SS-2

This is a test of your ability to choose a correct path from among several choices. In the picture below is a box with dots marked S and F. S is the starting point and F is the finish. You are to follow the line from S, through the circle at the tcp of the picture and back to F.



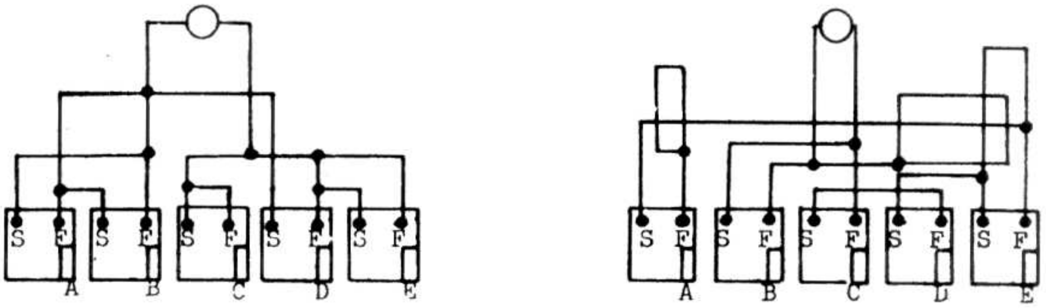
In the problems in this test there will be five such boxes. Only one box will have a line from the S, through the circle, and back to the F in the same box. Dots on the lines show the only places where connections can be made between lines. If lines meet or cross where there is no dot, there is no connection between the lines. Now try this example. Show which box has the line through the circle by blackening the space at the lower right of that box.



The first box is the one which has the line from S, through the circle, and back to F. The space lettered A has therefore been blackened.

Figure C.2: Instructions for the Pathfinding Test (Ekstrom *et al.*, 1976)

Now try the next two practice examples.



For the first example you should have marked the space lettered D. For the second example the answer is B.

Figure C.3: Two Sample Problems from the Pathfinding Test (Ekstrom *et al.*, 1976)

C.2 Programs used in the Experiment

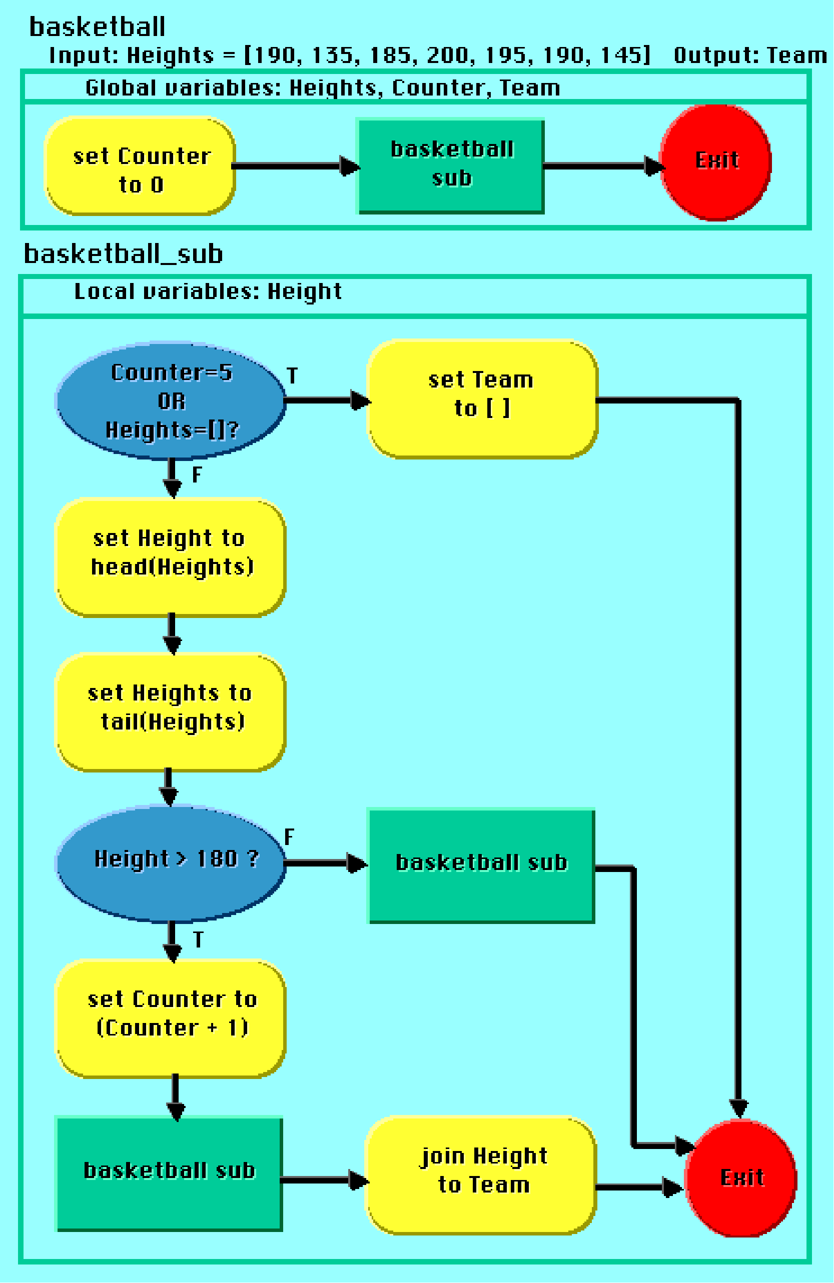


Figure C.4: The `basketball` Program: Control Flow Version

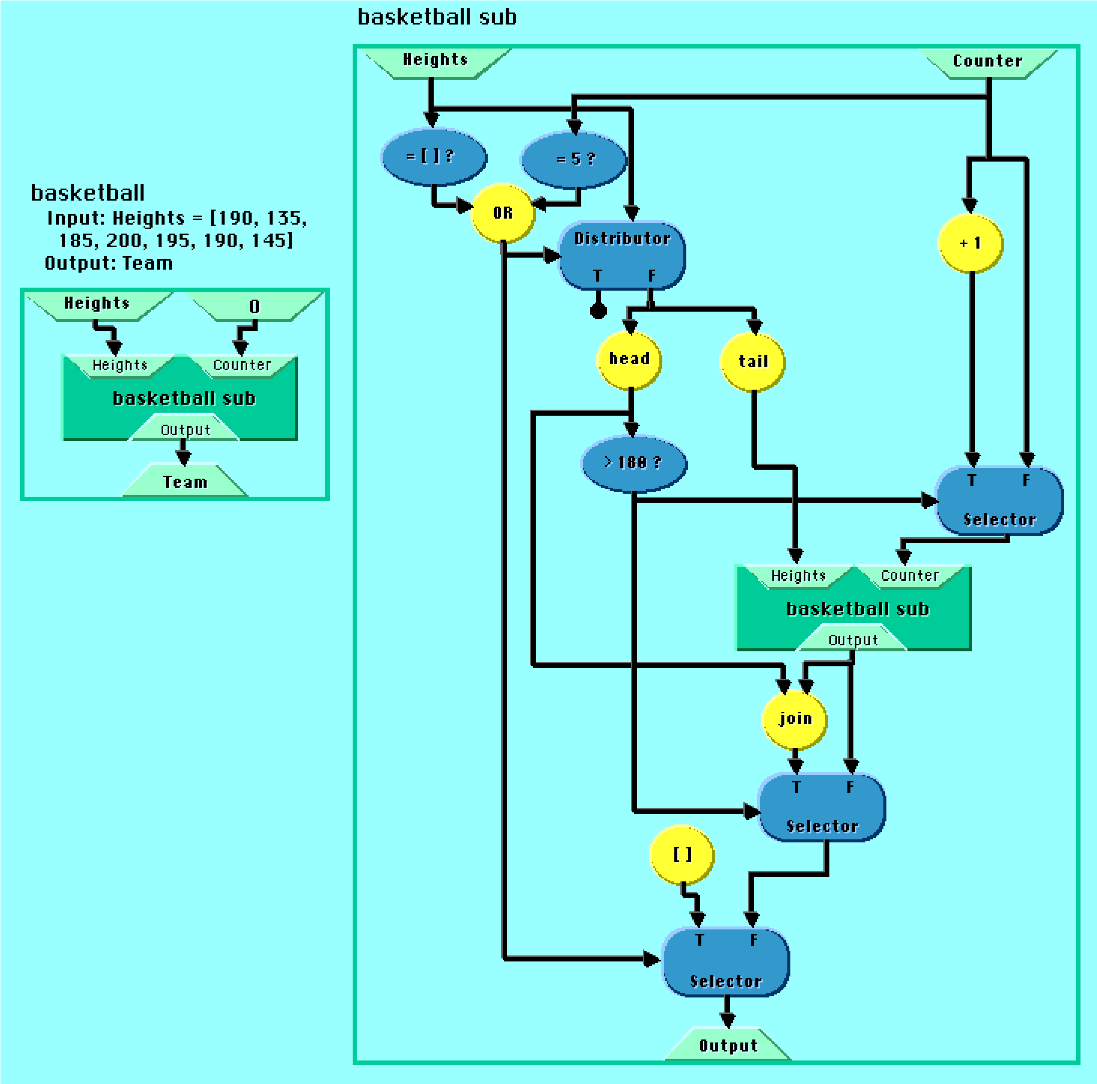


Figure C.5: The `basketball` Program: Data Flow Version

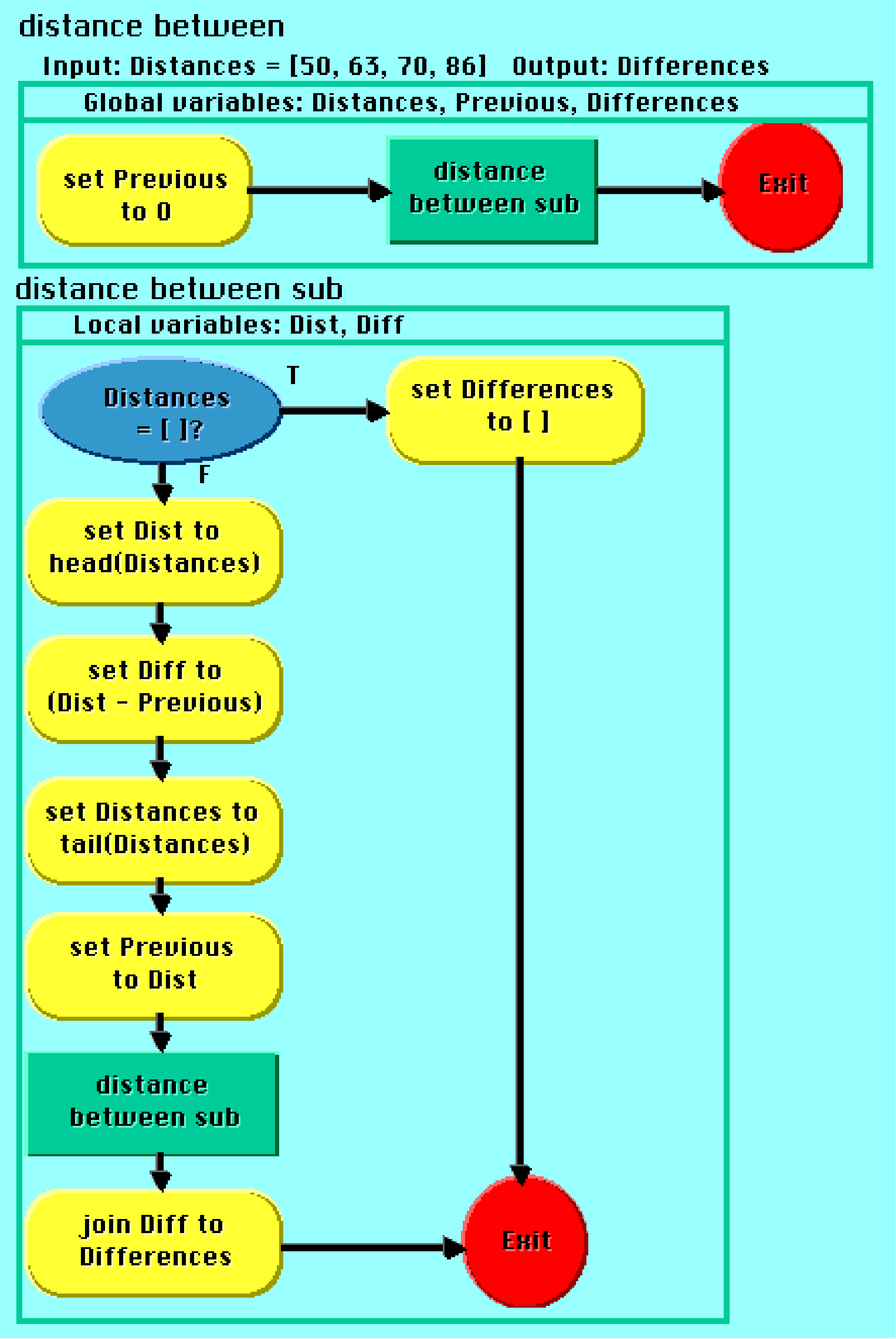

distance between
Input: Distances = [50, 63, 70, 86] Output: Differences
Global variables: Distances, Previous, Differences
set Previous to 0
distance between sub
Exit
distance between sub
Local variables: Dist, Diff
Distances = [ ]?
T
F
set Differences to [ ]
set Dist to head(Distances)
set Diff to (Dist - Previous)
set Distances to tail(Distances)
set Previous to Dist
distance between sub
join Diff to Differences
Exit

Figure C.6: The `distance between` Program: Control Flow Version

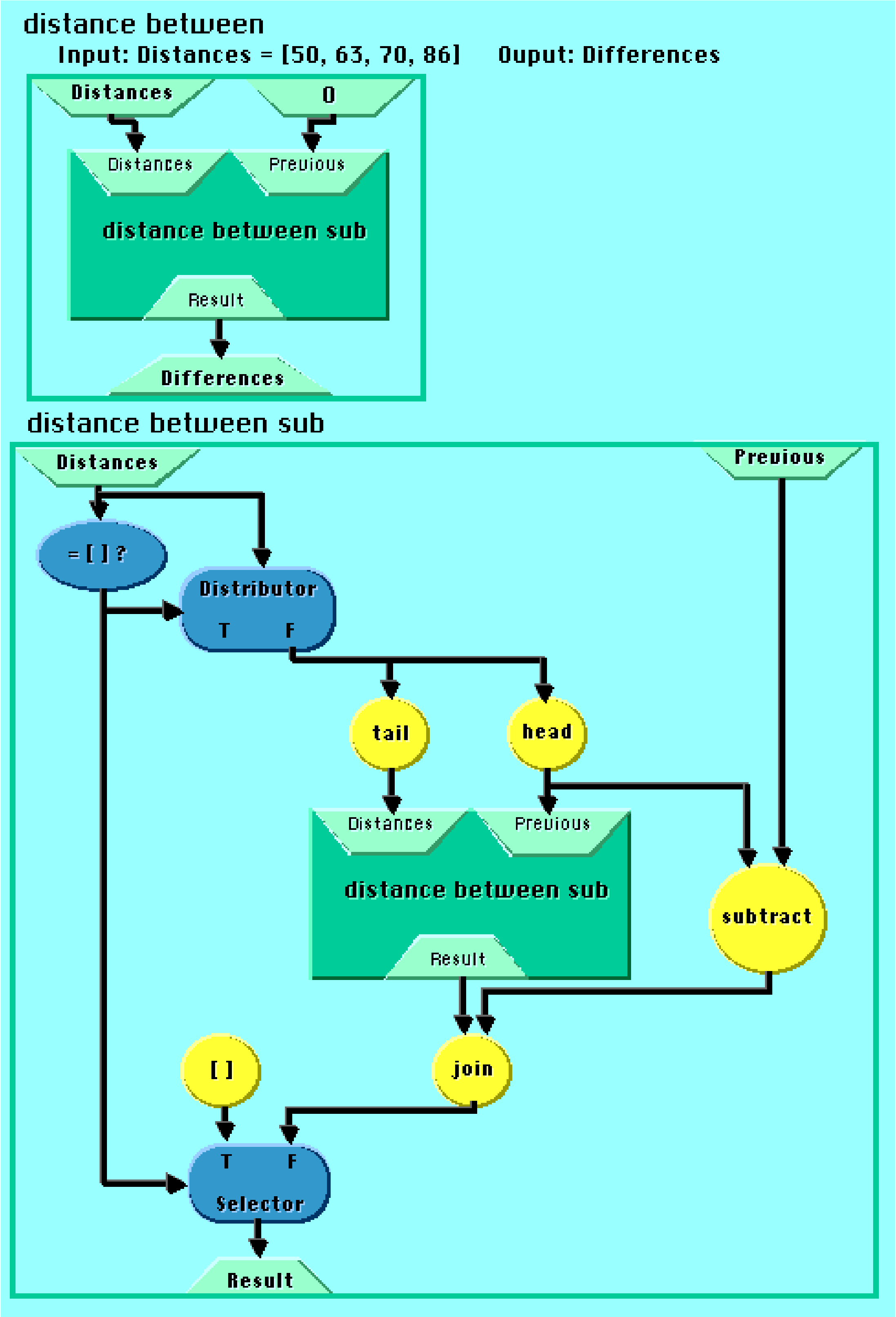


Figure C.7: The `distance between` Program: Data Flow Version

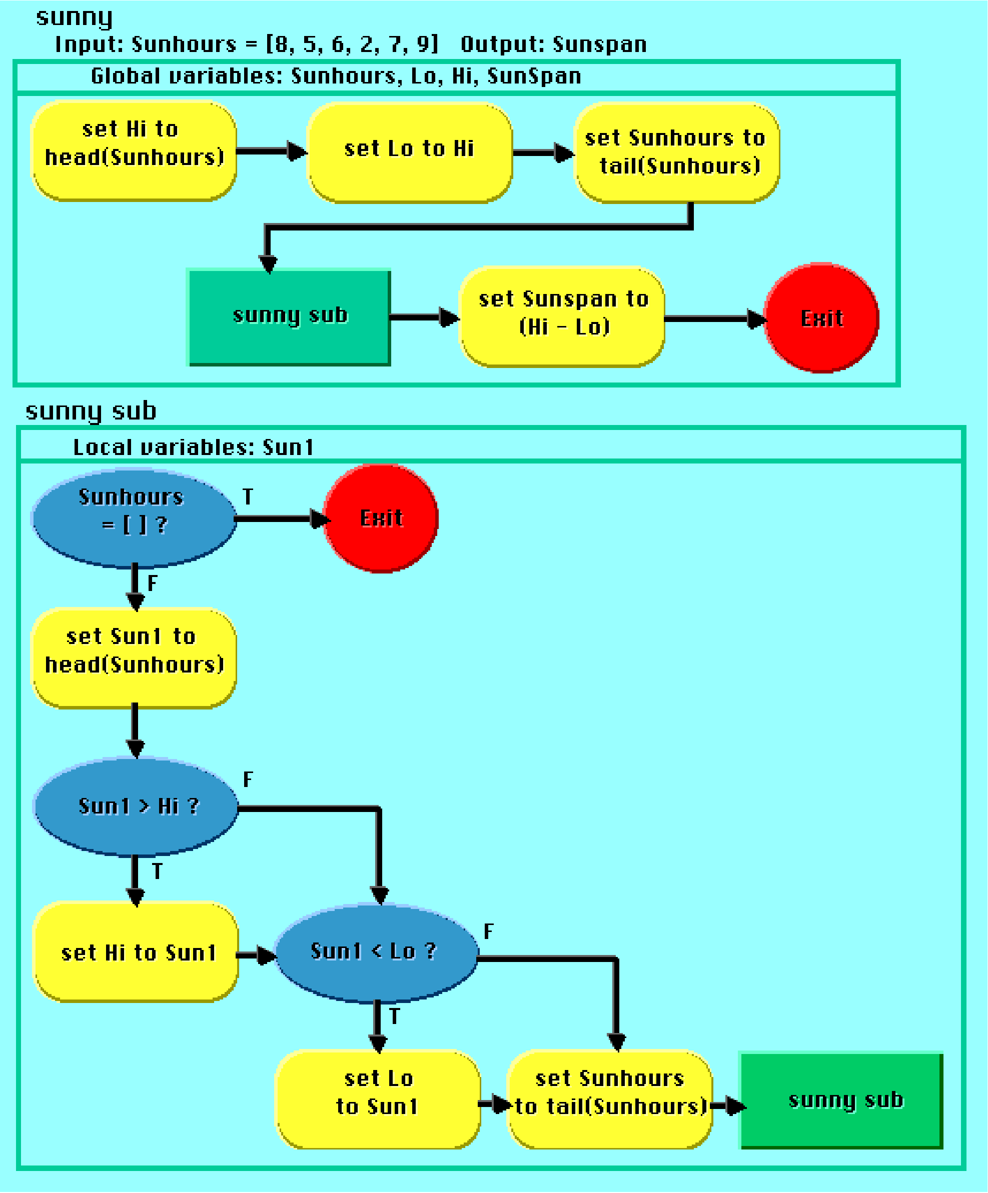


Figure C.8: The sunny Program: Control Flow Version

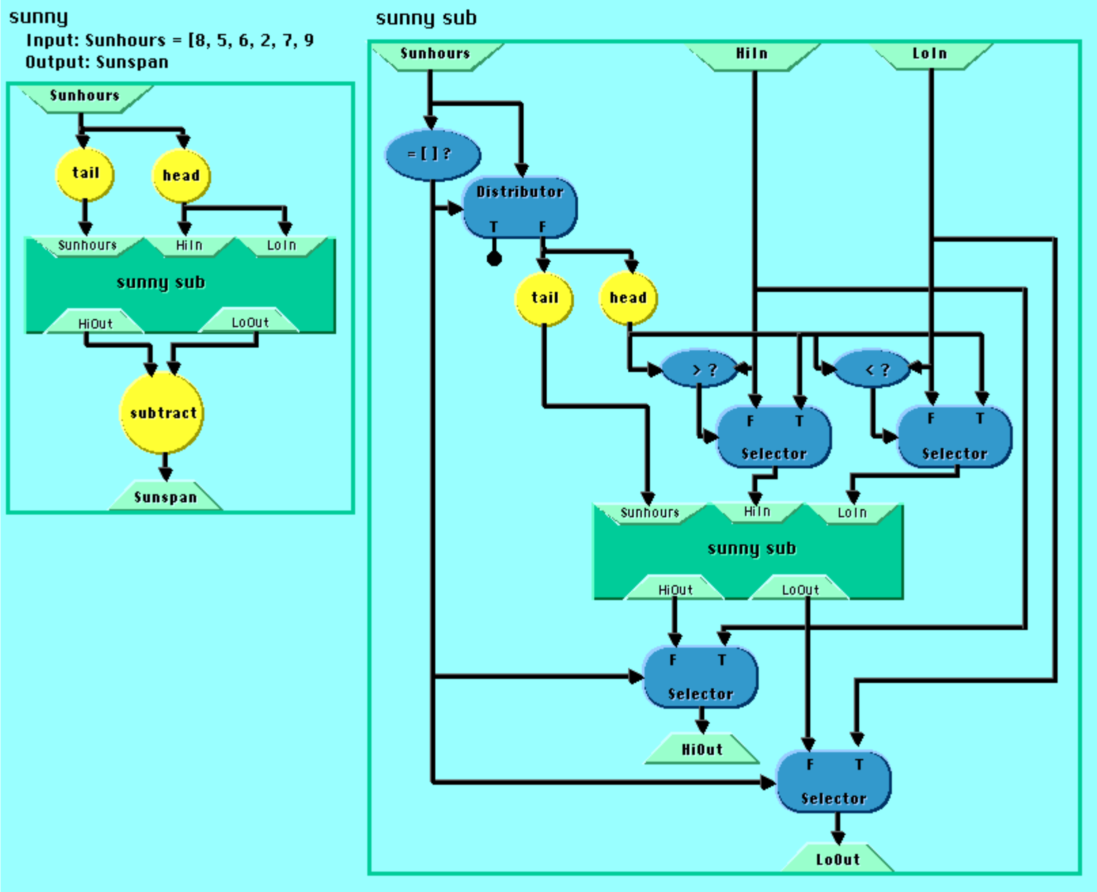


Figure C.9: The sunny Program: Data Flow Version

C.3 Comprehension Questions

BASKETBALL: FUNCTION

Choose the statement which best describes the goal the program seeks to achieve:

○ It produces the average height of the basketball players on the team

○ It assembles a team of up to five players by choosing only those who are over 180 cm

○ It excludes persons under a specified height

○ None of the above

BASKETBALL: CONTROL FLOW

If Counter $<$ 5, what happens next?

○ It depends on the value of Heights

○ Team gets set to [ ]

○ Height gets set to the head of Heights

○ None of the above

BASKETBALL: DATA FLOW

Given the example input to the program, is the head of Heights present in the Output of each recursive call?

○ Yes

○ No

○ Only if it is $>$ 180

○ None of the above

BASKETBALL: OPERATIONS

What test is performed on Counter?

○ Counter $=$ 5

○ Counter $=$ [ ]

○ Counter $+$ 1

○ None of the above

BASKETBALL: STATE

When Counter = 5, what is the value of Heights?

○ [ ]

○ [192]

○ [190, 145]

○ None of the above

DISTANCE BETWEEN: FUNCTION

Given a list of distances which represent, for example, the distance between a starting point and various cities along the route, does the program:

○ Calculate the distance between neighbouring cities

○ Work out the distance between the nearest and furthest cities

○ Find the shortest route between the starting point and the furthest city

○ None of the above

DISTANCE BETWEEN: CONTROL FLOW

Does distance between sub recurse over all of the elements of the list, or only some?

○ All

○ Some

○ It depends on the input

○ None of the above

DISTANCE BETWEEN: DATA FLOW

How does the value of Previous change at each recursive call?

○ It takes on the value of the head of the list in the previous recursive call

○ It has the same value as the current head of the list"

○ It doesn't, it remains the same throughout execution

○ None of the above

DISTANCE BETWEEN: OPERATIONS

What calculation requires the values of both Dist and Previous?

○ Equals

○ Subtract

○ Multiply

○ None of the above

DISTANCE BETWEEN: STATE

At the point when distance between sub is called with [70, 86], what will be the resulting value from the 'subtract' operation?

○ -7

○ 16

○ 7

○ None of the above

PASSES: FUNCTION

Choose the statement which best describes the goal the program seeks to achieve:

○ The program returns a list of marks > 65

○ The program compares pairs of student marks

○ The program counts the number of exam passes

○ None of the above

PASSES: CONTROL FLOW

Is Pass calculated before or after the recursive call?

○ Pass isn't calculated, it is just passed through

○ Before

○ After

○ None of the above

PASSES: DATA FLOW

Is Pass used in any tests?

○ Yes, the > test

○ Yes, the = test

○ No

○ None of the above

PASSES: OPERATIONS

What test is performed on Mark?

○ Mark < 65?

○ Mark = 65?

○ Mark = [ ]?

○ None of the above

PASSES: STATE

When Pass = 3, what is the value of Marklist?

○ [ ]

○ Given the input data, Pass never equals 3

○ [80]

○ None of the above

SUNNY: FUNCTION

Choose the statement which best describes the goal the program seeks to achieve:

○ The program calculates the average amount of sun per day

○ The program works out the difference between the sunniest and least sunny days

○ The program finds the day on which there was the most sun and the day on which there was the least sun

○ None of the above

SUNNY: CONTROL FLOW

Does sunny sub recurse over every element of its input list (Sunhours)?

○ No, it doesn't get to the end of Sunhours

○ No, it recurses over every other element of Sunhours

○ It depends on the input

○ None of the above

SUNNY: DATA FLOW

In Sunny, where do the values of Hi and Lo come from?

○ One from the head of Sunhours and one from the tail of Sunhours

○ Both come from the head of Sunhours

○ The first two elements of Sunhours

○ None of the above

SUNNY: OPERATIONS

What calculation is performed on Hi and Lo?

○ Multiplication

○ Addition

○ Subtraction

○ None of the above

SUNNY: STATE

At the time when sunny sub is called with Sunhours = [7,9], what is the value of Hi?

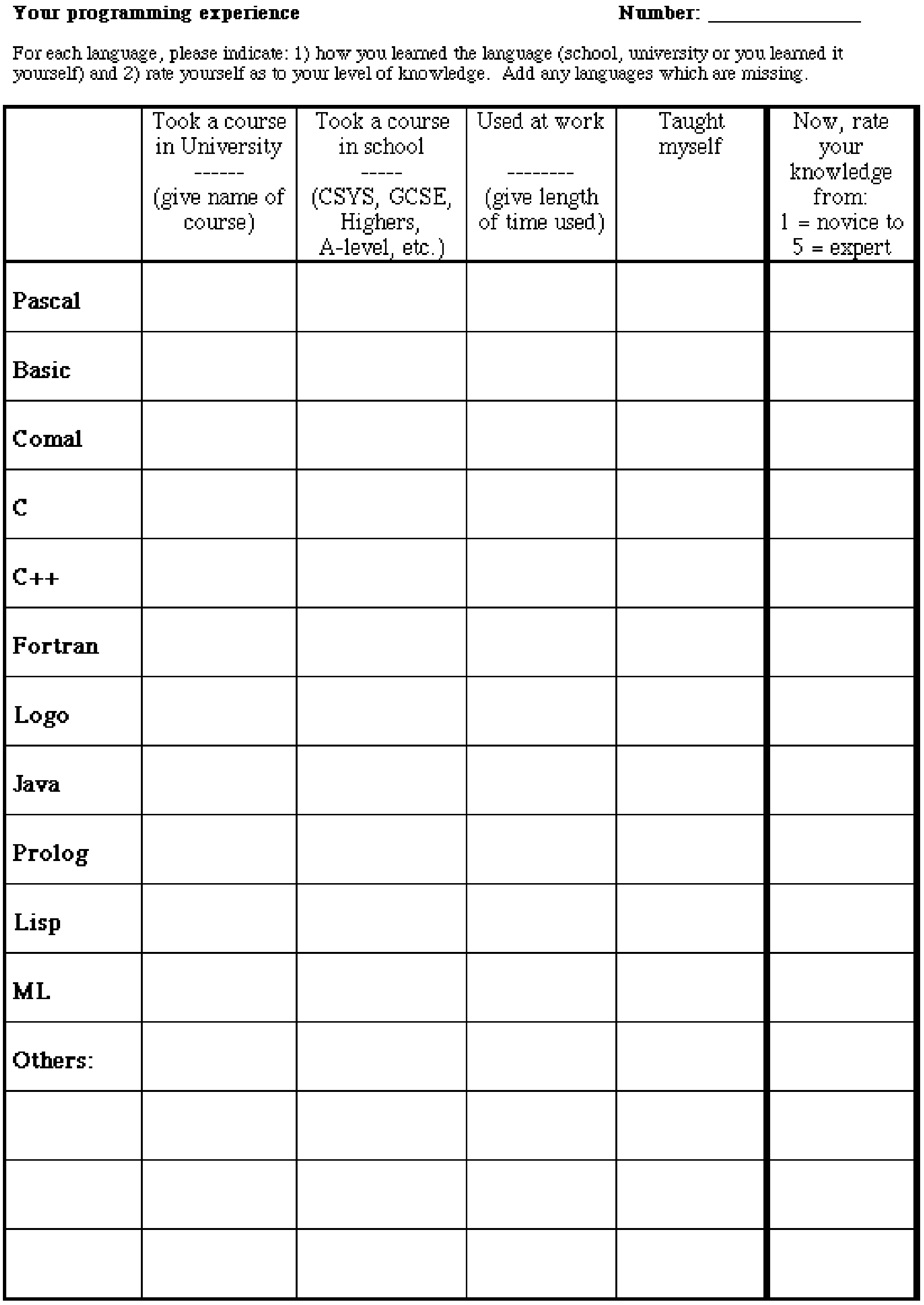
○ 8

○ 9

○ It doesn't have a value yet

○ None of the above

C.4 The Programming Self-Report Questionnaire



Your programming experience **Number:** ______________

For each language, please indicate: 1) how you learned the language (school, university or you learned it yourself) and 2) rate yourself as to your level of knowledge. Add any languages which are missing.

	Took a course in University ------ (give name of course)	Took a course in school ----- (CSYS, GCSE, Highers, A-level, etc.)	Used at work -------- (give length of time used)	Taught myself	Now, rate your knowledge from: 1 = novice to 5 = expert
Pascal					
Basic					
Comal					
C					
C++					
Fortran					
Logo					
Java					
Prolog					
Lisp					
ML					
Others:					

Appendix D

Program Summary Analysis: Coding Manuals

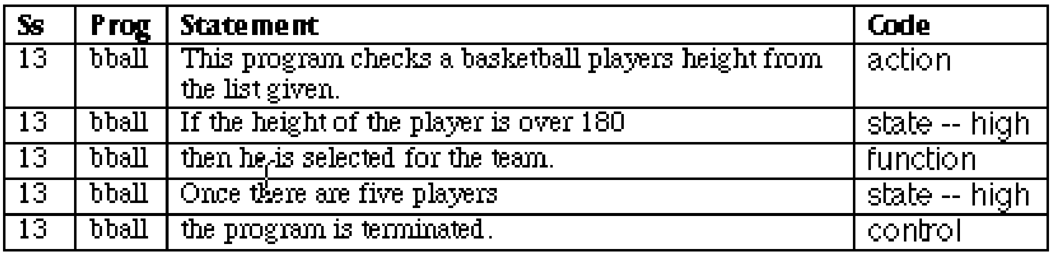
Information Types Coding Manual

The following sections describe how to code program summaries in terms of information types. It is assumed that the coder has read through a description of the information types coding scheme (see Chapter 8) and is familiar with the general distinctions between the categories.

Furthermore, it is assumed that the coding tools described in Chapter 8 will be used, therefore, some of the steps below describe how to prepare tha data in the format required by the coding tools.

The Coding Process

Before coding, the program summary should be segmented, dividing the summary up into short phrases consisting of a subject and a predicate (either of which may be implied). The segment to be coded should be placed in the third column of a table: the code itself will be inserted in the fourth column, as in Figure D.1 (although these parameters can obviously be changed).



Ss	Prog	Statement	Code
13	bball	This program checks a basketball players height from the list given.	action
13	bball	If the height of the player is over 180	state -- high
13	bball	then he is selected for the team.	function
13	bball	Once there are five players	state -- high
13	bball	the program is terminated.	control

Figure D.1: Information Types Coding

Coding can either be carried out sequentially (line by line), or by category, *i.e.* by several passes through the code in order to identify all segments of a particular type. The latter probably guarantees higher reliability, given that content based coding is very demanding, and switching between category definitions increases this load, however, it is time-consuming. I personally feel that the category based method makes it easier to ensure consistency within categories, but there is no reason that the step by step method cannot be used. The important thing is that the end result is as consistent as possible.

Sequential coding is simply a process of going through the summaries, line by line, and choosing the most appropriate code for each line, based on the decision process shown in Figure D.4. The code can be inserted into the table using the appropriate button on the coding panel.

Category based coding is carried out by going through the text and looking for instances of the category. It may be easiest to start with the most straightforward categories to code and, for each category, look for instances of that category, and use the coding button panel to label each case.

The following representation shows a decision process for discriminating between each category.

Examples of each type of category were provided with the category definitions in Chapter 8: further examples are not provided as Carletta[1] suggests that coding schemes based on the provision of extensive examples run the risk that coders code only those instances which are actually provided as examples.

[1] personal communication

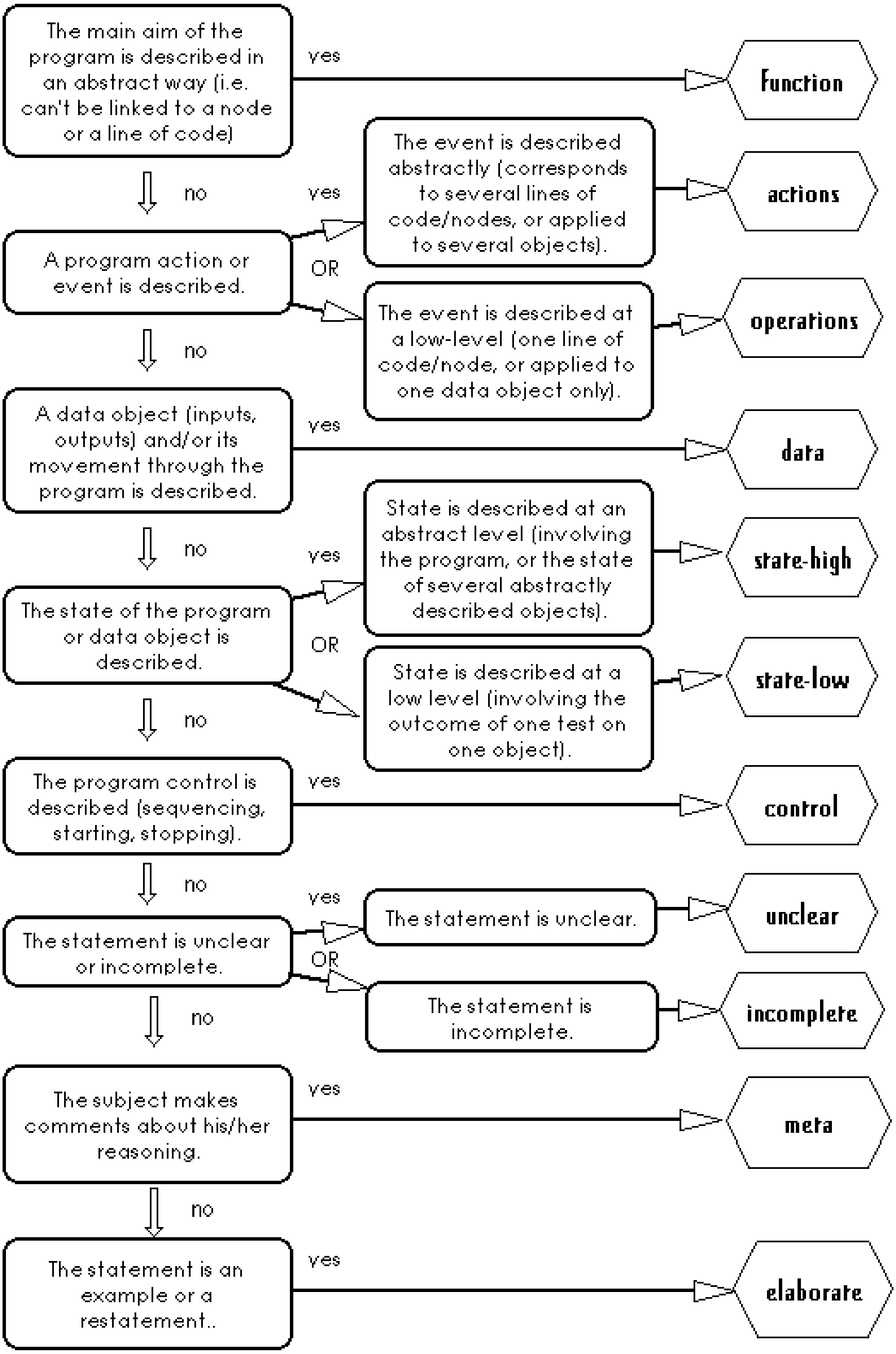

The main aim of the program is described in an abstract way (i.e. can't be linked to a node or a line of code)
yes
function
no
A program action or event is described.
yes
The event is described abstractly (corresponds to several lines of code/nodes, or applied to several objects).
actions
OR
The event is described at a low-level (one line of code/node, or applied to one data object only).
operations
no
A data object (inputs, outputs) and/or its movement through the program is described.
yes
data
no
The state of the program or data object is described.
yes
State is described at an abstract level (involving the program, or the state of several abstractly described objects).
state-high
OR
State is described at a low level (involving the outcome of one test on one object).
state-low
no
The program control is described (sequencing, starting, stopping).
yes
control
no
The statement is unclear or incomplete.
yes
The statement is unclear.
unclear
OR
The statement is incomplete.
incomplete
no
The subject makes comments about his/her reasoning.
yes
meta
no
The statement is an example or a restatement..
yes
elaborate

Figure D.2: Decision Process for Information Types Coding

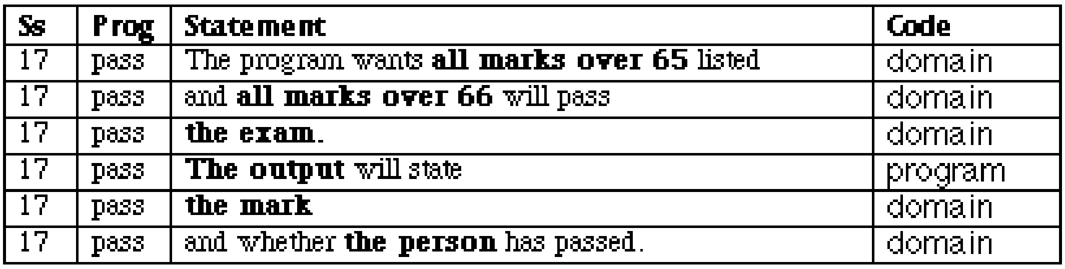
Object Description Coding Manual

The following sections describe how to code program summaries in terms of object descriptions. It is assumed that the coder has read through a description of the object descriptions coding scheme (see Chapter 8) and is familiar with the general distinctions between the categories.

Furthermore, as with the information types scheme described above, it is assumed that the coding tools described in Chapter 8 will be used, therefore, some of the steps below describe how to prepare tha data in the format required by the coding tools.

The Coding Process

Before coding, the program summary should be segmented, dividing the summary up in such a way that there is one *data object* description per segment. Segmenting should not occur based on any other objects (*e.g.* the program, actions/events within the program, such a recursive call, iteration). It is helpful to highlight the data object in the segment in some way so as to distinguish it from other objects, but this is not necessary. The segment to be coded should be in the third column of a table: the code itself will be inserted in the fourth column, as in Figure D.3 (although these parameters can obviously be changed).



Ss	Prog	Statement	Code
17	pass	The program wants **all marks over 65** listed	domain
17	pass	and **all marks over 66** will pass	domain
17	pass	**the exam.**	domain
17	pass	**The output** will state	program
17	pass	**the mark**	domain
17	pass	and whether **the person** has passed.	domain

Figure D.3: Object Coding

Coding can either be carried out sequentially (line by line), or by category, *i.e.* by several passes through the code in order to identify all segments of a particular type. The latter probably guarantees higher reliability, given that content based coding is very demanding, and switching between category definitions increases this load, however, it is time-consuming. I personally feel that the category based method makes it easier to ensure consistency within categories, but there is no reason that the step by step method cannot be used. The important thing is that the end result is as consistent as possible.

Sequential coding is simply a process of going through the summaries, line by line, and choosing the most appropriate code for each line, based on the decision process shown in Figure D.4. The code can be inserted into the table using the appropriate button on the coding panel.

Category based coding is carried out by going through the text and looking for instances of the category. It may be easiest to start with the most straightforward categories to code. In this case, these tend to be *indirect* and *program only* (as it is easy to draw up a list of objects which only occur within the program: for the programs described here, the only 'program only' object was the counter). Then, for each category, look for instances of that category, and use the coding button panel to label each case.

The following representation shows a decision process for discriminating between each category.

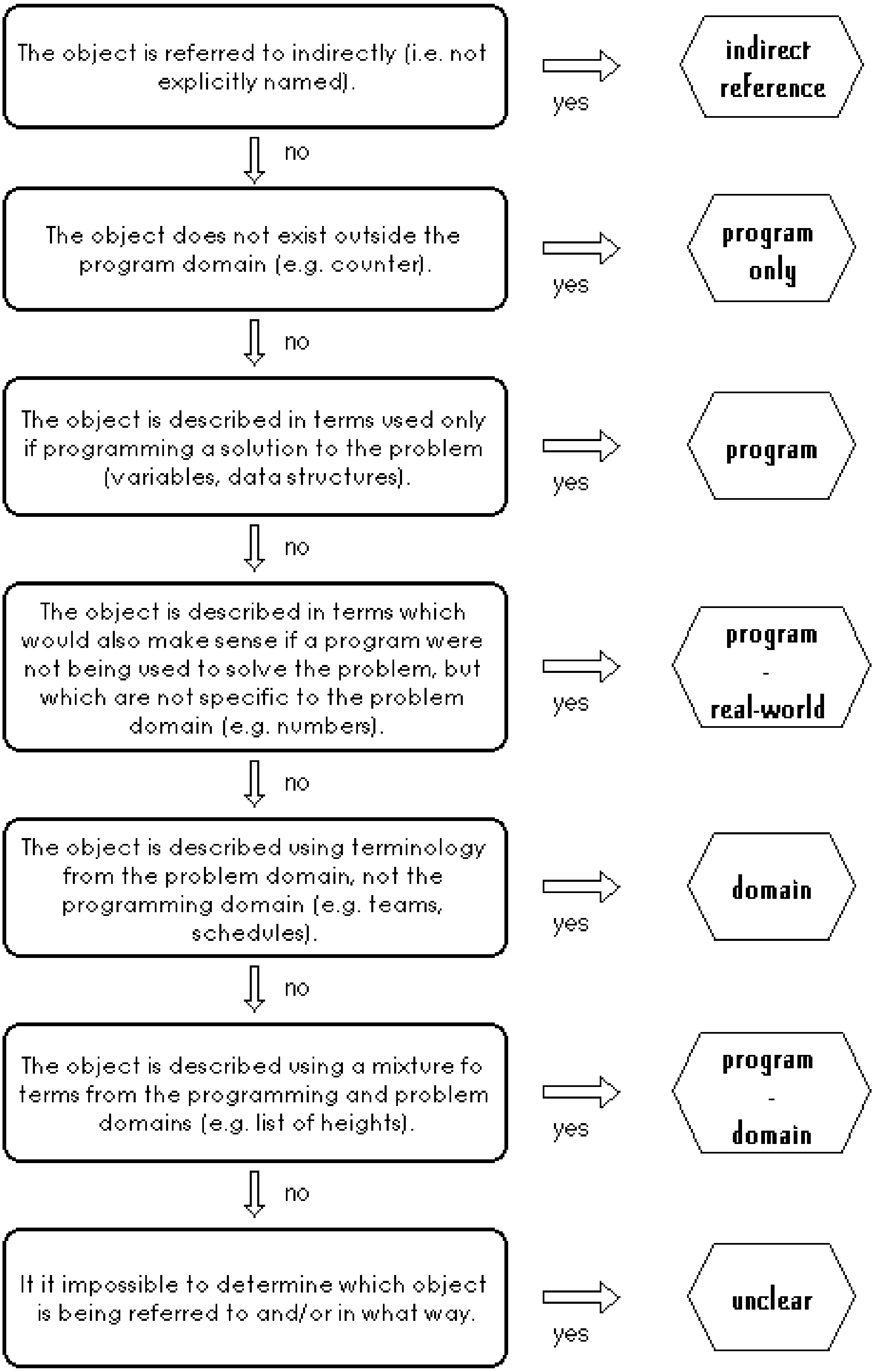

The object is referred to indirectly (i.e. not explicitly named).
yes
indirect reference
no
The object does not exist outside the program domain (e.g. counter).
yes
program only
no
The object is described in terms used only if programming a solution to the problem (variables, data structures).
yes
program
no
The object is described in terms which would also make sense if a program were not being used to solve the problem, but which are not specific to the problem domain (e.g. numbers).
yes
program - real-world
no
The object is described using terminology from the problem domain, not the programming domain (e.g. teams, schedules).
yes
domain
no
The object is described using a mixture fo terms from the programming and problem domains (e.g. list of heights).
yes
program - domain
no
It it impossible to determine which object is being referred to and/or in what way.
yes
unclear

Figure D.4: Decision Process for Object Coding

Appendix E

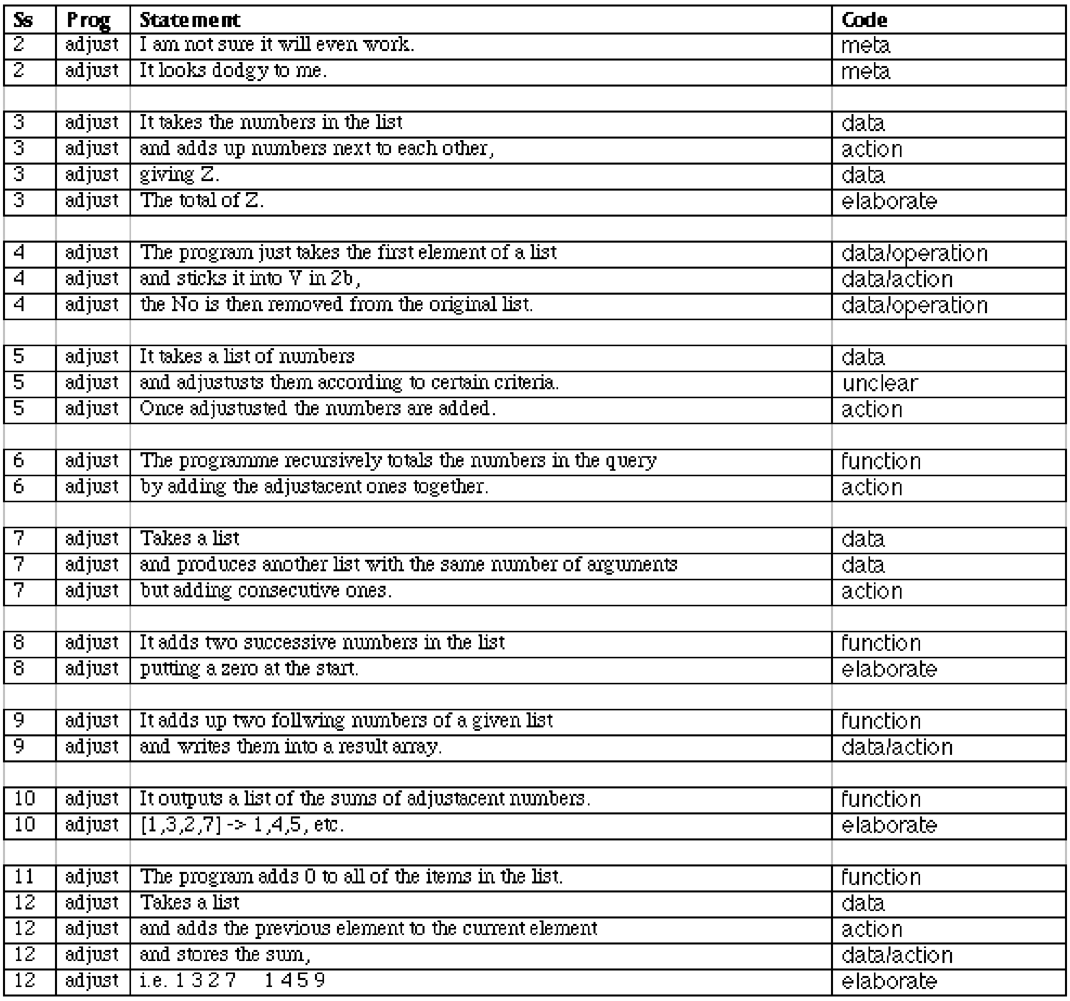
Coded Transcripts

This appendix provides further coding examples using the schemes described in Chapter 8. The colour has been removed from the coding, as it is primarily used for obtaining an overview of a summary and/or comparing summaries.

The following examples are shown:

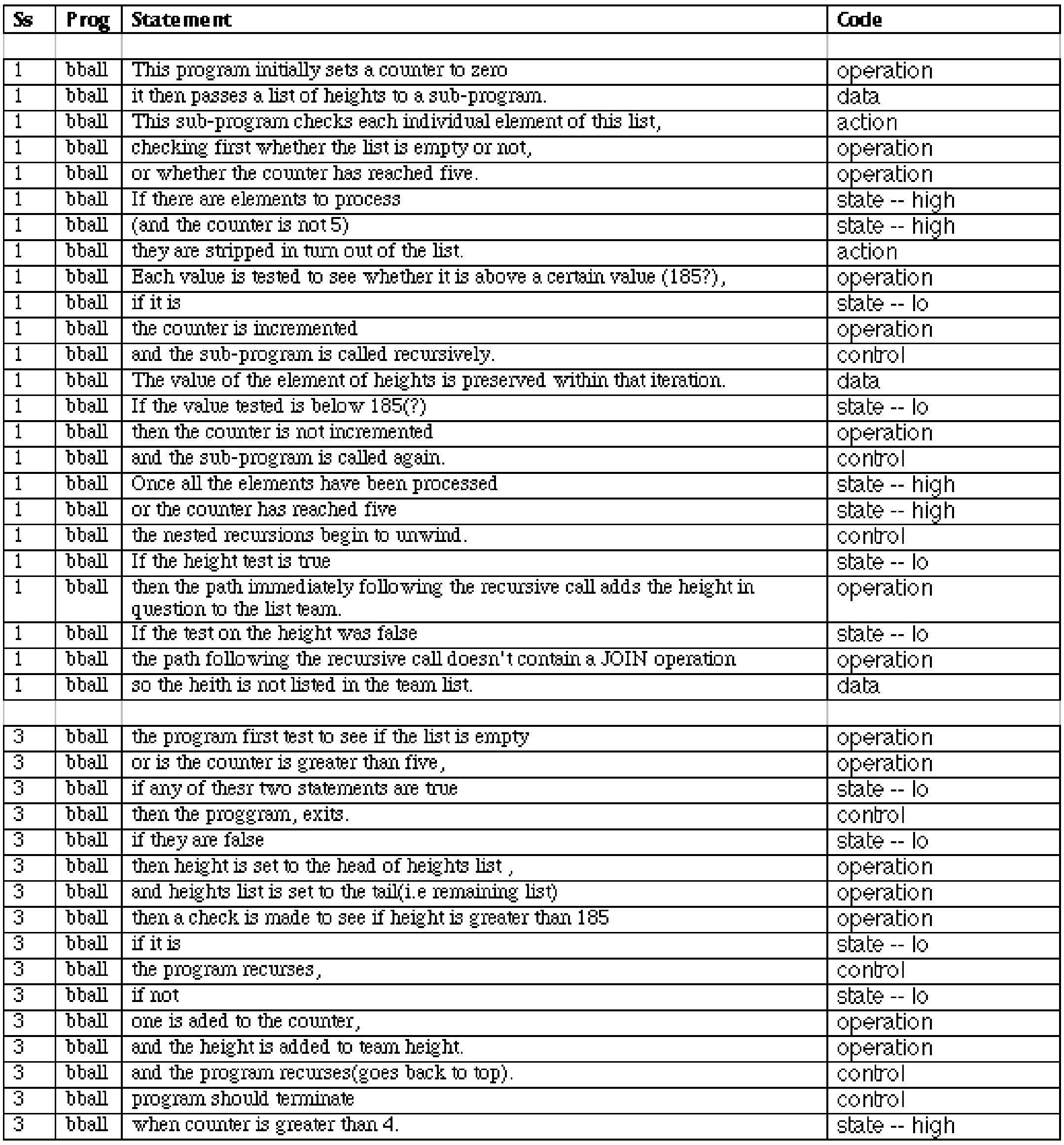
- Information Types Coding:
  - Prolog Experiment;
  - VPL Experiment: Control Flow Group;
  - VPL Experiment: Data Flow Group.
- Object Descriptions Coding:
  - VPL Experiment: Control Flow Group;
  - VPL Experiment: Data Flow Group.

The summaries from the Prolog experiment were based on the `adjust` program (shown in Appendix A), while the summaries from the VPL experiment are based on the `basketball` program (shown in Appendix C).



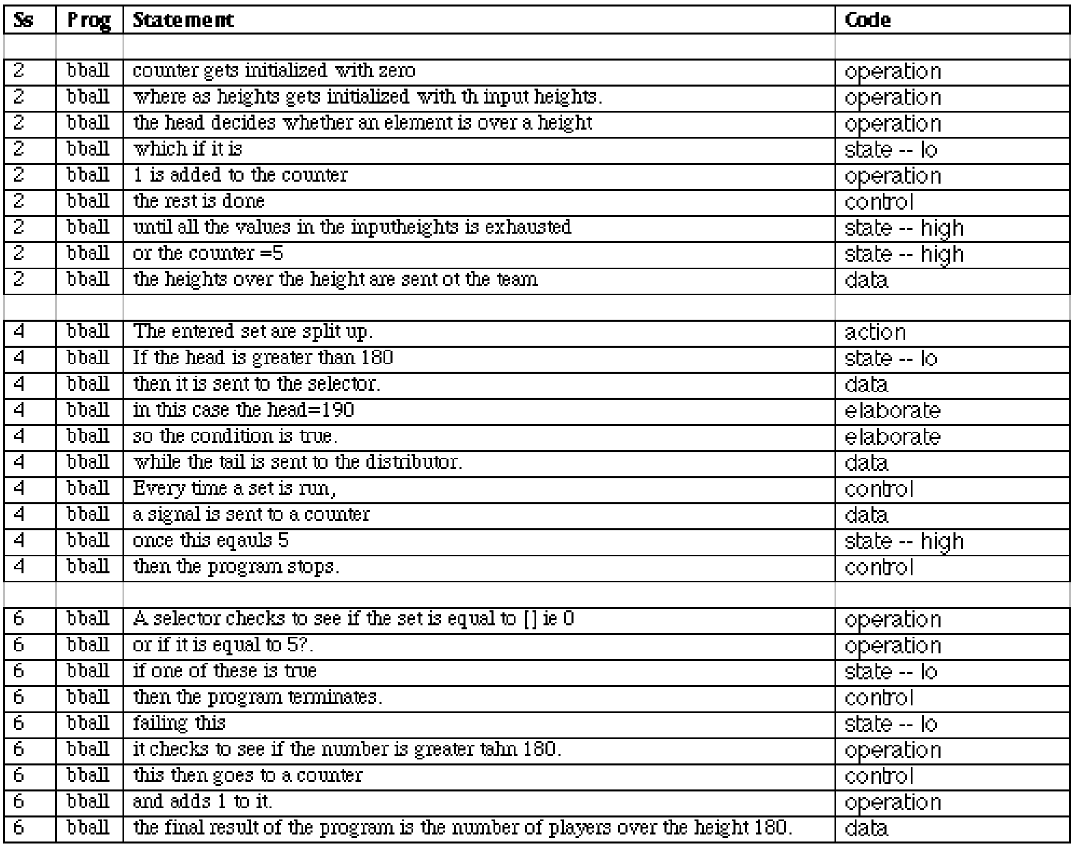
Ss	Prog	Statement	Code
2	adjust	I am not sure it will even work.	meta
2	adjust	It looks dodgy to me.	meta
3	adjust	It takes the numbers in the list	data
3	adjust	and adds up numbers next to each other,	action
3	adjust	giving Z.	data
3	adjust	The total of Z.	elaborate
4	adjust	The program just takes the first element of a list	data/operation
4	adjust	and sticks it into V in 2b,	data/action
4	adjust	the No is then removed from the original list.	data/operation
5	adjust	It takes a list of numbers	data
5	adjust	and adjustusts them according to certain criteria.	unclear
5	adjust	Once adjustusted the numbers are added.	action
6	adjust	The programme recursively totals the numbers in the query	function
6	adjust	by adding the adjustacent ones together.	action
7	adjust	Takes a list	data
7	adjust	and produces another list with the same number of arguments	data
7	adjust	but adding consecutive ones.	action
8	adjust	It adds two successive numbers in the list	function
8	adjust	putting a zero at the start.	elaborate
9	adjust	It adds up two follwing numbers of a given list	function
9	adjust	and writes them into a result array.	data/action
10	adjust	It outputs a list of the sums of adjustacent numbers.	function
10	adjust	[1,3,2,7] -> 1,4,5, etc.	elaborate
11	adjust	The program adds 0 to all of the items in the list.	function
12	adjust	Takes a list	data
12	adjust	and adds the previous element to the current element	action
12	adjust	and stores the sum,	data/action
12	adjust	i.e. 1 3 2 7 1 4 5 9	elaborate

Figure E.1: Information Types Coding: Prolog Experiment



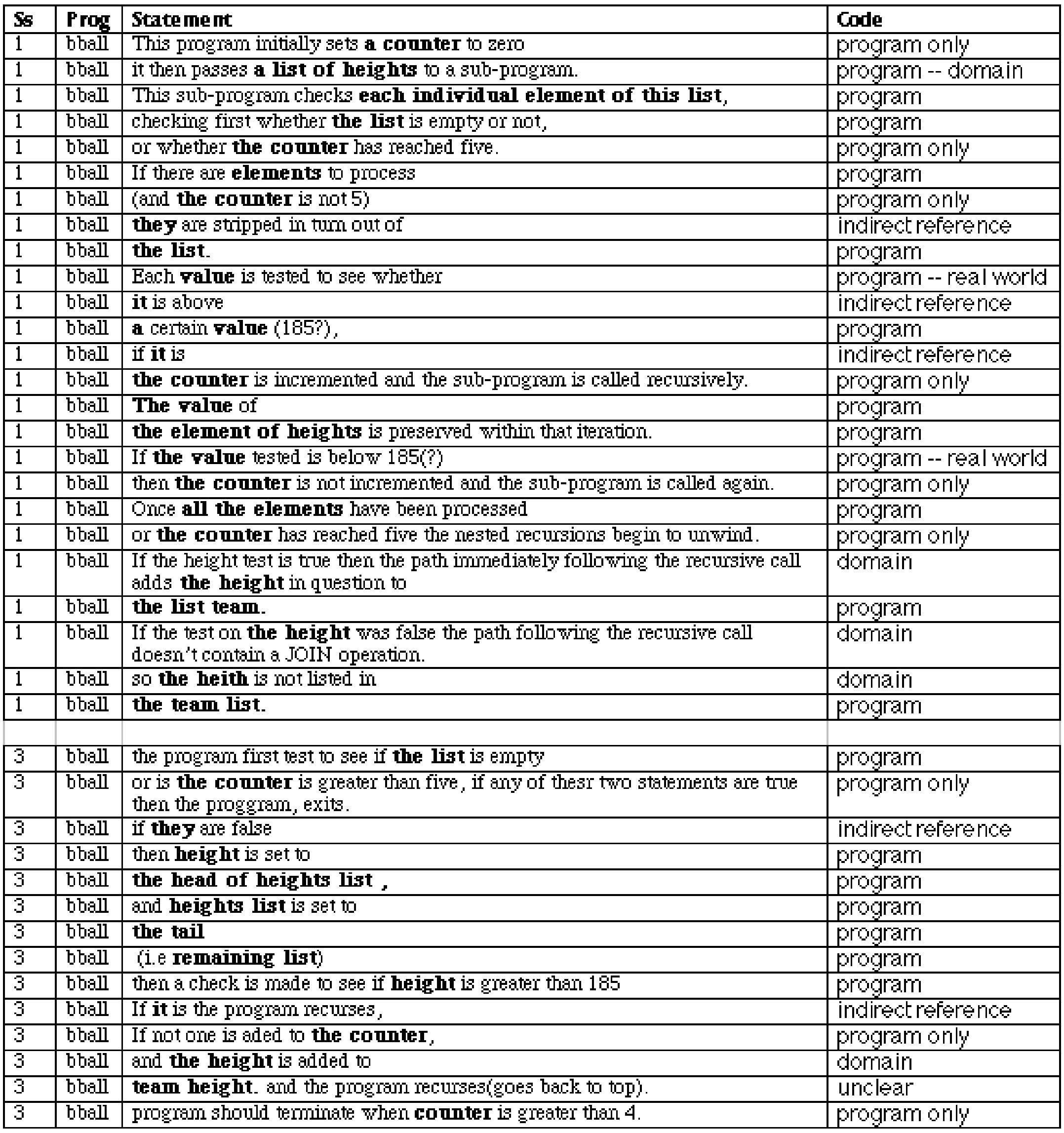
Ss	Prog	Statement	Code
1	bball	This program initially sets a counter to zero	operation
1	bball	it then passes a list of heights to a sub-program.	data
1	bball	This sub-program checks each individual element of this list,	action
1	bball	checking first whether the list is empty or not,	operation
1	bball	or whether the counter has reached five.	operation
1	bball	If there are elements to process	state -- high
1	bball	(and the counter is not 5)	state -- high
1	bball	they are stripped in turn out of the list.	action
1	bball	Each value is tested to see whether it is above a certain value (185?),	operation
1	bball	if it is	state -- lo
1	bball	the counter is incremented	operation
1	bball	and the sub-program is called recursively.	control
1	bball	The value of the element of heights is preserved within that iteration.	data
1	bball	If the value tested is below 185(?)	state -- lo
1	bball	then the counter is not incremented	operation
1	bball	and the sub-program is called again.	control
1	bball	Once all the elements have been processed	state -- high
1	bball	or the counter has reached five	state -- high
1	bball	the nested recursions begin to unwind.	control
1	bball	If the height test is true	state -- lo
1	bball	then the path immediately following the recursive call adds the height in question to the list team.	operation
1	bball	If the test on the height was false	state -- lo
1	bball	the path following the recursive call doesn't contain a JOIN operation	operation
1	bball	so the heith is not listed in the team list.	data
3	bball	the program first test to see if the list is empty	operation
3	bball	or is the counter is greater than five,	operation
3	bball	if any of thesr two statements are true	state -- lo
3	bball	then the proggram, exits.	control
3	bball	if they are false	state -- lo
3	bball	then height is set to the head of heights list ,	operation
3	bball	and heights list is set to the tail(i.e remaining list)	operation
3	bball	then a check is made to see if height is greater than 185	operation
3	bball	if it is	state -- lo
3	bball	the program recurses,	control
3	bball	if not	state -- lo
3	bball	one is aded to the counter,	operation
3	bball	and the height is added to team height.	operation
3	bball	and the program recurses(goes back to top).	control
3	bball	program should terminate	control
3	bball	when counter is greater than 4.	state -- high

Figure E.2: Information Types Coding: VPL Experiment, Control Flow Group



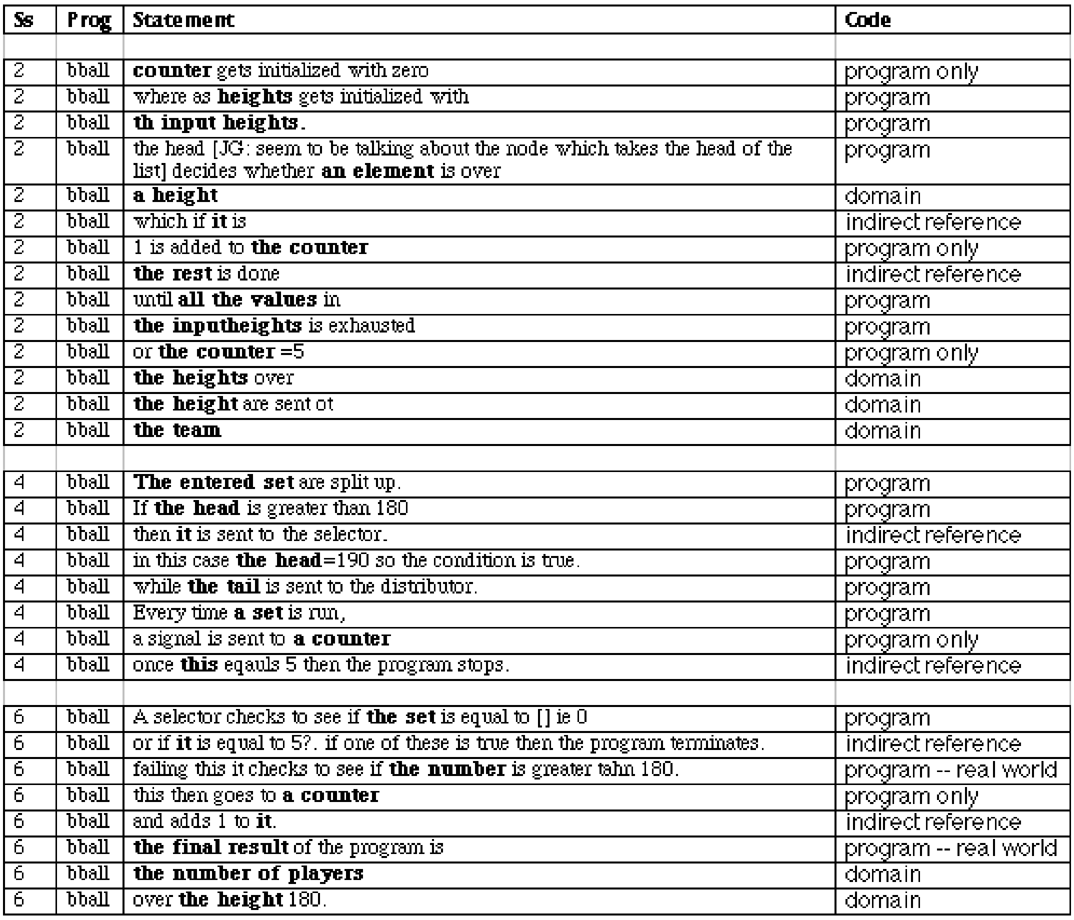
Ss	Prog	Statement	Code
2	bball	counter gets initialized with zero	operation
2	bball	where as heights gets initialized with th input heights.	operation
2	bball	the head decides whether an element is over a height	operation
2	bball	which if it is	state -- lo
2	bball	1 is added to the counter	operation
2	bball	the rest is done	control
2	bball	until all the values in the inputheights is exhausted	state -- high
2	bball	or the counter =5	state -- high
2	bball	the heights over the height are sent ot the team	data
4	bball	The entered set are split up.	action
4	bball	If the head is greater than 180	state -- lo
4	bball	then it is sent to the selector.	data
4	bball	in this case the head=190	elaborate
4	bball	so the condition is true.	elaborate
4	bball	while the tail is sent to the distributor.	data
4	bball	Every time a set is run,	control
4	bball	a signal is sent to a counter	data
4	bball	once this eqauls 5	state -- high
4	bball	then the program stops.	control
6	bball	A selector checks to see if the set is equal to [] ie 0	operation
6	bball	or if it is equal to 5?.	operation
6	bball	if one of these is true	state -- lo
6	bball	then the program terminates.	control
6	bball	failing this	state -- lo
6	bball	it checks to see if the number is greater tahn 180.	operation
6	bball	this then goes to a counter	control
6	bball	and adds 1 to it.	operation
6	bball	the final result of the program is the number of players over the height 180.	data

Figure E.3: Information Types Coding: VPL Experiment, Data Flow Group



Ss	Prog	Statement	Code
1	bball	This program initially sets **a counter** to zero	program only
1	bball	it then passes **a list of heights** to a sub-program.	program -- domain
1	bball	This sub-program checks **each individual element of this list**,	program
1	bball	checking first whether **the list** is empty or not,	program
1	bball	or whether **the counter** has reached five.	program only
1	bball	If there are **elements** to process	program
1	bball	(and **the counter** is not 5)	program only
1	bball	**they** are stripped in turn out of	indirect reference
1	bball	**the list**.	program
1	bball	Each **value** is tested to see whether	program -- real world
1	bball	**it** is above	indirect reference
1	bball	**a** certain **value** (185?),	program
1	bball	if **it** is	indirect reference
1	bball	**the counter** is incremented and the sub-program is called recursively.	program only
1	bball	**The value** of	program
1	bball	**the element of heights** is preserved within that iteration.	program
1	bball	If **the value** tested is below 185(?)	program -- real world
1	bball	then **the counter** is not incremented and the sub-program is called again.	program only
1	bball	Once **all the elements** have been processed	program
1	bball	or **the counter** has reached five the nested recursions begin to unwind.	program only
1	bball	If the height test is true then the path immediately following the recursive call adds **the height** in question to	domain
1	bball	**the list team**.	program
1	bball	If the test on **the height** was false the path following the recursive call doesn't contain a JOIN operation.	domain
1	bball	so **the heith** is not listed in	domain
1	bball	**the team list**.	program
3	bball	the program first test to see if **the list** is empty	program
3	bball	or is **the counter** is greater than five, if any of thesr two statements are true then the proggram, exits.	program only
3	bball	if **they** are false	indirect reference
3	bball	then **height** is set to	program
3	bball	**the head of heights list ,**	program
3	bball	and **heights list** is set to	program
3	bball	**the tail**	program
3	bball	(i.e **remaining list**)	program
3	bball	then a check is made to see if **height** is greater than 185	program
3	bball	If **it** is the program recurses,	indirect reference
3	bball	If not one is aded to **the counter**,	program only
3	bball	and **the height** is added to	domain
3	bball	**team height**. and the program recurses(goes back to top).	unclear
3	bball	program should terminate when **counter** is greater than 4.	program only

Figure E.4: Object Description Coding: VPL Experiment, Control Flow Group



Ss	Prog	Statement	Code
2	bball	**counter** gets initialized with zero	program only
2	bball	where as **heights** gets initialized with	program
2	bball	**th input heights.**	program
2	bball	the head [JG: seem to be talking about the node which takes the head of the list] decides whether **an element** is over	program
2	bball	**a height**	domain
2	bball	which if **it** is	indirect reference
2	bball	1 is added to **the counter**	program only
2	bball	**the rest** is done	indirect reference
2	bball	until **all the values** in	program
2	bball	**the inputheights** is exhausted	program
2	bball	or **the counter** =5	program only
2	bball	**the heights** over	domain
2	bball	**the height** are sent ot	domain
2	bball	**the team**	domain
4	bball	**The entered set** are split up.	program
4	bball	If **the head** is greater than 180	program
4	bball	then **it** is sent to the selector.	indirect reference
4	bball	in this case **the head**=190 so the condition is true.	program
4	bball	while **the tail** is sent to the distributor.	program
4	bball	Every time **a set** is run,	program
4	bball	a signal is sent to **a counter**	program only
4	bball	once **this** eqauls 5 then the program stops.	indirect reference
6	bball	A selector checks to see if **the set** is equal to [] ie 0	program
6	bball	or if **it** is equal to 5?. if one of these is true then the program terminates.	indirect reference
6	bball	failing this it checks to see if **the number** is greater tahn 180.	program -- real world
6	bball	this then goes to **a counter**	program only
6	bball	and adds 1 to **it**.	indirect reference
6	bball	**the final result** of the program is	program -- real world
6	bball	**the number of players**	domain
6	bball	over **the height** 180.	domain

Figure E.5: Object Description Coding: VPL Experiment, Data Flow Group