# A Support Environment for the Teaching of Programming

by

Rosanne Stewart

Submitted in fulfilment of the academic requirements
for the degree of Master of Science
in the Department of Computer Science and Information Systems,
University of Natal

Pietermaritzburg
1996

# ABSTRACT

This thesis examines the effectiveness of a specially constructed computer based support environment for the teaching of computer programming to novice programmers. In order to achieve this, the following distinct activities were pursued. Firstly, an in-depth investigation of programming misconceptions and techniques used for overcoming them was carried out. Secondly, the educational principles gained from this investigation were used to design and implement a computer based environment to support novice programmers learning the Pascal language. Finally, several statistical methods were used to compare students who made use of the support environment to those who did not and the results are discussed.

# PREFACE

The experimental work described in this dissertation was conducted in the Department of Computer Science and Information Systems, University of Natal, Pietermaritzburg, under the supervision of Professor Vevek Ram.

These studies represent original work by the author and have otherwise not been submitted in any form for any degree or diploma to any University. Where use has been made of the work of others it is duly acknowledged in the text.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

## ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Professor Vevek Ram whose guidance in the preparation of this dissertation was most valuable. In addition I would like to thank Mrs. Sue Brittain, of the Department of Statistics and Biometry, who assisted me with the statistical analysis included in this thesis. My thanks also to the students who participated in this study.

My parents have provided constant support and for this I am most grateful.

My brother, Peter assisted with the proof reading and Leonard Els provided his technical expertise. I appreciate their help.

Finally, Mr. Christopher Scogings, a former acting-head of the department of Computer Science, introduced me to programming. I owe my love of programming to him.

This work was supported in part by the Foundation for Research Development (FRD)

# 1. INTRODUCTION

## 1.1 Introduction

When Elliot Soloway, an associate Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan, asked "Should We Teach Students to Program? " the reply from several leading programming instructors was an emphatic "yes"[1]. Although there was much disagreement about the nature of the programming instruction and programming environment, all agreed that in the future, programming is not only going to be taught to Computer Scientists, but that more and more people are going to require programming instruction to assist in their chosen career (Soloway, 1993). For this and other reasons, programming has been included in school syllabi at varying levels and is studied at university level by students from almost all faculties. However, regardless of one's age or specific field of interest or expertise, learning to program can be difficult.

The primary objective of this research is to examine how novice programmers come to understand fundamental programming concepts, and to examine the potential of one particular approach to aid novice programmers' understanding of these fundamental concepts. The approach examined in this research is the design and implementation of a computer based support environment, Patman. Patman was developed with the intention of reducing student misconceptions through the use of specially designed programs and the glass box approach. It differs from other systems that have implemented the glass box approach[2], in that the design process emphasised the pedagogical aspects, rather than the interface characteristics. The design of Patman, was influenced by misconceptions documented by other researchers and those found in the course of this study. The example problems, included in the system, were designed with the intention of addressing these misconceptions.

[1] The instructors who replied to Soloway's question included M. Clancy, M. Linn, A. DiSessa and others.

[2] See sections 3.4 and 4.2.

## 1.2 Research Objectives

The main objective of this research is

> the evaluation of the effectiveness of a support environment in reducing students' misconceptions and assisting in the acquisition of general programming knowledge.

This involved the pursuit of two sub-objectives, namely:

1. the investigation of novice programmer's problems and misconceptions
2. the development of a support environment to alleviate these problems and misconceptions.

This research was motivated by the desire to address the following problems which hamper the teaching of programming to novice programmers:

- the varying ability of the enrolled students
- the prevalence of misconceptions, and
- lecturer to student ratios that are not optimal.

These factors are discussed in turn.

Several factors contribute to one's ability to learn a programming language. Broadly, these can be broken up into two groups: background and psychological factors. Background characteristics include previous access to computers and previous programming experience while psychological factors are those factors that are unique to an individual, such as their motivation level or their desire to learn to program. Previous studies have found that background factors are most influential in determining one's ability to program under conventional methods of teaching. This poses significant problems for programming instruction at South African Universities. It can be argued that South African Universities differ from other Universities, especially those of First World countries, in that students entering university have vastly different background characteristics. Some will have had full access to computers while others will have had negligible contact. Assuming the validity of the previous studies, a large number of South African students are not sufficiently prepared to benefit from conventional computer education methods.

Novice programmers are prone to hold many misconceptions about a programming language. Several studies have attempted to categorise student misconceptions. The majority of these studies were conducted in First World countries where English is the mother tongue[3]. The portability of these findings to the South African context can be questioned. Firstly, because of the vastly different background characteristics which students have, and secondly, because not all students have English as their first language. For this reason, this study incorporates an investigation into misconceptions held by novice programmers at a South African University.

When this research was initiated, approximately fifty percent of Introduction to Programming students failed this course at a South African University[4]. This is an indication that conventional teaching methods are inadequate. To further complicate matters, the number of students taking first year computer courses is growing rapidly each year, yet the number of lecturers, at best, remain the same. When one considers that increasing numbers of non-computer career oriented students are likely to request programming instruction (Soloway, 1993), the challenge is considerable. Unless the nature of programming instruction changes, programming instructors are unlikely to cope. It is feasible to investigate and evaluate additional teaching methods that would enable lecturers to spend less of their limited time explaining the details of syntax matters and rather focus on the general principles of programming, such as algorithm design and correctness.

The rationale of this research is threefold. Firstly, to teach programming effectively, it is essential to have an understanding of how students acquire programming knowledge, the problems they experience and the misconceptions that they have. Secondly, support environments have the potential to change the nature of teaching. Lecturers are likely to have more time to focus on the problem solving aspects of programming since support environments have the potential to provide students with greater control of their learning process. Not only are students likely to be less dependent on the instructor during formal

---

[3] See section 2.8.

[4] The course was offered in the Computer Science and Information Systems Department, University of Natal, Pietermaritzburg.

instruction periods, but they can have access to the support environment out of these times. This could contribute towards an increase in learning of the greatest number of students for a given amount of lecturer effort. Thirdly, there is evidence to indicate that support environments have the potential to compensate for a limited or negatively oriented background with respect to computer programming. Previous research shows background characteristics, such as prior computer experience, had less of an impact when the conventional teaching method was supplemented with a computer based support environment. Here the success of the student was more related to psychological orientations such as motivation and self-confidence. This would be advantageous in the South African context.

## 1.3 Limitations

This research attempts to illustrate that support environments, which are easily developed and implemented, can be effective in minimising many of the misconceptions that occur in programming. It is not the intention of this research to design or develop a computer based tutoring system. Computer based tutoring systems, and in particular Intelligent tutoring systems, which include features such as student testing and student modelling, tend to require considerable development time and testing. Nonetheless, this research provides strong evidence that the support environment discussed here, is an effective learning tool and thus can be used as a prototype for the development of a computer based tutoring system.

## 1.4 Overview of the Chapters

The thesis is structured around the research objectives. The next chapter investigates the manner in which novice programmers acquire their programming knowledge and the difficulties that they experience. The third chapter evaluates teaching strategies with the intention of minimising programming difficulties commonly encountered among students. The design of the support environment is the focus of the fourth chapter, and the fifth chapter describes the experimental process. The sixth chapter addresses the evaluation of the support environment and a discussion of the results. Details of these chapter contents are given below.

Chapter 2 addresses the understanding of how programming concepts are learnt, in particular:

- psychological aspects of learning new concepts
- psychological aspects of learning programming concepts
- how programming concepts are represented and structured
- successful and unsuccessful programmer traits
- differences between novice and expert programmers
- novice programmer's misconceptions

In Chapter 3 several teaching strategies are discussed. Although some researchers have adopted the approach of changing the programming language specifications, this chapter focuses on those teaching strategies that can be realistically adopted by programming instructors. Of particular interest, is the glass box approach.

In chapter 4, the design of the support environment based on the findings of chapter 2 and 3 is presented. The system can be considered to be a generic support environment that dynamically displays what is happening in memory during the execution of statements in Pascal programs written by an expert. The term generic implies that all students are assumed to have the same amount of programming knowledge. The system does not treat users individually, but rather the user is seen as a novice programmer.

Chapter 5 is concerned with the evaluation of the support environment to determine its effectiveness in reducing student programming errors, in improving general programming ability and in compensating for a disadvantageous background. To evaluate the system a formal experimental approach was adopted, and thus control and experimental groups were required. Chapter 5 commences with a discussion of the conventional and experimental teaching methods and the allocation of students to groups for the purposes of this study. The procedures for the collection of student data, such as students' background and psychological characteristics, and the evaluation of the data are then discussed. Chapter 6 includes considerable statistical testing, and hence a brief explanation of the statistical tests used is included.

Chapter 6 deals with the comparison of the student groups, in terms of examination and final results, noted misconceptions, performance on worksheet questions and other student characteristics. This discussion is based on statistical tests. Finally, student opinions of the support environment are given.

Chapter 7 concludes the thesis with an overview of the research, a discussion of the main findings of this research and suggestions for possible future research.

# 2. An Overview of the Psychology of Programming

## 2.1 Introduction

This chapter addresses different aspects of learning to program. Some psychological aspects of learning new ideas, and in particular programming concepts are outlined. This is followed by a detailed study of common difficulties encountered by novice programmers, and in particular, programming misconceptions. Some language specific difficulties are also discussed. Furthermore, an attempt is also made to categorise successful and unsuccessful learner traits. The rationale being that if we understand how students learn to program, and if we have knowledge of their difficulties, we will be better equipped to teach programming.

## 2.2 Learning of new concepts

Learning to program, and more specifically the planning and debugging of programming tasks, require "...a cognitive involvement which goes beyond rote memorisation or other low level thinking abilities." (Dalbey & Linn, 1985, p. 254) Cognitive psychologists, and more specifically Piagetian psychologists, refer to the process of assimilating new information with existing information as the process of 'meaningful learning' (Jones, 1986; Mayer, 1975, 1976, 1981). Meaningful learning results in understanding which Mayer describes as "...the ability to use learned information in problem-solving tasks that are different from what was explicitly taught" (Mayer, 1981, p. 122).

The human cognitive system can be broken down into two parts: short-term memory and long-term memory. Short-term memory has a limited capacity and duration and can be considered the active or working memory. The data in short-term memory is the information to which a person can give active conscious attention. Long-term memory is regarded as permanent and of unlimited capacity and holds all the knowledge a person has acquired. Although there are several models of the way information is organised, or structured in long term memory all the models agree that long-term memory is highly organised (Jordaan & Jordaan, 1984).

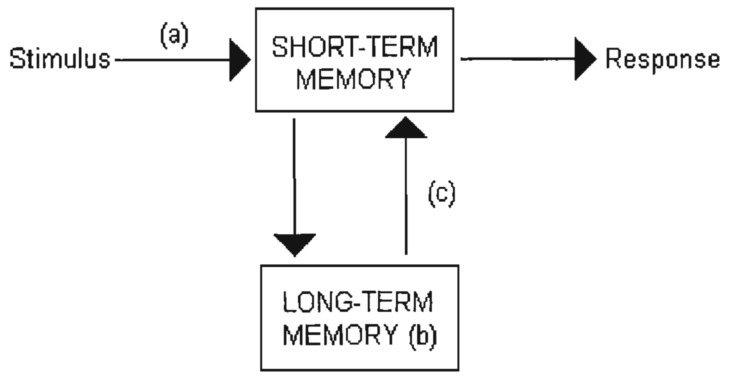


**Figure 2.1: Some information processing components of meaningful learning. Condition (a) is transfer of new information from outside to short-term memory. Condition (b) is availability of assimilative context in long-term memory. Condition (c) is activation and transfer of old knowledge from long-term memory to short-term memory (Mayer, R.E., 1981, p. 122).**

Mayer (1981) states three conditions have to be met for meaningful learning to occur, namely: reception, availability and activation. Firstly the learner must pay attention to the new information, this will allow the information to reach short-term memory, this is called reception and is depicted as arrow (a) in Figure 2.1 above. The second step towards meaningful learning is the availability step, depicted as arrow (b): the learner must possess appropriate prerequisite knowledge in long-term memory to use in assimilating the new information. The final step involves the learner actively using this existing knowledge during learning so that the new concepts may be connected with it (arrow (c) in Figure 2.1). Meaningful learning thus involves learners coming into contact with the new material (i.e. bringing into short-term memory), then searching long-term memory for 'appropriate anchoring ideas' or 'ideational scaffolding' (Ausubel, 1968[5] as cited by Mayer, 1981, p. 122) and then transferring those ideas to short term memory so that they can be combined with new incoming information.

---

[5]Original source: AUSUBEL, D. P. (1968). Educational Psychology: A Cognitive View, Holt, Rinehart and Winston, New York.

If any of these conditions are not met, 'meaningful learning' cannot take place and the learner will experience difficulties and be forced to learn each new piece of information by rote (Mayer, 1981).

## 2.3 Programming - what needs to be learnt

Linn & Dalbey (1985) have identified an ideal chain of activities, which involve considerable cognitive skill, required for the acquisition of programming knowledge. This is useful in that it breaks the acquisition of programming knowledge into three identifiable steps, and is discussed below. The chain has three main links: (a) single language features, (b) design skills, and (c) general problem-solving skills.

(a) Language Features

Students need to learn and understand the features of the language being studied. Although this knowledge is a prerequisite for the writing of programs, it is of little general use or benefit. Students with only an understanding of language features cannot develop code to solve a particular problem.

(b) Design Skills

Design skills are the group of techniques used to combine language features to form a program that solves a problem, and includes templates and procedural skills. These skills are essential for students to generate their own code to solve a problem of any complexity, and are defined thus.

**Templates**

Linn & Dalbey (1985, p. 192) refer to templates as "...stereotypic patterns of code that use more than a single language feature..." examples being the running-total template which accumulates the sum of a finite set of input and finding the minimum of a group of numbers template. These templates assist students in solving many problems without inventing new code. Templates can also reduce the cognitive demands of programming. Students can break down problems until they can implement the solution using one of their acquired templates. Students who have acquired templates are able to write more complicated programs than those who do not.

**Procedural Skills**

Procedural skills are used to combine templates and language features to solve new programming problems and include: planning, testing and reformulating of programming plans. Planning is required to solve complex problems. Novices rarely work on programs complex enough to demand substantial planning, but experts spend considerable part of their programming time engaged in planning (Kurland *et al.*, 1984). Testing is required to ensure that a program is reliable and robust. However novices seldom test their programs substantially and tend to use obvious or normal forms of input. Experts develop this skill. Reformulating involves the modification of program code, usually as a result of problems found in the testing phase. Novices seldom make substantial changes to their code, rather focusing on localised changes.

(c) Problem-Solving Skills

This is the last link in the chain of activities. These are templates and procedural skills common to many or all formal systems. For example, templates such as the insertion sort can be used in several programming languages and can be applied to other domains, such as card playing. The acquisition of these skills usually require a substantial amount of programming experience.

## 2.4 Learning of Programming concepts

Many authors have considered the process of learning to program as the closest one gets to the tabular rasa, or blank slate, situation of childhood (Sheil, 1981; Jones, 1985; Clements, 1986; Putnam, 1986). Most novice programmers do not have an appropriate cognitive framework in which to incorporate the new knowledge (Dalbey & Linn, 1985; Jones, 1985). As a result students initially rely on inappropriate analogies within their existing knowledge domain or "memorise each piece of new information by rote as a separate item to be added to memory" (Mayer, 1981). Both strategies hinder the effective learning of programming. The use of inappropriate analogies results in students possessing many misconceptions (Bayman & Mayer, 1983), many of which take months, or even years, of programming experience to eradicate[6]. The use of the rote learning strategy prevents

[6] See section 2.8.

meaningful learning from taking place, as the new information is not assimilated with the existing knowledge. As a result students will not understand the new concepts, and will be unable to transfer the new concepts to unique situations.

Generally, when someone initially learns to program they have to deal with three interwoven, yet separate cognitive tasks:

- "...orientation, finding out what programming is for, what kinds of problems can be tackled and what the eventual advantages might be of expending effort in learning the skill." (Du Boulay, 1986, p. 57)
- "...understanding the general properties of the machine that one is learning to control, the notional machine..." (Du Boulay, 1986, p. 57).
- learning the syntax and semantics of the formal language that has to be learnt.

Du Boulay (1986) comments that "...much of the 'shock' of the first few encounters between the learner and the system are compounded by the student's attempt to deal with all these different kinds of difficulty at once." The student is overloaded with new concepts as a result of the inherent complexity of the problem.

Initially, students have difficulty understanding what a program can do for them, and how it can be used to solve a particular problem and consequently have difficulty structuring their algorithms (Du Boulay & O'Shea, 1981). This problem is compounded by the usual analogy used by teachers to assist programmers in the initial programming stages: novices are instructed to write algorithms as if they were giving directions to a friend, or creating a recipe. "Part of a novice's difficulty in planning is caused by the disparity between the familiar conventions for specifying a plan to a human being...and a computer program" (Du Boulay & O'Shea, 1981). Miller (1975)[7], as cited by Du Boulay and O'Shea, found that a fundamental difference is the 'qualificational' human specification as opposed to the 'conditional' computer specification. Students have difficulty translating from the more natural 'qualificational' specifications of the form "PUT RED THINGS IN BOX 1" to 'conditional' specifications of the form "IF THING IS RED THEN PUT IN BOX 1" (Du

[7] Original source: MILLER, L.A. (1975). Naive programmer problems with specification of transfer-of-control. *Proceedings of the AFIPS National Computer Conference*, Vol. 44, p. 657-663.

Boulay & O'Shea, 1981). Novices tend to under specify algorithms and develop algorithms for specific instances of a problem rather than a general algorithm.

A necessary requirement for students to learn to program, is the internalising of a 'mental model' of the programming system (Hoc, 1977; Dalbey & Linn, 1985). The term 'mental model' is borrowed from cognitive psychologists who use the term mental model to refer to the organisation of memory into structures (Merrill, 1991). Bayman and Mayer (1983) refer to the mental model of the programming system as "the user's conception of the 'invisible' information processing that occurs inside the computer between input and output" (Bayman & Mayer, 1983, p. 677). The mental model of the language and computer, develops over time, as a student uses the programming system (Bayman & Mayer, 1983). Unfortunately little instructional effort is given to assist students in developing an accurate mental model of the language or system (Bayman & Mayer, 1983). Jones (1984) noted that novices scrutinise any aspect of the available learning resources, such as screens and notes, for confirmation of their own mental models. This may result in students developing a mental model that is inaccurate or incoherent (Bayman & Mayer, 1983; Jones, 1984). Du Boulay states that students frequently develop "...reasonable theories of how the system works, given their limited experience, except that the theories are incorrect" (Du Boulay, 1986, p. 72).

During the acquisition of the syntax and semantics of the formal language, several errors occur. Syntactic errors are incorrect statements which result in a compiler or interpreter error. Semantic errors involve the misuse of correct statements, such as the output of a variable value, before it has been initialised. Novices make semantic errors more frequently and they find these harder to diagnose, compared to syntactic errors (Du Boulay & O'Shea, 1981; Allwood, 1986). It has also been noted than these errors are not randomly distributed among the language constructs, but rather are clustered around certain constructs (Du Boulay & O'Shea, 1981). These semantic errors are referred to as misconceptions in this thesis, and they are generally said to occur as a result of a student possessing a faulty mental model of the programming system.

Another difficulty in acquiring programming expertise, is the acquisition of 'standard structures' (Du Boulay, 1986), 'conceptual chunks' (Jones, 1985), 'schemas' , 'plans'

(Soloway *et al.*, 1982; Joni & Soloway, 1986; Détienne & Soloway, 1990) or 'templates' (Linn & Dalbey, 1985; Anderson, 1986; Linn & Clancy, 1992a, 1992b, 1993). This equates to Linn & Dalbey's (1985) second link in the chain of cognitive skills. These standard structures represent abstract segments or chunks of code, which are customised depending on the specific programming task currently being undertaken (Soloway *et al.*, 1982). Comparative studies of novice and expert programmers have shown that novices are yet to acquire these standard structures, or conceptual chunks (Dalbey & Linn, 1985; Jones, 1985; Spohrer & Soloway, 1986). Novices "...tend to isolate and memorise rather than integrate and organise" (Linn, 1992, p. 121).

In addition, the standard structures or chunks that students do learn are often constraining because they are learned from a limited set of examples (Anderson *et al.*, 1984). As a result, novices are not able to keep up with the memory demands of the new language (Anderson *et al.* 1984; Linn & Dalbey, 1985).

Lastly the student has to develop strategies for putting everything together to solve a specific problem: "... mastering the pragmatics of programming, ...how to specify, develop, test and debug a program ..." (Du Boulay, 1986, p. 58). Generally novices are not expected to develop a great deal of expertise in problem analysis and specification in an introductory programming course (Dalbey & Linn, 1985), consequently this is not a concern of this thesis. Rather this thesis is concerned with students' understanding of the syntax and semantics of a programming language, the accuracy and completeness of their mental models and the degree to which they have acquired standard programming structures.

## 2.5 Differences between novice and expert programmers

A comparison of novice and expert programmers is useful in developing an understanding of how programming knowledge is structured or organised, and what needs to be done to develop that knowledge representation in novice programmers. It can also highlight the shortcomings of programming instruction. Novice programmers are those that have not yet successfully completed a programming course. Expert programmers are those that have successfully completed a programming degree and have had several years of programming experience.

Schneiderman *et al.* (1976) studied expert and novice programmer's abilities to recall programs. When a meaningful program was given, the expert programmers were able to recall more lines of code than novice programmers. When a program consisted of random program statements no difference in recall ability was noted. This is seen as evidence of expert programmers structuring their programming knowledge into chunks of meaningful code or program schemata. Novice programmers are less able to form these chunks. Analogous findings were found in novice and expert chess players by Chase and Simon[8] in 1973 (Allwood, 1986).

In Adelson's (1981) study of novice and expert programmers, subjects were required to recall lines of code, at most 16, which had been shown to them one at a time. Experts recalled more lines and had a different organisation in the recall compared to the novices. It was also found that novices had a smaller chunk size, and a less stable and less hierarchical organisation of the programming concepts than the experts.

More recent research has been undertaken by Rist (1986). Rist asked novice and experts programmers to group lines of code that were "...related to each other in their action" (Rist, 1986, p. 32), but gave no basis for this division. Although, novices did use some plan based groupings, they reverted to more syntactic groupings when programs became more complex. Generally, novice programmers grouped more lines of code together based on their syntactic characteristics, while expert programmers used plan based groups. This indicates that although novices have acquired some templates or plans, they are not as fully developed as those of expert programmers.

Expert programmers have a more accurate and well-constructed 'mental model' than novice programmers (Hoc, 1977). Expert programmers are more adequately able to cope with the complexities of writing a program to solve a particular problem as they have more highly organised domain knowledge. Developing these standard structures is an important factor in developing programming expertise (Dalbey & Linn, 1985). Novices need to be

[8] Original Source: CHASE, W.G. & SIMON, H.A. (1973). Perception in chess. *Cognitive Psychology*, Vol. 4, p. 55-81.

encouraged to acquire an organisation that is more like that of an expert, but they need help in this endeavour. This is a responsibility of programming instructors.

## 2.6 Successful and Unsuccessful programmer traits

In an attempt to understand the process of learning to program in more detail, many authors have examined the traits of successful and unsuccessful programmers. Coombs *et al.* (1982) believe "...analysis of the contrasting learning strategies used by successful and unsuccessful learners should provide data on the nature of computing information itself and on the cognitive skills required for its acquisition".

### 2.6.1 Cognitive Style

Coombs *et al.* (1981, 1982) note that writers on cognitive style, a pattern of strategies a thinker uses to handle information, generally make their definitions "in terms of polar dispositions" and cite the following examples (p. 296 & 454):

| | | |
|---|---|---|
| convergent thinking | - divergent thinking | (Hudson, 1966) |
| vertical thinking | - lateral thinking | (de Bono, 1967) |
| analytic | - gestalt | (Levy-Agresti & Sperry, 1968) |
| verbal | - spatial | (Paivo, 1971) |
| sequential | - simultaneous | (Luria, 1966) |
| field independence | - field dependence | (Witkin, Dyk, Faterson, Goodenough & Karp, 1962) |

Coombs *et al.* (1981, 1982) noted that all of the above dichotomies are similar, although not identical, in that they all describe cognitive style in terms of two contrasting modes. Coombs *et al.* describe these dichotomies as follows (Coombs *et al.*, 1981, p. 296; Coombs *et al.*, 1982, p. 454):

> "...(a) a mode that is active, analytical, articulated, specific and critical;
>
> (b) a mode that is passive, global, vague, diffuse and uncritical."

Coombs *et al.* (1981, 1982) classified their learners into operational or comprehension learners, following Pask's (1976)[9] categorisation of cognitive style based on human information processing. Operation learners, adopt a bottom-up approach, concentrating of the details of the new material and work 'upwards' towards a general understanding. These learners concentrate on the logical relations between the material. Comprehension learners, adopt a top-down approach, concentrating on a global picture and then deal with the specifics. These learners may not be able to perform the operations required to use the new material, as they often ignore the relations that connect the different material together (Du Boulay & O'Shea, 1981; Coombs *et al.*, 1981, 1982; Dalbey & Linn, 1985).

From their empirical research on first time programmers learning FORTRAN at the university level, Coombs *et al.* (1981, 1982) were able to distinguish some discernible behavioural differences between the comprehension and operation learners, namely:

| | **operation learners** | **comprehension learners** |
|---|---|---|
| during lectures | • gave priority to writing the facts down | • focused on understanding the information |
| practical exercises | • completed more exercises | • completed less exercises |
| | • accepted exercises as given and solvable | • questioned exercises validity |
| | • extended the problems once they had solved the initial specifications | • rarely experimented with the language outside the given exercises |
| general or conceptual questions | • rarely asked such questions | • commonly asked such questions |

Coombs *et al.* (1981, 1982) concluded that the successful learners, namely the operation learners, produced a more abstract representation of language structures, and are less bound to the context of the specific examples, than the comprehension learners. Coombs *et al.* (1981, 1982) thus conjecture that the operation learners would be more flexible in their

[9] Original source: PASK, G, (1976). Conversation theory: Application in Education and Epistemology, Elsevier, Amsterdam, p. 85-86.

problem solving. From these experiments the authors were able to conclude that students who are more successful in acquiring programming skill were those learners who "paid close attention to detail, systematically abstracted the critical features of programming structures and represented structural relations in terms of rule form." (Coombs *et al.*, 1982, p. 474)

It can be argued that one way of assisting students in programming knowledge would be to lead them towards an operational strategy. However it is necessary to take heed of Pask's warning, as cited by Coombs *et al.*. Students find it very hard to adopt a differing learning style, and even when they do they are significantly less effective than those who have it naturally. Taking heed of these findings, some researchers (Dalbey & Linn, 1985; Cavaiani, 1989) have suggested that in order to overcome the consequences of the mismatch of teaching methods to learning style, different teaching strategies should be adopted for different students based on their learning style.

### 2.6.2 Cognitive Abilities

High general ability students perform well in programming classes (Linn & Dalbey, 1985). However, Mayer *et al.* (1986) caution that this might not signal high general ability as a necessary requirement for programming, but rather the ability of the test "...to predict success in academic learning under a wide variety of situations" (p. 608). It remains necessary to determine the specific skills required for programming.

However, in the few studies undertaken, conflicting factors have been seen to be most influential. Snow (1980)[10], as cited by Mayer, reported that 'diagramming', non-verbal logical reasoning, and mathematics problem solving correlated with learning BASIC. Webb (1984) stated that a mathematical test consisting of word problems and computation problems was the best predictor of success in learning Logo. Other contributing factors were non-verbal logical reasoning and spatial ability. Clements (1986) found that mathematical ability, logical operations, creativity and field dependence were all related to

[10] Original source: SNOW, RE. (1980). *Aptitude processes*, in Vol. 1, Aptitude, Learning and Instruction, Snow, R.E.; Federico, P. & Montague, W.E. (eds.), Erlbaum, Hillsdale, N.J., p. 27-63.

most components of Logo programming. However, Clements' study reveals a lack of consistency in the relationship between cognitive abilities and the ability to perform programming tasks for first and third graders. For example, "Creativity and mathematics achievement best predicted the total off-computer score of first graders..." but "... field dependence and creativity were most predictive of the total off-computer score.." of third graders. Although some of these differences can be understood in terms of developmental differences, it must be noted that measures of cognitive ability "... may be differentially predictive for various age groups, who may use divergent solution processes". Clements concluded from his empirical research that "...Logo programming is not just for the mathematically proficient." This contrasts with Goodwin and his colleagues (Goodwin & Sanati, 1986; Goodwin & Wilkes, 1986) studies of Pascal programmers. They found mathematics to be a good predictor of success in an introductory programming course.

However Wileman *et al.* (1981) also found no strong evidence to indicate that BASIC programming success was based on mathematical or scientific ability. They found Reading Comprehension, Alphabetic and Numeric Sequences, Logical Reasoning, Algorithmic Execution and Alphanumeric Translation skills to be more reasonable predictors of success.

Mayer and his colleagues (1986) found that learning to program in BASIC was related to general ability, especially logical reasoning and spatial ability. However, they also identified two specific thinking skills which tended to predict success in learning BASIC: "...ability to translate word problems into equations or answers (problem translation skill), and ability to predict the outcome of a procedure or set of directions that is stated in English (procedure comprehension skill)" (Mayer, 1986, p. 609).

Generally these results can be summarised by stating that general ability is correlated to success in programming, however more specific skills have been isolated. These include non-verbal skills, such as logical reasoning, and problem translation and procedure comprehension skills. Mathematical skill is also cited.

### 2.6.3 Computer Anxiety & Alienation

Computer anxiety is "the fear of impending interaction with a computer that is disproportionate to the actual threat presented by the computer"[11] and computer alienation refers to "generalised feelings of despair, discontent, or frustration" (Ray & Minch, 1990, p. 478). Generally, students who have less computer anxiety achieve higher scores on programming tasks (Ray & Minch, 1990; Chen & Vecchio, 1992). A similar relationship exists between computer alienation and programming tasks (Ray & Minch, 1990).

### 2.6.4 Access to Computers

Access to computers has been seen to be a contributing factor in learning to program. However this is only a significant factor when insufficient computer resources are available to the student. Linn & Dalbey (1985) categorised student access as high and low. High access students included students with home access and possible additional access. Low access students were those students who had no home access. In their research at several schools they found that their was a significant difference between high and low access students' performance only when there was insufficient in-school access.

### 2.6.5 Interest

Interest in the topic will make students more motivated, however this has not been found to be a substantial influencing factor in the success of students learning to program (Linn & Dalbey, 1985).

### 2.6.6 Gender

Generally it is agreed that in terms of programming ability there is no difference between male and female students (Mazlack, 1980; Linn & Dalbey, 1985).

---

[11] Original source: HOWARD, G.S.; MURPHY, C.M. & THOMAS, G.E. (1986). Computer Anxiety considerations for design of introductory computer courses. In Proceedings of the 1986 annual Meeting of the Decision Science Institute , Atlanta, GA: Decision Science Institute, p. 630-632.

### 2.6.7 Additional factors

**Introverts vs. Extroverts**

Introverted individuals tend to perform better than more extroverted individuals on certain programming tasks. Chen & Vecchio (1992) postulate that this is perhaps as a consequence of introverts being better able to focus their attention on cognitively demanding tasks.

**Attitudinal Characteristics**

A strong predictor of programming success is if the student expects the course to be difficult. This attitude has a negative impact on programming success. If students prefer problems with one rather than multiple solutions, this also negatively impacts their programming success (Goodwin & Wilkes, 1986). If students find the assignments easy and the programming instructor helpful, a positive impact is seen on programming success (Goodwin & Sanati, 1986).

Generally it can be said that programming is dependent on aptitude and personality characteristics of an individual (Goodwin & Sanati, 1986; Goodwin & Wilkes, 1986; Chen & Vecchio, 1992).

## 2.7 Novice Programmers' Misconceptions

A programming misconception is an incorrect notion of one, or a combination of many, programming constructs. It is a persistent misunderstanding which may be maintained for several years by the programmer. For example, a common misconception that novice programmers experience is that a variable named LARGEST would take on the value of the largest of several inputs.

A misconception is different from a programming error, as an error is often a result of carelessness or short-sightedness of the programmer. On the other hand, misconceptions often require the programmer to reconstruct their perception of the underlying machine or compiler. This process of reconstructing or fine tuning sometimes takes several months of programming. As a result, some so called expert programmers still have incorrect mental models (Du Boulay, 1986).

In this study, errors such as the most frequent Pascal syntactic error, the omission of a semicolon (Ripley & Druseikis, 1978), are not considered to be misconceptions. These are rather considered to be language specific problems which could be eradicated by improving the programming language specifications. This study of programming misconceptions focuses on those programming problems which require the student to reconstruct or modify their perception of the conceptual machine, and those problems which are language independent.

Misconceptions are widespread. They occur in primary to adult students (Pea, 1986). In addition, research has shown that they occur in most programming languages (Pea, 1986), although most research has examined novice programmers learning BASIC or Pascal. It has also be found that misconceptions are transferred from one programming language to another. Thus, students who learn to program in one programming language and then another, not only transfer their knowledge but their misconceptions as well.

Misconceptions occur as a result of insufficient knowledge of the required domain. Much of program instruction (incorrectly) treats learning to program as a new and independent skill which relies little on previous knowledge or learning. Pea (1986) refers to the novice programmer as being in a state which is the closest an adult will get to "situation of a tabula rasa" ( Pea, 1986, p. 26).

Novice programmers, having little knowledge on which to base their current learning experiences tend to work intuitively. As a result, they develop survival techniques, which are, at times, inappropriate. One such survival technique which is common among novices is to use the analogous situation of conversing with a human, when attempting to understand the process of programming. Unfortunately, this analogy results in several misconceptions as the computer is given additional interpretative powers which are beyond the power of a computer or compiler. Pea (1986) referred to conceptual 'bugs' and classified the 'superbug' as the "default strategy that there is a hidden mind somewhere in the programming language that has intelligent interpretative powers". Novices also rely on knowledge from other domains such as mathematical algebra.

Further problems arise as a result of the programming languages themselves. Most programming languages have chosen commands or instructions derived from English to assist the programmer in writing code. However, this often causes misconceptions in novice programmers, as they associate too much of the English meaning to the command. The Pascal 'while' looping construct is an example. Novice programmers associate the temporal nature of the 'while' English word to the construct and falsely believe that the loop is terminated as soon as the condition is no longer true[12]. This occurs as a result of "...the mismatch between the designer's and the user's understanding of what is implied..." by a certain command (Du Boulay, 1986). Novice programmers may also assume that the system has inference capabilities of a human because of the naturalness of the language (Du Boulay, 1986). The grammar of programming languages, which are mostly English-like, is also seen to be a problem, especially for programmers whose first language is not English. These programmers must not only learn the vocabulary but also the grammar of the programming languages (Du Boulay & O'Shea, 1981).

All these factors contribute to the novice programmer developing inappropriate mental models of the underlying machine on which the program runs. The remainder of this chapter discusses particular programming misconceptions in more detail. Initially, a conceptual classification of misconceptions is presented. The conceptual classification of misconceptions classifies the misconceptions based on the students' underlying thought processes. The chapter concludes with an in-depth discussion of programming constructs and their associated misconceptions.

### 2.7.1 A Conceptual Classification of Misconceptions

In Pea's (1986) study of language-independent conceptual 'bugs' in novice programmers, he has identified three classes of bugs: parallelism, intentionality and egocentrism bugs. Pea suggests that these misconceptions are a result of the "'superbug', the default strategy that there is a hidden mind somewhere in the programming language that has intelligent interpretative powers" (Pea, 1986, p. 26).

[12] See section 2.7.2.1.

Pea's classification is worthy of consideration as it is a first attempt at a classification of programming misconceptions, and gives the reader a global understanding of novice programmers' underlying thought processes. Consequently each of his three classes of bugs are discussed below. In Du Boulay's (1986) study of novices' difficulties in learning to program, he briefly highlighted three types of errors: misapplications of analogy, overgeneralisations and the inexpert handling of complexity. These error types can be used to categorise novice programmers misconceptions and are also discussed below.

### 2.7.1.1 Parallelism Bugs

This is the perception that several lines of code "... in a program can be active or somehow known to the computer at the same time, or in parallel" (Pea, 1986, p. 27). This misconception occurs in two main contexts.

The first, is in the context in which conditional statements occur outside of loops. Pea (1986) gives the following example (p. 27):

```
IF SIZE = 10, THEN PRINT "HELLO
```

where `SIZE`[13] is the variable name in the conditional statement. Later in the program a loop is introduced to increment (by one) the variable `SIZE` until it reaches ten.

```
FOR SIZE = 1 TO 10, PRINT "SIZE
NEXT SIZE
```

Eight out of the fifteen high school students in their second year of Computer Science predicted that `HELLO` would be printed when `SIZE` became 10. The novices failed to comprehend that the IF statement would be inactive by the time the variable `SIZE` became ten. Rather, the students stated that the IF statement was waiting for the variable to become ten. One student stated: "It looks at the program all at once because it is so fast" (Pea, 1986, p. 27). Pea interprets these comments by suggesting that the student attributes

[13] To distinguish program variables in the text, the convention in this and subsequent references to variables is the use of the `Courier` font.

the program to have the ability to monitor the status of every line in the program simultaneously.

This misconception is also noticeable when students are asked to predict the outcome of a program which includes a while statement. As many as a third of the Pascal students predicted that the while loop would be halted as soon as the exiting condition became true.

To understand the student perception of if and while statements, Pea examined the usage of conditionals in natural language as it was his contention that these bugs were as a result of the 'superbug'. In other words, he believed that students were using the inappropriate analogy of conversing with a human. Pea suggests that if you offer to help someone if they are having difficulty with something, the individual might take you up on your offer several hours, days, months or even years later. In fact, in English usage, the decision is rarely instantaneous, the temporal nature of the conditional is based on the context of the situation. However, in computer programming the conditional response is immediate.

If one examines the usage of a while expression in natural language, such as "while the highway is two lanes, continue north" (Pea, 1986, p. 28), it is understandable why novice programmers should have this misconception that the while loop is continuously monitored for the exit condition. The while statement in computer programming is at odds with the natural language interpretation of a while statement.

Novices, experiencing symptoms of the parallelism bug have more than likely applied their knowledge of natural language to the new domain of computer language. The students have incorrectly transferred their knowledge of natural language to the more structured programming language context.

The second context in which the parallelism bug is evident, is one in which variables are assigned values or initialised after the lines of code which use the variables. Pea (1986) gives the following lines of code as an example (p. 28):

```
AREA = Height X Width
Input Height
Input Width
PRINT "AREA
```

When the product of Height and Width are calculated on the first line, the current values, or if they have not yet been assigned a value, two arbitrary or default values, will be multiplied and assigned to AREA. However, many students assume that the product of the inputted Height and Width variables will be assigned to AREA and printed out in the fourth line. The students fail to realise that the programming language is unable to look ahead to determine the desired values for the variables Height and Width. They have given the programming language (or machine) the ability to process statements as a human listener or reader would be capable of doing. They have assembled all the given information to produce the required result. Pea notes that in natural language domains students are encouraged to scan ahead as a reading strategy, and it is this very principle that has led them astray in the more formal domain of programming language comprehension.

### 2.7.1.2 Intentionality Bugs

Intentionality bugs are those problems which arise as a result of students assigning predictive abilities to the program, or as Pea (1986) states "...goal directedness or foresightedness... " (p. 29). The students assume that the program is capable of determining the intention of the code, thereby indicating that they attribute human qualities to the program. This class of misconception is evident when students are required to comprehend or trace through a program. Pea (1986) demonstrates this misconception with the use of the following Logo program example (Pea 1986):

```
TO SHAPE :SIDE
IF :SIDE = 10 STOP
REPEAT 4 [FORWARD :SIDE RIGHT 90]
SHAPE :SIDE/2
END
```

When one types SHAPE 40, the program will draw a large square of size 40 and then a smaller one inside the first and then stop. Initially, SIDE has a value of 40. In the second line, the condition statement determines whether the code should be terminated, as the SIDE is not equal to ten, it will continue to execute the third line. In this line the process of moving forward forty units and then turning ninety degrees is repeated four times. This in effect draws the first square. The penultimate line repeats the execution of the code, but with a size of twenty. When the code is repeated after drawing a square of twenty units, the execution is stopped on the second line due to the condition statement.

When students are required to predict the outcome of the code, some students incorrectly interpret the conditional statement's usage. Erroneously, some students predicted that the code will result in the box of size 10 being drawn. They interpreted the conditional as a command 'encouraging' the computer to draw a square of size ten. When Pea (1986) questioned these students as to why a square of size 10 would be drawn, they responded by saying things such as "... because it wants to draw a square". These students have given the program intentional powers.

#### 2.7.1.3 Egocentrism bugs

This class of misconceptions takes its name from the psychological phenomenon which is common amongst children. Egocentrism is an overemphasis on the perspective of self relative to that of others. Egocentrism is usually manifest in tasks which place strenuous cognitive demands on the individual. In programming, egocentrism bugs are those bugs "...where students assume that there is more of their meaning for what they want to accomplish in the program than is actually present in the code they have written" (Pea 1986). Students give the program the ability to interpret what they intend and not necessarily what they have written in the program. As a consequence, variable values or essential lines of code are often omitted. These skeleton programs are not a result of sloppy work but rather a consequence of the student assuming that the program is able to fill in all the missing bits to accomplish the intend task.

Soloway *et al.* (1982) have also found the existence of a persistent misconception which can be considered to be a further example of the egocentrism bug. They labelled this misconception the 'mushed variable' bug. A quarter of their Pascal students erroneously used the same variable for more than one role. Soloway *et al.* (1982) refer to code in which students have used the variable `X` to both store the value of some input and to hold a running total of the input variables. In this instance, students are assuming that the program is able to determine in which role the variable is required to be used, and use it accordingly.

Egocentrism bugs occur when students are required to write code to accomplish a task. Intentionally bugs occur when students trace or depict the outcome of some correct code. Both classes of misconceptions are a result of the student attributing too much interpretative power or intelligence to the computer. In parallelism bugs the student assumes that more than one line of code can be active at any instance. In this case the student is attributing the computer with the intelligence to assimilate and process more than one bit of information at a time, a characteristic of human interactions. As has been discussed, all these errors result from the student erroneously applying the analogy of conversing with a human, when they are interacting with the program or machine.

Pea suggests that if we wish to overcome these errors, that methods to diagnose the misconceptions must be developed. The programs or problems in which misconceptions can arise should be frequently and explicitly given to the student by the teacher. Also, the novice programmer needs to be made aware of what must be explicitly expressed in the code and what is done by the compiler or interpreter.

#### 2.7.1.4 Misapplication of analogy

These types of errors arise as a result of a student attributing "... more structure or relationships from an analogy than are warranted" (Du Boulay, 1986, p. 58). Du Boulay refers to the infamous box analogy of a variable: students erroneously believe that a variable can hold more than one value. Pea's 'superbug' is a result of students misapplying the human analogy to the programming system.

#### 2.7.1.5 Overgeneralisations

These types of errors are a result of the novice overgeneralising one feature of the system to another. For example, because the formal parameters of a Pascal procedure are separated by semicolons, a student might erroneously separate the actual parameters with a semicolon. Although Du Boulay gives only syntactic examples he states that these types of errors are not necessarily limited to syntactic errors. Soloway *et al.* (1982) believe that some errors occur as a result of students overgeneralising the counter variable concept to input variable concepts[14]. In many instances, these errors can be considered to be as a result of inconsistencies of the programming language design.

#### 2.7.1.6 Inexpert handling of complexity

Du Boulay (1986) believes that inexpert handling of complexity is the cause of the third type of error. More specifically, students do not understand the interactions of different sub-parts of a program and thus improperly interleave them in the program.

## 2.8 Programming Construct Misconceptions

The above discussion focused on a conceptual classification of language independent bugs or misconceptions. Other authors have concentrated on studying novices' actual problems while studying particular languages. This section summarises these researchers results from a more language oriented approach. This is not necessarily in conflict with Pea's classification. Nonetheless, it is necessary to classify the misconceptions in terms of language constructs to appreciate the complexity of the problem.

There are three measures of errors: error frequency, error proneness and error-persistence. Error frequency determines how often a particular error is noted. Error proneness refers to the overall frequency of a particular error relative to its use. There is evidence that a small number of error types account for the majority of all errors. For example, conditions are highly error-prone even though they account for a relatively small number of actual errors. That is novices are highly likely to make an error when coding a conditional although they generally do not include many conditions in their programs (Young, 1974 as cited by Du Boulay & O'Shea, 1981). Error persistence refers to the rate at which errors are eliminated from the programs or the user's mental model. Errors with high persistence take longer to diagnosis and eradicate. The while loop's temporal error is an example of a high error persistence situation (Du Boulay & O'Shea, 1981).

[14] See Section 2.8.1.3.

This study is primarily concerned with error proneness. Given the constraints of most teaching situations, programming instructors need to focus their attention on those constructs or plans which have high error proneness.

The statistics discussed in this section are based on research on Pascal and BASIC novice programmers who attended college or high school programming courses in the USA during the 1980's (Bayman *et al.*, 1983; Putnam *et al.*, 1986; Sleeman *et al.*, 1986).

In Putnam *et al.*'s study, the students were high school BASIC novice programmers, from five different schools, who had completed a BASIC programming course. Ninety-six students were examined with a six item test. Four test items required students to predict the outcome of a four to ten-line program and two items required the students to debug a slightly more complex program. A written description of the intent of the programs was provided. Fifty-six students were subsequently interviewed to allow the researchers to understand the exact nature of the individual student's problems more fully.

Sleeman *et al.*'s study, (same authors as above), involved an analysis of high school Pascal novice programmers, from three different schools. Most students had some previous BASIC exposure. The method was the same as the BASIC experiment described in the previous paragraph except for two factors: the administered test was refined and fewer students were interviewed. The screening test included nine items in which the student had to predict the outcome and one debugging item. As the authors were limited to interviewing 35 of the 67 students by logistics, they selected the students with the most significant difficulties.

The third study (Bayman & Mayer, 1983) involved thirty college undergraduate students who had successfully completed an introductory BASIC course. In this study, the test required the students to explain in natural language (English) the steps that the computer would carry out to complete the programming statement.

### 2.8.1.1 Input Misconceptions

Input misconceptions are those misconceptions which students have regarding the reading of input. Given a series of input values, students with input misconceptions will falsely predict which value is accepted by the computer based on some characteristic of the program code. Students failed to understand that input is accepted in a sequential manner, without any implicit selection process. "More of the students ... had difficulties with READ statements than with any other aspect of the BASIC language" (Putnam *et al.*, 1986). The input misconceptions will be discussed in turn.

'Semantically constrained Reads' - In this instance the misconception is that the program can select values from the given input based on the basis of features of those inputted values (Sleeman *et al.* , 1986). The most common view was that a READ statement used with a meaningful variable name selected the most appropriate value for the variable in terms of the variable name (Putnam *et al.*, 1986; Sleeman *et al.*, 1986).

Sleeman *et al.* (1986) use the following Pascal program as an example:

```
PROGRAM B1;
VAR First, Smallest, Largest:INTEGER;
BEGIN
  WRITELN('Enter three numbers');
  READLN(Largest, Smallest, First);
END.
```

Given the input `5`, `10` and `1`, students with this misconception believe that the variable `Smallest` will take on the value `1`, `Largest` will take on the value `10` and `First` the value `5`.

Declaration order determines the order in which values are read into variables - Students with this misconception would assign values to variables based on the order of declaration, if the declaration order and input command (READ statement) were not the same (Sleeman *et al.*, 1986).

Below is a Pascal program which could serve to diagnose students who hold this misconception:

```
PROGRAM example2;
VAR B, A, C:INTEGER;
BEGIN
  WRITELN('Enter three numbers');
  READLN(C, B, A)
END.
```

Given the input: 25, 10 and 20 students with this misconception would argue that B gets the first value (25), A gets the second value (10) and C (20) gets the last value because of the order in which the variables were declared (Sleeman *et al.*, 1986).

Variables are assigned values based on the variables' alphabetical order - This misconception occurred when students "...tried to impose meaning on single-letter variable names in READ statements" (Putnam *et al.*, 1986, p. 464). The following BASIC program illustrates this misconception (Putnam *et al.*, 1986):

```
40 READ A
50 READ B
60 READ N
200 DATA 9, 38, -100, 5, 12
```

Students with this misconception stated that A gets the first value, namely 9 "...(because 9 is the first value in the DATA statement)...", B gets the second value (38) and N would be assigned the value 12 "...because N is near the end of the alphabet and 12 is the last number in the list of data" (Putnam *et al.*, 1986, p. 464-465).

Multiple-value variables - In this instance this misconception is that variables can be assigned more than one value at one time. This error often occurs with the semantically constrained input misconception. For example (Sleeman *et al.*, 1986):

```
PROGRAM B3;
VAR Even, Odd: INTEGER;
BEGIN
  WRITELN('Enter data: ');
  READLN(Even, Odd);
END.
Input:  3 2 10 5
```

Students consistently stated that Even would be assigned the values 2 and 10, and Odd would hold the values 3 and 5. One variation of the multiple value variable misconception is that students falsely believe that all the values are assigned to (all) the variables.

Another variation is that students thought that a variable name is capable of holding the same number of values as characters in the name, hence a 3 character variable e.g. `Odd` would be capable of holding 3 values simultaneously. These variations were less frequent than the general form of the multiple-value variable misconception.

Another error noted with input statements can be observed when students execute programs. Students do not understand where the to-be-input data comes from. In novice programmer situations this is invariably the keyboard, yet students do not understand that the control shifts from the computer to the user at this point (Bayman & Mayer, 1983; Du Boulay, 1986). Furthermore, in most languages, the syntax of an input statement disguises the fact that the variable mentioned in the input statement has it's value changed, or initialised, by the input statement (Du Boulay, 1986).

### 2.8.1.2 Output Misconceptions

These errors appear to be language dependent. Three errors were noted in Pascal regarding statements such as

```
WRITELN('Enter a number:');
```

1. caused a number to be read, similarly `WRITELN('Enter 5 numbers:')` would cause 5 numbers to be read.
2. caused the variable name and value to be printed.
3. after the statement has been executed the program can choose a number from the input values.

Bayman and Mayer (1983) have noted that novices have difficulty in conceiving that output statements merely display on the screen what they are instructed to do. They also assume functional capabilities when a semantic variable is included in the output statement.

Another error involved students not distinguishing the difference between the BASIC commands `PRINT C` and `PRINT "C"` or the Pascal commands `WRITE('C')` and `WRITE(C)`. Without inverted commas the value of the variable `C` is outputted, otherwise the character C is outputted. Students either ignored the quotation marks, or believed that the quotation marks would cause the value of the variable to be printed.

Other occasional errors noted by Putnam *et al.* (1986) are the repeated print and multiple-value print misconceptions. One student, who had major programming difficulties, thought that an output statement of the form `PRINT X` would cause the value of X to be printed several times. Putnam *et al.* (1986) labelled this the repeated print misconception. Students with the multiple-value print misconception believed that all previous values associated with a variable would be printed when the variable was printed.

#### 2.8.1.3 Variable Misconceptions

The most common misconception regarding variables is that a variable can hold more than one value simultaneously. This misconception manifests itself in various situations. One such situation is the multiple value read mentioned above under input misconceptions. Some students recognised that the values were READ in one at a time, but believed that previous variable values will still be known. They believed that the values were collated in some way into the variable - that the variable acted in a similar way as a stack data structure. Hence they believed that a subsequent output statement would result in all the values being displayed (Putnam *et al.*, 1986). This is probably as a result of the misapplication of the box analogy[15] for a variable. Students believe that the values overwritten are still available somewhere and can be retrieved (Du Boulay, 1986).

Confusion of variables - Here students confuse two variables in a program. Sleeman *et al.* (1986) illustrate this with the following Pascal statements:

```
READLN(P); Q:=Q+1;
```

Students with this misconception interpreted the line of code as:

```
Q=P+1;
```

In diagnosing errors which occur when students write their own programs, Soloway *et al.* (1982) have referred to 'mushed variables'. A 'mushed variable' is when a single variable

[15] See section 2.7.1.4.

is used  incorrectly for more than one role. An example is shown below (Soloway *et al.*, 1982, p. 50):

```
program Student26_Problem2;
var X, Ave : integer;
begin
  repeat
    read (X);
    X:=X + X
  until X + X > 100;
  Ave := X div Nx;
  Write (Ave)
end.
```

Here the student has used the variable X as both an input variable and a counter variable. Although the authors could not explain this error, they concluded that it was probably due to the student assuming that the computer had some interpretative skills. Students assumed that because they themselves could distinguish the different roles of the one variable, so could the computer and hence use the different values appropriately (Soloway *et al.*, 1982). This is an example of Pea's (1986) egocentrism bugs.

<u>Initial value of the variable is maintained rather than updated</u> - Putnam *et al.* (1986) postulate that this problem could merely be as a result of difficulties with tracing code. Students were simply overloaded due to the complexity of the problem. Du Boulay (1986) argues that this may be linked to the idea of a variable remembering its previous value and believes that this error concerns the temporal scope of the assignment statement. Students may think that the variable "...value does not fade away and hangs around until either explicitly changed, the contents of memory are erased or the machine switched off" (Du Boulay, 1986, p. 65).

<u>Printing of variable values</u> - two variations of this misconception are evident (Sleeman *et al.*, 1986):

a) Values of variables are printed whenever the variable is encountered on the LHS (left hand side) of an expression.
b) The value of the LHS variable is printed whenever its value changes.

When students have been required to write programs, a common error is for an input statement to be included before the loop, but not inside the loop. Soloway *et al.* (1982)

believe that this is not a result of carelessness, but rather the overgeneralisation of the concept of the counter variable to an input variable. Students with this misconception believe that decrementing an input variable by one will return the previous value of the variable, in the same way decrementing a counter variable by one returns the previous value of the counter. Further evidence is the selection of the `Prev_Num` variable, by the student, to hold this previous input value. This finding can also be as a result of the student misapplying the box analogy to a variable, and it supports Putnam *et al.*'s (1986) and Du Boulay's (1986) findings that students believe that a variable is able to collate all inputs using mechanisms such as a stack.

#### 2.8.1.4 Assignment Misconceptions

Reversal of the assignment statement - This is evident in both Pascal and BASIC novice programmers. For example, `A:=B` (Pascal) or `LET A = B` (BASIC) is interpreted by students with this misconception as meaning that `B` gets the value of variable `A` as opposed to the correct interpretation of `A` getting the value of `B`. However this seems to be a minor conceptual problem as most students with this misconception interpret statements of the form `A:= B + C` (Pascal) or `LET A:= B + C` (BASIC) correctly (Putnam *et al.*, 1986).

Assignment statement is equivalent to the Boolean comparison operation - students with this misconception believe that statements such as `LET C = C + 1` are invalid as `C` (say with a value 0) cannot be equal to `C + 1` (1), however statements such as `LET W = A + 1` were considered valid by the same students (Putnam *et al.*, 1986). Other students learning Pascal interpreted the assignment statement as a comparison. Thus `A:=B` is interpreted as the Boolean comparison of the variables `A` and `B`, the result of which is true only if the variables' values are the same (Sleeman *et al.*, 1986). Putnam *et al.* (1986) suspect that variations of this misconception appears to occur as a result of students incorrectly transferring knowledge from another domain, namely algebra.

Variables swap values - `A:=B` is interpreted as `temp:=A`, `A:=B` and `B:=temp` (Sleeman *et al.*, 1986).

Instantiated variable's value is printed - Given the sequence `A:=2; B:=3; A:=B` one student in Sleeman *et al.*'s 1986 study stated that 2 = 3 would be printed.

Links variables together - An assignment of the form `A:=B` links the variables `A` and `B` in some way, so that whatever happens to `A` in the future also happens to `B`. This is as a result of students failing to see the difference between the identity and equality of two variables (Du Boulay, 1986).

Other students believe that an assignment statement has no effect (Sleeman *et al.*, 1986).

Some authors have postulated that these errors are a result of the overloading of operators such as the equivalence symbol (Ripley & Druseikis, 1978; Du Boulay & O'Shea, 1981).

Soloway *et al.* (1982) have noted that novice programmers experience more difficulty with the running total variable concept compared to the counter variable concept, although both concepts require similar assignment statements. They put forward three hypotheses which could account for this observation.

1. The activity of counting, which uses the counter variable concept, and the activity of summing values, which uses the running total variable concept, are different activities.
2. "Students might learn the notion of a counter as a special entity..." and do not decompose the assignment statement into a left hand side variable getting the value of the right hand side expression. Consequently, when students are confronted with the running total variable concept, they have to "...confront their understanding of the particular type of assignment statement needed in this context" (Soloway *et al.*, 1982, p. 47).
3. The running total variable update requires the addition of a variable, whereas the counter variable update requires the addition of a constant. Soloway *et al.* (1982) postulate that novices find variable concepts harder than constants.

### 2.8.1.5 Loop Misconceptions

#### 2.8.1.5.1 General looping errors

The errors discussed in this section apply to all types of looping constructs such as the Pascal WHILE, FOR and REPEAT loop constructs.

WRITELN adjacent to loop is included in the loop. A Pascal program which illustrates this error is:

```
PROGRAM A5;
VAR I, X: INTEGER;
BEGIN
  FOR I:= 1 TO 3 DO
  BEGIN
    WRITELN('Enter a number.');
    READLN(X);
  END;
  WRITELN(X);
END.
```

Given the input `6 3 4 2 4 1 8` students indicate the output is `6 3 4` when in reality only the `4` will be displayed (Sleeman *et al.*, 1986, p. 13).

However, Sleeman *et al.* (1986) noticed that PASCAL novice programmers who had this misconception with FOR loops did not necessary have the same problem with WHILE loops and vice versa. The error appears to be consistent not with looping constructs in general but with a particular loop construct.

Data-driven looping - Here students believe the number of iterations is dependent on the number of data items to be read. A BASIC example is as follows:

```
10   FOR I = 1 TO 5
20   READ X
30   PRINT X
40   NEXT I
50   DATA 5, 8, 6, 3, 10, 11, 1, 25, 2
```

The following output was predicted by students: `5 8 6 3 10 11 25 2`. This program will only read and print the first five data values.

A variation of the data-driven looping is that students believe that the loop controls the number of times the process is repeated (the rows) and the number of data values determine the number of columns for each row. The Pascal example is as follows:

```
PROGRAM A2;
VAR I, X:INTEGER
BEGIN
  FOR I:= 1 TO 3 DO
  BEGIN
    WRITELN('Enter a number');
    READLN(X);
    WRITELN(X)
  END
END.
```

Given the input: `6 3 4 2 4 1 8`, students suggested the following output would result:

```
6 3 4 2 4 1 8
6 3 4 2 4 1 8
6 3 4 2 4 1 8
```

<u>Scope problems</u> - Three problems can be categorised as loop scope problems:

1. only the last instruction of the loop is executed multiple times. This misconception was only noticed by Sleeman *et al.* (1986) when the last instruction was a WRITE(LN) statement. The WRITELN instruction in the loop is executed the correct number of times, but all preceding loop instructions are only executed once.
2. BEGIN/END or indentation defines a loop in Pascal. For example:

   ```
   PROGRAM A3;
   VAR X:INTEGER;
   BEGIN
     WRITELN('Enter a number.');
     READLN(X);
     WRITELN(X)
   END.
   ```

   Given the input `6 3 4 2 4 1 8` students predicted that all the numbers in the data set would be printed out despite the absence of a looping construct (Sleeman *et al.*, 1986).
3. After a loop is executed control goes to the first statement of the program. This error occurred in "...short programs where the error could be interpreted as re-initialising variables each loop-cycle" (Sleeman *et al.*, 1986).

#### 2.8.1.5.2 Errors specific to FOR loops

Four errors were noted only in the context of FOR loops, these will be discussed in turn.

The control variable does not have a value inside the loop in the case of Pascal (Sleeman *et al.*, 1986).

The FOR loop statement acts as a constraint on the embedded READLN statement. For example (Sleeman *et al.*, 1986, p. 16):

```
PROGRAM A5;
VAR I, X:INTEGER;
BEGIN
  FOR I :=1 TO 3 DO
  BEGIN
    WRITELN('Enter a number.');
    READLN(X);
  END;
  WRITELN(X);
END.
```

Given the input `6  3  4  2  1  8`, students with this misconception believed that only values `3  2` and `1` would be read and displayed, as this is the range of the FOR loop control variable. This program will actual read the first three numbers and display the last value read in (`4`).

Other students believed that the FOR statement specified the number of times a variable's value would be displayed. Hence the above segment of code resulted in students predicting that each of the nine numbers in the data segment would be printed five times. The program would actually read in and print out the first five numbers only once.

A final FOR loop misconception involved the use of the control variable. Some Pascal programmers thought it was acceptable to change the value of the control variable within the loop. They also failed to realise that the control variable is simply a counter which is incremented with each iteration of the loop.

### 2.8.1.6 Conditional statements

If statements result in several misconceptions. All of the misconceptions seem to occur as a result of students failing to understand the flow of control of an IF statement, although most students understood the basic concept of a conditional (Putnam *et al.*, 1986).

Several errors were noted specifically when a conditional is false. These are outlined below.

When a conditional is false:

- program execution is halted if the condition is false and there is no ELSE branch (Sleeman *et al.*, 1986)
- execution of the entire program terminates (Putnam *et al.*, 1986)
- control is passed to the beginning of the program (Putnam *et al.*, 1986)

Both the THEN and the ELSE branches are executed (Sleeman *et al.*, 1986)

The THEN statement is executed whether or not the condition is true (Sleeman *et al.*, 1986)

When an IF statement results in a branch to a PRINT statement, both the variable and the value in the conditional expression are printed (Putnam *et al.*, 1986).

Example:

```
30    IF N = 0 THEN GOTO 60
...
60    PRINT SMALLEST
```

A student predicted the following output: `1  0`, where `1` is the value of the variable SMALLEST and `0` the value of the conditional. When the conditional in line 30 was changed to `N = -99`, the student predicted the output to be `1 -99`.

The statement directly after an IF statement without an ELSE branch is interpreted as the ELSE branch. Thus statements of the form `IF <a> THEN <b>; <c>;` is interpreted as `IF <a> THEN <b> ELSE <c>;`.

In writing conditional statements, students have greater difficulty with OR statements than AND statements, and the combination of the OR Boolean operator with a negative test expression results in high error frequency (Miller, 1974).

### 2.8.1.7 Procedure Misconceptions

Two misconceptions occur in procedures, both indicate a lack of understanding of flow of control. In the first, all statements, including those in procedures are executed in the order they appear in a top to bottom scan of the program. In the second, procedures are executed when they are encountered in a top to bottom scan of the code and again when they are called within the program.

### 2.8.1.8 Flow of Control

Putnam *et al.* (1986) found the following flow of control misunderstandings in their BASIC novice programmers.

All statements in a program must be executed at least once, even statements that might be skipped due to branching in the code. For example, one student when requested to trace the following program:

```
10    LET X = 1
20    LET Y = 2
30    IF X = 1 THEN GOTO 50
40    PRINT X
50    PRINT Y
60    END
```

correctly stated that the value of `Y` would be printed at line 50, but then incorrectly stated that because line 40 had been missed the computer would go back and execute line 40 before terminating.

Another similar misconception was that all PRINT statements in a program are executed.

Students with these flow of control misconceptions appear to be of the impression that all code in the program is there for a purpose and thus must be executed (Putnam *et al.*, 1986).

Generally, students experience difficulty with accurately following or predicting the flow of control of a program. Du Boulay (1986) observed that although students recognised that a program represents a sequence of instructions, they at times forgot that each instruction operates in an environment, or state, created by the previous instructions. Some students believe that the instructions are somehow executed all at once at the end of the program, while others fail to realise that the next instruction is always executed unless explicitly instructed otherwise (Du Boulay, 1986). Also, students using the divide and conquer design heuristic often fail to take into account the interaction between chunks of code (Soloway, 1982; Du Boulay, 1986).

### 2.8.1.9 Tracing and Debugging

This has been found to be one of the hardest programming tasks for novice programmers (Putnam, 1986; Sleeman, 1986). Generally students would infer the function of a program from a few statements and hence ignore or misinterpret lines of code that did not concur with their initial interpretation of the program (Putnam, 1986; Sleeman, 1986).
For example, students would incorrectly infer that the program below would find the smallest number of a set of numbers as a result of the dominance of the variable `Smallest` (Sleeman *et al.*, p. 17):

```
PROGRAM I1;
VAR smallest, Number: INTEGER;
BEGIN
  WRITELN('Enter a number:');
  READLN(Number);
  Smallest:= Number;
  WHILE Smallest <> 0 DO
  BEGIN
    IF Smallest > Number THEN
      Smallest:= Number;
    WRITELN('Enter a number:');
    READLN(Number);
  END;
  WRITELN(Smallest)
END.
```

In other situations, students would interpret the program based on what they consider to be reasonable output. Given the following code:

```
PROGRAM F2;
VAR number: INTEGER;
BEGIN
  WRITE('Enter a number:');
  READLN(Number);
  IF number = 7 THEN
    WRITELN('Unlucky number');
  IF number = 10 THEN
    WRITLEN('Lucky number');
  WRITELN('The Number was', Number)
END.
```

and the input `10` students correctly stated that the program would output '`Lucky Number`', but they then indicated that the program would terminate "...as there's no point in doing the next line as we know the value must be ten as it's a lucky number... " (Sleeman *et al.*, p. 18).

Also students would spend a lot of time debugging or understanding a particular section of the program, making assumptions about the other parts of the program. And lastly students had problems keeping track of the variables in a program. As mentioned earlier in this section, this could merely be a result of the complexity of the debugging or tracing task. However it appears that students fail to understand the rote behaviour of the computer in executing programs, and rather assume that the computer will act as a reasonable human, inferring or using intuition where necessary.

The following table serves to summarise programming constructs and their associated misconceptions discussed so far.

**Table 2.1: Summary of programming construct misconceptions and references**

| Programming construct and misconceptions | Reference and Frequency |
|---|---|
| 1. Input statements | |
| a) Semantically constrained reads | Sleeman *et al.* - frequent & consistent<br>Putnam *et al.* - frequent & consistent |
| b) Declaration order determines the order in which values are read into variables | Sleeman - fairly frequent & consistent |
| c) Alphabetic order of variables determines the order in which values are read into variables | Putnam *et al.* - occasional |
| d) Multiple-value variables | Sleeman *et al.* - frequent & consistent<br>Putnam *et al.* - frequent & consistent |

| | |
|---|---|
| 2. Output statements | |
| a) an output statement causes a number to be read | Sleeman *et al.* - occasional |
| b) output statement causes variable name and its value to be displayed | Sleeman *et al.* - occasional |
| c) after an output statement has been executed the program can choose a value from input | Sleeman *et al.* - occasional |
| d) Misinterpretation of statements of the form: `PRINT "C"` (BASIC) or `WRITE('C')` (Pascal) | Putnam *et al.* - occasional<br>Bayman & Mayer (1983) - 7% of students |
| e) Repeated print | Putnam *et al.* - occasional |
| f) Multiple-value print | Putnam *et al.* - occasional |
| 3. Variables | |
| a) Multiple-value variables | Sleeman *et al.* - frequent<br>Putnam *et al.* - frequent |
| b) Confusion of variables | Sleeman *et al.* - occasional<br>Putnam *et al.* - not quantified |
| c) Initial value maintained | Putnam *et al.* - not quantified |
| d) Printing of variables when variable : | |
| i) on LHS of expression | Sleeman *et al.* - occasional |
| ii) value changes | Sleeman *et al.* - occasional |
| 4. Assignment Statements | |
| a) reversal | Putnam *et al.* - occasional & inconsistent |
| b) comparison operator | Sleeman *et al.* - occasional |
| c) variables swap values | Sleeman *et al.* - occasional |
| d) instantiated variable's value is printed | Sleeman *et al.* - occasional<br>Bayman & Mayer (1983) - 7% |
| e) links variables together | Du Boulay (1986) - not quantified |
| f) no effect | Sleeman *et al.* - occasional |
| 5. Loop Statements | |
| a) output statement following loop included within scope of loop | Sleeman *et al.* - frequent<br>Putnam *et al.* - fairly frequent |
| b) data-driven looping | Sleeman *et al.* - several<br>Putnam - occasional |
| c) scope | |
| i) if the last instruction of a loop is a WRITE(LN), only this statement is executed multiple times | Sleeman *et al.* - occasional |
| ii) BEGIN/END or indentation defines a loop | Sleeman *et al.* - occasional |
| iii) after a loop is executed control goes to the first statement of the program | Sleeman *et al.* - occasional |
| d) Errors specific to FOR loops: | |
| i) control variable has no value | Sleeman *et al.* - fairly frequent |
| ii) control value acts as input constraint | Sleeman *et al.* - occasional<br>Putnam *et al.* - fairly frequent |
| iii) for loop specifies no. of times variables' values are displayed | Putnam *et al.* - occasional |
| iv) acceptable to change control variable's value within the loop | Putnam *et al.* - not quantified |

| | |
|---|---|
| 6. IF statements | |
| a) When a conditional is false | |
| i) program execution is halted if there is no ELSE branch | Sleeman *et al.* - occasional |
| ii) execution of the entire program terminates | Putnam *et al.* - occasional |
| iii)control is passed to the beginning of the program | Putnam *et al.* - occasional |
| b) Both THEN and ELSE branches are executed | Sleeman *et al.* - occasional |
| c) the THEN branch is always executed | Sleeman *et al.* - occasional |
| d) conditional value and variable value are displayed | Putnam *et al.* - occasional<br>Bayman & Mayer (1983) - 10% |
| e) statement following IF statement becomes ELSE branch | Sleeman *et al.* - occasional |
| 7. Procedures | |
| a) In order | Sleeman *et al.* - frequent |
| b) In order + call | Sleeman *et al.* - fairly frequent |
| 8. Flow of control | |
| a) all statements are executed | Putnam *et al.* - occasional |
| b) all prints executed | Putnam *et al.* - occasional |

Sleeman *et al.* (1986): Total population (35) Frequent = 25%+(8+ students), fairly frequent = 4-7, occasional = 1-3.

Putnam *et al.* (1986): Total population (56) Frequent = 25%+ (14+students), fairly frequent = 6-13, occasional = 1-5.

## 2.9 Chapter Summary

In this chapter, programming knowledge representations and the learning process involved in acquiring this knowledge was discussed. The acquisition of a mental model of the programming environment, and general programming plans were seen as critical in the development of programmer expertise. Novice programmers' misconceptions were discussed at a general and specific level and it was evident from this discussion that novice programmers have considerable difficulties with low-level programming knowledge. Most of the novice programmers never pass the semantic level understanding of programming languages in introductory programming courses. Students also have faulty mental models and acquire few general programming plans.

# 3. Strategies for Teaching Programming

## 3.1 Introduction

This section discusses several strategies for teaching programming so as to minimise the difficulties students experience while learning to program in a more effective manner. Teaching methods can make a difference. In exemplary programming classes, medium ability students do as well as high ability programmers, and better than medium ability students in typical programming classes (Linn & Dalbey, 1985). Shackelford & Badre (1993) go further. They argue that students experience difficulty in solving simple programming tasks not because of the difficulty of the problem, nor due to the required algorithmic thinking skills, nor the inherent difficulty of programming languages, but rather as a consequence of the consistently incomplete, and sometimes inaccurate, instructional treatment of language construct usage. Furthermore, it is not realistic to change the programming language environment, but one can realistically address the quality of the teaching strategy.

Generally it can be said that programming instruction must encourage understanding of concepts, i.e. meaningful learning, rather than rote learning, because programming involves the transfer of existing knowledge to new situations (Mayer, 1981). Students are expected to write novel programs, and consequently, students need to find a way of connecting the new information to existing knowledge.

## 3.2 Explicit instruction on programming misconceptions

Common sense might suggest that this approach would be the most effective. An experiment, undertaken by Stemler (1989), which looked at the effect of instruction on programmers misconceptions, resulted in disappointing results. The subjects were junior and senior school BASIC programming students. All students were given the same assignments, used the same textbook and received 55 minutes of instructional time per day for the semester. The control group, a class of 11 students, worked through the text book and completed all questions at the end of each chapter in addition to the class programming assignments and tests. The experimental group, two classes of 11 students each, received verbal instruction, which focused on assisting students in overcoming or avoiding misconceptions. In addition to the exercises at the end of each textbook chapter, students in the experimental group were also required to complete homework exercises of different natures: completing a program or predicting the output of short programs. At the end of the study there was no significant difference in the groups based on the misconceptions test. It was noted, however, that students who received explicit instruction on common flow of control misconceptions appeared to perform better in programming tasks. Stemler notes that this finding verifies a study (Sleeman & Gong, 1985)[16] in which it was reported that "many misconceptions could be remediated effectively through a combination of: (a) explicit training about the syntax and semantics of specific constructions in the programming languages, (b) requiring a learner to predict outcomes of short programs, and (c) providing students with interactive computer feedback" (Stemler, 1989, p. 31). Significantly Stemler noticed, however, that students in both the experimental and control groups "...had difficulty with reading and predicting the output of programs" (p. 32). Even those students who were relatively more efficient in generating correct programs had difficulty following the logic of a program that was not theirs. Stemler concluded that "...explicit instruction in the misconception areas was beneficial, but that many students still needed more experience with tracing programs and predicting the output" (p. 33).

## 3.3 Concrete models

The most popular suggestion for encouraging meaningful learning in programming is that of providing the students with a concrete model (Mayer, 1976; Bayman & Mayer, 1983; Lieberman, 1984). Mayer proposes that providing students with a concrete model gives them a set into which new information can be assimilated. Mayer (1976) found that students who were given a concrete model excelled in tasks requiring a moderate amount of transfer presumably because the model allowed the students to integrate or assimilate the new technical information to meaningful existing concepts. Concrete models have their

[16] Original source: SLEEMAN, D. & GONG, B. (1985). From clinical interviews to policy recommendations: A case study in high school programming. Stanford: Stanford university School of Education, March. (ERIC Document Reproduction Service NO. ED 257 415).

strongest effect in situations where learners are unlikely to possess useful prerequisite concepts. They are thus most useful when the material is unfamiliar, such as programming, and for low-ability or inexperienced students (Mayer, 1981). Furthermore through his empirical research, Mayer (1981), noted that for a concrete model to be effective, it must be available to the student prior to or during, as opposed to after, the learning process.

A further argument for the use of a concrete model is that providing students with a concrete model assists them in developing a more accurate and consistent mental model of the system. Even if learners are not given a view of the computer, students will form their own impressions which "...may be rather impoverished, relying on coincidence, and may be insufficient to explain much of the observed behaviour" (Du Boulay, 1986, p. 59).

## 3.4 The 'glass box' approach

The glass box approach is an example of a concrete model which enables the learner to 'see' what goes on inside the computer. Each command results in an observable change in the computer. The rationale is that if students can see what is happening inside the computer they will be able to develop more accurate mental models. However it is not necessary for the learner to be able to see all changes, only those necessary to assist the user in understanding the new concepts (Mayer, 1979; 1981). Mayer refers to the appropriate level of description, as the transaction level, where "a transaction consists of an action, an object, and a location in the computer" (Mayer, 1981, p. 137). For each programming statement there is a transaction, and thus Mayer suggests that the glass box should illustrate these transactions.

Du Boulay *et al.*'s (1981) notional machine also utilises the glass box approach. Their notional machine is an idealised model of the computer implied by the constructs of the programming language. Two key design principles of the notional machine are simplicity and visibility. The authors argue that a central difficulty in teaching novice programming is describing what the machine can be instructed to do or how it manages to do it. The notional machine addresses this difficulty by showing the learner certain of its workings in action. Seeing the internal workings of the programming commands allows students to encode information in a more coherent and useful way (Du Boulay, 1981), and thus they can develop a more accurate mental model. The glass box assists the student in learning the relation between a "program on the page and the mechanism it describes" (Du Boulay, 1986, p. 59). Although this concept is essential for the development of programming skills, it is a concept which is grasped only gradually (Du Boulay, 1986).

Goodwin and Sanati's (1986) Paslab learning package is a further example of the glass box approach. Essentially the package was designed with the intention of allowing students to understand what is happening inside the computer relative to statements in Pascal programs constructed by an expert. A comparison of final results obtained by students in the introductory programming courses who had and had not used the system, revealed that there was a change in the distribution of results. In the population of students who made use of Paslab, there was a sharp rise in students moving out of the failing category (11%, as opposed to 23% of non-Paslab students) and into the category of 'acceptable' performance (68% compared to 55%, but the percentage of students receiving a grade of 'distinction' did not alter (21%, compared to 22%). The inference drawn by Goodwin & Sanati (1986) was that the Paslab learning system helped students with lesser backgrounds but did not provide a 'boost' to higher ability students. These results concur with Mayer's (1981) findings that concrete models are more beneficial to low-ability students. A further observation made by Goodwin and Sanati (1986) was that the Paslab system changed the factors that affected the students' performances in the course. Under conventional teaching conditions, that is the non-Paslab conditions, background characteristics of students, such as previous computer experience, had a major impact on the final results those students received. Under Paslab conditions, the dominant factors shifted to the psychological characteristics of the student, such as motivational level. However the authors cautioned that these findings might be unique to technical institutions, and in particular the Worcester Polytechnic Institute, and thus need to be replicated in non-technical institutions. This is one of the objectives of this study.

## 3.5 Example Programs

Schemata or plans are an essential part of an expert programmer's domain knowledge. Novices need to acquire these schemata or plans to become proficient in programming. Most novices experience difficulty in abstracting plans from the limited examples they are given, although the most successful manage. Students are assisted in acquiring these general programming strategies by the use of numerous examples of similar problems (Jones, 1984). Not only is the inclusion of numerous examples beneficial to the development of organised domain knowledge, but students like the use of lots of examples (Dalbey & Linn, 1985).

## 3.6 Case Studies

Another technique, which has more recently been suggested by Linn and Clancy and colleagues, is the use of case studies for programming (Linn, 1992; Linn & Clancy, 1992a, 1992b; Schank *et al.*, 1993). Linn and Clancy offer an alternative to the more traditional syntax oriented organisation of most program instruction. They propose the use of case studies to assist students in the acquisition of program design skills, similar to those of an expert programmer.

Each of their case studies include:

- a statement of the programming problem
- a narrative description of the process used by an expert to solve the problem
- a listing of the expert's code
- study questions to provide practice in program design, problem solving and analysis,
- test questions to assess student's understanding of the program solution (Linn & Clancy, 1992b, p. 121).

Linn & Clancy (1992b) suggest that the expert narrative included in the case studies model expert program design skills by implementing eight principles of program design. These are listed below (1992a, 1992b):

1. The Recycling principle - reuse of ideas and templates: chunks of code or programming plans.
2. The Multiple Representation principle - consider multiple representations of each design template such as natural language, pseudecode etc. to attain a robust understanding of programming templates.
3. The Alternative Paths Principle - generate and evaluate alternative designs for problems

4. The Reflection Principle - reflect on alternative designs for programming problems and on problem-solving processes.
5. The Fingerprint Principle - develop effective debugging skills by associating symptoms of bugs with the appropriate bug.
6. The Divide-and-Conquer Principle - code and test complex programs a piece at a time to isolate program bugs and to reduce the cognitive demands of programming.
7. The Persecution Complex Principle - test for all possible weaknesses in the program using typical and extreme cases.
8. The Literacy Principle - produce code that is self-documenting so that it is easy to understand, modify, and debug.

Although the use of case studies has been found to improve pre-university students' design abilities, it is not an approach that can easily adopted by program instructors. Firstly, no case study design guidelines are discussed by the authors to assist instructors in developing their own case studies. Secondly, the development of reasonable case studies is time-consuming - the expert commentary was typically 20 or more pages long in Linn and Clancy's examples. Thirdly, students found the case studies difficult to read although the authors have addressed this issue, to some degree, through the use of on-line template libraries and hypermedia tools (Linn, 1992; Schank *et al.*, 1993). Nevertheless, this approach does appear to benefit students in acquiring program design skills.

## 3.7 Teaching of programming plans

Mayer (1981) postulates that teaching students programming schemata will assist in their understanding of programming as their knowledge can be more highly organised. Program schemata give statements a higher level meaning and alleviate some cognitive overload.

## 3.8 Putting technical information into own words

Mayer (1981) has suggested the technique of encouraging students to relate the material to a familiar situation, and more specifically getting them to put technical information into their own words. Although this strategy does not assist students in developing an accurate mental model of the programming environment, it does encourage students to assimilate the new knowledge, and consequently meaningful learning can occur (Linn and Clancy, 1992a, 1992b).

## 3.9 The computer as a teaching tool

Computers are seen as a useful instructional environment for programming because it has the capacity to provide the feedback needed to assess one's performance. Some examples of systems that have used computers in the teaching of programming are the Stanford BIP project (Barr *et al.*, 1976), Pascal Tutoring Aid (Doukidis *et al.*, 1989) and Paslab (Goodwin & Sanati, 1986). All address the teaching of programming to novice programmers.

A further advantage of the computer as a teaching tool is that it provides a mechanism for guided discovery. Dalbey & Linn (1985) suggest that guided discovery makes the learner responsible for gathering and using feedback while learning, and that this is advantageous as these are the skills that programmers ultimately require when designing their own programs. However, Dalbey & Linn caution that some computer control maybe necessary to develop all the skills necessary for programming.

Computer Assisted Learning tools are not uncommon although very few are implemented in programming courses, and generally the shift is towards intelligent tutoring systems. However, to be effective these systems tend to require substantial development time to be of tangible benefit to the students. Consequently these approaches are beyond the scope of this thesis.

## 3.10 Additional teaching strategies

These teaching strategies attempt to assist students in writing programs. The first deals with the appropriate selection of a looping construct, and the second puts forward the usefulness of providing novice programmers with stylistic guidelines to assist in the writing of their own programs.

### 3.10.1 Loops

Shackelford & Badre (1983) have developed the 'constructive use rule' for loop constructs which attempts to assist students in the selection of an appropriate loop construct.

The constructive use rule:

(a) if the value of the control variable is simply a count of the number of iterations, use a "FOR" loop.

(b) if the value of the control variable exists apart from the loop (you need only access it), use a "WHILE" loop;

(c) if the value of the control variable exists only after computation within the loop, use a "REPEAT" loop.

**Figure 3.1: Two decision rules for construct selection.** **(Shackelford & Badre, 1993, p. 988)**

They categorised the three Pascal loop constructs into order-influenced loop schema as shown below (Shackelford & Badre, 1993, p. 988):

```
FOR (test)            REPEAT                Get a value
BEGIN                   Get a value         WHILE (test)
  Get a value           Process value       BEGIN
  Process value       UNTIL (test)            Process value
END (for)                                     Get a value
                                            END (while)
```

Essentially the FOR and REPEAT loop constructs imply a 'Get-Process' ordering and the WHILE construct implies a 'Process-Get' ordering. They observed that students' preferred looping construct was the WHILE loop (44% of all loop implementations). In 75% of the WHILE loop implementations the unnatural 'Get-Process' ordering was used with a 21% success rate. When the WHILE loop was implemented with the more natural 'Process-Get' ordering, a success rate of 70% was noted. All other loop attempts had a success rate of 57%.

It appears that conventional program instruction does not assist students in selecting the most appropriate looping construct. Consequently there is an absence of any effective basis for decisions regarding the selection of loop constructs. However for novices it clearly matters which loop implementation is used. Subject performance improved dramatically when both the definition and the application of the WHILE loop were constrained and hence Shackelford & Badre (1983) suggest that the WHILE loop should be repositioned relative to the FOR and REPEAT constructs with each having comparable status with respect to application. This approach, which modifies the instructional treatment of the programming language, results in a significant improvement in novice programmers' performance.

### 3.10.2 Discourse Rules

Expert programmers also have knowledge about discourse rules, which are the stylistic conventions in programming, and which assist in the readability of programs (Letovsky, 1986). Joni & Soloway (1986) found that in 90% of the novice programs they analysed, some discourse rules had been violated. They suggest that instructors encourage students to develop good programming practices and develop a set of maxims and discourse rules to assist students in the acquisition of these practices.

## 3.11 Chapter Summary

Several strategies for dealing with effective program instruction have been suggested. All of which attempt to improve, with varying success, the acquisition of one or more types of programming knowledge.

# 4. The Design of the PATMAN Support Environment

## 4.1 Introduction

Several factors have been cause for concern in entry level programming courses taught at the University of Natal, Pietermaritzburg. These include the varying background characteristics of students, the high failure rates, and in particular the high failure rate of students from disadvantaged backgrounds, and the increasing student to lecturer ratios. These problems will be discussed in turn.

In the past, students enrolling in computer courses have had vastly different computer experience: some students are computer literate while others may have never seen a computer before enrolling in the course. This placed, and still does place, an additional difficulty on the instructor who must attempt to accommodate the needs of both groups. This problem still exists as students who have no experience are taught in the same lecture as student who have studied Computer Science as a matriculation subject.

Student failure rates have been alarmingly high. Prior to this research, on average fifty percent of  students failed the Introduction to Programming course. This was not unique to this course as other programming courses at the same university have failure rates above forty percent. Moreover, the majority of black students have failed programming courses. Goodwin and Sanati (1986) found that under traditional teaching methods, background characteristics were most influential in determining a student's success in the course, while under their experimental approach, which incorporated the use of the Paslab, they found that psychological factors, such as learning style were more important.

To further complicate matters, student to lecturer ratios are worsening and as a result, lecturers are not able to attend to individual student's needs adequately. This problem is likely to persist and possibly worsen as tertiary institutions battle for adequate financial support.

An additional resource that gives students greater control of their learning pace would be valuable under these circumstances. Furthermore a system that would minimise the role of background characteristics would be beneficial. This research is a first attempt at addressing these problems. It is particularly concerned with investigating the affects of a support environment on students' misconceptions and the individual characteristics that determine the students' success in the course. This research also seeks to determine whether the approach adopted in this research to solve these problems is adequate.

The system was developed using the 'glass box' approach as a mechanism for providing students with insightful examples. The effectiveness of this support environment in minimising student misconceptions and improving general programming ability is the focus of this study. The name given to the system is Patman, which is derived from Pascal Assistant Tutor.

## 4.2 The 'glass box' approach

In terms of novice programmers misconceptions, Pea (1986) has postulated that these errors occur as a result of the novice attributing the computer with human characteristics. A system that implements the glass box approach is not merely capable of showing the user the effects of the programming language constructs, but also the manner in which the computer executes commands. The glass box approach is favourable in that students are visually made aware of the step-by step nature of computers. This can be used to dispel the notion that computers are intuitive, or that more than one program construct can be active at any one time.

Many researchers have suggested the possibility of this approach and postulated its effect (Peele, 1975; Du Boulay, O'Shea & Monk, 1981; Allen, 1982; Lieberman, 1984; Goodwin & Sanati, 1986; Pea, 1986), as discussed under teaching strategies (see section 3.4). Some of these researchers comments are listed below.

- Du Boulay, O'Shea & Monk (1981) - "the use of a notational machine which is an idealised model of a computer implied by the constructs of the programming language"
- Allen (1982) - "schematic illustrations of a computer's action facilitate the learning of programming of skills"

- Lieberman (1984) - “Watching a program work step-by-step”...“greatly facilitates understanding of the internal workings of a program”.

## 4.3 Learning by example

Other researchers (Jones, 1984; Linn & Clancy, 1985, 1992a, 1992b, 1993) have suggested that learning by example can assist students in arranging their knowledge. Experts think about problem solutions abstractly in terms of plans or templates while novices think of actual code and in classification of code experiments, experts classify code according to the actions, while novices classify according to some superficial characteristic of the code. (Schneiderman, 1976; Adelson, 1981; Rist, 1986). Linn and Clancy found that by providing students with case studies, students were able to improve their sorting mechanisms. Linn and Clancy, in their research into the benefits of providing students with case studies noted seven principles. The benefits of two of these principles, the recycling principle and the multiple representation principle, were a result of giving complete program segments to students. The recycling principle refers to the fact that providing students with examples encourages students to use existing knowledge and templates rather than reinventing the wheel. The multiple representation principle involves showing different representations of the same templates to assist students in acquiring more robust ones.

## 4.4 Functional requirements

The following issues were considered in the development of the functional requirements of the system.

- The users are novice programmers with varying degrees of computer literacy and consequently the system should be easy to use without confounding or interfering with the learning process. The system also needs to be robust and intuitive as students would be using the system without the assistance of an experienced user.
- The users are required to use Turbo Pascal 5.0 as their programming environment. Therefore, to assist in the ease of use of the system the Turbo Pascal menu system and hotkeys ought to be mimicked to maximise transfer of knowledge about the use of the

one system to the other. The system output and error messages must be compatible with the Turbo Pascal 5.0 environment.

- Students have varying abilities, weak points and learning speeds and consequently the system should allow students to control their learning process.

## 4.5 The system model

The system is comprised of three components, as shown in Figure 4.1.

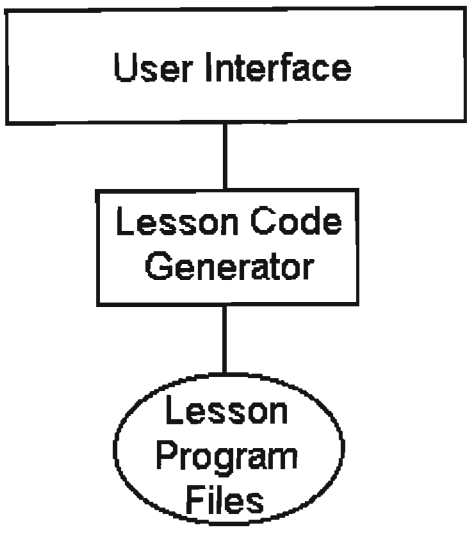


**Figure 4.1: System Diagram**

### 4.5.1 The user interface

The user interface is responsible for the dialogue with the user and is intended to provide a robust, simple, intuitive interface similar to that of the student's programming environment. The user may use the mouse or keyboard to select all menu options. Menu options that are not available in the current context are disabled. For example, the run menu option is disabled until the user opens a lesson as it would be inappropriate to run a program prior to it being open. One menu option that differs from the Turbo Pascal environment is the Next Lesson option, which is enabled once a student has opened a lesson. This was included in the system at the request of the students.

The 'glass box' approach is implemented through the use of four window areas which are visible once a student has opened a lesson. The program window contains the program code, the display window shows the output generated from the program, the input window shows the user input and the variable window displays all declared variables and their values.

### 4.5.2 The lesson code generator

The lesson code generator parses the selected lesson's program and generates a series of control statements when the lesson is initially selected. This generated code includes the code necessary to update the four windows and to control the flow of control. As the user steps or runs the selected program, the actions described in the generated code are performed. These actions are typically:

- create variable in variable window
- get user input
- display input value in input window
- edit variable value in variable window
- show output in display window
- move to program line x.
- remove local variables from variable window.

### 4.5.3 Lesson Program Files

The design of the programs formed the backbone of the system. The success or failure of the system, in reducing students' misconceptions, could be attributed in part to the design of the lesson program files. It was imperative to include programs that would force students to re-adjust an incorrect or inaccurate conceptual model of the programming and computer environment by presenting the students with programs that would contradict these inaccurate conceptual models. All the programs included in the system were written by the author with the intention of addressing at least one misconception per program. Often it was possible to address several misconceptions in one program. For example, the program below addresses the following misconceptions:

- Line 1 shows that a WRITELN output statement is different from a READLN input statement as only two numbers are accepted for input.

- Line 2 addresses variable concepts in general, and in particular the multiple value and semantic input misconceptions.
- Lines 3 and 4 address output statement misconceptions and variable concepts.

```
PROGRAM MultInt2;
VAR num_bigger_10,num_smaller_10:INTEGER;
BEGIN
  WRITELN('Enter 4 numbers: ');                            {line 1}
  READLN(num_smaller_10,num_bigger_10);                    {line 2}
  WRITELN('Numbers greater than 10 are ',num_bigger_10);{line 3}
  WRITELN('Numbers less than 10 are ',num_smaller_10);   {line 4}
END.
```

To prevent the overloading of concepts, in most instances the programs were restricted to deal with at most three major misconceptions.

Other considerations in the design of the programs were the illustration of:

- general algorithms or plans such as finding a maximum value (Soloway *et al.*, 1982; Joni & Soloway, 1986)
- the particular programming constructs in suitable problem solving situations (Soloway *et al.*, 1982)
- good programming principles (Joni & Soloway, 1986).

This said, it was sometimes necessary to develop code that made use of inappropriate program constructs, or that were not user friendly. For example the program discussed above violates several good program design principles. The program has an inappropriate user prompt and uses inappropriate variable names and produces incorrect output. However, the intention is for students to re-evaluate their perceptions of these statements based on the generated output.

Other programs required students to compare outputs from several programs. For example, similar programs were written, each using a different looping construct, in an attempt to illustrate the appropriate (or inappropriate) use of each looping construct. This is illustrated in Figure 4.2.

```
PROGRAM for_guess;
VAR mynum,yournum,counter:INTEGER;
BEGIN
  mynum:=13;
  WRITE('Guess the number I am thinking of!
');
  READLN(yournum);
  FOR counter:=9 TO 12 do BEGIN
    IF (yournum > mynum) THEN
      WRITE('Too High.. guess again: ');
    IF (yournum < mynum) THEN
      WRITE('Too Low.. guess again: ');
    READLN(yournum);
  END;
  IF (yournum = mynum) THEN
    WRITELN('You guessed correctly the
fourth time!')
  ELSE WRITELN('You failed to guess
correctly even after four guesses!');
END.
```

```
PROGRAM repeat_guess;
VAR mynum,yournum,try:INTEGER;
BEGIN
  mynum:=5;   try:=1;
  WRITE('Guess the number I am thinking of!
');
  READLN(yournum);
  REPEAT
    IF (yournum > mynum) THEN
       WRITE('Too High.. guess again: ');
    IF (yournum < mynum) THEN
       WRITE('Too Low.. guess again: ');
    IF yournum <> mynum THEN BEGIN
      try:=try+1;
      READLN(yournum);
    END;
  UNTIL (yournum = mynum);
  WRITE('You guessed my number in ',try,'
guesses.');
END.
```

```
PROGRAM While_guess;
VAR mynum,yournum,try:INTEGER;
BEGIN
  mynum:=18;   try:=1;
  WRITE('Guess the number I am thinking of!
');
  READLN(yournum);
  WHILE (yournum <> mynum) DO
  BEGIN
    IF (yournum > mynum) THEN
      WRITE('Too High.. guess again: ')
    ELSE
      WRITE('Too Low.. guess again: ');
    try:=try+1;
    READLN(yournum);
  END;
  WRITELN('You guessed my number in ',try,'
guesses');
END.
```

**Figure 4.2: Three implementations of the guessing game - each with a different looping construct**[17]

It was hoped that inappropriate use of a looping construct would result in students having to determine why the program had not behaved as expected, and thereby force the students to change their conceptual model of the looping constructs. These comparison programs also attempted to allow students to distinguish the difference between particular programming constructs and to learn general algorithms. The objective of showing students general algorithms, have been mentioned earlier in this work, namely aligning the novices knowledge structuring to be more closer to that of an expert and to facilitate the recycling principle.

A total of seventy one programs were included in the system[18]. The programming constructs included in the system were restricted to the following:

- Input constructs: READLN statement
- Output constructs: WRITE and WRITELN statements
- If statements
- Assignment statements

---

[17] To display the programs next to each other in the figure, some WRITELN statements have been wrapped around to the next line; in Patman these statements appear on one line.

[18] Appendix A contains a complete listing of all programs included in the system.

- Looping constructs: repeat, while and for loops
- Data Types: strings, integers, arrays and Boolean variables
- Procedures and functions with call-by-value and call-by-reference parameters and local and global variables

The programs used by Sleeman *et al.* (1986) in diagnosing student misconceptions were a valuable source of information for the development of the these programs. These programs formed the basis of several programs included in Patman.

## 4.6 Hardware

As the system was intended for novice programmers with minimal computer literacy, the system had to be user friendly. Furthermore, as the system was a prototype, the potential to expand the system into a fully fledged tutoring system for teaching novice programmers, was a design consideration. Windows, with its built in graphical user interface capabilities, was thus considered a desirable platform on which to build the system. The system was developed for a Windows platform using Turbo Pascal for Windows 1.5.

The minimal requirements of the system are:

- a computer capable of running Windows 3.x or higher
- 2 Mb hard drive space
- VGA capable monitor
- a mouse (optional)

## 4.7 Design of the system

The remainder of this section discusses a typical Patman session with the intention of illustrating design considerations and decisions.

The student initiates the learning process by selecting a lesson category (see Figure 4.3) and program.

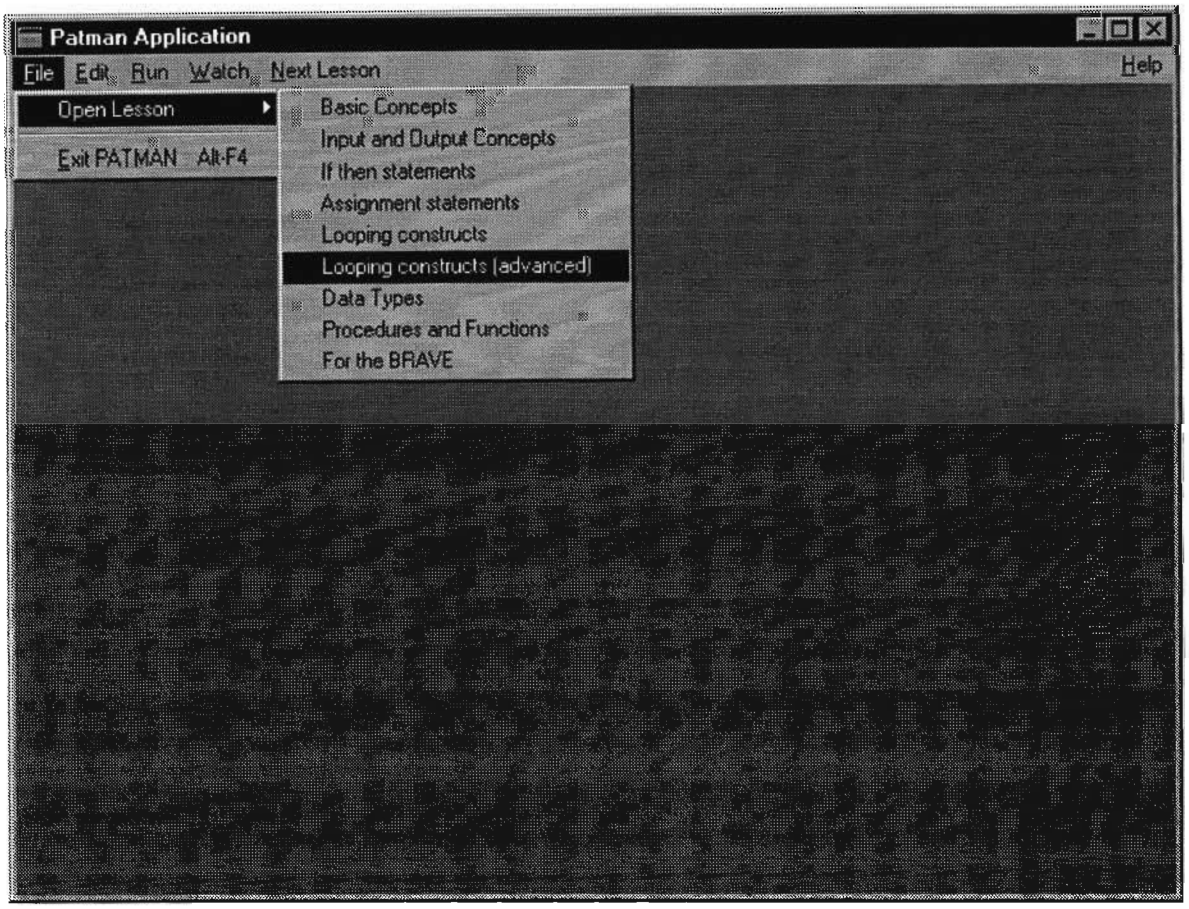
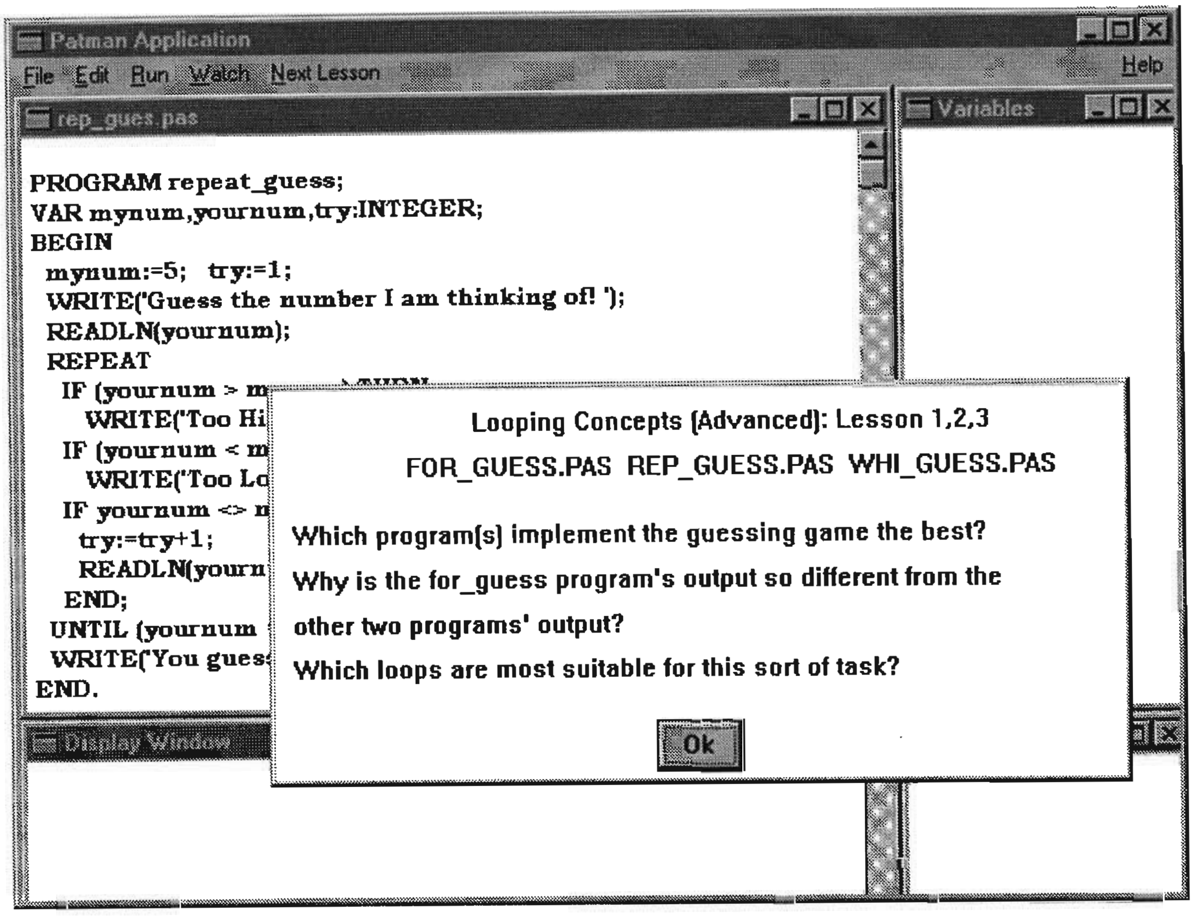


**Figure 4.3: Student selects a lesson category**

The selected program is then displayed in the program window. To encourage students to notice peculiarities and to benefit from the program's intended learning objectives, a comment dialogue is displayed before the student can commence with the lesson (Figure 4.4).



```
PROGRAM repeat_guess;
VAR mynum,yournum,try:INTEGER;
BEGIN
  mynum:=5;  try:=1;
  WRITE('Guess the number I am thinking of! ');
  READLN(yournum);
  REPEAT
   IF (yournum > m
     WRITE('Too Hi
   IF (yournum < m
     WRITE('Too Lo
   IF yournum <> n
    try:=try+1;
     READLN(yourn
   END;
  UNTIL (yournum
  WRITE('You gues
END.
```

Looping Concepts (Advanced): Lesson 1,2,3

FOR_GUESS.PAS REP_GUESS.PAS WHI_GUESS.PAS

Which program(s) implement the guessing game the best?

Why is the for_guess program's output so different from the other two programs' output?

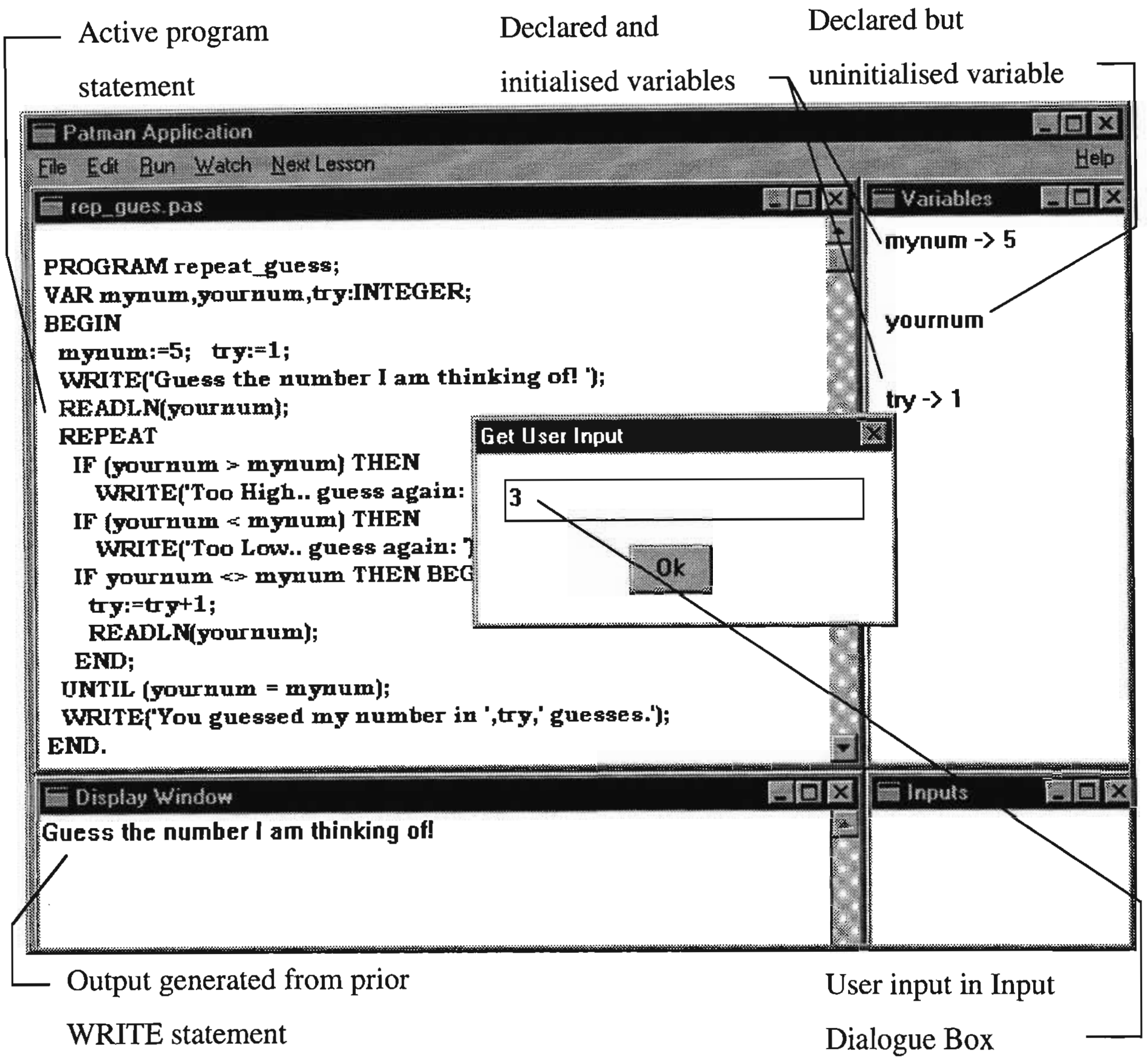
Which loops are most suitable for this sort of task?

**Figure 4.4: A comment dialogue box prompts the student to think about some issues**

These comments were designed to encourage students to note the difference between several program segments and to determine which programming construct is most suitable for the particular task. Often the comments were phrased in the form of a question and it was the responsibility of the student to answer the question by executing the program code.

The user can now open a new lesson, or step or run through the program code. When a student steps through the code they are required to select the step function to proceed to the next program statement. When a program has been completed and the user opts to step through the code again, a dialogue box appears indicating that the program has been completed and the code will be reset. This is in accordance with the Turbo Pascal environment. When a student selects the run option the program is automatically executed. During the execution of a program, the active program statement is highlighted using red text, to assist the students in following the flow of control through the program (see Figure

4.5). Whenever necessary, the other windows are updated to reflect the resultant action of the program statement. These statements will now be discussed in turn.


Active program statement
Declared and initialised variables
Declared but uninitialised variable
Patman Application
File Edit Run Watch Next Lesson Help
rep_gues.pas
Variables
Get User Input
Ok
Display Window
Inputs
Output generated from prior WRITE statement
User input in Input Dialogue Box

```
PROGRAM repeat_guess;
VAR mynum,yournum,try:INTEGER;
BEGIN
  mynum:=5;  try:=1;
  WRITE('Guess the number I am thinking of! ');
  READLN(yournum);
  REPEAT
    IF (yournum > mynum) THEN
       WRITE('Too High.. guess again:
    IF (yournum < mynum) THEN
       WRITE('Too Low.. guess again: ')
    IF yournum <> mynum THEN BEG
      try:=try+1;
      READLN(yournum);
    END;
  UNTIL (yournum = mynum);
  WRITE('You guessed my number in ',try,' guesses.');
END.
```

```
mynum -> 5
yournum
try -> 1
```

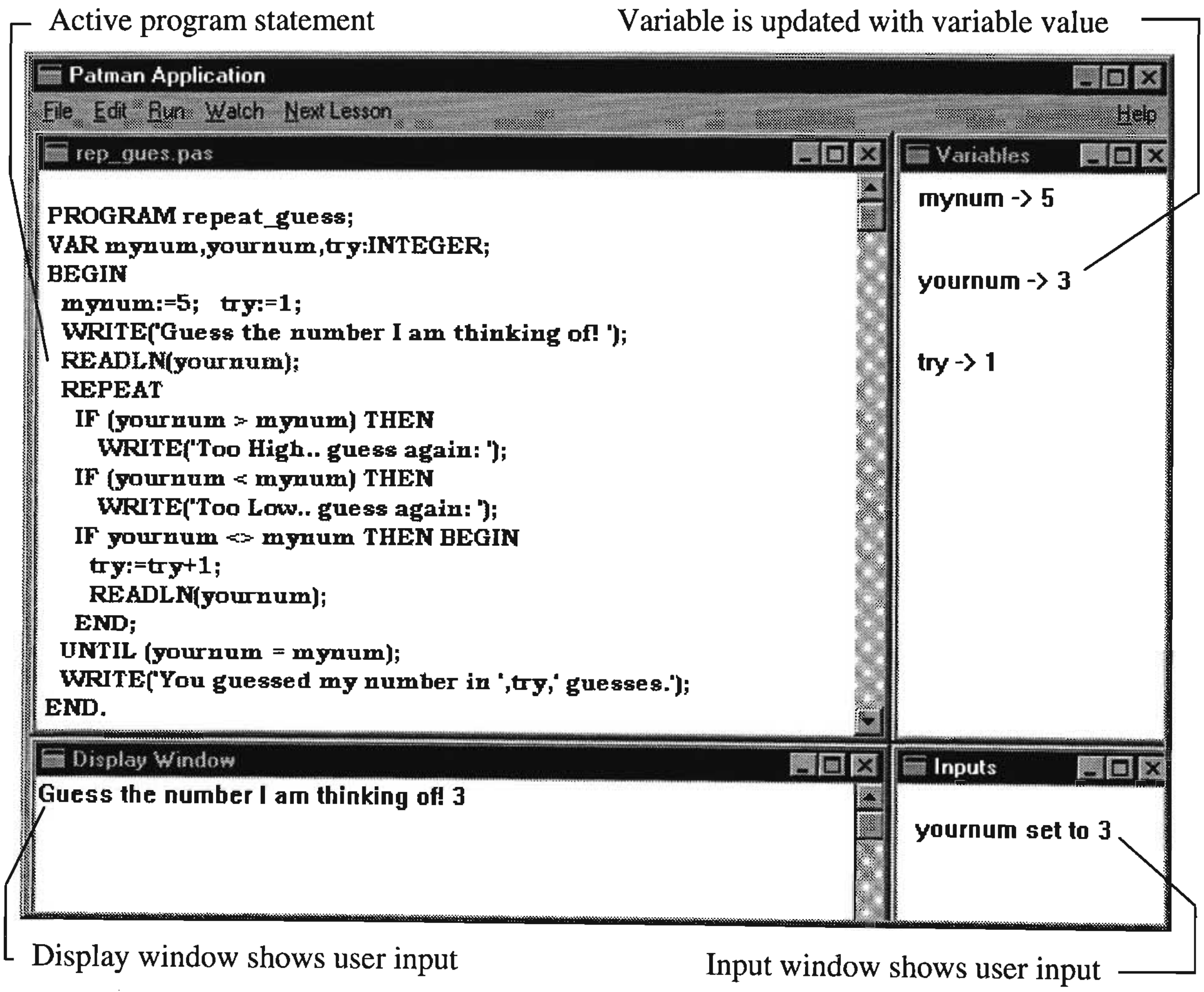
```
3
```

```
Guess the number I am thinking of!
```

**Figure 4.5: Stepping through the program - the READLN input statement is active.**

During the execution of a declaration statement, the declared variables are written in the variable window. All variables appear in the variable window based on the declaration order. During the execution of a declaration statement no value is associated with a variable (Figure 4.5). This was done to illustrate the need for the initialisation of variables. In one input lesson, a variable value is displayed before being assigned a value to further demonstrate this concept. It was decided to keep the variable value blank, rather than to associate some miscellaneous value with the variable to minimise confusion. Once the variable is given a value, an arrow (->) links the variable and its value. Parameters and local variables required special representation. This will be discussed later.

During the execution of an input statement, a series of actions are performed. Initially, an input dialogue box is displayed (Figure 4.5). Although this is different from the Turbo Pascal environment, this deviation was considered justified as it reinforces the distinction between input and output statements. All keyboard responses are restricted to this box which was designed to react in the same way as the default input environment of Turbo Pascal. To close the dialogue box students can use the OK button or the enter key. If the entered value is not of the required type, an error occurs and the program execution is terminated. If the user does not enter an adequate number of inputs the dialogue box reappears. This mimics the Turbo Pascal environment. If a valid variable value is entered, this value is written in three windows as illustrated in Figure 4.6.


Active program statement
Variable is updated with variable value
Patman Application
File Edit Run Watch Next Lesson Help
rep_gues.pas
Variables
Display Window
Inputs
Display window shows user input
Input window shows user input

```
PROGRAM repeat_guess;
VAR mynum,yournum,try:INTEGER;
BEGIN
  mynum:=5;  try:=1;
  WRITE('Guess the number I am thinking of! ');
  READLN(yournum);
  REPEAT
    IF (yournum > mynum) THEN
      WRITE('Too High.. guess again: ');
    IF (yournum < mynum) THEN
      WRITE('Too Low.. guess again: ');
    IF yournum <> mynum THEN BEGIN
      try:=try+1;
      READLN(yournum);
    END;
  UNTIL (yournum = mynum);
  WRITE('You guessed my number in ',try,' guesses.');
END.
```

```
mynum -> 5

yournum -> 3

try -> 1
```

```
Guess the number I am thinking of! 3
```

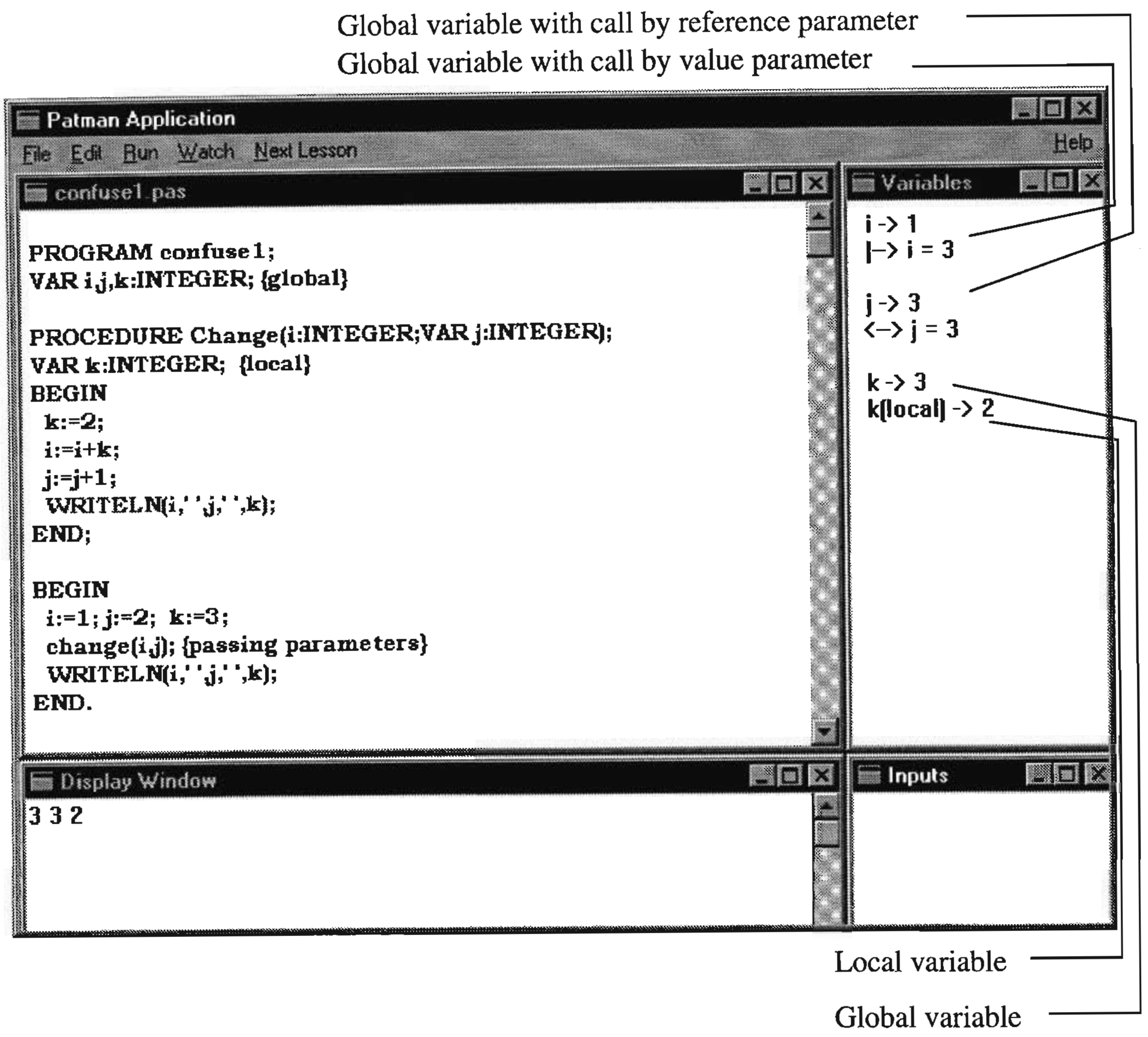
```
yournum set to 3
```

**Figure 4.6: Stepping through the code - after user has entered a value in dialogue box**

Firstly, the input window reflects the entered value. The input window was considered necessary to allow students to do a walk-through of the program and verify the resultant output. Secondly, the entered variable value is shown in the display window, this was done to coincide with the Turbo Pascal environment, in which the default input and output devices are shown in the same space. Lastly the entered value is shown in the variable window, in which the variable value is updated.

An output statement results in the output being shown in the display window, and an assignment statement results in the variable value being updated in the variable window.

Procedure and function parameters required special representation in the variable window as did local variables. Parameters are associated with their global variable in the variable window as shown in (Figure 4.7). Call-by-reference parameters are represented using the <-> symbol to reflect the two way interchange of data. Call-by-value parameters are represented by the | -> symbol to reflect the one way data exchange. Local variables are indicated by including the word 'local' in parenthesis after the variable name.


Global variable with call by reference parameter
Global variable with call by value parameter
Patman Application
File Edit Run Watch Next Lesson Help
confuse1.pas
Variables
Display Window
Inputs
Local variable
Global variable

```
PROGRAM confuse1;
VAR i,j,k:INTEGER; {global}

PROCEDURE Change(i:INTEGER;VAR j:INTEGER);
VAR k:INTEGER;  {local}
BEGIN
  k:=2;
  i:=i+k;
  j:=j+1;
  WRITELN(i,' ',j,' ',k);
END;

BEGIN
  i:=1; j:=2;  k:=3;
  change(i,j); {passing parameters}
  WRITELN(i,' ',j,' ',k);
END.
```

```
i -> 1
|-> i = 3

j -> 3
<-> j = 3

k -> 3
k[local] -> 2
```

```
3 3 2
```

**Figure 4.7: Variable window contains call by value and call by reference parameters as well as local and global variables**

Once the procedure or function has been completed these lines disappear from the variable window to indicate that they are only available within the scope of the procedure or function.

Other statements such as looping constructs, if statements, procedure and function calls, affect the flow of control and are thus visible through the sequence of highlighted statements. It is the responsibility of the student to notice this change in flow of control.

At any stage during the stepping through of a program, the user is able to select a menu option. However the user is not able to edit the program code, or the contents of the display, input and variable windows.

## 4.8 Extensions to the system

A possible extension to the system would have been to provide users with the ability to edit the program code or type in their own code. This would have required the inclusion of a compiler to check the syntax and semantics of the user generated code. In itself, this would not have presented a problem as Turbo Pascal provides a stand alone compiler. A more serious problem would have been the extension of the lesson parser to be able to cope with all Pascal statements. The extension of the system, although useful, was considered beyond the scope of the current research objectives.

A further extension to the system would have been to test students' understanding of concepts. Moreover, these test results could have directed students to additional lessons based on their apparent understanding of the tested concepts. In this case some of the user control would have been given to the computer. Although this could have been included in the system with relative ease, this was seen to be a confounding issue in the system as the research would have entered the realms of computer assisted tutoring which was not the intention of this research. However the inclusion of a testing mechanism to provide students with feedback and additional motivation would have been desirable.

# 5. Experimental Method

## 5.1 Introduction

Students who enrolled in the Introduction to Programming course, at the University of Natal , Pietermaritzburg in 1992 and 1993, were used as subjects for the experimental work. The Introduction to Programming students were considered to be suitable subjects for a number of reasons. They were taught Pascal programming and little else and no previous computer knowledge or experience was required for the course. Furthermore, the course was ideal for testing general principles of programming, as students from different faculties and of different years of study tended to enrol for the course. Students also had different motivations for enrolling in the course. Some students required the credit for their chosen major subject while other students enrolled for the course as they had a desire to learn to program. Moreover, the students had a mixed background of Pascal programming knowledge and were representative of a large spectrum of the University population in terms of their courses, faculties, age, year of study, race and gender.

Students who enrolled for the Introduction to Programming course of 1992, were monitored to determine their programming misconceptions. This data was used to determine if there were any misconceptions peculiar to the South African context, and more specifically, the course taught at the University of Natal. This data influenced the design of the Patman support environment, as the program examples included in Patman were designed, as part of this research, to accommodate these misconceptions.

A comparative experimental approach was decided upon to determine the effectiveness of Patman in reducing novice programmers' misconceptions and its ability to improve their general programming ability. A formal experimental method was planned, hence a control group and an experimental group were required. The control group would be exposed to the conventional teaching methods and would thus provide evidence of novice programmers' misconceptions under these teaching methods. The experimental group would be the same in all respects as the control group except that they would be allowed to make use of an additional resource, the Patman program. The two student groups would then be compared to determine if the Patman support environment provided any benefits to the experimental group. Students who enrolled for the Introduction to Programming class of 1993 formed these two groups, namely the control group (CONTROL '93) and the experimental group (PATMAN). While the primary reason for investigating the 92 student group (CONTROL '92) was to identify misconceptions for the design of Patman, additional data gathered could also be used to allow the comparison of:

- the CONTROL '92 and CONTROL '93 unsupported student groups
- the CONTROL '92 and PATMAN student groups
- the CONTROL '93 and PATMAN student groups

The first comparison is useful to determine the stability of the Introduction to Programming course under conventional teaching methods from year to year. The second comparison, although not of primary concern, provides additional information of the effectiveness of the Patman support environment. The third comparison, which is of primary concern, provides a clear indication of the performance of those students who made use of the additional resource, namely the Patman support environment, compared to those students who were taught using conventional teaching methods.

## 5.2 Teaching process

The Introduction to Programming course was taught during a 13 week semester. Students were required to attend two lectures and one tutorial per week. In addition, students were also required to submit a weekly programming assignment. Tutorials were more interactive than lectures, with approximately 30 students per group. During each tutorial, students were expected to complete an algorithm of the weeks' programming assignment. It was the responsibility of the tutor to assist students, usually on an individual basis, with any problems. Students later completed their programming assignment during their own time. To do this, students invariably made use of the campus computer laboratories, although a small percentage used home computers. The maintenance of hardware and software was the responsibility of the non-academic Computer Services Division (CSD) of the University of Natal, Pietermaritzburg. A Computer Science postgraduate student was also available to assist students while using the PCs.

Student assessment was broken down into two components, namely the class mark and the examination mark. The class mark was comprised of ten practical assignments and two class tests and contributed 30% towards the final student assessment. A two hour written examination at the end of the course formed the remaining 70% of the final mark.

The above course description and assessment applied to all students who participated in the experiment and in this thesis is considered to be the conventional method of teaching. Although the students who formed the experimental group were taught and assessed in the same manner as both control groups, they were also able to make use of the Patman support environment. Experimental group students used Patman for a maximum of one hour at a fixed time on a weekly basis, due to limited computer resources. During this time students were allowed to run any of the available lessons as many times as they liked. Although students were not prevented from interacting with fellow students, little interaction was noted during the sessions. Occasionally students would ask their neighbour for some assistance or comment on something they had discovered, but generally students appeared to be serious and deep in concentration while using Patman. Most students finished the lessons before the hour was up, but some students continued using the system for the full hour. It was also observed that most students ran the same lesson three or four times, while some students spent considerably longer on each lesson. On average, students used Patman for a total of 3 hours and 25 minutes during the eight available Patman sessions, or 35 minutes per session.

## 5.3 Allocation of students to groups

Of the 106 students who initially enrolled for the 1993 Introduction to Programming course, 50 students were randomly selected to attend the Patman sessions, but some reshuffling occurred to accommodate students' time table clashes. Students who did not attend any Patman sessions were considered part of the 1993 control group.

To assist in the evaluation of students' misconceptions and the effectiveness of the system, all students were required to complete 2 questionnaires and 3 worksheets during the semester.

Of the 50 selected Patman students, five students were dropped from the experimental group as they attended three or less Patman sessions. Two students did not complete the course and three students did not complete all necessary worksheets and questionnaires, and hence were also excluded from the experimental group. Of the original 56 students included in the CONTROL '93 group, 35 students completed the course and 33 students completed all worksheets and questionnaires. Thus, of the 83 students who completed the 1993 Introduction to Programming course, 40 students formed the PATMAN experimental group and 33 students formed the CONTROL '93 group. Of the 64 students who wrote the 1992 Introduction to Programming final examination, 61 students completed all worksheets and questionnaires and thus formed the CONTROL '92 group.

## 5.4 Testing and Evaluation Procedure

### 5.4.1 Determination of possible influencing background and psychological factors

Two questionnaires[19] were developed: one was completed during the first tutorial and the other a few weeks later once students had made use of the computers and completed a few practical assignments. The first questionnaire included items which would not be influenced by the course, such as the student's matriculation results, their preferred problem type, their previous exposure to computers. The second questionnaire mainly included psychological factors, such as the student's computer anxiety and alienation, their opinion of the helpfulness of tutorials, tutors, lectures, practical assignments and whether they had considered dropping the course.

The questionnaires were primarily based on Goodwin and colleagues experiments of programmer traits (Goodwin & Sanati, 1986; Goodwin & Wilkes, 1986) and Matta and Kern's (1989) literary review. Although programmer aptitude tests are traditionally unreliable (Calitz, 1984), an attempt was made to determine some programmer characteristics that might affect their success in the course. This data was required to

[19] The questionnaires are included in Appendix B.

determine if there was a substantial difference between the control and experimental groups.

### 5.4.2 Determination of the student misconceptions

Three worksheets were designed to determine the students' misconceptions[20]. These were based on the work done by Sleeman and colleagues' (Putnam *et al.*, 1986; Sleeman *et al.*, 1986) experimental work on programming misconceptions. All three worksheets were completed by students during their tutorial sessions, after 4 weeks, 9 weeks and 12 weeks of programming instruction. Although there was no time limit most students completed the worksheet within 20 minutes. The worksheets were administered once the relevant programming concepts had been lectured and after the students could have been expected to obtain a working knowledge of the concepts by completing the programming assignment dealing, either directly or indirectly, with the particular construct.

On completion of the worksheets, the responses were analysed by the experimenter to determine which misconceptions were evident. This data was used to determine the error proneness of programming constructs and whether there was a significant difference between the control and experimental groups in the frequency of misconceptions and also to determine if there were any misconceptions specific to the South African context.

### 5.4.3 Programming ability

The students' examination and final results for the introductory programming courses were used as a measure of the students' programming abilities. Both results were obtained independently of the experimenter and was the responsibility of the lecturers and external examiners. These measures of programming ability were used to determine any differences in the student groups, and hence the success of Patman.

[20] The worksheets are included in Appendix D.

## 5.5 Statistical tests used to evaluate the data

### 5.5.1 Z-proportion test

A Z-proportion test provides a mechanism to make statistical comparisons between two groups. In this thesis it has been used to answer questions of the form:

1. Is the proportion of students in the CONTROL '92 group who considered deregistering from the course different from the proportion of CONTROL '93 group's students who considered deregistering?
2. Did a larger proportion of the CONTROL '93 group's students experience a particular misconception than the PATMAN group's students?

As this statistical tool tests hypotheses about the difference between 2 population proportions, tests were required for each pair of data, namely:

- CONTROL '92 versus CONTROL '93
- CONTROL '92 versus PATMAN group
- CONTROL '93 versus PATMAN group.

When this test was used to compare the proportion of students who experienced a particular error, the control groups were compared to determine if the proportion of students were the same. However the experimental group was compared with each control group in turn, to determine whether a smaller proportion of the experimental group experienced a particular difficulty compared to the control group. This equates to questions of the form 2, whereas the control group questions are of the form 1.

In statistical terminology the above description can be phrased as follows.
The two control groups were analysed using the hypotheses:

$H_O$: $p_{\text{CONTROL '92}} = p_{\text{CONTROL '93}}$ where p = proportion of students with correct answers

$H_A$: $p_{\text{CONTROL '92}} \neq p_{\text{CONTROL '93}}$

The experimental group was separately compared to each control group using the hypotheses:

$H_O$: $p_{\text{PATMAN}} \geq p_{\text{CONTROL}}$ where p = proportion of students with correct answers

$H_A$: $p_{\text{PATMAN}} < p_{\text{CONTROL}}$

For hypothetical tests, such as the Z-proportion test, it is also necessary to determine an alpha level. An alpha level, represents the confidence level at which one wants to make the decision to reject the null hypothesis ($H_O$). If there is a notable difference in the populations being tested, an alpha level indicates how likely it is that the noted difference is actually due to chance. An alpha level of 0.05 indicates that there is a 5% likelihood that the proportions are actually from the same population, with an alpha level of 0.01 this likelihood is only 1%. Statistically, an alpha level of 0.05 is considered reasonable while with an alpha level of 0.01 one can be even more confident that the null hypothesis has been correctly rejected. In this research, both alpha levels are used as the groups were often found to be significantly different even at an alpha level of 0.01.

The decision to reject or accept the null hypotheses is based on the following decision rule:

$$\text{If } Z \leq Z_{\text{critical}}, \text{ accept } H_O.$$

$$\text{If } Z > Z_{\text{critical}}, \text{ reject } H_O.$$

For the control group comparison

$Z_{\text{critical}} = 1.96 \quad (\alpha = 0.05)$ $\qquad$ $Z_{\text{critical}} = 2.576 \ (\alpha = 0.01)$

and for the experimental versus control group comparisons

$Z_{\text{critical}} = 1.645 \ (\alpha = 0.05)$ $\qquad$ $Z_{\text{critical}} = 2.326 \ (\alpha = 0.01)$

### 5.5.2 Analysis of variance

Analysis of variance (ANOVA) is a statistical procedure used to examine the variation in populations to determine whether the populations are equal. In this research it was used to answer questions of the form:

- Are the mean final results for each student group different?

Once again, it is necessary to select an alpha level to determine the necessary confidence level of the decision. A further indication of the likelihood of the noted differences actually being due to chance is the probability factor. A probability factor (p) of 0.001 indicates that there is only a one tenth percent of a chance that the results are due to chance. Any value of $p < 0.01$ is considered reasonable.

In statistical terms the null and alternative hypotheses were:

$H_O$: $\mu_{CONTROL\ '92} = \mu_{CONTROL\ '93} = \mu_{PATMAN}$ where $\mu_i$ = mean of student group $i$

$H_A$: not all means are equal

### 5.5.3 Scheffé's multiple comparison method

Once an ANOVA test has indicated that the means of several populations are sufficiently different, Scheffé's method can be used to determine which populations are different. This is done on a pairwise comparison basis, and hence three comparisons were required as was the case for the Z-proportion test.

Analysis of variance analysis and Scheffé's multiple comparison method together provide a way of determining whether there are any significant differences between groups' means. The Z-proportion test is a method to determine whether there are any significant differences between groups' proportions.

### 5.5.4 The multiple regression model

In multiple regression, a set of independent variables are used to model a dependent variable. In this thesis multiple regression was used to determine those background and psychological factors that influenced a student's success in the programming course. For each student group the final result and examination results were independently modelled based on the set of independent variables, or predictors, which were obtained from the questionnaires and included such factors as the students' matriculation results, learning style and previous experience with computers.

In experiments of this nature, the possibility exists that some attribute of one or more student groups, which is external to the experimental process and which account for the observed difference in the student groups may apply. This is of particular concern. Since the groups in this experiment were randomly allocated, it is possible that there could be some significant difference in one or more of the student groups' psychological and background characteristics. For each student group, multiple regression modelling was used to determine which of these student characteristics were influential in determining a student's success in the course in terms of their final and examination results. It was then possible to determine whether there were any significant differences between the groups in terms of these influencing characteristics.

# 6. Results and Discussion

The results of this investigation will be presented in five separate sections. The first three correspond to the three methods used for obtaining them. This is followed by a comparison of the demographics of the student groups and, finally, the student's evaluation of the support environment itself.

## 6.1 Analysis of Correct Responses for Worksheet Questions

This section compares the proportion of students in the three groups who correctly answered each worksheet question. This comparison facilitates an evaluation of the Patman support environment. If the proportion of PATMAN's students who correctly answer a question is greater than the proportion of control group students who do so, the difference can be attributed to the support environment. It would also be an indication that the support environment was successful in improving students' acquisition of programming knowledge.

A student response to a worksheet question was considered to be correct if the student answered the question exactly as required. The majority of the worksheet questions, eleven out of fourteen, required the student to indicate the output of the program code. In these questions, there were lots of opportunities for students to make careless mistakes. Only exact answers were accepted. If the student had carelessly written output on the same line instead of on separate lines, the answer was not considered correct. Likewise, if the student used the incorrect input values, but everything else was correct, the response was considered incorrect.

For each question the proportion of correct responses were analysed using a Z-proportion test which tested the hypotheses about the difference between two population proportions. The results of the three worksheets are discussed separately.

### 6.1.1 Worksheet 1

Worksheet 1 was administered in the fourth week of the semester after the experimental group students had spent a mean time of 33 minutes on Patman. Figure 6.1 shows the percentage of correct responses for each question per student group. The percentage of correct responses for the experimental group was consistently higher than either control group. Statistically, the proportion of correct responses for the experimental group was significantly higher than the CONTROL '92 group for Question 1 ($\alpha = 0.01$), Question 2 ($\alpha = 0.05$) and Question 4 ($\alpha = 0.01$). The proportion of correct responses for the experimental group was significantly higher than those of the CONTROL '93 group for Question 3 ($\alpha = 0.05$).

Although the results appear to indicate that the CONTROL '92 group was worse than the CONTROL '93 group for questions 1, 2 and 4, statistically there was no significant difference between the control groups for all four questions.

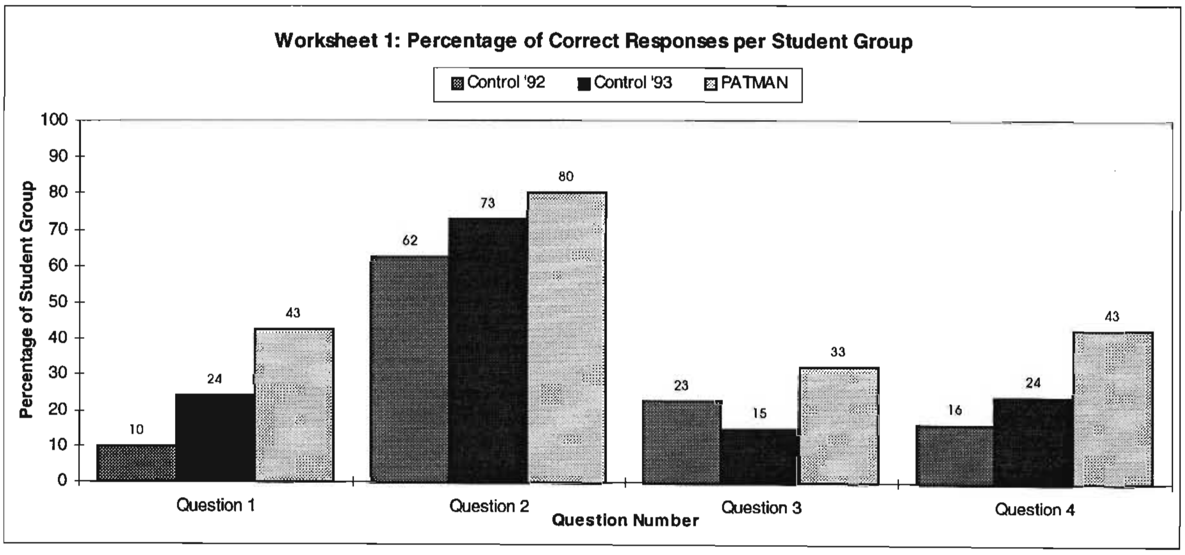


**Figure 6.1: Worksheet 1 - Percentage of correct responses**

The total number of correct responses per student for worksheet one are reflected in Table 6.1

**Table 6.1: Worksheet 1 - Total number of correct responses per student**

| Number of Correct Answers | Percentage of Student Group | | |
|---|---|---|---|
| | Control '92 | Control '93 | PATMAN |
| 0 | 31 | 21 | 10 |
| 1 | 41 | 48 | 23 |
| 2 | 16 | 9 | 40 |
| 3 | 8 | 15 | 15 |
| 4 | 3 | 6 | 13 |
| **Mean** | **1.11** | **1.32** | **1.98** |

It should be noted that more than 50% of the experimental group correctly answered 2 or more questions, in contrast the majority of the control groups' students answered at most 1 question correctly. Analysis of variance and Scheffé's method indicated that there was a significant difference ($\alpha$=0.01, p =0.00093) between the CONTROL '92 and PATMAN groups' means.

## 6.1.2 Worksheet 2

Worksheet 2 was administered in the ninth week of the semester after the experimental group had spent a mean time of 1 hour and 58 minutes on the system. The percentage of correct responses for each question are shown in Figure 6.2.

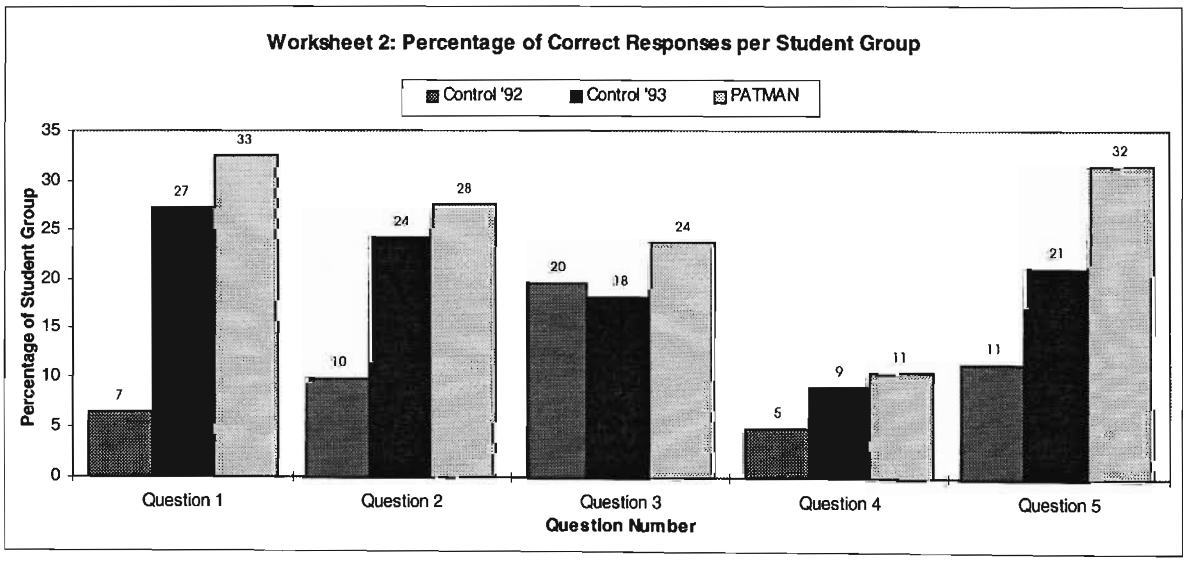


**Figure 6.2: Worksheet 2 - Percentage of Correct Responses**

The proportion of correct responses for the experimental group was consistently higher than both of the control groups. Statistically, there was no significant difference between the CONTROL '93 group and the PATMAN group, however the proportion of correct responses for the experimental group was significantly higher than the CONTROL '92 group for Question 1 ($\alpha = 0.01$), Question 2 ($\alpha = 0.05$) and Question 5 ($\alpha = 0.01$). There was no significant difference between the three student groups for Questions 3 and 4.

In general, the students performed poorly on this worksheet. The lowest percentage of correct responses was achieved on question 4. This question included an if statement within a while statement, and required considerable tracing expertise. Only 5%, 9% and 11% of the CONTROL '92, CONTROL '93 and PATMAN groups respectively correctly answered this question. The percentage of correct responses for Question 1, which dealt with assignment statements, was unexplainably low for the CONTROL '92 group (7%) compared to the other two student groups. In addition to being significantly lower than the PATMAN group's 33% it was also significantly different ($\alpha = 0.01$) from the CONTROL '93 group's 27%.

The total number of correct responses per student on this worksheet are shown in Figure 6.2.

**Table 6.2: Worksheet 2 - Total number of correct responses per student**

| | Percentage of Student Group | | |
|---|---|---|---|
| Number of Correct Answers | Control '92 | Control '93 | PATMAN |
| 0 | 70 | 52 | 50 |
| 1 | 15 | 21 | 15 |
| 2 | 8 | 12 | 8 |
| 3 | 5 | 6 | 18 |
| 4 | 2 | 9 | 10 |
| 5 | 0 | 0 | 0 |
| **Mean** | **0.52** | **0.97** | **1.23** |

Analysis of variance and Scheffé's method revealed that there was a significant difference ($\alpha = 0.05$, P = 0.001) between the CONTROL '92 and PATMAN groups' means.

## 6.1.3 Worksheet 3

Worksheet 3 was administered in the twelfth week of the semester after students had spent a mean time of 3 hours and 25 minutes on Patman. The percentage of correct responses per question for worksheet 3 are reflected in Figure 6.3.

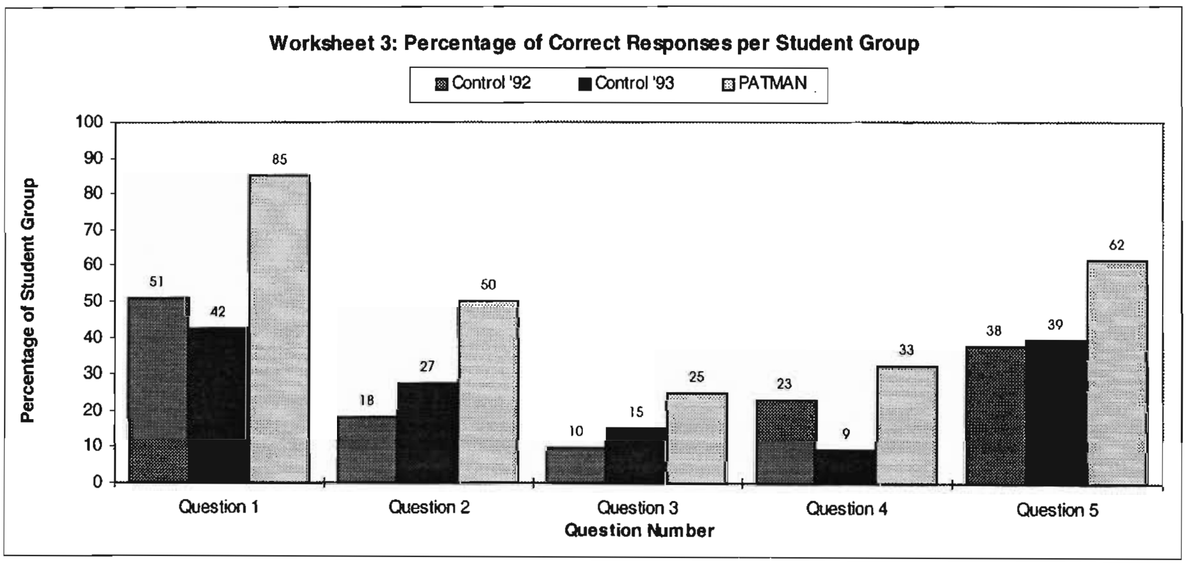


**Figure 6.3: Worksheet 3 - Percentage of Correct Responses**

Again, the proportion of correct responses for the experimental group was consistently higher than both of the control groups for all questions. The PATMAN group was significantly better than the CONTROL '92 group on Question 1 ($\alpha = 0.01$), Question 2 ($\alpha = 0.01$), Question 3 ($\alpha = 0.05$) and Question 5 ($\alpha = 0.01$). The experimental group was also better than the CONTROL '93 group on Question 1 ($\alpha = 0.01$), Question 2 ($\alpha = 0.05$), Question 4 ($\alpha = 0.01$) and Question 5 ($\alpha = 0.05$). There was no significant difference between the control groups.

The largest percentage of correct responses was achieved for Question 1. The '92 and '93 control groups achieved 51% and 42% success respectively, compared to the experimental group's success of 85%.

Based on the total number of correct responses per student, students generally did considerably better on worksheet 3 compared to the previous worksheets. See Figure 6.3.

**Table 6.3: Worksheet 3 - Total number of correct responses per student**

| | Percentage of Student Group | | |
|---|---|---|---|
| Number of Correct Answers | Control '92 | Control '93 | PATMAN |
| 0 | 38 | 36 | 13 |
| 1 | 23 | 24 | 8 |
| 2 | 13 | 18 | 33 |
| 3 | 18 | 15 | 20 |
| 4 | 5 | 3 | 18 |
| 5 | 3 | 3 | 10 |
| **Mean** | **1.39** | **1.29** | **2.53** |

Analysis of variance and Scheffé's method indicated that there was a significant difference ($\alpha = 0.01$, $P < 0.001$) between PATMAN and both control groups' means.

### 6.1.4 Summary of Results

The results discussed in this section are summarised below.

In terms of the proportion of correct responses per student group, the experimental group was consistently higher than both control groups for all fourteen worksheet questions. Compared to the CONTROL '92 group, the experimental group proportion was significantly better for ten questions and compared to the CONTROL '93 group it was significantly better for four questions. The control groups were significantly different for one question.

In terms of the mean number of correct responses per student for each worksheet, the experimental group was better than both control groups for all three worksheets. The PATMAN group was significantly better than the CONTROL '92 group for worksheets one and two, and significantly better than both control groups for worksheet 3.

It can be concluded from these results that there was a noticeable difference between the two control groups and the experimental groups, in terms of the correct responses for worksheet question. Hence, it can be said that the Patman support environment assisted the experimental group students in acquiring programming knowledge.

## 6.2 Analysis of Examination and Final results by Student Group

In many respects the worksheet questions can be considered an unfair representation of a student's ability to program, especially as the questions were designed to trap the student's errors. In the context of this research, which aimed to determine if common misconceptions noted in novice programmers could be minimised through the use of Patman, it was appropriate. However it was not the only aim of this research. A further objective was to determine if the Patman system could assist students in becoming better programmers.

Examination and final results were used as an indication of a student's general programming ability. The examination process was independent of this research. The lecturers of the course set the examination and marked the students' scripts, overseen by an external examiner. It was the responsibility of the external examiner to ensure a reasonable standard was applied and that the scripts were marked fairly. Thus it is assumed that the examination results were a reasonable representation of the students' programming abilities. The examination and final course results are the focus of this section.

### 6.2.1 Analysis of Examination Results

A histogram of the examination results are shown in Figure 6.4. The mean percentages acquired in the examinations were 48.58%, 55.21% and 68.40% for the CONTROL '92, CONTROL '93 and PATMAN groups respectively.

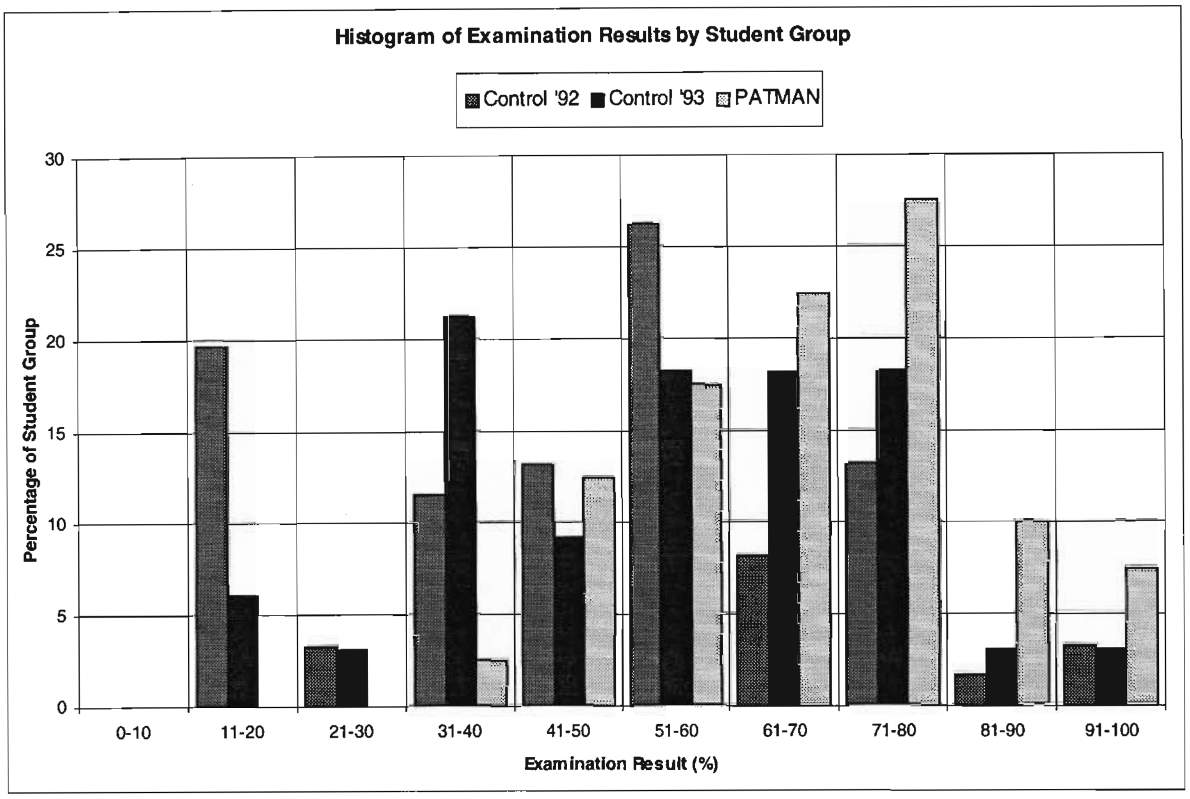


**Figure 6.4: Examination Results by Student Group**

Using ANOVA and Scheffé's method, the PATMAN group was found to be significantly higher than the CONTROL '92 group ($\alpha = 0.01$, $p < 0.001$) and the CONTROL '93 ($\alpha = 0.05$, $p < 0.001$) group. However, as a result of the large proportion of control '92 group students in the range 11-20%, the assumption of normality could not readily be assumed. The examination results were then tested using a Kruskal-Wallis one way analysis of variance test. Using this test the student groups were still found to be significantly different ($p < 0.001$).

The percentage of students who failed the examination i.e. obtained a result less than 50, are also noteworthy: 46% of the CONTROL '92 group and 39% of the CONTROL '93 group failed, as opposed to the 8% of the experimental group.

### 6.2.2 Analysis of Final Course Results

The final results were calculated by including

- the two test marks, which each counted 9% percent,

- the weekly practicals, which contributed 12% percent, and
- the examination result.

The final result gives an impression of a student's overall understanding of the course throughout the semester. The final results are depicted in Figure 6.5.

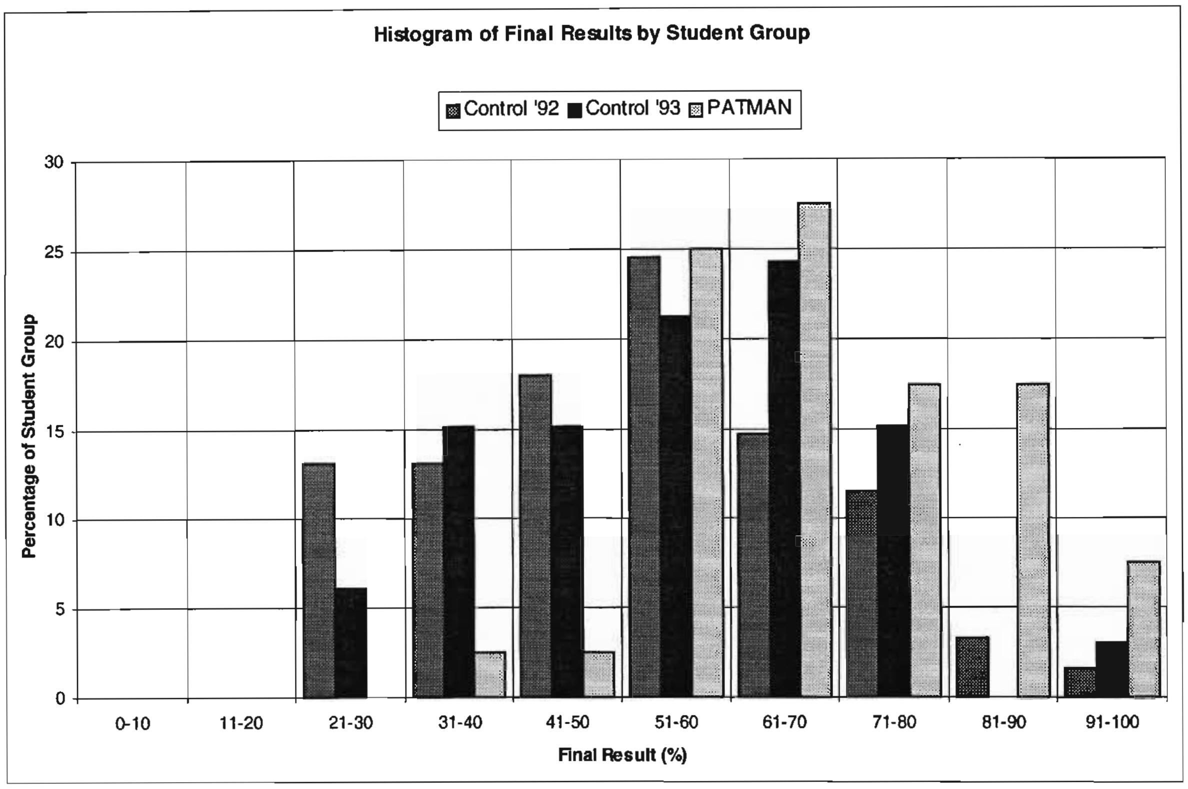


**Figure 6.5: Final Results by Student Group**

The average results for the three student groups were 53.70%, 55.47% and 68.39% respectively. Again, using an ANOVA test and Scheffé's method, the PATMAN groups final results were found to be significantly higher ($\alpha = 0.01$, $p < 0.001$) than either control groups' results. The percentage of students who failed were as follows: 33% of both control groups and 5% of the experimental group.

### 6.2.3 Summary of Results

From the analysis of the examination and final results it can be concluded that the experimental group students benefited significantly from using the PATMAN system. The experimental group results were considerably higher than both control groups for the examination and the overall course. The percentages of PATMAN students who failed the examination and course were significantly lower than both CONTROL groups. This confers with Goodwin and Sanati's (1986) finding that support environments of this nature assist students with disadvantaged backgrounds. There was no significant differences between the control groups with respect to examination and final course results.

The previous two sections have examined the difference between the experimental and control groups with regard to the final and examination results and the number of correct responses on the worksheet questions. In all cases, the experimental group was considerably better than either control group. This indicates that the PATMAN support environment was successful in assisting novices in acquiring general programming knowledge. The following section examines the differences in student groups in terms of the programming construct misconceptions to determine whether this improvement in the PATMAN students' performance is as a result of a reduction in misconceptions.

## 6.3 Analysis of Misconceptions

Student responses to all 14 worksheet questions were analysed to categorise all errors and misconceptions. Where necessary, students were interviewed to fully understand their thought processes and to enable accurate categorisation of their problems. These interviews reinforced Pea's (1986) findings in many respects. Students frequently attributed human characteristics to the computer and that they also frequently interpreted the simplest constructs as conceptually complex.

Initially all misconceptions noted in Sleeman *et al.*'s (1986) and Putnam *et al.*'s (1986) research were used as possible errors, however it soon became evident that students were making additional errors consistently. These errors are included in the data presented in this chapter, although it must be noted that these errors are not necessarily deep conceptual errors. Some might simply be a result of carelessness.

To minimise experimenter bias, the categorisation of errors was done without reference to the students. All student responses were analysed using the same error categories. This was done on a question by question basis. Thereafter, error categories were grouped together and a total per student was obtained. At this stage, each student had a value for each error category. A value of zero indicated that no occurrences of that error had been noted. Finally, a error percentage was obtained for the three student groups: CONTROL '92, CONTROL '93 and PATMAN. Whenever results are given in the text they appear in this order.

The data presented in the Tables in this section are all percentages to assist comparison of the student groups by the reader. These percentages were obtained by dividing the value of each error category by the number of students in each student group, multiplied by the number of times that error was noted. For example, a misconception which occurred frequently, was the multiple valued variable misconception. This error appeared in response to three questions. Fifty nine occurrences were noted for the CONTROL '92 group. The percentage represented in this section (see Table 6.4), was calculated with the numerator of 59, and the denominator of 3 multiplied by 61, the size of the group.

It should be noted that in the Tables misconceptions and errors are grouped together. Although Patman was designed with the intention of minimising the deeper underlying misconceptions, surface level errors have also been included. This was decided upon firstly to present a complete analysis of the student responses, and secondly it was sometimes difficult to categorise student errors. If the experimenter was in any doubt regarding the categorisation of the student error, the surface level error was recorded. These errors might also be of interest to anyone who is an instructor of programming, and in particular Pascal programming. For the remainder of this section, misconceptions and errors are more generally referred to as errors, unless the distinction is significant. Furthermore, the errors tabulated for each construct are presented in descending order of frequency for the whole population. Whenever it was feasible to calculate the percentage of students who correctly understood a particular programming construct, these percentages have been included as the last entry in the Table.

All errors are presented and discussed by programming constructs: output statements, input statements, variable concepts, assignment statements, looping constructs, if statements and procedures. In each section, actual student responses are presented to illustrate the noted errors. This is followed by a discussion of previous research in the field. Where appropriate, differences between the proportion of students in each student group who experienced a particular misconception are noted.

### 6.3.1 Output statements

The output statement errors were the most difficult to categorise. In many instances the experimenter had to resort to categorising errors as 'incorrect output statement'. Furthermore, in the initial worksheet questions, some output errors were so severe that they indicated a complete lack of understanding of output statements. The were labelled 'no understanding of output statements'. To illustrate the noted misconceptions and errors associated with output statements, the following program (worksheet 1 - question 3) will be used[21].

```
PROGRAM three;
VAR x: INTEGER;
BEGIN
  WRITELN('Enter a number.');
  READLN(x);
  WRITELN(x);
  WRITELN('The value of x is 5');
  WRITELN(x);
END.
```

This worksheet question was designed specifically to reveal students misconceptions with output statements. Given the following input: 6 3 4 2 4 1 8 students were requested to specify the output of the program when executed.

Correct Output:

```
Enter a number.
6
The value of x is 5
6
```

To illustrate the errors actual student responses will be given and discussed.

Student Response 1:

[21] For relative percentages and significance levels of the output statement misconceptions see Table 6.4.

```
6
THE VALUE OF X IS 5
6
```

This student omitted the output generated from the first output statement, namely

```
WRITELN('Enter a number.');
```

Statements of this form are usually included in programs to provide the user with instructions about what input is required of them. This error was categorised as the 'no Enter' output error, which occurred frequently in all student groups. Invariably, user-friendly statements of this form are followed by an input statement, as is the case here. In the example above, the student appears to have interpreted the WRITELN and READLN statement as one input statement. This supports Sleeman *et al.*'s (1986) finding that students believed output statements of this form caused a number to be read. Similarly, WRITELN('Enter 4 numbers:') caused four numbers to be read. This error was found to be a persistent error, as it was evident in all three worksheets.

Student Response 2:

```
Please enter the number
6,3,4,2,4,18
The value of the number is 5
```

This student was categorised as having several errors. Firstly the student illustrated that he had no understanding of output statements as he haphazardly changed the contents of the output statements. The student also did not display the output of the last write statement, this error is classified as 'no output after WRITELN('The value of x is 5');' in Table 6.4. Although this error claimed the highest proportion of all students, this error is not a consistent error as it was peculiar to this program. It appears that students interpreted the previous output statement as constraining all future output. This error was documented by Sleeman *et al.* (1986) and Putnam *et al.* (1986). The final error noted in this program was the 'multiple valued variable' misconception. This will be further discussed under the variable misconceptions section.

Student Response 3:

```
The value of 6 3 4 2 4 1 8 is 5
```

This student was categorised into the 'no Enter' error category and the 'no output after WRITELN('The value of x is 5');'. These errors have been discussed above. The student also substituted the value of variable x for the character x in the output statement, thus indicating that he had difficulty discerning between output statements of the form `WRITELN('x')` and `WRITELN(x)`. Statements of the latter form indicate that the value of variable x must be displayed; statements of the former form indicate that everything between quotation marks must be displayed exactly as stipulated. This error was found to be an occasional error by Sleeman *et al.* (1986) and Putnam *et al.* (1986). Furthermore the student illustrated a general misunderstanding of output statements as he omitted additional output statements and was hence included in the 'incorrect output statement' category. Lastly the student indicated that 'multiple valued variables' are possible. This error will be further discussed with other variable misconceptions and errors.

Student Response 4:

```
Enter a number: 6 3 4 2 4 1 8
6
The value of x is 6
6
```

This example demonstrates 2 errors. The additional numbers at the end of the fist line have been included by the student to demonstrate the user input. This in itself is not incorrect. However, by doing so, the student has demonstrated two concepts. Firstly, it can be seen that the student correctly understands the concept that a variable can only take on one value. The student understands that although the user enters several values, only the initial value is assigned to the variable. Secondly, they have carelessly ignored, or even misunderstood, the distinction between the WRITE and WRITELN output statements. WRITELN and WRITE have exactly the same syntax except that at the end of a WRITELN statement the cursor is positioned at the beginning of the following line. The more serious error evident in this example is that the student has 'changed the output to make it semantically correct'. As variable x has a value of 6 and not 5 they have changed the second to last output statement to `WRITELN('The value of x is 6')`. Using

Pea's (1986) classification of misconceptions, this error can be categorised as an intentionality bug.

Student Response 5:

```
6
The value if x is 5
5
```

This student was categorised into two error categories. Firstly, the 'no Enter' error category as they omitted the first output generated from the first WRITELN statement. Secondly, they erroneously indicated that output statements are capable of changing variable values and hence were included in the 'WRITE('x') changes value of variable x error category. This can be considered to be a serious misconception, as the student is not merely misinterpreting the notation of the output statement, but in addition changing the value of the variable.

Student Response 6:

```
6
3
The value of x is 5
4
```

This example, once again demonstrates the 'no enter' error, however this student has also erroneously interpreted statements of the form `WRITELN(x)` as input statements. This concurs with Sleeman *et al.*'s (1986) research.

Generally, the output errors noted in this investigation, coincide with Bayman and Mayer's (1983) findings that students have difficulty in conceiving that output statements only display what they have been instructed to do. They often predict what the intended output is, and thus fall prey to the intentionality bug.

The output errors are shown in Table 6.4. For each error the following information is shown:

- the relative percentage of occurrences for each student group

- the significance of any differences between groups in terms of the percentage of error occurrences. If the cell contains a hyphen this indicates that there is no significant difference between the groups.

**Table 6.4: Output statement errors - Percentages and Significance levels**

| Error | Percentage | | | Significance | | |
|---|---|---|---|---|---|---|
| Output statements | Control '92 | Control '93 | PATMAN | 92 vs. 93 | 92 vs. PAT | 93 vs. PAT |
| no output after WRITELN('The value of x is 5'); | 44 | 36 | 18 | - | 0.01 | 0.05 |
| no understanding of output statements | 43 | 33 | 20 | - | 0.01 | - |
| no Enter | 39 | 27 | 27 | 0.01 | 0.01 | - |
| incorrect output statement | 35 | 29 | 22 | - | 0.01 | 0.05 |
| No distinction between WRITE and WRITELN statements | 15 | 9 | 14 | 0.05 | - | - |
| WRITE('x') changes value of x | 10 | 12 | 13 | - | - | - |
| Changed output to make it semantically correct | 10 | 3 | 0 | - | 0.05 | - |
| WRITELN(x) interpreted as input statement | 7 | 2 | 4 | - | - | - |
| No distinction between WRITELN(x) and WRITELN('x') statements | 3 | 3 | 5 | - | - | - |

In terms of the milder 'incorrect output statement' error category the experimental group was significantly better than either control group. In terms of the 'no write understanding' error category, a significantly smaller proportion of the experimental group fell into this category compared to the CONTROL '92 group. In both instances there was no significant difference between control groups.

Students frequently failed to display the output from statements such as WRITELN('Enter a number'). This occurred in several questions and in 39%, 27% and 27% of all instances for the respective groups. It should be noted that in this instance, the CONTROL '92 group was significantly different ($\alpha$=0.01) from the CONTROL '93 group, and significantly higher ($\alpha$=0.01) than the experimental group.

Another error category which indicated that there was a significant difference between the control groups, was that where students carelessly disregarded the difference between the WRITE and WRITELN output statements. In this case, the control groups were significantly different at an alpha level of 0.05. The proportion of PATMAN students who omitted the output after the WRITELN('The value of x is 5') statement was significantly lower than both control groups. As regards the 'output changed to be semantically correct' error, the proportion of the experimental group was significantly lower than the CONTROL '92 group.

Unless stated above, there were no significant differences between the proportion of students with these misconceptions. This may be attributable to the experimental group only having spent a mean time of 33 minutes on Patman prior to completing the first worksheet.

### 6.3.2 Input statements

Several worksheet questions (13/14) required that students interpret the READLN input statement. However most of the errors were detected in the earlier worksheet questions, when students were still familiarising themselves with the elementary programming concepts. A noticeable improvement over time was noted by the examiner.

To enable easier trapping of student errors, students were required to make use of given input values whenever required. This was achieved by supplying students with input sequences below the segment of code, as done by Sleeman *et al.* (1986) in their research. As this was a potential problem for students, a verbal explanation was given at the beginning of each worksheet session in addition to the written instructions included on each worksheet.

The input statement errors were easier to categorise than the output statement errors and all of the errors indicate deeper underlying misconceptions[22].

The 'semantically constrained input' misconception is the most frequent misconception associated with input statements. Students with this misconception believe that input statements are capable of selecting the most appropriate input value for a variable, in terms

[22] For relative percentages and significance levels of the input statement errors see Table 6.5.

of the semantic meaning of the variable name. The following program illustrates this misconception:

```
PROGRAM four;
VAR max, min, first, last:INTEGER;
BEGIN
  WRITELN('Enter a list of numbers');
  READLN(max, min, first, last);
  WRITELN('Largest Number:',max);
  WRITELN('Smallest Number:',min);
  WRITELN('Last Number:',min);
  WRITELN('First Number:',first);
END.
```

Given the input `5 13 1 6`, the correct output for this program would be:

```
Enter a list of numbers
Largest Number: 5
Smallest Number: 13
Last Number: 6
First Number: 1
```

Students frequently indicated that the output would be as follows:

```
Enter a list of numbers
Largest Number: 13
Smallest Number: 1
Last Number: 6
First Number: 5
```

They have selected appropriate input values for each variable based on its semantic meaning.

The following example will be used to illustrate some of the other input concept misconceptions:

```
PROGRAM two;
VAR b,c,a:INTEGER;
BEGIN
  WRITELN('Enter three numbers');
  READLN(c,b,a);
END.
```

This worksheet question required students to indicate the value of variables `a`, `b` and `c` after line 5 had been executed, given the following input: `15 25 20`. As READLN statements associate each input with the stipulated variables in a sequential manner, variable `c` would have the value `15`, variable `b` the value `25` and variable `a` the value `20`.

Correct response:

```
a _20_ b _25_ c _15_
```

Student Response 1: Declaration order

```
a _20_ b _15_ c _25_
```

Here the student has looked at the declaration order of variables, which in this case is `b`, `c` and `a` and associated the input values with the variables based on this order. Hence variable `b` gets the initial input value, variable `c` the second and variable `a` the last input value.

Student Response 2: Alphabetic order

```
a _15_ b _25_ c _20_
```

Here the alphabetic order of the variables has determined the associativity of the variables and the input values. Hence variable `a` has been assigned the initial value, `b` the next and `c` the last input value.

Student Response 3: Numeric order

```
a _15_ b _20_ c _25_
```

Here the alphabetic order and the numeric order of the variables have been matched. Variable `a` is assigned the lowest numeric input value, `b` the middle numeric value, and `c` the largest numeric input value. This could be considered to be a sub-class of the semantically constrained input error, however it appears to be restricted to variables with no meaningful name. Students seem to impose meaning on the variables based on their alphabetic order.

The final error noted with input statements is the belief that the input statement was capable of selecting some arbitrary input value.

All the input statement errors discussed above were noted by Sleeman *et al.* (1986) and Putnam *et al.* (1986), with the exception of the matching of the alphabetic order of variables to the numeric ordering of input values.

All noted errors are shown in Table 6.5 along with any significant differences between the student groups.

**Table 6.5: Input statement errors - Percentages and Significance levels**

| | Percentage | | | Significance | | |
|---|---|---|---|---|---|---|
| Input statements | Control '92 | Control '93 | PATMAN | 92 vs. 93 | 92 vs. PAT | 93 vs. PAT |
| Semantically constrained input | 32 | 27 | 19 | - | 0.05 | - |
| Declaration order determines the order in which variables are read into variables | 15 | 3 | 5 | - | - | - |
| Variables are assigned values based on the variables alphabetic order | 8 | 10 | 3 | - | - | - |
| Variables are assigned values based on their alphabetic order and the inputs numeric order | 10 | 6 | 3 | - | - | - |
| Variable selects value | 3 | 2 | 5 | - | - | - |

A significantly smaller proportion of the experimental group compared to the CONTROL '92 group had the semantically constrained input misconception. With all other input statement misconceptions there were no significant differences between the student groups.

### 6.3.3 Variable concepts

During the discussion of output statement errors, reference was made to multiple valued variables in Examples 3 and 4. Students incorrectly assign more than one value to a variable. This misconception is frequently associated with the semantically constrained input misconception[23].

For example, given the following program:

[23] For relative percentages and significance levels of the variable concept errors see Table 6.6.

```
PROGRAM one;
VAR Even, Odd: INTEGER;
BEGIN
  WRITELN('Enter four numbers:');
  READLN(Even,Odd);
END.
```

and the following input: `3  2  10  5`, variable `Even` would have the value 3 and variable `Odd` would have the value `2`. Forty three percent of all students indicated that variable `Even` would have the values `2` and `10`, and variable `Odd` the odd input values, namely: `3` and `5`.

A variation of this multiple valued variable misconception is the belief that a variable holds the accumulated total, or running total of several values. Students with this misconception indicated that variable `Even` would have the value `12` (i.e. the sum of 2 and 10) and `Odd 8` (i.e. the sum of 3 and 5). Another variation of this multiple valued variable misconception is the belief that the variable indicates the number of values that have been assigned to the variable, hence variable `Even` would have the value `2` as would variable `Odd`.

As noted by Sleeman *et al.* (1986), students were misled by the output statement `WRITELN('Enter four numbers:')`. Students believed that this caused four numbers to be read, and hence devised methods of dealing with the inputs.

Some students had a general problem of tracing through programs and keeping track of variable values. This error was most noticeable in the responses to worksheet two's fourth question. This question had two variables, `p` and `q`, which were used for different purposes, but students frequently confused the variables. Some students believed that variables maintain their initial value. This variable will be included in the discussion of assignment statement errors.

All except one of the above mentioned misconceptions were also noted by Sleeman *et al.* (1986) in their research. The misconception peculiar to this research is the 'variable value is the number of values assigned to variable' misconception. The noted variable misconceptions are summarised in Table 6.6 below.

**Table 6.6: Variable concept errors - Percentages and Significance levels**

| Error | Percentage | | | Significance | | |
|---|---|---|---|---|---|---|
| Variables | Control '92 | Control '93 | PATMAN | 92 vs. 93 | 92 vs. PAT | 93 vs. PAT |
| Multiple-valued variables | 32 | 42 | 33 | - | - | - |
| Initial value maintained | 16 | 17 | 9 | - | - | - |
| Variable holds accumulated total | 16 | 9 | 0 | - | 0.01 | 0.05 |
| Variable value is the number of values read into variable | 11 | 9 | 0 | - | 0.05 | 0.05 |
| Confusion of variables | 6 | 11 | 3 | - | 0.05 | 0.01 |
| Printing of variable when value changes | 3 | 0 | 8 | - | - | - |

The proportion of the experimental group's students was significantly lower than both control groups for three misconceptions, namely: 'variable holds accumulated total' of all input, 'variable value is the number of values assigned to variable' and 'confusion of variables'.

### 6.3.4 Assignment statements

Two worksheet questions dealt with assignment statements, namely questions 2.1 and 2.4

Worksheet question 2.1 will be used to illustrate some of the assignment misconceptions[24].

```
PROGRAM One;
VAR a,b,c:INTEGER;
BEGIN
  WRITELN(' Enter two numbers: ');
  READLN(a,b);
  WRITELN(a);
  WRITELN(b);
  b:=a;
  a:=a+1;
  c:=a + b;
  WRITELN(a);
  WRITELN(b);
  WRITELN(c);
END.
```

[24] For relative percentages and significance levels of the assignment statement errors see Table 6.7.

Given the input: `2 3 4` the correct output is:

```
Enter two numbers:
2
3
3
2
5
```

Student Response 1:

```
2
3
4
2
6
```

Here the student has demonstrated two errors, firstly the 'no enter' error and secondly they have interpreted the `a:=b` assignment statement as a swap statement. However, as noted by Sleeman *et al.* (1986) this was not a consistent error as assignment statements of the form `a:=a+1` and `c:=a + b` were correctly interpreted. This was also seen in this research.

Student Response 2:

```
Enter two numbers:
2
3
2
3
5
```

Here the student has kept the initial values of variables `a` and `b`, thereby ignoring the first two assignment statements, but correctly assigned the summation of variables `a` and `b` to variable `c`. Although this error was noted primarily with assignment statements it was included under variable misconceptions as mentioned earlier.

Student Response 3:

```
Enter two numbers
2
3
2
3
5
3
2
5
```

Here the student included three additional numbers in the output. These corresponded to the three assignment statements. As the variables changed, their new value was written to output. This was also noted by Sleeman *et al.* (1986).

A less frequent assignment statement error was the 'reversal of the assignment operator'. Students interpreted statements of the form `a:=b`, which means variable `a` takes on the value of variable `b`, as variable `b` is assigned the value of variable `a`. This was not a consistent error as students with this misconception interpreted assignments of the form `a:=b+1` correctly.

There was no evidence to indicate that students had interpreted the assignment statement as a comparison operator. This is contrary to Sleeman *et al.*'s findings.

Table 6.7 summarises all assignment statement information.

**Table 6.7: Assignment statements - Percentages and Significance levels**

| Error | Percentage | | | Significance | | |
|---|---|---|---|---|---|---|
| Assignment Statements | Control '92 | Control '93 | PATMAN | 92 vs. 93 | 92 vs. PATMAN | 93 vs. PATMAN |
| swaps variable values | 11 | 18 | 5 | - | - | 0.05 |
| printing of variable values when the variable is on the LHS of a statement | 12 | 6 | 6 | - | - | - |
| reversal of assignment operator | 5 | 9 | 5 | - | - | - |
| **assignment correct** | **48** | **47** | **62** | - | **0.05** | **0.05** |

The proportion of the experimental group that exhibited symptoms of the assignment statement 'swaps variable values' misconception was significantly less than the

CONTROL '93 group ($\alpha$ =0.05). Less frequent misconceptions were also noted in all student groups regarding the 'printing of variable value when the variable is on the left hand side of a statement' and the 'reversal of the assignment operator'. There were no significant differences between all student groups with regard to either misconception. A comparison of the proportion of students who correctly interpreted the assignment statement indicated that the experimental group was significantly better than both control groups ($\alpha$=0.05).

### 6.3.5 Looping constructs

Loop statements were mainly dealt with in worksheet 2. FOR loops were the focus of 3 questions (2.2, 2.5 and 3.1), REPEAT loops the focus of one question (2.3) and WHILE loops the focus of another question (2.4). This balance was decided upon as a result of Sleeman *et al.*'s (1986) and Putnam *et al.*'s (1986) research, as they found that the majority of errors occurred in FOR loop constructs. However, in retrospect it appears that the questions concerning loop statements might have benefited from being formulated differently for two reasons. Firstly, the WHILE loop question's complexity was compounded by the nested if statement. Secondly, students found worksheet two considerably harder than the other worksheets. This could have been as a result of the inherent complexity of some of the questions or merely that students were tested on concepts prior to them having sufficient time to digest the concepts. However, when the worksheet questions were scheduled, the lecturer was confident that the students had been adequately prepared. It might have been beneficial to have delayed the administration of worksheet 2. Furthermore, had more time been available during the semester it might have been beneficial to have had an additional worksheet which dealt with less complex looping constructs prior to the more complex ones. Unfortunately, it was necessary that the worksheets be scheduled to provide minimal disruption to the normal class requirements. Moreover once fixed for the CONTROL '92 group, the tests for the '93 student groups were administered at a corresponding time in 1993 to provide consistency within years. These comments do not apply equally to FOR loops as they were covered more frequently in questions with simpler programs.

Notwithstanding the above, noteworthy results were obtained from the analysis of the looping construct errors[25].

The following program will be used to illustrate some of the noted looping misconceptions:

```
PROGRAM Five;
VAR num,val:integer;
BEGIN
  FOR num:=1 TO 3 DO
  BEGIN
    WRITELN('Enter a number:');
    READLN (val);
  END;
  WRITELN(val);
END.
```

Given the input `[6 3] [3 4 5] [2 1] [8]`[26] the correct output is:

```
Enter a number:
Enter a number:
Enter a number:
2
```

Student Response 1:

```
Enter a number:
Enter a number:
Enter a number:
```

Here the student has omitted the output generated from the WRITELN statement immediately after the loop. A similar error was noted by Sleeman *et al.* (1986) in short programs, after a loop is executed students believed that control goes to the first statement of the program

Student Response 2:

```
Enter a number:
Enter a number:
Enter a number:
Enter a number:
8
```

[25] For relative percentages and significance levels of the looping construct errors see Table 6.8.

[26] The brackets are used to represent the different sequence of inputs. When the program first requires input 6 and 3 will be entered, the second time 3, 4 and 5 will be entered and so on.

Here the student has ignored the loop's control variable value and looped for all input given. This error is not restricted to looping constructs, as the error is also evident in programs without such constructs. For example, students interpreted BEGIN and END statements, as well as indentation as looping mechanisms. Students with this misconception assumed that the code would be repeated for all input. This was categorised by Sleeman *et al.* (1986) and Putnam *et al.* (1986) as the 'data-driven looping' misconception.

Student Response 3:

```
Enter a number:
2
```

Here the student has only generated the output from the first output statement of the loop once. This error is only noted for output statements which give the user some guidance, all other output statements, and statements in general, within the scope of the loop are correctly executed each time the loop is executed.

Student Response 4:

```
Enter a number:
6
Enter a number:
3
Enter a number:
2
```

This student has included the output statement immediately after the loop within the scope of the loop. This error is peculiar to output statements, and as noted by Sleeman *et al.* (1986) and Putnam *et al.* (1986), students who had this misconception for a particular looping construct did not necessarily have it for all looping constructs. This is a peculiar misconception which eludes explanation.

Student Response 5:

```
Enter a number: 6
6 6 6
Enter a number: 3
3 3 3
Enter a number: 2
2 2 2
Enter a number: 8
8 8 8
```

This example demonstrates four errors. Firstly the student has not distinguished between WRITE and WRITELN statements, secondly she has included the last WRITELN within the scope of the FOR loop, and thirdly by using all input she has fallen into the 'data driven looping' error category. Finally the student has indicated that each value is displayed three times, hence the FOR loop specifies the number of times a variable value is displayed. This error was also noted by Sleeman *et al.* (1986) and Putnam *et al.* (1986).

Student Response 6:

```
Enter a number:
3
```

Here the student has selected an input value for the variable within the range of the FOR loops control variable. Hence `3` was assigned to variable `val`, and execution of the loop halted. Students with this misconception have interpreted the FOR loop as a constraint on input. If no input values satisfy the range of the control variable students indicate that an error will occur.

Student Response 7:

```
Enter a number: 6
Enter a number: 3
3
Enter a number: 2
2
Enter a number:8
```

Once again, this example illustrates several misconceptions. The student has failed to distinguish between WRITE and WRITELN statements, she has included the WRITELN statement immediately after the loop within the scope of the loop, and she has demonstrated 'data driven looping'. The student has also selected to output the value of variable `val`, only when its value is within the range of the control variable. She has interpreted the FOR loop control variable as an output constraint.

When students were requested to generate the output from the following program

```
PROGRAM One;
VAR number:INTEGER;
BEGIN
  FOR number:=1 TO 3 DO BEGIN
    WRITELN(number*2);
  END;
END.
```

several students were unable to generate the output. They either indicated that an error would be generated or requested input. These students thus believed that a FOR loop 'control variable has no value within the scope of a loop'. Sleeman *et al.* (1986) found that some students did not realise that the control variable is a counter which is incremented with each iteration of the loop. No evidence of this error was noted in this research.

Three additional errors were noted with the following program:

```
PROGRAM Three;
VAR letter:CHAR;
BEGIN
  WRITE('Enter a character:');
  READLN(letter);
  REPEAT
    WRITELN('You entered letter: ',letter);
    WRITE('Enter a character:');
    READLN(letter);
  UNTIL (letter='N') or (letter='n');
  WRITLEN(letter);
END.
```

Given the following input: `[h] [Q] [n] [N] [r]` the correct output is:

```
Enter a character:
You entered letter: h
Enter a character:
You entered letter: Q
Enter a character:
n
```

Student Response 1:

```
Enter a character:
You entered letter: h
Enter a character:
You entered letter: Q
Enter a character:
You entered letter: n
```

Here the student has understood that the loop must terminate once character 'n' had been read into variable `letter`. However he has proceeded to generate the output from the first output statement of the WHILE loop, and then terminated the program. He also omitted the output statement which follows the WHILE loop. This student has changed the output to make it more meaningful. He appears to have given the computer interpretative powers.

Student Response 2:

```
Enter a character:
You entered letter: h
Enter a character:
You entered letter: Q
```

Here the student has stopped too soon. She has anticipated that the next input character will satisfy the stopping criteria, and hence she has terminated prior to getting the input from the user. This error was classified as the 'Terminate loop too soon' error. She has also omitted the output generated from the output statement directly after the loop. This is an example of an intentionality misconception.

Student Response 3:

```
Enter a character:
You entered letter: h
Enter a character:
You entered letter: Q
Enter a character:
You entered letter: n
Enter a character:
N
```

This student has terminated the loop after both uppercase and lowercase `n`'s have been entered. This student, although understanding that the loop terminates after the until conditional has been met, has misinterpreted the until conditional to mean both upper and lowercase n. This could be a surface level error, or a symptom of a more serious misconception.

The looping errors are summarised in Table 6.8. In addition, the percentage of students who ignored the looping construct, and the percentage of students who correctly interpreted the looping constructs have been included.

**Table 6.8: Looping constructs - Percentages and Significance levels**

| Error | Percentage | | | Significance | | |
|---|---|---|---|---|---|---|
| Loop Statements | Control '92 | Control '93 | PATMAN | 92 vs. 93 | 92 vs. PATMAN | 93 vs. PATMAN |
| No write after loop | 32 | 21 | 8 | - | 0.01 | 0.01 |
| You entered letter n | 15 | 18 | 21 | - | - | - |
| data-driven looping | 25 | 9 | 4 | 0.01 | 0.01 | 0.05 |
| both N and n | 21 | 9 | 8 | - | 0.05 | - |
| Enter once only | 14 | 11 | 13 | - | - | - |
| Terminate loop too soon | 17 | 15 | 5 | - | 0.01 | 0.01 |
| WRITELN immediately after loop included in scope of loop | 14 | 2 | 4 | 0.01 | 0.05 | - |
| scope problem | 6 | 4 | 3 | - | - | - |
| FOR loops: | | | | | | |
| control variable can be changed in loop | 6 | 11 | 3 | - | 0.05 | 0.01 |
| control variable has no value within the scope of the loop | 13 | 6 | 0 | - | 0.01 | - |
| FOR loop specifies no. of times variable value is displayed | 7 | 9 | 3 | - | 0.05 | 0.05 |
| control value acts as input constraint | 8 | 8 | 3 | - | 0.05 | 0.05 |
| control value acts as output constraint | 8 | 7 | 2 | - | 0.01 | 0.05 |
| Ignored WHILE loop statement | 33 | 30 | 34 | - | - | - |
| Ignored FOR loop statement | 17 | 10 | 7 | - | 0.01 | - |
| Ignored REPEAT loop statement | 16 | 6 | 5 | - | 0.05 | - |
| WHILE loop correct | 16 | 18 | 21 | - | - | - |
| REPEAT loop correct | 23 | 21 | 29 | - | - | - |
| FOR loop correct | 41 | 43 | 66 | - | 0.01 | 0.01 |

Considering all 13 looping errors the experimental group was significantly better than the CONTROL '92 group in 10 instances, and 7 instances compared to the CONTROL '93 group. However in 2 instances the CONTROL '92 group was also considerably worse than the CONTROL '93 group, in addition to being worse than the PATMAN group. Namely in 'data driven looping' and the inclusion of a WRITELN following a loop into the loop.

Included in the Table above, is the proportion of students who ignored the looping construct. The WHILE loop was ignored by 33%, 30% and 34% of '92, '93 CONTROL and PATMAN groups. This could be an indication that the question was asked before the students had adequate time to digest the concept, and reinforces the comments made earlier. The FOR loop was ignored by 17%, 10% and 7% of the respective groups and similarly for the REPEAT loop (16%, 6%, 5%). There was no significant difference between the groups with regard to the WHILE loop, however the experimental group fared significantly better that the CONTROL '92 group with regard to the ignoring of the REPEAT and FOR loops.

Finally, the analysis of the proportion of students who correctly applied the looping constructs indicated that an average of 24%, 18% and 50% of all students correctly interpreted the REPEAT, WHILE and FOR loops. There was no significant difference in the student groups regarding the WHILE and REPEAT loops, however, for FOR loops the experimental group was significantly better than both control groups at and alpha level of 0.01.

### 6.3.6 If statements

The most frequent error of the experimental group was that of stopping the output if the condition evaluated to false[27]. This error was noted for question 2.4 in which the if statement was included in a WHILE loop:

[27] For relative percentages and significance levels of the if statement errors see Table 6.9.

```
PROGRAM Four;
VAR p,q:INTEGER;
BEGIN
  q:=0;
  WRITE('Enter a number:');
  READLN(p);
  WHILE p <> 0 DO
  BEGIN
    IF p > 0 THEN
      q:=q+1;
    WRITE('Enter a number:');
    READLN(p);
  END;
  WRITELN(q);
END.
```

As mentioned earlier, this program was possibly the hardest worksheet question as it included the if statement within the while statement. Only seven percent of all students correctly answered this question. It was also not often possible to establish students' thought processes for this question, and hence students were interviewed to establish their problems.

Given the input: `[1] [-1] [-3] [4] [0]` the program generates the following output:

```
Enter a number:
Enter a number:
Enter a number:
Enter a number:
Enter a number:
2
```

Several errors occur in conditional statements of the form IF (condition) THEN (action). If the condition is false students reacted in several ways.

Student Response 1:

```
Enter a number:
Enter a number:
1
```

This student has terminated execution of the program when the conditional statement became false. Statements which provide an alternative action in the form of IF THEN

ELSE statements, do not yield such drastic termination. This error was also noted by Sleeman *et al.* (1986) and Putnam *et al.* (1986).

Student Response 2:

```
Enter a number:
Enter a number:
Enter a number:
2
```

It is not immediately obvious how the student generated this output, but after interviewing the student, it was determined that she has treated the output statement immediately after the IF THEN statement as the ELSE component of the conditional statement. This error was also noted by Sleeman *et al.* (1986).

Student Response 3:

```
Enter a number:
Enter a number:
q=1
Enter a number:
q=1
Enter a number:
Enter a number:
2
```

Here the student has printed the value of q when the conditional was false. Although several students had their own variation of how and what was generated when the conditional was false, they all had one concept in common: they generated some output to indicate the current status of the program at that stage when the condition was false.

Another error made by students was the interpretation of the if statement condition as an assignment. If the condition was not satisfied students would arbitrarily assign a value to the variable to satisfy the condition. This error was not found by Sleeman *et al.* (1986) or Putnam *et al.* (1986).

The other conditional statement errors will be demonstrated using the program below.

```
PROGRAM Three;
VAR number:INTEGER;
BEGIN
  WRITE('Enter a number:');
  READLN(number);
  IF number = 7 THEN
   WRITELN('Unlucky number');
  IF number = 10 THEN
    WRITELN('Lucky number');
  WRITELN('The number was',number);
END.
```

The correct output for this program, given the input: `[4]` `[10]` `[7]` is as follows:

```
Enter a number:
The number was 4
```

Although this program was designed to determine students' conditional statement misconceptions, a more serious misconception was noted in several students' responses. Students appear to have given the computer interpretative capabilities and hence have edited the generated output to make it more meaningful. Bayman and Mayer (1983) found this to be a common problem.

Student Response 1:

```
The number was Lucky Number
The number was Unlucky number
```

This student demonstrates three errors. Firstly the student has generated output for all input and thus fall into the 'data driven looping' error category. Secondly, he has changed the output to make it more meaningful to himself. He has substituted output of the form:

```
Unlucky number
The number was 7
```

for output of the form:

```
The number was Lucky Number
```

Lastly he has not generated any input when variable `number` had a value of `4`. Other students with this misconception indicated that an error would be generated because `4` was not a lucky number. This misconception is referred to as 'relationship between conditional and input value' error. This error is another example of the intentionality bug, students appear to have given the computer interpretative capabilities to generate output which is more meaningful.

Student Response 2:

```
Enter a number
Unlucky number
Lucky number
The number was 4
```

Here the student has executed the if statement regardless of the conditional value being true or false. This was also noted by Sleeman *et al.*. (1986).

A final error noted by Sleeman *et al.*. (1986), Putnam *et al.*. (1986) and also in this investigation, was that students believed that the THEN action of an IF THEN ELSE statement was always executed.

The percentage of occurrences for the if statement errors are tabulated below (Table 6.9).

**Table 6.9: If Statements - Percentages and Significance levels**

| Error | Percentage | | | Significance | | |
|---|---|---|---|---|---|---|
| If statements | Control '92 | Control '93 | PATMAN | 92 vs. 93 | 92 vs. PAT | 93 vs. PAT |
| Halt program if the conditional of an IF THEN statement is false | 11 | 24 | 45 | - | **0.01** | **0.05** |
| statement following an IF THEN statement is considered the ELSE branch | 25 | 21 | 10 | - | 0.05 | - |
| Relationship between conditional and Input value error. | 21 | 24 | 3 | - | 0.01 | 0.01 |
| If the conditional is false, output to indicate the current status of the program is generated | 23 | 12 | 0 | - | 0.01 | 0.05 |
| if = assignment | 11 | 14 | 1 | - | 0.01 | 0.01 |
| Always THEN | 6 | 6 | 9 | - | - | - |
| **If Correct** | **52** | **58** | **74** | **-** | **0.01** | **0.01** |

The proportion of experimental students exhibiting the 'halt if the conditional of an IF THEN statement is false' error was, significantly higher ($\alpha$=0.01) than both control groups. No explanation can be given as to why such a large proportion of the experimental group fell into this error category. Considering all other misconceptions the experimental group fared significantly better than one or both control groups, in four of the five instances. The proportion of experimental group students who correctly interpreted the if statements was significantly ($\alpha$=0.01) better than both control groups.

### 6.3.7 Procedures

Only one worksheet question dealt with procedures. The following program will be used to illustrate the noted procedure misconceptions[28]. These errors indicate students difficulty with tracing the flow of control during program execution.

The following program:

```
PROGRAM Four;
VAR number:INTEGER;

PROCEUDRE Letters;
BEGIN
  WRITELN('ijkl');
  WRITELN('mnop');
END;

BEGIN
  WRITE('qrst');
  Letters;
  Letters;
END.
```

will generate the following output when executed:

```
qrstijkl
mnop
ijkl
mnop
```

Student Response 1:

```
ijkl
mnop
```

Here the student has traced the program in a top-down scan of the code and terminated execution at the end of the procedure. This error indicates that the student is unaware of

[28] For relative percentages and significance levels of the procedure errors see Table 6.10.

procedural abstraction and the general flow of control of programs. This error is classified in Table 6.10 as 'executed procedure only'.

Student Response 2:

```
qrst
ijkl
mnop
```

Here the student has traced the program in the correct manner but only executed the procedure once. Possibly the student has seen no advantage in calling the procedure twice and hence omitted the second call to the procedure. This was the most common error noted with procedures. This error was not noted by Sleeman *et al.*. (1986) and Putnam *et al.*. (1986) The student has also failed to distinguish between WRITE and WRITELN statements. This is surprising as this is a surface level error. One would expect students to be able to distinguish the difference between the statements by the twelfth week of programming instruction.

Student Response 3:

```
ijkl
mnop
qrst
```

Here the student has done a top-down scan of the code and generated the output in that order. This illustrates a general misunderstanding with flow of control of programs and procedural abstraction. This error was called the 'order appear' error.

Student Response 4:

```
ijkl
mnop
qrst
ijkl
mnop
ijkl
mnop
```

Here the student has generated the output from the procedure prior to execution of the main part of the program. During the execution of the main body of the program they have correctly called and executed the procedure. Students with this misconception, understand the concept of procedural abstraction, but have a problem with the flow of control beginning in the main body of the program.

Student Response 5:

```
    qrst
```

The final error regarding procedures is illustrated in this student's response. She has simply omitted the procedure calls, thus demonstrating a misunderstanding with procedural abstraction.

The errors regarding procedures are summarised in the following Table.

**Table 6.10: Procedures - Percentages and Significance levels**

| Error | Percentage | | | Significance | | |
|---|---|---|---|---|---|---|
| Procedures | Control '92 | Control '93 | PATMAN | 92 vs. 93 | 92 vs. PAT | 93 vs. PAT |
| Procedure only once | 39 | 6 | 10 | 0.01 | 0.01 | - |
| Order appear | 21 | 24 | 3 | - | 0.01 | 0.01 |
| Procedure only | 3 | 21 | 0 | 0.01 | - | 0.01 |
| Order appear + call | 2 | 9 | 3 | - | - | - |
| No procedure call | 2 | 9 | 3 | - | - | - |
| **Procedure correct** | **39** | **33** | **78** | **-** | **0.01** | **0.01** |

With respect to the procedure errors, there was a significant difference between the control groups (Table 6.10) on two accounts. In one instance, in which students only executed the procedure once, the CONTROL '92 group was significantly different from the CONTROL '93 group and significantly higher than the experimental group. In the second instance, in which only the procedure was executed, the CONTROL '93 group was significantly different from the CONTROL '92 group and significantly higher than the experimental group. Also, significantly fewer experimental group students executed the statements in the order they appeared, compared to both control groups. Finally, a significantly larger proportion (78%) of the PATMAN group correctly executed the code, compared to the 39% and 33% of the '92 and '93 CONTROL groups. This is an indication that the support environment was successful in demonstrating the flow of control through a program.

### 6.3.8 Summary of Results

From the above discussion of the programming constructs and misconceptions, the results can be summarised as follows:

- In one error instance the experimental group was significantly higher than both control groups. This is unexplainable.
- Out of the 49 documented errors, the experimental group was significantly better than one or both control groups in 29 instances, and in the other instances there appeared to be no significant difference between the groups.
- It could also be said that there was more significant difference between the experimental and control groups with regard to the constructs that were tested in the later worksheet questions and less significant difference between the groups with regard to the earlier testing of constructs. This suggests that the more time students spent on the Patman system, the more benefit the system was to them.
- The proportion of experimental students who correctly interpreted the assignment, while, repeat, for, if and procedure statements was greater than that of either control group. In four of the six instances in which these constructs were tested, these proportions were significantly higher ($\alpha \leq 0.05$) than either control group. There were no significant differences between the control groups.

Many misconceptions noted in previous research by Sleeman *et al.* (1986) and Putnam *et al.* (1986) were verified by this investigation. A few misconceptions, peculiar to this research were also identified. The analysis of the worksheets, which tested for misconceptions in this research, provides strong evidence that the Patman support environment was successful in minimising several misconceptions.

## 6.4 The Effect of PATMAN on Specific Misconceptions over Time

At the outset of this research it was necessary to decide between two experimental approaches. The first being a longitudinal study in which the benefits of the use of PATMAN for each student would be monitored throughout the semester. In this approach the focus would have been to determine students misconceptions prior to using PATMAN, and then to assess whether these students had maintained these same misconceptions after using PATMAN. The second approach would have been to compare two groups of students, one which made use of PATMAN and one which did not. In this approach the intention would be to determine if there was any significant difference between the groups of students, and if so this could be attributed to the use of PATMAN. The first approach was abandoned for several reasons. The PATMAN lessons were not designed to address particular misconceptions but rather to assist students in the acquisition of accurate conceptual models of the programming and computer environment. To allow for the accurate assessment of pre-PATMAN misconceptions it would have been necessary to delay the use of PATMAN until a student had been exposed to all the programming concepts under normal learning conditions, and only then could students commence with the additional PATMAN lessons. This would have necessitated students attending the PATMAN lessons after the course had been completed, and as a result was not feasible. This said, the first experimental approach could have been adopted if the scope of the research, and hence PATMAN, was limited to a small subset of misconceptions. However, it was felt that the examination of only a small set of misconceptions would have contradicted the research objectives. Novice programmers commonly have a wide variety of misconceptions and it was the intention of this research to identify these misconceptions. An hence the second experimental approach was adopted.

Nevertheless, it would still have been possible for a stage wise comparison of students over time if worksheets two and three included additional questions which attempted to trap the same misconceptions which had been noted in previous worksheets. Unfortunately this was not possible as limited time was available for the administration of worksheets with the course's time constraints. Nevertheless, it was possible to determine the affect of the use of PATMAN over time with regard to a few misconceptions. These misconceptions were those that were noted in more than one worksheet. The percentage of PATMAN students who were categorised as having a particular misconception at different times during the semester are shown in Table 6.11. Worksheet one, was administered during the fourth week of the semester, worksheet two, during the ninth week and worksheet three during the twelfth week.

**Table 6.11: Stage wise comparison of misconceptions**

| Error | Percentage of PATMAN Students | | |
|---|---|---|---|
| | Worksheet 1 Week 4 | Worksheet 2 Week 9 | Worksheet 3 Week 12 |
| no Enter (Output statements) | 35.0 | 35.0 | 37.5 |
| WRITELN(x) interpreted as input statement (Output statements) | 2.5 | 5.0 | 0.0 |
| No distinction between WRITE and WRITELN statements (Output statements) | | 60.0 | 17.5 |
| Multiple-valued variables (Variables) | 42.5 | 7.5 | |
| Confusion of variables (Variables) | | 10.0 | 0.0 |
| FOR Loop: control variable can be changed in loop (Looping constructs) | | 10.0 | 0 |
| FOR loop specifies no. of times variable value is displayed (Looping constructs) | | 5.0 | 2.5 |
| Ignored FOR loop statement (Looping constructs) | | 12.5 | 2.5 |
| If statements: always THEN (If statements) | | 20.0 | 2.5 |
| All input used | 45.0 | 30.0 | |

In Table 6.11 the empty cells indicate that the misconception was not included in the relevant worksheet. Of the ten misconceptions, there was a reduction in the percentage of students over time with regard to nine misconceptions. This indicates that PATMAN was able to assist students in overcoming these particular misconceptions. However, it appears that the opposite was true for one misconception. That is, the 'no Enter' output error category. At the time of worksheets one and two, 35 % of the PATMAN students were categorised with this error and at the time of worksheet three, 37.5%. As discussed early this was categorised as a surface level error as it was not persistent, and it could be a result of carelessness on behalf of the student. Nevertheless, as there was no substantial change in the percentage of students who fell into this error category over time, it can be assumed that PATMAN did not adequately address this problem.

## 6.5 Demographics of Student Groups

As discussed previously, a primary concern in experiments of this nature is the possibility that some attribute of one or more of the student groups which is external to the experimental process might account for the observed differences in the groups. Although it was not possible to ensure that there were no such significant differences, it is possible to ascertain whether any of the noted differences might be confounding variables in the experimental process.

To accomplish this task, two multiple regression models were established for each student group: one modelled final results and the other examination results. Each model determined a set of predictors, from a possible set of 56, which could have influenced the final or examination results of a student in any of the student groups. The set of predictors were based on the students' background and psychological characteristics obtained from Questionnaires 1 and 2 (see Appendix B). Appendix C contains a summary of student background and psychological characteristics. The Tables in Appendix C also indicate, for each characteristic, any statistical significant differences between the student groups, based on Z-proportion tests or ANOVA and Scheffé's multiple comparison method.

For each model, the set of significant predictors chosen included no student characteristic that was significantly different from the other student groups. A background characteristic which one might have expected to influence the validity of the results obtained in this experiment is the difference in mean matriculation points for the CONTROL '93 and PATMAN groups. However in the CONTROL '93 and PATMAN regression models this characteristic did not emerge as part of the set of significant predictors. This indicates that although there is a significant difference between the two student groups, in terms of matriculation points, this is not a factor which influenced the students' success in the course. The results of the multiple regression analysis thus indicate that in this regard any differences between the student groups' performance did not influence the experimental process.

The anxiety and alienation measures were not found to be significant predictors of students' success in the programming course, nonetheless this characteristic is of interest. Of all characteristics evaluated, this is the only one that indicates that the PATMAN group is significantly different from both control groups, and that there is no significant difference between the control groups. As this characteristic was a summation of several items from both questionnaires, it appears that the Patman support environment had a beneficial side effect of reducing students' computer anxiety and alienation.

With all studies of this nature, it is necessary to ensure that the noted improvements in performance are not due to external factors such as the Hawthorn effect, a psychological phenomena which refers to the effect of the experimental process on subjects, or the additional time experimental group students spent acquiring programming knowledge. In the current context it is unlikely that the Hawthorn effect took place as all students were part of the experiment and thus one would assume that all students would improve. Nor is it likely that the additional time, on average 35 minutes per student per week, spent on Patman had such a marked impact on student performance when one considers that the mean time students spent on their programming assignments was more than five hours per week. The noted improvements in student performance can more likely be attributed to the pedagogical aspects of the support environment. However these considerations could be the basis of further research

## 6.6 Students opinion of Patman

Finally the students opinion of the support environment is considered.

Students were requested to complete a questionnaire on Patman (see Appendix B). The data presented here is a summary of the experimental groups' responses to the questions on this questionnaire. The responses were rated on a 5-point Rating scale: (1) Strongly Disagree, (2) Disagree, (3) Neither Agree nor Disagree, (4) Agree, (5) Strongly Agree. Table 6.12 shows the average rating for the experimental group.

**Table 6.12: Student responses to 5-point rating scale questions**

| Statement | Average Rating | Number of negative responses (max. = 40) |
|---|---|---|
| I benefited from using Patman | 4.17 | 1 |
| The Patman programs/lessons were interesting | 3.92 | 0 |
| The Patman programs/lessons were enjoyable | 3.64 | 1 |
| I liked the fact that I could work at my own pace and decide which lesson I was going to study and how often | 4.47 | 0 |
| I found the Patman tutorials more beneficial than the standard tutorials | 3.69 | 5 |
| I would have preferred to have had access to Patman at anytime and for any length of time | 4.36 | 2 |

From these results it can be seen that students responded favourably to the Patman system. All seven averages are positive. Most students appreciated the fact that they could work at their own pace (mean = 4.47, no negative responses) and believed that they benefited from using the system (mean = 4.17, one negative response). Only one student felt that he had not benefited from using the system. The benefits of using the system as indicated by the experimental group are noted in Table 6.12.

**Table 6.13: Benefits of using Patman**

| Benefit | Number of students (max. = 40) |
|---|---|
| Patman helped with the understanding of programs | 29 |
| Patman helped with the learning of programming constructs | 26 |
| Patman helped with the weekly program assignments | 7 |
| other benefits of Patman (specified by students) | |
| learning new functions and reserved words | 2 |
| walk-through of programs | 2 |
| understanding computer output | 1 |
| learning at my pace ensures I understand the program before continuing | 1 |

As expected, students would have preferred to have made use of the Patman in their own time via the undergraduate network and computer laboratories (4.36). This is reinforced by comments and suggestions made by the students (see Table 6.12). Given this facility students indicated that they would have like to have used Patman 2.2 (mean) days per week and for 2.8 hours per week. It appears that students felt that more time on Patman was needed to cover the lessons adequately. Twenty two students felt it was necessary to attend all tutorials in order to keep up with the lessons. Twelve students felt there was adequate time to complete the lessons.

Finally, students were asked to suggest any improvements or make any comments regarding Patman. A selection of these are noted in Table 6.13.

**Table 6.14: Students' comments and suggestions about Patman**

| Student Comments regarding Patman support environment |
|---|
| But in my own time. Otherwise I had to rush from lunch to the extra tutorial |
| found Patman helpful |
| Given output and have to write a program or vice verse so that we can see our mistakes at that moment and get help |
| Have more programs available to work through with possible exam questions from past papers |
| I don't think we did enough work in the Patman tutorials More programs should be included |
| I would like to have access to Patman after finishing the course to help me with revision. Early tutorials of Patman did not help much because I was not yet familiar with the course |
| I would like to see Patman testing the users knowledge after a lesson so as to ensure that the user understood what he was doing. |
| It definitely reinforced the lectures and playing with the mouse was great too. |
| It is great and helpful |
| It might be helpful if there are programs very similar to those we do in lecturers |
| It would be better to write our own programs and be corrected if wrong then every time we are given a program and how to run it only |
| Maybe give printout of the program in the lessons so can refer to them later as gained most from construction etc. of programs |
| None I can think of it's a good system |
| Obtain a printout out of programs in Patman for later reference |
| Other than having more access to the program, more loops Exercises and Text variables etc. |
| Patman Tutorials were very good & allowed one to see how programs work & taught me how to write out & understand programs. |
| Put it on the LAN |
| Should be made easily accessible - people should be able to attend anytime should they want to |
| There must at least be like after going on through programs be given one exercise to test our understanding and answers to those exercises but they must have no contribution to our Tutorial marks |
| there wasn't anything on textfiles |
| Try to build it up into LAN, so that student can have more access to it |

## 6.7 Chapter Summary

The Patman support environment was evaluated using the proportion of correct responses for each worksheet question per student group, the final and examination results of the students, and the proportion of occurrences of each misconception per student group. All evaluations indicated a noticeably higher rate of success for the Patman students as compared to the control group students. There was also found to be no confounding experimental variables, hence the noted difference can be attributed to the Patman support environment. Finally, the students who made use of the Patman support environment reacted favourably towards the system.

# 7. Conclusions and Further Research

In learning to program students often formulate inaccurate or incomplete mental models of the programming language environment. This may be attributed to both the nature of the task and inadequate programming instruction. This thesis has examined documented evidence of the misconceptions which commonly hamper the progress of novice programmers. It has also investigated the role that support environments can play in minimising the effects of these misconceptions.

Novice programmers are in a situation in which they have insufficient previous knowledge of programming. As a result, they invariably rely on inappropriate knowledge and learning strategies. Students misapply knowledge from other domains, and over apply analogies, the most common being that of equating programming with conversing with a human. Pea (1986) has called this the 'superbug'. Furthermore, most programming instruction concentrates on the syntax of the programming language and many instructors may be unaware of the way in which students acquire programming knowledge. Invariably significant numbers of students develop inaccurate mental models which result in misconceptions such as parallelism bugs, egocentrism bugs and intentionality bugs.

As documented in this thesis, a Pascal support environment, Patman, was developed with the objective of reducing student misconceptions. The support environment interface was based on the glass box approach which allows students to see programming language constructs being executed at the transaction level. Each programming statement results in some visible change to one or more of the following: the variable, input, display windows or the flow of control display. The objective of this glass box approach is to assist students in developing accurate mental models of the programming language. The Patman support environment used this approach as a mechanism for showing students example programs. The example programs were designed with the intention of providing examples that would contradict inaccurate or incorrect mental models. Other design considerations included the illustration of general algorithms or plans, the demonstration of good programming principles and the use of particular programming constructs in appropriate ways. The intention of these design considerations was to assist students in structuring their programming knowledge in a manner similar to expert programmers. Comparative studies of expert and novice programmers have shown that novice programmers structure their knowledge based on surface characteristics of the syntax of the programming constructs, whereas expert programmers structure their knowledge based on functional characteristics of larger segments of code, called plans.

The Patman support environment was tested in an introductory programming course to determine whether the support environment was capable of reducing student misconceptions and improving the students' general programming ability. Three students groups were used: CONTROL '92, CONTROL '93 and PATMAN. The PATMAN student group made use of the support environment in addition to the conventional teaching methods. The control groups were used to determine if there were any differences between the students taught under conventional teaching methods and the students who made use of the additional Patman learning resource. The CONTROL '92 student group was used to investigate misconceptions, which assisted in the design of Patman, as well as to determine the year to year variance. The main findings of this empirical work are listed below:

- a smaller proportion of the PATMAN group were found to possess misconceptions
- a larger proportion of the PATMAN group correctly answered the worksheet questions
- the PATMAN students' examination and final results were noticeably higher than those of both control groups
- fewer of the experimental group's students failed the examination or the course compared to both control groups
- minimal differences were noted between the control groups
- most experimental group students reacted favourably to the Patman support environment.

These findings provide strong evidence that the support environment was influential in reducing student's programming misconceptions and that it was capable of assisting students in acquiring programming knowledge in an introductory programming course.

On entry into South African Universities, students have vastly different computer knowledge and experience, and thus it seems appropriate to determine whether support environments of this nature are capable of assisting students to overcome disadvantageous background characteristic. In the current research, when students were classified as either coming from a disadvantageous background (i.e. no previous computer experience, no home computer, disadvantaged schooling) or not, the largest benefit was noted in students with disadvantageous backgrounds. Those students making use of the support environment outperformed those who did not. Although, it could be an anomaly of the current research it does provide an indication that support environments, of a similar nature to the one discussed here, could address some of the problems facing South African University students in learning to program.

In terms of the difficulties students experience while learning to program, more research needs to be directed towards comparative studies of programming languages. Research in the 1980's investigated BASIC versus FORTRAN versus Pascal as an initial programming language, but as more universities move away from Pascal to C and C++ as their introductory programming language, the impact of this shift needs to be investigated. Another issue is the implications of teaching novices event-driven programming languages such as Visual Basic or Delphi. What difficulties will novices experience when learning these languages and how must programming instruction adapt? These issues are of particular concern when one takes heed of predictions that programming is no longer going to be restricted to the domain of computer science. People from all walks of life are going to develop their own programs in order to solve their particular problems.

Several extensions to the support environment are possible. Firstly the system can be developed into a Computer Assisted Tutoring System, by including testing mechanisms, which would give students feedback about their knowledge acquisition. Such testing mechanisms could include the use of cloze tests (Robinson, 1981). Cloze procedures, which have often been used as a measure of prose comprehension, have also been found to be a simple method for measuring software comprehension. They are of particular interest because the construction of a cloze test is easier than multiple choice type quizzes. They only take a few hours to construct, require little skill and are potentially automatable. This is significant in a context such as University, where programming instructors tend to be under increasing pressure. Furthermore initial studies show they are reliable as a measure of software comprehension (Hall & Zweben, 1986).

The current support environment has focused on misconceptions of the semantics of the Pascal programming language. This was appropriate as this is prerequisite knowledge for the design of programs. However, the system could be extended to provide assistance to students for the design of programs, possibly providing algorithm animation, in a similar vein to the Algorithm Animator and Programming Toolbox developed at the Witwatersrand University (Sanders & Gopal, 1991).

Further extensions to the system could incorporate principles of intelligent tutoring systems, such as student modelling, which provide an adaptive and flexible learning environment. In the current context it would be essential for the student model to include not only the knowledge of the student as a subset of an expert programmer, but also knowledge of programming misconceptions. The design of an Intelligent Tutoring System, which could accommodate varying learning styles of students, could also be beneficial as the mismatch of teaching and learning styles has been found to have had a negative impact on the acquisition of programming knowledge.

# REFERENCES

ADELSON, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition*, Vol. 9, p. 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*, Vol. 10, p. 483-495.

AHO, A.V.; SETHI, R. & ULLMAN, J.D. (1986). Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing Company, Reading, Massachusetts.

ALLEN, R.B. (1982). Cognitive Factors in Human Interaction with Computers. In Directions in Human-Computer Interaction, Badre, A. & Schneiderman, B. (eds.), Ablex Publishing Corporation, Norwood, New Jersey, p. 1-26.

ALLWOOD, C.L. (1986). Novices on the Computer: a Review of the Literature. *International Journal of Man-Machine Studies*, 25, p. 633-658.

ALLWOOD, C.M. & BJÖRHAG, C.-G. (1990). Novices' debugging when programming in Pascal. *International Journal of Man-Machine Studies*, 33, p. 707-724.

ANDERSON, J.R. & SKWARECKI, E. (1986). The Automated Tutoring of Introductory Computer Programming. *Communications of the ACM*, Vol. 29, No. 9, p. 842-849.

ANDERSON, R.C. (1986). Some reflections on the acquisition of knowledge. *Educational Researcher*, 13, p. 5-10.

ANDERSON, J.R.; FARELLEL, R. & SAUERS, R. (1984). Learning to Program in LISP. *Cognitive Science*, 8, p. 87-129.

APPELBAUM, S.H. & PRIMMER, B. (1990). An $HR_x$ for Computer Anxiety. *Personnel*, September, p. 8-11.

BARR, A.; BEARD, M. & ATKINSON, R. (1976). The Computer as a Tutorial Laboratory: the Stanford BIP Project. *International Journal of Man-Machine Studies*, 8, p. 567-596.

BAYMAN, P. & MAYER, R.E. (1983) A Diagnosis of Beginning Programmer's Misconceptions of BASIC Programming Statements. *Communications of the ACM*, Vol. 26, p. 677-679.

BELL, D. (1976). Programmer Selection and Programming Errors. *The Computer Journal*, Vol. 19, No. 3 , p. 202-206.

BONAR, J.G. & CUNNINGHAM, R. (1988). Bridge: Tutoring the Programming Process. In Intelligent Tutoring Systems: Lessons Learned, Psotka, J.; Massey, D.L. & Mutter, S.A. (eds.), Erlbaum Associates, Hillsdale, New Jersey, p. 409-434.

BOYLE, T. (1990). The Core Approach to Developing Learning Environments for Programming. *Monitor*, Vol. 1, No. 1, p. 7-10.

BRANSON, R.K. (1989). Necessary Conditions for Success in Instructional Systems Development Projects. *Educational Technology*, February, p. 46-47.

BROOKS, R. (1977). Towards a Theory of the Cognitive Processes in Computer Programming. *International Journal of Man-Machine Studies*, 9, p. 737-751.

BROOKS, R.E. (1980). Studying Programmer Behaviour Experimentally: The Problems of Proper Methodology. *Communications of the ACM*, Vol. 23, No. 4, p. 207-213.

CALHOUN, M.; STALEY, D.; HUGHES, L. & MCLEAN, M. (1989). The Relationship of Age, level of Formal Education, Duration of Employment Toward Attitudes in the Workplace. *Journal of Medical Systems*, Vol. 13, No. 1, p. 1-9.

CALITZ. A.P. (1984). Evaluation of attitudes of prospective Computer Science Students with a view to the Development of a Computer Aided Testing Program. *M.Sc. Thesis*, University of Port Elizabeth.

CARDINALE, L.A. & SMITH, C.M. (1994). The Effects of Computer-Assisted Learning-Strategy Training on the Achievement of Learning Objectives. *Journal of Educational Computing Research*, Vol. 10(2), p. 153-160.

CAVAIANI, T.P. (1989) Cognitive Style and Diagnostic Skills of Student Programmers. *Journal of Research on Computing in Education*, Summer, p. 411-420.

CHEN, H.-G. & VECCHIO, R.P. (1992). Nested IF-THEN-ELSE Constructs in End-User Computing: Personality and Aptitude as Predictors of Programming Ability. *International Journal of Man-Machine Studies*, 36, p. 843-859.

CLARIANA, R.B. (1993). The Motivational Effect of Advisement on Attendance and Achievement in Computer-Based Instruction. *Journal of Computer-Based Instruction*, Vol. 20, No. 2, p. 47-51.

CLARKE, J.A. (1990). Producing that First CAL Program: Experiences of a Novice. *Interactive Learning International*, Vol. 6, p. 143-151.

CLEMENTS, D.H. (1986). Developmental Differences in the Learning of Computer Programming: Achievement and Relationships to Cognitive Abilities. *Journal of Applied Developmental Psychology*, 7, p. 251-266.

COHEN, J. (1984). The Interaction Between Methods of Instruction and Individual Learning Style. *Educational Psychology*, Vol. 4, No. 1, p. 51-60.

COOMBS, M.J.; GIBSON, R & ALTY, J.L. (1981). Acquiring a First Computer Language: A Study of Individual Differences. In Computing Skills and the User Interface, Coombs, M.J. & Alty, J.L. (eds.), Academic Press, London, p. 289-313.

COOMBS, M.J.; GIBSON, R. & ALTY, J.L. (1982). Learning a First Computer Language: Strategies for Making Sense. *International Journal of Man-Machine Studies*, 16, p. 449-486.

DALBEY, J. & LINN, M.C. (1985). The Demands and Requirements of Computer Programming: A Literature Review. *Journal of Educational Computing Research*, Vol. 1(3), p. 253-274.

D'ARCY, J. (1985). Learning Pascal after BASIC. In Human-Computer Interaction - INTERACT '84, Shackel, B. (ed.), Elsevier Science Publishers, North-Holland, p. 771-775.

DAVIS, E.A.; LINN, M.C. & CLANCY, M.J. (1995). Students' Off-line and On-line Experiences. *Journal of Educational Computing Research*, Vol. 12 (2), p. 109-134.

DÉTIENNE, F. & SOLOWAY, E. (1990). An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies*, 33, p. 323-342.

DISESSA, A.A & ABELSON, H. (1986). Boxer: A Reconstructable Computational Medium. *Communications of the ACM*, Vol. 29, No. 9, p. 859-868.

DOUKIDIS, G.I.; ROGERS, R.A. & ANGELIDES, M.C. (1989). Developing a Pascal Tutoring Aid. *Computers and Education*, Vol. 13, No. 4, p. 367-378.

DU BOULAY, B & O'SHEA, T. (1981). Teaching Novices Programming. In Computing Skills and the User Interface, Coombs, M. & Alty, J. (eds.), Academic Press, London, p. 147-200.

DU BOULAY, B. & SOTHCOTT, C. (1987). Computers Teaching Programming: An Introductory Survey of the Field. In Artificial Intelligence and Education: Learning Environments and Intelligent Tutoring Systems, Lawler, R. W. & Yazdani, M. (eds.), Ablex Publishing, Norwood, New Jersey, p. 345-372.

DU BOULAY, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, Vol. 2(1), p. 57-73.

FALZON, P. (1990). Human-Computer Interaction: Lessons from Human-Human Communication. In Cognitive Ergonomics: Understanding, Learning and Designing Human-Computer Interaction, Falzon, P. (ed.), Academic Press, London, p. 51-65.

FAY, A.L. & MAYER, R.E. (1994). Benefits of Teaching Design Skills Before Teaching LOGO Computer Programming: Evidence for Syntax-Independent Learning. *Journal of Educational Computing Research*, Vol. 11(3), p. 187-210.

FRIEND, C.L. & COLE, C.L. (1990). Learner Control in Computer-Based Instruction: A Current Literature Review. *Educational Technology*, November, p. 47-49.

GOFORTH, D. (1994). Learner Control = Decision Making + Information: A Model and Meta-Analysis. *Journal of Educational Computing Research*, Vol. 11(1), p. 1-26.

GÖKTEPE, M.; ÖZGÜÇ, B. & BARAY, M. (1989). Design and Implementation of a Tool for Teaching Programming. *Computers and Education*, Vol. 13, No. 2, p. 167-178.

GOODWIN, L. & SANATI, M. (1986). Learning Computer Programming through Dynamic representation of computer functioning: evaluation of a new learning package for Pascal. *International Journal of Man-Machine Studies*, 25, p. 327-341.

GOODWIN, L. & WILKES, J.M. (1986). The Psychological and Background Characteristics Influencing Students' Success in Computer Programming. *AEDS Journal*, Fall, p. 1-9.

GOULD, J.D. (1975). Some Psychological Evidence on How People Debug Computer Programs. *International Journal of Man-Machine Studies*, 7, p. 151-182.

HARRISON, C. (1990). CAL in the Teaching of Initial Software Engineering. *The CTISS File*, September, p. 52-55.

HAZEN, M. (1992). Academic Computing: How to Address the Teaching and Learning Challenge. *New Directions for Teaching and Learning*, No. 51, p. 43- 53.

HOC J.M. (1977). Role of Mental Representations in Learning a Programming Language. *International Journal of Man-Machine Studies*, Vol. 9, p. 87-105.

JONES, A. (1985). How Novices Learn to Program. In Human-Computer Interaction - INTERACT '84, Shackel, B. (ed.), Elsevier Science Publishers, North-Holland, p. 777-783.

JONES, C. (1995). Gaps in Programming Education. *Computer*, April, p. 70-71.

JONI, S.-N.A. & SOLOWAY, E. (1986). But My Program Runs! Discourse Rules for Novice Programmers. *Journal of Educational Computing Research,* Vol. 2(1), p. 95-125.

JORDAAN, W.J. & JORDAAN, J.J. (1984). *Man in Context,* McGraw-Hill Book Company, Johannesburg.

KONVALINA, J.; WILEMAN, A. & STEPHENS, L.J. (1983). Math Proficiency: A Key to Success for Computer Science Students. *Communications of the ACM,* Vol. 26, No. 5, p. 377-382.

LANZA, A. & ROSELLI, T. (1989). An Evaluation of a CBI System for Computer Programming Language. *Journal of Computer-Based Instruction,* Vol. 16, No. 4, p. 126-128.

LEIBLUM, M.D. (1989). Implementing Computer-Aided Learning in a University Environment: Some Practical advice to a CAL Agency. *Educational Technology,* February, p. 27-31.

LEUTNER, D. (1993). Guided Discovery Learning with Computer-Based Simulation Games: Effects of Adaptive and Non-Adaptive Instructional Support. *Learning and Instruction,* Vol. 3, p. 113-132.

LIEBERMAN, H. (1987). An Example-Based Environment for Beginning Programmers. In Artificial Intelligence and Education: Learning Environments and Intelligent Tutoring Systems, Lawler, R. W. & Yazdani, M. (eds.), Ablex Publishing, Norwood, New Jersey, p. 135-151.

LIEBERMAN, H. (1984). Seeing what your programs are doing. *International Journal of Man-Machine Interaction,* Vol. 21, p. 311-331.

LINN, M. (1992). How can Hypermedia Tools Help Teach Programming? *Learning and Instruction,* Vol. 2, p. 119-139.

LINN, M.C. & CLANCY, M.J. (1992a). Can Expert's explanations help students develop program design skills?. *International Journal of Man-Machine Studies,* 36, p. 511-551.

LINN, M.C. & CLANCY, M.J. (1992b). The Case for Case Studies of Programming Problems. *ACM,* Vol. 35, No. 3, p. 121-132.

LINN, M.C. & DALBEY, J. (1985). Cognitive Consequences of Programming Instruction, Access, and Ability. *Educational Psychologist,* Vol. 20, No. 4, p. 191-206.

LUKEY, F.J. (1980). Understanding and Debugging programs. *International Journal of Man-Machine Studies*, 12, p. 189-202.

MADDUX, C.D. (1989). The Harmful Effects of Excessive Optimism in Educational Computing. *Educational Technology*, July, p. 23-29.

MARTIN, B. & HEARNE, J.D. (1990). Transfer of Learning and Computer Programming. *Educational Technology*, January, p. 41-44.

MATTA, K.F. & KERN, G.M. (1989). A Framework for Research in Computer-Aided Instruction: Challenges and Opportunities. *Computers in Education*, Vol. 13, No 1, p. 77-84.

MAYER, R.E. (1975). Different Problem-Solving Competencies Established in Learning Computer Programming With and Without Meaningful Models. *Journal of Educational Psychology*, Vol. 67, No. 6, p. 725-734.

MAYER, R.E. (1976). Some Conditions of Meaningful Learning for Computer Programming: Advance Organizers and Subject Control of Frame Order. *Educational Psychology*, Vol. 68, No. 2 , p. 143-150.

MAYER, R.E. (1981). The Psychology of How Novices Learn Computer Programming. *Computing Surveys*, Vol. 13, No. 1, p. 121-141.

MAYER, R.E; DYCK, J.L. & VILBERG, W. (1986). Learning to Program and Learning to Think: What's the Connection?. *Communications of the ACM*, Vol. 29, No. 7, p. 605-610.

MAZLACK, L.J. (1980). Identifying Potential to Acquire Programming Skill. *Communications of the ACM*, Vol. 23, No. 1, p. 14-17.

MERRILL, M.D. (1991). Constructivism and Instructional Design. *Educational Technology*, May, p. 45-53.

MESSICK, S. (1984). The Nature of Cognitive Styles: Problems and Promise in Educational Practice. *Educational Psychologist*, Vol. 19, No. 2, p. 59-74.

MILLER, L.A. (1974). Programming by Non-programmers. *International Journal of Man-Machine Studies*, 6, p. 237-260.

MILLER, M. L. (1978). A Structured Planning and Debugging Environment for Elementary Programming. *International Journal of Man-Machine Studies*, 11, p. 79-95.

PAXTON, A.L. & TURNER, E.J. (1984). The application of human factors to the needs of the novice computer user. *International Journal of Man-Machine Studies*, 20, p. 137-156.

PEA, R.D. (1986). Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research*, Vol. 2(1), p. 25-35.

PENNINGTON, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19, p. 295-341.

PERKINS, D.N. & MARTIN, F. (1986). Fragile Knowledge and Neglected Strategies in Novice Programmers. In Empirical Studies of Programmers, Soloway, E. & Iyengar, S. (eds.), Ablex Publishing Corporation, Norwood, New Jersey, p. 213-229.

PUTNAM, R.T.; SLEEMAN, D.; BAXTER, J.A. & KUSPA, L.K. (1986) A Summary of Misconceptions of High School BASIC Programmers. *Journal of Educational Computing Research,* Vol. 2(4), p. 459-473.

PYOTT, S. & SANDERS, I. (1991). ALEX: An Aid to Teaching Algorithms. *SIGCSE Bulletin*, Vol. 23, No. 3, p. 36-44.

RAY, N.M. & MINCH, R.P. (1990). Computer Anxiety and Alienation: Toward a Definitive and Parsimonious Measure. *Human Factors*, 32(4), p. 477-491.

REEVES, T.C. (1993). Pseudoscience in Computer-Based Instruction: The Case of Learner Control Research. *Journal of Computer-Based Instruction*, Vol. 20, No. 2, p. 39-46.

RIPLEY, G.D. & DRUSEIKIS, F.C. (1978). A Statistical Analysis of Syntax Errors. *Computer Languages*, Vol. 3, p. 227-240.

RIST, R.S. (1986). Plans in programming: Definition, Demonstration, and development. In Empirical Studies of Programmers, Soloway, E. & Iyengar, S. (eds.), Ablex Publishing Corporation, Norwood, New Jersey, p. 28-47.

ROBERTSON, S.P. & YU, C.-C. (1990). Common Cognitive Representations of Program Code Across Tasks and Languages. *International Journal of Man-Machine Studies*, 33, p. 343-360.

ROBINSON, C.G. (1981). Cloze procedure: A Review. *Educational Research*, Vol. 23,, No. 2 p. 128-133.

ROBSON, C. (1990). Designing and Interpreting Psychological Experiments. In Human Computer Interaction, Preece, J. & Keller, L. (eds.), Prentice Hall Press, New York, p. 257-367.

SALISBURY, D.F. (1990). Cognitive Psychology and its Implications for Designing Drill and Practice Programs for Computers. *Journal of Computer-Based Instruction*, Vol. 17, No. 1, p. 23-30.

SAMURÇAY, R. (1990). Understanding the Cognitive Difficulties of Novice Programmers: A Didactic Approach. In Cognitive Ergonomics: Understanding, Learning and Designing Human-Computer Interaction, Falzon, P. (ed.), Academic Press, London, p. 187-198.

SANDERS, I. & GOOPAL, H. (1991). AAPT: Algorithm Animator and Programming Toolbox. *Technical Report, 1991-14*, Computer Science Department, University of the Witwatersrand.

SCHANK, P.K.; LINN, M. & CLANCY, M.J. (1993). Supporting Pascal Programming with an On-Line Template library and Case Studies. *International Journal of Man-Machine Studies*, 38, p. 1031-1048.

SCHNEIDER, M.L. (1982). Models for the Design of Static Software User Assistance. In Directions in Human-Computer Interaction, Badre, A. & Schneiderman, B. (eds.), Ablex Publishing Corporation, Norwood, New Jersey, p. 1-26.

SCHNEIDERMAN, B. (1976). Exploratory Experiments in Programmer Behaviour. *International Journal of Computer and Information Sciences*, Vol. 5, No. 2, p. 123-143.

SHACKELFORD, R.L. & BADRE, A.N. (1993). Why Can't Smart Students Solve Simple Programming Problems? *International Journal of Man-Machine Studies*, 38, p. 985-997.

SHEIL, B.A. (1981). The Psychological Study of Programming. *Computing Surveys*, Vol. 13, No. 1, p. 101-120.

SIME, M.E. (1981). The Empirical Study of Computer Language. In Man-Computer Interaction: Human Factors Aspects of Computers & People, Shackel, B. (ed.), Sijthoff & Noordhoff, Netherlands, p. 215-244.

SLEEMAN, D. (1986). The Challenges of Teaching Computer Programming. *Communications of the ACM*, 29(9), p. 840-841.

SLEEMAN, D.; PUTNAM, R.T.; BAXTER, J. & KUSPA, L. (1986) Pascal and High School Students: A Study of Errors. *Journal of Educational Computing Research*, Vol. 2(1), p. 5-23.

SOLOWAY, E. (1986). Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, Vol. 29, No. 9, p. 850-859.

SOLOWAY, E. (1993). Should We Teach Students to Program. *Communications of the ACM*, Vol. 36, No. 10, p. 21-24.

SOLOWAY, E.; EHRLICH, K.; BONAR, J. & GREENSPAN, J. (1982). What Do Novices Know About Programming? In Directions in Human-Computer Interaction, Badre, A. & Schneiderman, B. (eds.), Ablex Publishing Corporation, Norwood, New Jersey, p. 27-54.

SPOHRER, J.C. & SOLOWAY, E. (1986). Analyzing the High Frequency Bugs in Novice Programs. In Empirical Studies of Programmers, Soloway, E. & Iyengar, S. (eds.), Ablex Publishing Corporation, Norwood, New Jersey, p. 230-251.

SPOHRER, J.C. & SOLOWAY, E. (1986). Novice Mistakes: Are the Folk Wisdoms Correct?. *Communications of the ACM*, Vol. 29, No. 7, p. 624-632.

STEINBERG, E.R. (1989). Cognition and Learner Control: A Literature Review. *Journal of Computer-Based Instruction*, Vol. 16, No. 4, p. 117-121.

STEMLER, L. (1989). Effects of Instruction on the Misconceptions About Programming in BASIC. *Journal of Research on Computing in Education*, Fall, p. 26-35.

TENNYSON, R.D. (1993). MAIS: A computer-based integrated instructional system. *Behavior Research Methods, Instruments, & Computers*, 25(2), p. 93-100.

VINEGRAD, M.D. (1989). Self Assessed Learning States and Computer Based Instruction. *Interactive Learning International*, Vol. 5, p. 117-119.

WEBB, G.I. (1988). A Knowledge-Based Approach to Computer-Aided Learning. *International Journal of Man-Machine Studies*, 29, p. 257-285.

WEBB, N.M. (1984). Microcomputer learning in small groups: cognitive requirements and group processes. *Journal of Educational Psychology*, Vol. 76, No. 6, p. 1076-1088.

WIEDENBECK, S. (1991). The Initial Stage of Program Comprehension. *International Journal of Man-Machine Studies*, 35, p. 517-540.

WILEMAN, S.; KONVALINA, J. & STEPHENS, L.J. (1981). Factors Influencing Success in Beginning Computer Science Courses. *Journal of Educational Research*, Vol. 74, No. 4, p. 223-226.

# A. PATMAN SUPPORT ENVIRONMENT PROGRAMS

## A.1 Basic concepts

### A.1.1 Trivial

```
program Trivial;
begin
end.
```

### A.1.2 Wrtsome

```
program wrtsome;
begin
   Writeln('This program');
   Writeln('actually does something.');
end.
```

### A.1.3 Writes1

```
program Writes1;
begin
  write('This program does nothing much!');
end.
```

### A.1.4 Writes2

```
program Writes2;
begin
  write('This program ');
  write('does nothing much!');
end.
```

### A.1.5 Writes3

```
program Writes3;
begin
  writeln('This program ');
  write('does nothing much!');
end.
```

### A.1.6 Wrtlines

```
program WrtLines;
begin
   Write('This will ');
   Write('all be ');
   Writeln('on one line.');
end.
```

### A.1.7 Goodform

```
program Good_Programming_Style;
begin
   Write('Programming style ');
   Write                  ('is a matter of ');
   Writeln                               ('personal choice');
   Write('Each person ');
   Write            ('can choose ');
   Writeln                     ('their own style');
   Write('They can be ');
   Write          ('very clear, or ');
   Writeln                       ('extremely messy');
end.
```

### A.1.8 Uglyform

```
program Ugly_Programming_Style;begin  Write('Programming style ')
;Write                    ('is a matter of ');
Writeln('personal choice');Write('Each person ');
Write('can choose ');Writeln
('their own style');Write('They can be ');Write
    ('very clear, or ');
Writeln('extremely messy');end.
```

### A.1.9 Comments

```
program Comments;
begin { This is the start of the main program }
(* This is a comment that is ignored by the Pascal compiler *)
{  This is also ignored }
   Writeln('Hi - Mom and Dad');
   Writeln('I am broke.');   {All students are!}
(*
   Writeln('Send money');
   Writeln('Send money');
 *)
   Writeln('Bye');  {writeln('this is the last line');}
end. (* This is the end of the main program *)
```

## A.2 Input and Output Concepts

### A.2.1 StrIN1

```
PROGRAM StrsIN1;
VAR firstname,surname:STRING;
BEGIN
   WRITELN('Enter your first name: ');
   READLN(firstname);
   WRITELN('Enter your surname: ');
   READLN(surname);
   WRITELN('Your surname is :',surname);
   WRITELN('Your first name is ',firstname);
END.
```

### A.2.2 StrIN2

```
PROGRAM StrsIN2;
VAR firstname,surname:STRING;
BEGIN
   WRITELN('Enter your first name: ');
   READLN(surname);
   WRITELN('Enter your surname: ');
   READLN(firstname);
   WRITELN('Your surname is :',surname);
   WRITELN('Your first name is ',firstname);
END.
```

### A.2.3 StrsIN1

```
PROGRAM StrsIN1;
VAR firstname,surname:STRING;
BEGIN
   WRITELN('Enter your first name: ');
   READLN(firstname);
   WRITELN('Enter your surname: ');
   READLN(surname);
   WRITELN('Your surname is :',surname);
   WRITELN('Your first name is ',firstname);
END.
```

### A.2.4 StrsIN2

```
PROGRAM StrsIN2;
VAR firstname,surname:STRING;
BEGIN
   WRITELN('Enter your first name: ');
   READLN(surname);
   WRITELN('Enter your surname: ');
   READLN(firstname);
   WRITELN('Your surname is :',surname);
   WRITELN('Your first name is ',firstname);
END.
```

### A.2.5 StrsIN3

```
PROGRAM StrsIN3;
VAR firstname,surname:STRING;
BEGIN
   WRITELN('Enter your first name: ');
   READLN(surname);
   WRITELN('Enter your surname: ');
   READLN(firstname);
   WRITELN('Your surname is :',firstname);
   WRITELN('Your first name is ',surname);
END.
```

### A.2.6 WrtRead

```
PROGRAM WrtRead;
VAR name:STRING;
BEGIN
  WRITELN('Enter your name: ');
  READLN(name);
  WRITELN(name);
  WRITELN('Your name is set to STUDENT');
  WRITELN(name);
END.
```

### A.2.7 MultRd1

```
PROGRAM MultiRead1;
VAR first,second:STRING;
BEGIN
  WRITELN('Enter two words: ');
  WRITELN(first);
  WRITELN(second);
END.
```

### A.2.8 MultRd2

```
PROGRAM MultiRead2;
VAR first,second,third:STRING;
BEGIN
  WRITELN('Enter two words: ');
  READLN(second,third,first);
  WRITELN(first);
  WRITELN(second);
  WRITELN(third);
END.
```

### A.2.9 MultRd3

```
PROGRAM MultiRead3;
VAR first,second:STRING;
BEGIN
  WRITELN('Enter two words: ');
  READLN(second,first);
  WRITELN(first);
  WRITELN(second);
  READLN(first);
  WRITELN(first);
  WRITELN(second);
END.
```

### A.2.10 IntIn1

```
Program Intin1;
var temp:integer;
begin
  writeln('Please enter an integer');
  readln(temp);
  writeln('User entered:',temp);
end.
```

### A.2.11 MultInt1

```
PROGRAM MultInt1;
VAR favourite,worst:INTEGER;
BEGIN
   WRITELN('Enter some numbers: ');
   READLN(favourite,worst);
   WRITELN('Your favourite number is ',favourite);
   WRITELN('Your worst number is ',worst);
END.
```

### A.2.12 MultInt2

```
PROGRAM MultInt2;
VAR num_bigger_10,num_smaller_10:INTEGER;
BEGIN
   WRITELN('Enter 4 numbers: ');
   READLN(num_smaller_10,num_bigger_10);
   WRITELN('Numbers greater than 10 are ',num_bigger_10);
   WRITELN('Numbers less than 10 are ',num_smaller_10);
END.
```

### A.2.13 MultInt3

```
PROGRAM MultInt3;
VAR first,second:INTEGER ;
BEGIN
  WRITELN('Enter two numbers: ');
  READLN(second,first);
  WRITELN(first);
  WRITELN(second);
  READLN(first);
  WRITELN(first);
  WRITELN(second);
END.
```

## A.3 If then statements

### A.3.1 IfThen1

```
PROGRAM ifthen1;
VAR number,lucky:INTEGER;
BEGIN
  WRITELN('ENTER A LUCKY NUMBER');
  READLN(lucky);
  WRITELN('ENTER ANOTHER NUMBER');
  READLN(number);
  If (number = lucky) then
     WRITELN('You entered a lucky number!');
END.
```

### A.3.2 IfThen2

```
PROGRAM ifthen2;
VAR number:INTEGER;
BEGIN
  WRITELN('ENTER A NUMBER');
  READLN(number);
  If number > 10 then
     WRITELN('The number is greater than 10 and');
  WRITELN('the number is ',number);
END.
```

### A.3.3 Ifelse1

```
PROGRAM ifelse1;
VAR number,lucky:INTEGER;
BEGIN
  WRITELN('ENTER A LUCKY NUMBER');
  READLN(lucky);
  WRITELN('ENTER ANOTHER NUMBER');
  READLN(number);
  IF (number = lucky) THEN
     WRITELN('You entered a lucky number!')
  ELSE WRITELN('You entered an unlucky number!');
END.
```

### A.3.4 Ifelse2

```
PROGRAM ifelse2;
VAR age:INTEGER;
BEGIN
  WRITELN('Please enter your age:');
  WRITE('Don''t lie');
  READLN(age);
  IF (age > 40) or (age < 16) THEN
  BEGIN
    WRITELN('You liar!')
    WRITELN('I can''t trust you!');
  END
  ELSE
  BEGIN
    WRITELN('You are an honest person!');
    WRITELN('You deserve my trust!');
  END;
END.
```

### A.3.5 Ifdemo1

```
program Demonstrate_Conditional_Branching1;
var One,Two,Three:integer;
begin
   writeln('enter three numbers');
   readln(one,two,three);
   if Three = (One + Two) then
      Writeln('three is equal to one plus two');
   if Three = 3 then begin
      Write('three is ');
      Write('equal to ');
      Write('one plus two');
      Writeln;
   end;
   if Two = 2 then
      Writeln('two is equal to 2 as expected')
   else
      Writeln('two is not equal to 2... rather strange');
end.
```

### A.3.6 Ifdemo2

```
program Demonstrate_Conditional_Branching2;
var One,Two,Three:integer;
begin
   writeln('Enter two numbers:');
   readln(one,two,three);
   if Two = 2 then
     if One = 1 then
        Writeln('one is equal to one')
     else
        Writeln('one is not equal to one')
   else
     if Three = 3 then
        Writeln('three is equal to three')
     else
        Writeln('three is not equal to three');
end.
```

## A.4 Assignment statements

### A.4.1 Assign1

```
PROGRAM assign1;
VAR num:INTEGER;
BEGIN
  num:=3;
  WRITELN(num);
END.
```

### A.4.2 Assign2

```
PROGRAM assign2;
VAR input,num:INTEGER;
BEGIN
  WRITELN('Enter a number: ');
  READLN(input);
  num:=input;
END.
```

### A.4.3 Assign3

```
PROGRAM assign3;
VAR input,num:INTEGER;
BEGIN
  WRITELN('Enter a number: ');
  READLN(input);
  num:=input;
  WRITELN(num);
  WRITELN('Enter a number: ');
  READLN(num);
  WRITELN(num);
END.
```

### A.4.4 Swop

```
PROGRAM swop;
VAR x,y,z:INTEGER;
BEGIN
  WRITELN('Enter some numbers: ');
  READLN(x,y,z);
  x:=y;
  y:=z;
  z:=x;
  WRITELN(x,' ',y,' ',z);
END.
```

## A.5 Looping constructs

### A.5.1 Forloop1

```
PROGRAM forloop1;
VAR i,num:INTEGER;
BEGIN
  WRITELN('Hello');
  FOR i:= 1 TO 3 DO
  BEGIN
    WRITE('Enter a number: ');
    READLN(num);
  END;
  WRITELN('You entered number ',num);
  WRITELN('Goodbye');
END.
```

### A.5.2 Forloop2

```
PROGRAM forloop2;
VAR i,num:INTEGER;
BEGIN
  WRITELN('Hello');
  FOR i:= 1 TO 3 DO
  BEGIN
    WRITE('Enter a number: ');
    READLN(num);
    WRITELN('You entered number ',num);
  END;
  WRITELN('Goodbye');
END.
```

### A.5.3 EasyWhile

```
PROGRAM easywhile;
VAR j:INTEGER;
BEGIN
  j:=3;
  WHILE j < 7 DO
  BEGIN
    WRITELN('Smile!');
    j:=j+3;
  END;
END.
```

### A.5.4 While1

```
PROGRAM while1;
VAR i,num:INTEGER;
BEGIN
  WRITELN('Hello');
  i:=1;
  WHILE (i < 4) DO
  BEGIN
    WRITE('Enter a number:');
    READLN(num);
    i:=i+1;
  END;
  WRITELN('You entered number ',num);
  WRITELN('Goodbye');
END.
```

### A.5.5 While2

```
PROGRAM while2;
VAR i,num:INTEGER;
BEGIN
  WRITELN('Hello');
  i:=1;
  WHILE (i < 4) DO
  BEGIN
    WRITE('Enter a number:');
    READLN(num);
    i:=i+1;
    WRITELN('You entered number ',num);
  END;
  WRITELN('Goodbye');
END.
```

### A.5.6 EasyRepeat

```
PROGRAM easyrepeat;
VAR j:INTEGER;
BEGIN
  j:=3;
  REPEAT
    WRITELN('Happy Birthday!');
    j:=j+3;
  UNTIL (j > 7);
END.
```

### A.5.7 Repeat1

```
PROGRAM repeat1;
VAR i,num:INTEGER;
BEGIN
  WRITELN('Hello');
  i:=1;
  REPEAT
    WRITE('Enter a number:');
    READLN(num);
    i:=i+1;
  UNTIL (i > 3);
  WRITELN('You entered number ',num);
  WRITELN('Goodbye');
END.
```

### A.5.8 Repeat2

```
PROGRAM repeat2;
VAR i,num:INTEGER;
BEGIN
  WRITELN('Hello');
  i:=1;
  REPEAT
    WRITE('Enter a number:');
    READLN(num);
    i:=i+1;
    WRITELN('You entered number ',num);
  UNTIL (i > 3);
  WRITELN('Goodbye');
END.
```

### A.5.9 BigTest

```
PROGRAM BigTest;
VAR name:string;
BEGIN
  BEGIN
   WRITELN('What is your name?');
   READLN(name);
  END;
  WRITELN('Your name is ',name);
END.
```

### A.5.10 DataIn

```
PROGRAM datain;
VAR i,num:INTEGER;
BEGIN
  WRITE('Enter some numbers: ');
  READLN(i);
  FOR i:= 1 TO 3 DO
  BEGIN
    WRITE('Enter a number: ');
    READLN(num);
  END;
  WRITELN('You entered number ',num);
  WRITELN('Goodbye');
END.
```

## A.6 Looping Constructs (advanced)

### A.6.1 For_Guess

```
PROGRAM for_guess;
VAR mynum,yournum,counter:INTEGER;
BEGIN
  mynum:=13;
  WRITE('Guess the number I am thinking of! ');
  READLN(yournum);
  FOR counter:=9 TO 12 do BEGIN
    IF (yournum > mynum) THEN
      WRITE('Too High.. guess again: ');
    IF (yournum < mynum) THEN
      WRITE('Too Low.. guess again: ');
    READLN(yournum);
  END;
  IF (yournum = mynum) THEN
    WRITELN('You guessed correctly the fourth time!')
  ELSE WRITELN('You failed to guess correctly even after four guesses!');
END.
```

### A.6.2 Rep_Guess

```
PROGRAM repeat_guess;
VAR mynum,yournum,try:INTEGER;
BEGIN
  mynum:=5;    try:=1;
  WRITE('Guess the number I am thinking of! ');
  READLN(yournum);
  REPEAT
    IF (yournum > mynum) THEN
       WRITE('Too High.. guess again: ');
    IF (yournum < mynum) THEN
       WRITE('Too Low.. guess again: ');
    IF yournum <> mynum THEN BEGIN
      try:=try+1;
      READLN(yournum);
    END;
  UNTIL (yournum = mynum);
  WRITE('You guessed my number in ',try,' guesses.');
END.
```

### A.6.3 Whi_Guess

```
PROGRAM While_guess;
VAR mynum,yournum,try:INTEGER;
BEGIN
  mynum:=18;    try:=1;
  WRITE('Guess the number I am thinking of! ');
  READLN(yournum);
  WHILE (yournum <> mynum) DO
  BEGIN
    IF (yournum > mynum) THEN
      WRITE('Too High.. guess again: ')
    ELSE
      WRITE('Too Low.. guess again: ');
    try:=try+1;
    READLN(yournum);
  END;
  WRITELN('You guessed my number in ',try,' guesses');
END.
```

### A.6.4 Whi_Test

```
PROGRAM while_test;
VAR result:INTEGER;
BEGIN
  WRITE('Enter your test result (%): ');
  READLN(result);
  WHILE (result < 0) or (result >100) DO
  BEGIN
    WRITE('Please re-enter your result (0..100): ' );
    READLN(result);
  END;
  IF result > 75 then
   WRITELN('Well done !')
  ELSE IF result > 50 THEN
    WRITELN('Good!')
  ELSE WRITELN('Better luck next time!');
END.
```

### A.6.5 Rep_Age

```
PROGRAM repeatage;
VAR age:INTEGER;
BEGIN
  REPEAT
    WRITE('Please enter your age: ');
    READLN(age);
    IF (age >= 30) or (age <=16) THEN
      WRITELN('You liar!');
  UNTIL (age > 16) and (age < 30);
  WRITELN('You are ',age,' years old');
END.
```

## A.6.6 Square

```
PROGRAM square;
VAR size,i,j:INTEGER;
BEGIN
  WRITE('Enter the size of the square:');
  READLN(size);
  FOR i:=1 TO size DO
  BEGIN
    FOR j:=1 TO size DO
      WRITE('*');
    WRITELN;
  END;
END.
```

## A.6.7 Rectangle

```
PROGRAM rectangle;
VAR width,height,col,row:INTEGER;
BEGIN
  WRITE('Enter the width of the rectangle: ');
  READLN(width);
  WRITE('Enter the height of the rectangle: ');
  READLN(height);
  FOR col:=1 TO height DO
  BEGIN
    FOR row:=1 TO width DO
      WRITE('X');
    WRITELN;
  END;
END.
```

## A.6.8 Rep_Result

```
PROGRAM repeat_results;
VAR num,nostudents,result:INTEGER;
BEGIN
  WRITE('How many student''s marks would you like to enter? ');
  READLN(nostudents);
  num:=1;
  REPEAT
    REPEAT
      WRITE('Enter mark for student ',num,' :');
      READLN(result);
    UNTIL (result >= 0) and (result <=100);
    num:=num+1;
  UNTIL (num > nostudents);
  WRITELN('Data entry is complete!');
END.
```

## A.6.9 Whi_Result

```
PROGRAM while_results;
VAR num,nostudents,result:INTEGER;
BEGIN
  WRITE('How many student''s marks would you like to enter? ');
  READLN(nostudents);
  num:=1;
  WHILE (num < nostudents) DO
  BEGIN
    REPEAT
      WRITE('Enter mark for student ',num,' :');
      READLN(result);
    UNTIL (result >= 0) and (result <=100);
    num:=num+1;
  END;
  WRITELN('Data entry is complete!');
END.
```

### A.6.10 For_Result

```
PROGRAM for_results;
VAR num,nostudents,result:INTEGER;
BEGIN
  WRITE('How many student''s marks would you like to enter? ');
  READLN(nostudents);
  FOR num:=1 to nostudents DO
  BEGIN
    REPEAT
      WRITE('Enter mark for student ',num,' :');
      READLN(result);
    UNTIL (result >= 0) and (result <=100);
  END;
  WRITELN('Data entry is complete!');
END.
```

## A.7 Data Types

### A.7.1 Arrays1

```
PROGRAM arrays1;
VAR nums:ARRAY [1..3] OF INTEGER;
BEGIN
  nums[1]:=1;
  nums[2]:=nums[1]+1;
  nums[3]:=nums[1]+nums[2];
  nums[1]:=4;
  nums[2]:=nums[1]-nums[3];
  WRITELN(nums[1]);
  WRITELN(nums[2]);
  WRITELN(nums[3]);
END.
```

### A.7.2 Arrays2

```
PROGRAM arrays2;
VAR nums:ARRAY [1..3] OF INTEGER;
    i:INTEGER;
BEGIN
  nums[1]:=1;
  nums[2]:=nums[1]+1;
  nums[3]:=nums[1]+nums[2];
  nums[1]:=10;
  nums[2]:=nums[1]-nums[3];
  FOR i:=1 TO 3 DO
    WRITELN(nums[i]);
END.
```

### A.7.3 StuMarks

```
PROGRAM stumarks;
VAR marks:ARRAY [0..2] OF INTEGER;
    j:INTEGER;
BEGIN
  FOR j:=0 TO 2 DO BEGIN
    WRITE('Enter mark for student ',j,': ');
    READLN(marks[j]);
  END;
  marks[2]:=marks[2]-1;
  FOR j:=0 TO 2 DO
    WRITELN(marks[j]);
END.
```

## A.8 Procedures and Functions

```
ProBasic
PROGRAM probasic;

PROCEDURE message;
BEGIN
  WRITELN('HELLO');
END;

BEGIN
  message;
  message;
END.
```

### A.8.1 FunBasic

```
PROGRAM funbasic;
VAR count:INTEGER;

FUNCTION first:INTEGER;
BEGIN
  first:=count+1;
END;

BEGIN
  count:=1;
  IF (first>1) THEN
    WRITELN('First Hello')
  ELSE WRITELN('Goodbye');
END.
```

### A.8.2 Greeting

```
PROGRAM greeting;
VAR character:CHAR;

PROCEDURE message(ch:CHAR);
BEGIN
  IF ch IN ['g','G'] then WRITELN('Good Luck')
  ELSE IF ch IN ['h','H'] then WRITELN('Hello')
       ELSE WRITELN('No message selected');
END;

BEGIN
  WRITE('Enter a character: ');
  READLN(character);
  message(character);
END.
```

### A.8.3 Greet2

```
PROGRAM greet2;
VAR character:CHAR;

FUNCTION valid(ch:CHAR):BOOLEAN;
BEGIN
  IF ch IN ['g','G','h','H'] THEN valid:=TRUE
  ELSE BEGIN
    valid:=FALSE;
    WRITELN('Not a valid character!');
  END;
END;

PROCEDURE message(ch:CHAR);
BEGIN
  IF ch IN ['g','G'] then WRITELN('Good Luck')
  ELSE IF ch IN ['h','H'] then WRITELN('Hello');
END;

BEGIN
  WRITE('Enter a character: ');
  READLN(character);
  IF valid(character) THEN
    message(character);
END.
```

### A.8.4 Adding1

```
PROGRAM adding1;
VAR total,number,i,n:INTEGER;

PROCEDURE calctotal; {Calcultates the sum of all numbers inputted}
BEGIN
  total:=0;
  FOR i:=1 TO 4 DO BEGIN
    WRITE('Enter number ',i,': ');
    READLN(number);
    total:=total+number;
  END; {for-loop}
END; {findhighest}

BEGIN  {main program}
  calctotal;
  WRITELN('The sum of the 4 numbers was ',total);
END.
```

### A.8.5 Average

```
PROGRAM average;
VAR ave,number,i:INTEGER;

FUNCTION total:INTEGER; {Returns the sum of all numbers inputted}
VAR sum:INTEGER;
BEGIN
  sum:=0;
  FOR i:=1 TO 4 DO BEGIN
    WRITE('Enter number ',i,': ');
    READLN(number);
    sum:=sum+number;
  END; {for-loop}
  total:=sum;
END; {findhighest}

BEGIN  {main program}
  ave:=total DIV 4;
  WRITELN('The average of the nubers is', ave);
END.
```

### A.8.6 Inorder1

```
PROGRAM testord1;
VAR num1,num2,num3:INTEGER;

FUNCTION inorder:BOOLEAN;
BEGIN
  IF num1 <= num2 THEN
    inorder:=(num2<=num3)
  ELSE inorder:=FALSE;
END;

BEGIN
  WRITE('Enter 3 values: ');
  READLN(num1,num2,num3);
  IF inorder THEN
    WRITELN('Numbers are in order')
  ELSE WRITELN('Numbers are not in order');
END.
```

## A.8.7 Inorder2

```
PROGRAM testord2;
VAR num1,num2,num3:INTEGER;

FUNCTION inorder(x1,x2,x3:INTEGER):BOOLEAN;
BEGIN
  IF x1 <= x2 THEN
    inorder:=(x2<=x3)
  ELSE inorder:=FALSE;
END;

BEGIN
  WRITE('Enter 3 values: ');
  READLN(num1,num2,num3);
  IF inorder(num1,num2,num3) THEN
    WRITELN('Numbers are in order')
  ELSE
    IF inorder(num3,num2,num1) THEN
      WRITELN('Numbers are in reverse order')
    ELSE WRITELN('Numbers are not in order');
END.
```

## A.8.8 Add2

```
PROGRAM add2;
VAR i,thetotal:INTEGER;                         {Global Variables}

FUNCTION total(num:INTEGER):INTEGER;
VAR sum,i,number:INTEGER;               {Local Variables}
BEGIN
sum:=0;
FOR i:=1 TO num DO BEGIN
WRITE('Enter number ',i,': ');
READLN(number);
sum:=sum+number;
END; {for-loop}
total:=sum;
END; {calc total}

BEGIN  {main program}
WRITE('How many numbers would you like to add:');
READLN(i);
Thetotal:=total(i);
WRITELN('The total of the ',i,' numbers is ', TheTotal);
END.
```

## A.8.9 Confuse1

```
PROGRAM confuse1;
VAR i,j,k:INTEGER; {global}

PROCEDURE Change(i:INTEGER;VAR j:INTEGER);
VAR k:INTEGER;  {local}
BEGIN
  k:=2;
  i:=i+k;
  j:=j+1;
  WRITELN(i,' ',j,' ',k);
END;

BEGIN
  i:=1; j:=2;  k:=3;
  change(i,j); {passing parameters}
  WRITELN(i,' ',j,' ',k);
END.
```

### A.8.10 Confuse2

```
PROGRAM confuse2;
VAR x,y,z:INTEGER;

PROCEDURE Change(VAR x:INTEGER;y:INTEGER);
VAR z:INTEGER;
BEGIN
  z:=13;
  WRITELN(x,' ',y,' ',z);
  x:=11;
  y:=12;
END;

BEGIN
  x:=1; y:=2;  z:=3;
  change(x,y);
  WRITELN(x,' ',y,' ',z);
END.
```

## A.9 For the BRAVE

### A.9.1 Printing

```
PROGRAM printing;

PROCEDURE print(ch:CHAR;tot:INTEGER);
VAR i:INTEGER;
BEGIN
  FOR i:= 1 to tot DO
    WRITE(ch,',');
END;

BEGIN
  print('*',2);
  print('$',6 DIV 2);
  WRITELN;
  print('@',1);
END.
```

### A.9.2 MyAbsolute

```
PROGRAM myabsolute;
VAR num1,num2:REAL;

FUNCTION myabs(num:REAL):REAL;
BEGIN
  IF (num < 0 ) THEN myabs:= 0 - num
  ELSE myabs:=num;
END;

BEGIN
  WRITE('Enter 2 numbers');
  READLN(num1,num2);
  WRITELN('The absolute value of ',num1,' is ',myabs(num1));
  WRITELN('The absolute value of ',num2,' is ',myabs(num2));
  WRITELN('The absolute value of ',num2:2:2,' is ', myabs(num2):2:2);
END.
```

### A.9.3 Judges

```
PROGRAM Judges;
VAR score: ARRAY[1..4] OF INTEGER;
    max:INTEGER;

PROCEDURE getscores;
VAR i:INTEGER;
BEGIN
  FOR i:=1 TO 4 DO BEGIN
    WRITE('Enter mark for judge number ',i,' :');
    READLN(score[i]);
  END;
END;{getscores}

FUNCTION findmax:INTEGER;
VAR tempmax,i:INTEGER;
BEGIN
  tempmax:=score[1];
  FOR i:= 2 TO 4 DO BEGIN
    IF score[i] > tempmax THEN
      tempmax:=score[i];
  END;
  findmax:=tempmax;
END;{findmax}

BEGIN
  getscores;
  max:=findmax;
  WRITELN('The maximum score was ',max);
END.
```

### A.9.4 Sorting

```
PROGRAM sorting;
VAR score: ARRAY[1..3] OF INTEGER;

PROCEDURE getscores;{gets judges' scores from user}
VAR i:INTEGER;
BEGIN
  FOR i:=1 TO 3 DO BEGIN
    WRITE('Enter mark for judge number ',i,' :');
    READLN(score[i]);
  END;
END;{getscores}

PROCEDURE swopscores(VAR m,n:INTEGER); {swops elements m and n}
VAR temp:INTEGER;
BEGIN
  temp:=m;
  m:=n;
  n:=temp;
END;{swopscores}

PROCEDURE sortscores; {sorts the scores into ascending order}
VAR i:INTEGER;
    swop:BOOLEAN;
BEGIN
 REPEAT
   swop:=FALSE
   FOR i:= 1 TO 2 DO BEGIN
     IF score[i] > score[i+1] THEN BEGIN
       swopscores(score[i],score[i+1]);
       swop:=TRUE;
     END;
   END;
 UNTIL not swop;
END;{sortscores}

BEGIN
  getscores;
  sortscores;
  WRITELN('The scores in order are: ',score[1],score[2],score[3]);
END.
```

## A.9.5 Telephone

```
PROGRAM teledirectory;
VAR phone:ARRAY [1..3] OF REAL;
    name:ARRAY[1..3] OF string;
    continue:CHAR;

PROCEDURE makedirectory;
VAR i:INTEGER;
BEGIN
  FOR i:=1 TO 3 DO
  BEGIN
    WRITE('Enter name ',i,' :');
    READLN(name[i]);
    WRITE('Enter telephone number for ',name[i],' :');
    READLN(phone[i]);
  END;
END;{makedirectory}

PROCEDURE searchnumber;
VAR i:INTEGER;
    person:STRING;
BEGIN
  WRITE('Enter name: ');
  READLN(person);
  i:=0;
  REPEAT
    inc(i);
  UNTIL (i > 3) OR (name[i] = person);
  IF i > 3 THEN WRITELN('Sorry selected person not listed!')
  ELSE WRITELN('Telephone number: ',phone[i]:8:0);
END;

BEGIN
  makedirectory;
  REPEAT
    searchnumber;
    WRITE('Do you want to continue (y/n): ');
    READLN(continue);
  UNTIL (continue IN ['n','N']);
END.
```

## B. QUESTIONNAIRES 1, 2 AND PATMAN EVALUATION QUESTIONNAIRE

# Intro. to Programming
# Questionnaire 1

Student Number: ______

Please note

1. All information collected from this questionnaire will be treated as confidential.
2. The information will be used for research only. It will not effect your final result.

# Intro. to Programming: Questionnaire 1

## Examples

a. Red is my favourite colour (Y/N): ______
Answer either Y (for Yes) or N (for N).

b. For every statement, place a cross in the column that best describes your feelings.

| Colour preference | Strongly disagree | Disagree | Neither Agree nor Disagree | Agree | Strongly Agree |
|---|---|---|---|---|---|
| I like the colour red | | | | | |
| I like the colour green | | | | | |

c. Circle the letter of the statement that best describes your personal preference.

a. I prefer the colour red to green
b. I prefer the colour green to red

Questions

1. Home language: ____________________

2. Parent's occupations
Father: ________________________________________
Mother: ________________________________________

3. Number of years at University: ______ years

4. Number of computer courses completed? ______ (Excluding Computer Science at school)

5. Have you worked with a computer before this term? (Y/N) ______
If you answered yes, what did you use it for?

| | | | |
|---|---|---|---|
| Word processing | ______ | Spread Sheets | ______ |
| Programming | ______ | Games | ______ |
| Data Base | ______ | Other (Specify) | ______ |

6. Do you (or your family) have a personal computer at home? (Y/N) ______

7. Matric Subjects and Results    Examining Department ______________

| | Subject | Result | | Subject | Result |
|---|---|---|---|---|---|
| 1. | ______________ | ___ | 5. | ______________ | ___ |
| 2. | ______________ | ___ | 6. | ______________ | ___ |
| 3. | ______________ | ___ | 7. | ______________ | ___ |
| 4. | ______________ | ___ | 8. | ______________ | ___ |

Signature: ______________ (You make look at my varsity records to obtain results.)

8. Circle the letter next to the statement that best describes your personal preference.

   a. Most of the time I like to learn alone.
   b. Most of the time I prefer learning in pairs to learning alone.
   c. Most of the time I prefer to be taught than to learn by myself.

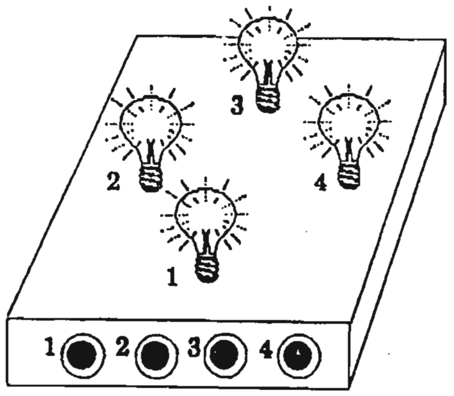
9. Circle the letter next to the statement that best describes your personal preference.

   a. A well-defined problem with several possible solutions based on one's approach.
   b. A well-defined problem with a single unique solution, which can be proven to be correct or incorrect.

10. For each statement, mark the block which best describes your feelings or thoughts.

| Statements | Strongly Disagree | Disagree | Neither Agree or Disagree | Agree | Strongly Agree |
|---|---|---|---|---|---|
| I expect the present course (i.e. IP) to be difficult for me. | | | | | |
| I hesitate to use a computer for fear of making mistakes that I cannot correct. | | | | | |
| I am confident that I could learn computer skills. | | | | | |
| Our country relies too much on computers. | | | | | |
| Computers are changing the world too rapidly. | | | | | |
| The computer interferes with professional relationships among people. | | | | | |
| The best computer programmers are creative. | | | | | |
| The best computer programmers plan work carefully to spend as little time as possible at the terminal. | | | | | |
| The best computer programmers prefer to write simple, specific programs to solve particular tasks. | | | | | |

11. The figure below shows four light bulbs (labelled 1,2,3,4). Four switches are connected so that each switch controls the light bulb with the corresponding number.



General Procedure

1. Turn on the light bulb that is directly across from the single light bulb that is on.
2. If any odd-numbered light bulb is on, go to step 4.
3. Turn off the lowest numbered light bulb, and go to step 5.
4. Turn off the highest numbered light bulb.
5. Turn on the bulb next to the highest numbered bulb that is on, in the clockwise direction.
6. Turn off any even-numbered bulbs which might be on, and stop.

Answer the following questions.

i. Assume only light bulb #1 is on. Perform the procedure, starting with step 1.
When you stop in step 6, which is/are correct. (Circle the letter next to the correct statement/s.)

a. Light bulbs #3 and #4 are on.
b. No light bulbs are on.
c. Only light bulb #1 is on.
d. Only light bulb #2 is on.
e. None of the above.

ii. Perform the procedure again. This time assume only light bulb #2 is on in the beginning. When you stop in step 6, which is/are correct. (Circle the letter next to the correct statement/s.)

a. Only light bulb #1 is on.
b. Light bulbs #2 and #3 are on.
c. At least three light bulbs are on.
d. Only two light bulbs are on.
e. None of the above.

iii. Again perform the procedure, this time assuming only light bulb #3 is on initially. When you stop in step 6, which is/are correct. (Circle the letter next to the correct statement/s.)

a. Only light bulb #2 is on.
b. Only light bulb #3 is on.
c. Only light bulb #4 is on.
d. All light bulbs will be on.
e. None of the above.

iv. Finally, perform the procedure assuming only light bulb #4 was initially on. When you stop in step 6, which is/are correct. (Circle the letter next to the correct statement/s.)

a. Light bulbs #2 and #4 are on.
b. Light bulbs #1 and #3 are on.
c. At least one even-numbered bulb will be on.
d. At least one odd-numbered bulb will be on.
e. None of the above.

v. Based on your experience in performing this procedure, which is/are correct. (Circle the letter next to the correct statement/s.)

a. The instructions can be applied regardless of the number of light bulbs initially turned on.
b. Regardless of which light bulb was initially on, when we stop in step 6 all light bulbs will be off.
c. Regardless of which light bulb was initially on, when we stop in step 6 only light bulb #1 will be on.
d. When an even-numbered bulb is initially turned on, then when we stop in step 6 only light bulb #3 will be on.
e. None of the above.

---

If you have had no prior programming experience STOP HERE !

---

12. Have you attended Introduction to Programming lectures prior to this year? _____

    If yes, when: ____________________________
    Did you complete the course, but failed? (Y/N) ______
    If you did not complete the course, for how long did you attend
    the lectures?____________________________________weeks/months

13. How knowledgeable are you of the following computer languages?
    (For each row, mark the block which best describes your knowledge)

| Programming Language | No Knowledge | Little Knowledge | Average Knowledge | Expert Knowledge |
|---|---|---|---|---|
| Basic | | | | |
| FORTRAN | | | | |
| Pascal | | | | |
| C | | | | |
| Other (specify) ____________________ | | | | |

14. Circle the statement(s) that is (which are) not a Pascal programming statement.

a. writeln b. y:=y DIV 3 c. readln(sum) d. while e. go

15. After the following program is executed, what is the final value stored in variable x? (Circle the letter next to the correct value.)

```
PROGRAM xtest;
VAR i,x:INTEGER;
BEGIN
  x:=400;
  FOR i:=1 TO 3 DO
  BEGIN
    x := x + i;
    WRITE(x);
  END;
END;
```

a. 403 b. 400 c. 6 d. 406. e. 3

# Intro. to Programming
# Questionnaire 2

Student Number: _____

Please note
1. All information collected from this questionnaire will be treated as confidential.
2. The information will be used for research only. It will not effect your final result.

## Intro. to Programming: Questionnaire 2

Please answer all questions in the stipulated format.

1. What is your (intended)Major : ______________________________
   Faculty : ______________________________

(E.g. Arts, Agric., Commerce, Social Science, Science)

2. Have you previously attempted a university programming course? (Y/N) _____

If yes, please complete the following:
course name(s): ______________________________________
year(s): ________________________________
final grade(s): ________

If you did not complete the course, approximately how long did you attend the course?____

3. I have at times thought seriously of dropping this course. (Y/N) _____

4. Do you intend using the knowledge you have gained in this course? (Y/N) ______

If yes, where or when:(e.g. work/personal projects) _________________________

5. Mark the answers which best describe your response to both statements.

The lecturer for this course was helpful.

| | |
|---|---|
| Strongly Disagree | ______ |
| Disagree | ______ |
| Agree | ______ |
| Strongly Agree | ______ |

The tutor (Mr McKenzie/Mr Tooke) for this course was helpful.

| | |
|---|---|
| Strongly Disagree | ______ |
| Disagree | ______ |
| Agree | ______ |
| Strongly Agree | ______ |

6. The homework assignments have been more beneficial than the classroom presentations. (Y/N) ______

7. Place a mark in the block which best describes your attitude or response.

| The program assignments | Strongly Disagree | Disagree | Neither Agree nor Disagree | Agree | Strongly Agree |
|---|---|---|---|---|---|
| have been more difficult than I expected | | | | | |
| have been more time consuming than I expected | | | | | |
| have been more frustrating than I expected | | | | | |
| were easy to do | | | | | |
| went smoothly | | | | | |

8. Mark the block which best describes your response to every statement.

| Statements | Strongly Disagree | Disagree | Neither Agree nor Disagree | Agree | Strongly Agree |
|---|---|---|---|---|---|
| I clearly understand what input computers want. | | | | | |
| I don't feel helpless when using the computer. | | | | | |
| I am sure of my ability to interpret a computer output. | | | | | |
| I don't understand computer output. | | | | | |
| Working with computers is so complicated it is difficult to understand what is going on. | | | | | |
| I like to use computers. | | | | | |
| I don't care what other people say, computers are not for me. | | | | | |
| The computer interferes with my work. | | | | | |
| The computer doesn't interfere with my personal relationships with people. | | | | | |

# PATMAN Tutorial Questionnaire

1. Do you think that you have benefited from using PATMAN?

| Strongly Disagree | Disagree | Neither Agree nor Disagree | Agree | Strongly Agree |
|---|---|---|---|---|
| | | | | |

Indicate in which way/ways PATMAN helped you (if any)

understanding programs ____________ with the weekly program assignments __

learning programming constructs _____ OTHER(specify) __________________

2. Did you find the PATMAN programs/lessons

| | Strongly Disagree | Disagree | Neither Agree nor Disagree | Agree | Strongly Agree |
|---|---|---|---|---|---|
| interesting | | | | | |
| enjoyable | | | | | |

3. Did you like the fact that you could work at your own pace and determine which lessons you were going to study when and how often?

| Strongly Disagree | Disagree | Neither Agree nor Disagree | Agree | Strongly Agree |
|---|---|---|---|---|
| | | | | |

4. Did you find the PATMAN tutorials more beneficial than the standard tutorials?

| Strongly Disagree | Disagree | Neither Agree nor Disagree | Agree | Strongly Agree |
|---|---|---|---|---|
| | | | | |

5. Would you have preferred to have had access to PATMAN so you could use it anytime (i.e. possibly in the STUDEN LAN room) and for any length of time?

| Strongly Disagree | Disagree | Neither Agree nor Disagree | Agree | Strongly Agree |
|---|---|---|---|---|
| | | | | |

Indicate how frequently you would have liked to have used PATMAN ____________

& how many hours a week you would have liked to have used PATMAN __________

6. Did you find it necessary to attend all PATMAN tutorial to enable you to keep up? Y/N _

7. Pleases suggest any improvements that you would have like to have seen in PATMAN (or any other comments you would like to make regarding PATMAN [good or bad])

______________________________________________________________

__________________________________________

# C. STUDENT BACKGROUND AND PSYCHOLOGICAL CHARACTERSTICS

## C.1 Background Characteristics

| Background Characteristics | Time | Control '92 | Control '93 | PATMAN | 92 vs 93 | 92 vs PAT | 93 vs PAT |
|---|---|---|---|---|---|---|---|
| Matric Points (Mean[1]) | T1 | 34.8 | 32.4 | 35.5 | - | - | 0.05 |
| Number of years at University (Mean) | T1 | 1.9 | 2.2 | 1.7 | - | - | - |
| Number of computer courses completed. (Mean) | T1 | 0.3 | 0.3 | 0.2 | - | - | - |
| Previous computer experience (none=0, max. = 5). (Mean) | T1 | 2.0 | 1.5 | 1.8 | - | - | - |
| Previous programming experience. (%[2]) | T1 | 21.31 | 24.24 | 10.00 | - | - | - |
| Agricultural faculty students (%) | T1 | 8.20 | 6.06 | 5.00 | - | - | - |
| Science Faculty students (%) | T1 | 86.89 | 90.91 | 87.50 | - | - | - |
| English home language students (%) | T1 | 68.85 | 57.58 | 55.00 | - | - | - |
| Black students (%) | T1 | 19.67 | 42.42 | 40.00 | - | 0.05 | - |
| Indian students (%) | T1 | 19.67 | 6.06 | 5.00 | - | 0.05 | - |
| White students(%) | T1 | 60.66 | 51.52 | 55 | - | - | - |
| Female students (%) | T1 | 49.18 | 60.61 | 40.00 | - | - | - |
| Family has personal computer (%) (#[3]) | T1 | 47.54 | 42.42 | 45.00 | - | - | - |
| Father: post school education (%) | T1 | 39.34 | 42.42 | 30.00 | - | - | - |
| Mother: post school education (%) | T1 | 16.39 | 27.27 | 25.00 | - | - | - |

[1] Mean score/rating for student group.

[2] Percentage of student group.

[3] Goodwin and Sanati (1986).

## C.2 Attitudinal and Psychological Characteristics

| Attitude and Psychological Characteristics | Time | Control '92 | Control '93 | PATMAN | 92 vs 93 | 92 vs PAT | 93 vs PAT |
|---|---|---|---|---|---|---|---|
| Problem Type: prefer a well-defined problem with several possible solutions based on one's approach.(%) | T1 | 49.18 | 57.58 | 62.50 | - | - | - |
| Problem Type: prefer a problem with a single unique solution, which can be proven to be correct or incorrect. (%) | T1 | 50.82 | 42.42 | 37.50 | - | - | - |
| Learning Style: I like to learn alone. (%) | T1 | 67.21 | 63.64 | 77.50 | - | - | - |
| Learning Style: I prefer learning in pairs to learning alone. (%) | T1 | 22.95 | 24.24 | 17.5 | - | - | - |
| Learning Style: I prefer to be taught than to learn by myself. (%) | T1 | 9.84 | 12.12 | 5.00 | - | - | - |
| I intend using the knowledge I have gained in this course. (%) | T2 | 55.74 | 75.76 | 75.00 | - | 0.05 | - |
| I expect the present course to be difficult for me. (Mean) (5-point[4]) (#) | T1 | 3.2 | 2.8 | 2.9 | - | - | - |
| Algorithmic ability i.e. light bulb questionnaire item (Min. = 0, Max. = 5) (Mean) | T1 | 3.7 | 4.0 | 3.6 | - | - | - |
| The lecturer for this course was helpful. (Mean) (4-point[5]) (#) | T2 | 2.3 | 3.3 | 3.4 | 0.01 | 0.05 | - |
| The tutor for this course was helpful. (Mean) (4-point) (#) | T2 | 2.6 | 3.1 | 3.0 | - | - | - |
| The best computer programmers are creative. (Mean) (5-point) (#) | T1 | 3.5 | 3.5 | 3.6 | - | - | - |
| The best computer programmers plan work carefully to spend as little time as possible at the terminal. (Mean) (5-point) (#) | T1 | 3.4 | 3.2 | 3.3 | - | - | - |
| The best computer programmers prefer to write simple, specific programs to solve particular tasks. (Mean) (5-point) (#) | T1 | 3.5 | 3.6 | 3.6 | - | - | - |
| I have at times thought seriously of dropping this course. (%) (#) | T2 | 54.10 | 27.27 | 12.50 | 0.05 | 0.05 | - |
| The homework assignments have been more beneficial than the classroom presentations. (%) (#) | T2 | 77.05 | 57.58 | 67.50 | 0.05 | - | - |

[4] Rated on a 5-point Strongly Agree, Agree, Neither Agree nor Disagree, Disagree, Strongly Disagree scale.

[5] Rated on a 4-point Strongly Agree, Agree, Disagree, Strongly Disagree scale.

## C.3 Computer Anxiety and Alienation

These questionnaire items are the 14 Revised Anxiety and Alienation scale items proposed by Ray and Minch (1990). The mean results reflected below are the mean results obtained for each statement based on a five point rating scale. Responses were originally scored on a scale of 1 to 5 as follows: (1) strongly agree, (2) agree, (3) neither agree nor disagree, (4) disagree, (5) strongly disagree. The overall Anxiety and Alienation measure is the summation of all 14 items scores. The mean results for each student group are reflected in the final row of the table. High scores indicate a high computer anxiety and alienation measure.

| Statement | Time | Control '92 | Control '93 | PATMA N | 92 vs 93 | 92 vs PAT | 93 vs PAT |
|---|---|---|---|---|---|---|---|
| I clearly understand what input computers want. | T2 | 3.2 | 2.9 | 2.5 | - | 0.01 | - |
| I don't feel helpless when using the computer. | T2 | 2.7 | 2.8 | 2.3 | - | - | - |
| I am sure of my ability to interpret a computer output. | T2 | 3.3 | 3.0 | 2.7 | - | 0.01 | - |
| I don't understand computer output. *[6] | T2 | 2.8 | 2.6 | 2.3 | - | 0.01 | - |
| Working with computers is so complicated it is difficult to understand what is going on. * | T2 | 2.8 | 2.6 | 2.1 | - | 0.01 | - |
| I hesitate to use a computer for fear of making mistakes I cannot correct. * | T1 | 2.2 | 2.5 | 2.1 | - | - | - |
| I am confident that I could learn computer skills. | T1 | 1.6 | 1.7 | 1.6 | - | - | - |
| I like to use computers. | T2 | 2.5 | 2.2 | 2.0 | - | 0.05 | - |
| I don't care what other people say, computers are not for me. * | T2 | 2.5 | 2.6 | 2.1 | - | - | - |
| The computer interferes with my work. * | T2 | 2.7 | 2.4 | 2.1 | - | 0.05 | - |
| Our country relies too much on computers. * | T1 | 2.7 | 3.0 | 2.8 | - | - | - |
| Computers are changing the world too rapidly. * | T1 | 2.9 | 3.5 | 3.0 | - | - | - |
| The computer interferes with professional relationships among people. * | T1 | 2.8 | 2.7 | 2.6 | - | - | - |
| The computer doesn't interfere with my personal relationships with people. | T2 | 2.2 | 2.2 | 2.3 | - | - | - |
| **Anxiety & Alienation** | **T1/2** | **36.9** | **36.5** | **32.4** | **-** | **0.01** | **0.05** |

[6] Asterisked items are reverse scored.

## C.4 Programming assignments

| The program assignments | Time | Control '92 | Control '93 | PATMAN | 92 vs 93 | 92 vs PAT | 93 vs PAT |
|---|---|---|---|---|---|---|---|
| have been more difficult than I expected. * | T2 | 2.0 | 2.4 | 2.6 | - | 0.05 | - |
| have been more time consuming than I expected. * | T2 | 1.8 | 2.2 | 2.1 | - | - | - |
| have been more frustrating than I expected. * | T2 | 1.9 | 2.4 | 2.5 | - | 0.05 | - |
| were easy to do. | T2 | 1.9 | 2.0 | 2.2 | - | - | - |
| went smoothly. | T2 | 1.9 | 2.2 | 2.4 | - | 0.05 | - |
| **Easy Assignments** | **T2** | **1.9** | **2.2** | **2.4** | **-** | **0.05** | **-** |

## D. WORKSHEETS 1, 2 AND 3

# Intro. to Programming: Worksheet 1

Student Number: _________

Please answer all questions in the stipulated format. Assume the user enters all the numbers in the input sequence at one time.

1. Given the following program code and input, what will be the value of the variables, after line 5 of the program has been executed.

```
PROGRAM one;
VAR Even, Odd : INTEGER;
BEGIN
  WRITELN ('Enter four numbers: ');
  READLN (Even,Odd);
END.
```

Input: 3 2 10 5 Variable Values Even:____ Odd: _____

How difficult did you find this question? (Circle the number which best describes your difficulty rating .)

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

2. Given the following program code and input, what will be the value of the variables, after line 5 of the program has been executed.

```
PROGRAM two;
VAR b, c, a : INTEGER;
BEGIN
  WRITELN ('Enter three numbers:');
  READLN (c, b, a );
END.
```

Input: 15 25 20 Variable Values a:____ b:____ c: ____

How difficult did you find this question? (Circle the number which best describes your difficulty rating .)

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

3. Given the following program code and input, write the output in the block provided.

```
PROGRAM three;
VAR  x : INTEGER;
BEGIN
  WRITELN ('Enter a number.');
  READLN (x);
  WRITELN (x);
  WRITELN ('The value of x is 5');
  WRITELN (x);
END.
```

Input : 6 3 4 2 4 1 8

Output

How difficult did you find this question? (Circle the number which best describes your difficulty rating .)

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

4. Given the following program code and input, write the output in the block provided.

```
PROGRAM four;
VAR max, min, first, last: INTEGER;
BEGIN
  WRITELN('Enter a list of numbers');
  READLN (max, min, first, last);
  WRITELN('Largest Number:', max);
  WRITELN('Smallest Number:',min);
  WRITELN('Last Number:', last);
  WRITELN('First Number:', first);
END.
```

Input: 5 13 1 6

Output

How difficult did you find this question? (Circle the number which best describes your difficulty rating .)

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

Please note: Your answers to this worksheet will be used for research only and in no way will it effect your final result.

# Intro. to Programming: Worksheet 2

Student Number: ________

1. For each program, examine the given program code and the input and then write the output in the block provided. Assume the user enters all elements in an input sequence at one time.
2. Also indicate how difficult you found the question. Circle the number which best describes your difficulty rating.

Please note: Your answers to this worksheet will be used for research only and in no way will it effect your final result.

```
PROGRAM One;
VAR a,b,c:INTEGER;
BEGIN
  WRITELN('Enter two numbers: ');
  READLN(a,b);
  WRITELN(a);
  WRITELN(b);
  b:=a;
  a:=a+1;
  c:=a+b;
  WRITELN(a);
  WRITELN(b);
  WRITELN(c);
END.
```

Input: [2 3 4][1 0]

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

```
PROGRAM Two;
VAR i,x:INTEGER;
BEGIN
  FOR i:=1 TO 3 DO
  BEGIN
    WRITELN('Enter a number: ');
    READLN(x);
    WRITELN(x);
  END;
END.
```

Input:[6 3][3 4 5][2 1][8]

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

```
PROGRAM Three;
VAR letter:CHAR;
BEGIN
  WRITE('Enter a character: ');
  READLN(letter);
  REPEAT
   WRITELN('You entered letter:',letter);
   WRITE('Enter a character: ');
   READLN(letter);
  UNTIL (letter = 'N') or (letter = 'n');
  WRITELN(letter);
END.
```

Input:[h][Q][n][N][r]

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

```
PROGRAM Four;
VAR p,q:INTEGER;
BEGIN
  q:=0;
  WRITE('Enter a number:');
  READLN(p);
  WHILE p <> 0 DO
    BEGIN
      IF p > 0 THEN
        q:=q + 1;
      WRITE('Enter a number: '):
      READLN(p);
    END;
  WRITELN(q);
END.
```

Input:[1][-1][-3][4][0]

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

```
PROGRAM Five;

VAR num,val:INTEGER;
BEGIN
  FOR num:=1 TO 3 DO
    BEGIN
      WRITELN('Enter a number: ');
      READLN(val);
    END;
  WRITELN(val);
END.
```

Input:[6][3][2][8]

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

# Intro to Programming: Worksheet 3

Student Number: ______

1. For each program, examine the given program code and the input and then write the output in the block provided. Assume the user enters all elements in an input sequence at one time.

2. Also indicate how difficult you found the question. Circle the number which best describes your difficulty rating.

Please note: Your answers to this worksheet will be used for research only. They will have no effect on your final result.

```
PROGRAM One;
VAR number:INTEGER;
BEGIN
  FOR number:=1 TO 3 DO BEGIN
    WRITELN(number*2);
  END;
END.
```

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

```
PROGRAM Two;
VAR x,y:INTEGER;
BEGIN
  WRITELN('Enter a number: ');
  READLN(x,y);
  IF x = 4 THEN
    WRITELN(x)
  ELSE WRITELN(y);
END.
```

Input:[3 5]

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

```
PROGRAM Three;
VAR number:INTEGER;
BEGIN
  WRITE('Enter a number: ');
  READLN(number);
  IF number = 7 THEN
    WRITELN('Unlucky number');
  IF number = 10 THEN
    WRITELN('Lucky number');
  WRITELN('The number was',number);
END.
```

Input:[4][10][7]

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

```
PROGRAM Four;
VAR number:INTEGER;

PROCEDURE Letters;
  BEGIN
    WRITELN('ijkl');
    WRITELN('mnop');
  END;

BEGIN
  WRITE('qrst');
  Letters;
  Letters;
END.
```

Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |

The following program should read a list of five test scores and report the number of failing scores (failing score is less than 50). The program, however produces the output listed below. Correct the program, marking the changes in the program code.

```
PROGRAM Failing;
VAR count,score,i:INTEGER;
BEGIN
  count:=0;
  FOR i:=1 TO 5 DO
  BEGIN
    WRITE('Enter a score: ');
    READLN(score);
    IF score < 50 THEN
      i:=i+1;
  END;
  WRITELN('Number of failing scores:',count);
END.
```

```
Enter a score: 45
Enter a score: 100
Enter a score: 55
Enter a score: 35
Enter a score: 60
Number of failing scores:0
```

Input:[45][100][55][35][60]

Faulty Output

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Very Easy | Easy | Average | Difficult | Very Difficult |